Parallel LZ77 Decoding with a GPU

Emmanuel Morfiadakis
Supervisor: Dr Eric McCreath
College of Engineering and Computer Science, ANU
Outline

- Background (What?)
- Problem definition and motivation (Why?)
- The chosen approach in detail (How?)
- Experimental results
- Conclusion/Future work
Data compression

- The process of converting digital data to a format that requires less bits to store/transmit than the original

- Many different compression/decompression algorithms exist nowadays
The LZ77 Compression Scheme

- Developed by Abraham Lempel and Jacob Ziv in 1977.
- Replaces occurrences of repeating data with tuples that reference previous occurrences of the same data.
- Useful when data repeats often, for example DNA sequences, literature works, etc.
Example of LZ77 encoding

“abcdefgabcdefg”

LZ77 Encoder

abcdefg(7,7)
Graphics Card Computing

- Early GPU usage revolved around rendering images, videos, and video game graphics.


- Currently, there are many APIs that allow usage of GPUs for general purpose computing, including CUDA from NVIDIA and OpenCL by the Khronos Group.
Parallel LZ77 Decoding with a GPU

- Speed up the decoding process of the LZ77 algorithm by completing this procedure on a graphics card

- Why? It is still used widely: pzip, gzip, Microsoft Windows, PDF, PNG, MNG, HTTP etc.
Approach

- Develop a serial implementation in C
- Develop a parallel implementation in CUDA
- Benchmark them against each other to determine if there are performance gains from the parallel implementation
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

  abd(3,3)c(1,1)(4,2)e(4,3)(2,2)
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

  \[ \text{abd}(3,3)c(1,1)(4,2)e(4,3)(2,2) \]
  \[ \text{abd} \]
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

  \[ abd(3,3)c(1,1)(4,2)e(4,3)(2,2) \]

  \[ \text{abd abd} \]
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

  $$\text{abd}(3,3)c(1,1)(4,2)e(4,3)(2,2)$$

  $\downarrow$

  $$\text{abd abd c}$$
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

  \[
  \text{abd}(3,3)\text{c}(1,1)(4,2)\text{e}(4,3)(2,2)
  \]

  \[
  \downarrow
  \]

  \[
  \text{abd abd c c}
  \]
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

  abd(3,3)c(1,1)(4,2)e(4,3)(2,2)

    abd abd c c bd
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

\[
\text{abd}(3,3)\text{c}(1,1)(4,2)\text{e}(4,3)(2,2)
\]

\[
\text{abd} \quad \text{abd} \quad \text{c} \quad \text{c} \quad \text{bd} \quad \text{e}
\]
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

  abd(3,3)c(1,1)(4,2)e(4,3)(2,2)

  ↓

  abd abd c  c  bd  e  cbd
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

```
abd(3,3)c(1,1)(4,2)e(4,3)(2,2)
```

```
abd abd c  c  bd  e  cbd  bd
```
The chosen approach - Serial decoding

- Simple: start from the beginning and decode one tuple/character at a time:

  \[
  \text{ab}(3,3)\text{c}(1,1)\text{e}(4,2)\text{d}(4,3)(2,2) = \text{abda}d\text{dbd}\text{cccc}\text{bdec}\text{bbdbd}
  \]
The chosen approach - Parallel decoding

- Main problem: If everything is done in parallel, how do we know that the part of the output that we require has been decoded?

- Example: abcd(1,1)(2,2)
The chosen approach - Parallel decoding

- The source of trouble is the backreference tuples, literals can be decoded without any issues in parallel.

- Adopted approach: resolve backreference tuples according to their depths.

- Depth is the maximum number of other tuples/literals that must be decoded before the current tuple can be resolved.
Going back to the previous example:

\[ abd(3,3)c(1,1)(4,2)e(4,3)(2,2) \]
Going back to the previous example:

$$\text{abd}(3,3)\text{c}(1,1)(4,2)e(4,3)(2,2)$$
Going back to the previous example:

$$\text{abd}(3,3)c(1,1)(4,2)e(4,3)(2,2)$$
Going back to the previous example:

\[ \text{abd}(3,3)\text{c}(1,1)(4,2)\text{e}(4,3)(2,2) \]

\[ \downarrow \quad \downarrow \quad \downarrow \]

\[ 0 \quad 1 \quad 2 \]
Parallel decoding

\[ abd(3,3)c(1,1)(4,2)e(4,3)(2,2) \]

0 1 0 1 2 0 3 4
Parallel decoding

\(\text{Round 0: abd c e}\)
Parallel decoding

\[ \text{Round 1: } \text{abdcde} \]
Parallel decoding

\[ \text{Round 2: } \text{ab}d\text{ab}d\text{c}c\text{b}\text{c}\text{b}\text{d}e \]
Parallel decoding

\[ \text{Round 3: } \text{abdabdccbdecbd} \]
Parallel decoding

$abd(3,3)c(1,1)(4,2)e(4,3)(2,2)$

Round 4: $abdabdccbdecdbdbd$
Parallel decoding

\[ \text{abd}(3,3)\text{c}(1,1)(4,2)\text{e}(4,3)(2,2) = \text{abdabdcdbdecdbdbd} \]
In the software

- Blocks of a constant number of elements
- Each element in position $i$ of the block is decoded/copied by thread $i$
- Synchronisation is done with a bit array
- When thread $i$ completes its workload it sets element $i$ of the array to 1
In the software - Initially

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0, a)</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>(0, 0, b)</td>
<td>(0, 0, e)</td>
</tr>
<tr>
<td>(0, 0, c)</td>
<td>(0, 0, c)</td>
</tr>
<tr>
<td>(2, 1)</td>
<td>(2, 2)</td>
</tr>
<tr>
<td>(4, 3)</td>
<td>(0, 0, a)</td>
</tr>
</tbody>
</table>

T0  T1  T2  T3  T4

1 - 1 0 - 2

[0, 0, 0, 0, 0]
In the software - Round 1
In the software - Round 2

Block 1

(0, 0, a) (0, 0, b) (0, 0, c) (2, 1) (4, 3)

T0 T1 T2 T3 T4

1 - 1 0 - 2

[1, 1, 1, 1, 1]

Block 2

(2, 1) (0, 0, e) (0, 0, c) (2, 2) (0, 0, a)
Experimental setup

- Intel Core i7 4510U running at 2 GHz with 4 GB of RAM and an NVIDIA GeForce 840M GPU, clocked at 1.029 GHz and with 2GB RAM

- Decode each dataset 10 times, calculate the average time taken

- Measure only the decoding part, not parsing, file I/O etc.
Evaluation

- Memory Transfer time: Is the time taken to transfer data to the GPU from the CPU a significant performance bottleneck?

- Synchronization Cost: Are there any slowdowns caused by thread interaction?

- Performance: Which one is faster? Parallel or serial?
Memory Transfer times

File Name

- Bible
- Char
- DNA
- HelloWorld
- Lorem
- Paper
- Paper2
- Random

Elapsed time (ms)
Memory Transfer times

- Grow slowly with respect to the parallel decoding time
- Only cause the parallel implementation to miss its opportunity to outperform the serial one for small files
- For larger file sizes (>200 KB) memory transfer times are much less than 1% of the total decoding time
Performance

![Bar chart showing elapsed time for different file names using parallel and serial decoding.]
Performance

- Parallel is much slower, except for smaller files

- But even then, the memory transfer times stall the parallel implementation long enough for the serial one to outperform it
Why the slowdown?
Three key reasons

- Memory/Execution dependencies
  - Must read/write and fetch a lot of data, this slows things down

- Synchronisation
  - A lot of time is spent waiting for other threads to complete their resolution rounds
Conclusion

- LZ77 is inherently sequential! A lot of overhead is introduced by trying to synchronise threads, calculating the writing locations of the output, etc.

- In contrast, the serial implementation only requires knowledge of the current writing position and the output up to that point.

- Memory transfers stop being significant after a certain threshold (~200 KB).

- Thus: parallelizable, but we should not expect good performance.
Future work and extensions

- Investigate other, less memory-intensive methods of synchronisation
- Using a different language for the parallel approach
- Determine if the scheme can benefit from parallel encoding
Thank you for listening - Q & A