CULZSS

LZSS Lossless Data Compression on CUDA

Adnan Ozsoy & Martin Swany

DAMSL - Distributed and MetaSystems Lab

University of Delaware
Indiana University-Bloomington
September 2011
OUTLINE

• Introduction
• Background
• Implementation Details
• Performance Analysis
• Discussion and Limitations
• Related Work
• Future Work
• Conclusion
• Q & A
INTRODUCTION

- Utilization of expensive resources usage
  - Memory
  - Bandwidth
  - CPU usage
- Data compression
  - Trade off in increasing running time
- Graphics Processing Units
- CULZSS
  - General purpose data compression
  - Reduces the effect of increasing time compared to CPU
  - Keeping the compression ratio
BACKGROUND

• LZSS Algorithm
  • Lempel–Ziv–Storer–Szymanski
  • A variant of LZ77
  • Dictionary encoding
    • Replace a string of symbols with a reference to a dictionary location of the same string
    • A minimum number matching is required
    • A bit flag to indicate encoding or not
  • Why LZSS
    • No prior knowledge, no transformation needed
BACKGROUND

- LZSS Algorithm
BACKGROUND

- LZSS Algorithm
- Example:

<table>
<thead>
<tr>
<th>Index</th>
<th>Original String</th>
<th>LZSS Representation</th>
<th>Total Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I meant what I said</td>
<td>I meant what I said</td>
<td>102</td>
</tr>
<tr>
<td>20</td>
<td>and I said what I meant</td>
<td>(12,7)(7,8)(2,5)</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>From there to here</td>
<td>(51,4)</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>from here to there</td>
<td>f(46,4)(51,8)(50,5)</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>I said what I meant</td>
<td>(24,19)</td>
<td></td>
</tr>
</tbody>
</table>

Total characters 102
IMPLEMENTATION DETAILS

- LZSS CPU implementation
- CUDA implementation has two versions
  - API
  - `gpu_compress( *buffer, buf_length, **compressed_buffer, &comp_length, compression_parameters);`
  - `gpu_decompress(*buffer, buf_length, **decompressed_buffer, &decomp_length, compression_parameters);`
  - Version 1
  - Version 2
- Decompression
- Optimizations
IMPLEMENTATION DETAILS

- LZSS CPU implementation
  - Straight forward implementation of the algorithm
  - Mainly adapted from Dipperstein’s work
  - To be fair to CPU – a thread version is implemented with POSIX threads.
  - Each thread is given some chunk of the file and chunks compressed concurrently, and reassembled.
IMPLEMENTATION DETAILS

- CUDA implementation
  - Version 1
    - Similar to Pthread implementation
    - Chunks of file is distributed among blocks and in each block smaller chunks are assigned to threads
IMPLEMENTATION DETAILS

- CUDA implementation
  - Version 2
    - Exploit the algorithm’s SIMD nature
    - The work distributed among threads in the same block is the matching phase of the compression for a single chunk
IMPLEMENTATION DETAILS

- Version 2
  - Matching phase
  - CPU steps
    - Encoded flags
    - Reassemble
    - Append block compression sizes
IMPLEMENTATION DETAILS

- Version 2 - Matching phase
IMPLEMENTATION DETAILS

- Decompression
  - Identical in both versions
  - Each character is read, decoded and written to output
  - Independent behavior exist among blocks
  - A list of compressed block sizes needs to be kept
  - The length of the list depends on the number of blocks
    - Negligible impact on compression ratio
IMPLEMENTATION DETAILS

- Optimizations
  - Coalesced access to global memory
    - Access fit into a block, done by just one memory transaction
    - Access to global memory is needed before and after matching stage.
    - In Version 1, each thread reads/writes buffer size of data
    - In Version 2, each thread reads/writes 1 byte of memory
    - For that reason test results give the best performance with 128 bytes of buffer size and 128 threads per block which leads one 128 bytes segment transaction in Fermi architectures
IMPLEMENTATION DETAILS

- Optimizations (cont.)
  - Shared memory usage
    - Shared memory is divided into banks, and each bank can only address one dataset request at a time, if all requests from different banks, all satisfied in parallel
    - Using shared memory for sliding window in Version 1 gave a %30 speed up
    - Access pattern on Version 2 is suitable for bank conflict free access where one thread can access one bank
    - Speed up is shown on analysis
IMPLEMENTATION DETAILS

- Optimizations (cont.)
  - Configuration parameters
    - Thread per block
    - V1 has limitations on shared memory
  - Buffer size
    - V2 consumes double buffer size than a single which limits the encoding offsets into 16 bit space
  - Buffer sizes 128 bytes
  - 128 threads per block
PERFORMANCE ANALYSIS

- Testbed Configurations
  - GeForce GTX 480
    - CUDA version 3.2
    - 480 cores – 1.5 GB of memory
  - Intel® Core™ i7 CPU 920 at 2.67 GHZ
    - 4 physical – 8 logical cores
    - 4 GB memory
PERFORMANCE ANALYSIS

- Datasets
  - Five sets of data
    - Collection of C files
    - Delaware State Digital Raster Graphics and Digital Line Graphs Server – basemaps for georeferencing and visual analysis
    - English dictionary – alphabetical ordered text
    - Linux kernel tarball
    - Highly compressible custom data set – contains repeating characters in substrings of 20
PERFORMANCE ANALYSIS

Compression Benchmark Average Running Times (in seconds)

<table>
<thead>
<tr>
<th></th>
<th>Serial LZSS</th>
<th>Pthread LZSS</th>
<th>BZIP2</th>
<th>CULZSS V1</th>
<th>CULZSS V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C files</td>
<td>50.58</td>
<td>9.12</td>
<td>20.97</td>
<td>7.28</td>
<td>4.26</td>
</tr>
<tr>
<td>DE Map</td>
<td>30.75</td>
<td>6.25</td>
<td>9.14</td>
<td>4.69</td>
<td>15.00</td>
</tr>
<tr>
<td>Dictionary</td>
<td>56.91</td>
<td>9.35</td>
<td>20.18</td>
<td>7.13</td>
<td>3.22</td>
</tr>
<tr>
<td>Kernel tarball</td>
<td>50.49</td>
<td>9.16</td>
<td>20.45</td>
<td>7.08</td>
<td>4.79</td>
</tr>
<tr>
<td>Highly Compr.</td>
<td>4.23</td>
<td>1.2</td>
<td>77.82</td>
<td>0.49</td>
<td>3.40</td>
</tr>
</tbody>
</table>

Speed Up

Data Sets

- Serial LZSS vs BZIP2
- Serial LZSS vs Pthread
- Serial LZSS vs CUDA Version 1
- Serial LZSS vs CUDA Version 2
PERFORMANCE ANALYSIS

Compression Ratios (smaller is better)

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>BZIP2</th>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C files</td>
<td>54.80%</td>
<td>15.60%</td>
<td>55.70%</td>
<td>63.49%</td>
</tr>
<tr>
<td>DE Map</td>
<td>33.90%</td>
<td>11.80%</td>
<td>34.20%</td>
<td>33.35%</td>
</tr>
<tr>
<td>Dictionary</td>
<td>61.40%</td>
<td>34.50%</td>
<td>61.80%</td>
<td>65.09%</td>
</tr>
<tr>
<td>Kernel tarball</td>
<td>55.10%</td>
<td>16.90%</td>
<td>56.50%</td>
<td>62.59%</td>
</tr>
<tr>
<td>Highly Compr.</td>
<td>13.50%</td>
<td>0.40%</td>
<td>13.90%</td>
<td>6.34%</td>
</tr>
</tbody>
</table>

Decompression Benchmark Average Running Times (in seconds)

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>LZSS</th>
<th>CULZSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C files</td>
<td>1.79</td>
<td>0.53</td>
<td></td>
</tr>
<tr>
<td>DE Map</td>
<td>1.21</td>
<td>0.49</td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td>2.02</td>
<td>0.55</td>
<td></td>
</tr>
<tr>
<td>Kernel tarball</td>
<td>1.77</td>
<td>0.56</td>
<td></td>
</tr>
<tr>
<td>Highly Compr.</td>
<td>0.71</td>
<td>0.27</td>
<td></td>
</tr>
</tbody>
</table>
DISCUSSION AND LIMITATIONS

- Version 1 gives better performance on highly compressible data
  - Can skip matched data
- Version 2 is better on the other data sets
  - Cannot skip matched data
  - Better utilization of GPU (coalescing accesses and avoiding bank conflicts) leads better performance
- Two version and the option to choose the version in API gives users the ability to use best matching implementation
RELATED WORK

- CPU based
  - Parallelizing with threads
- GPU based
  - Lossy data compression has been exposed to GPU community
    - Image/Video Processing, Texture Compressor, ... etc
  - Lossless data compression
    - O’Neil et al. GFC – specialized for double precision floating point data compression
FUTURE WORK

- Case study – network based application
- More detailed tuning configuration API
- Combined CPU and GPU heterogeneous implementation to benefit for future proof architectures that have both of them on the same die; AMD Fusion, Intel Nehalem, ...
- Multi GPU usage
- Overlapping computations with CPU and GPU in a pipelining fashion
CONCLUSION

- CULZSS shows a promising usage of GPUs for lossless data compression
- Main goal being achieved by better performance than CPU based implementation
- To our best knowledge first successful improvement attempt of lossless data compression for any kind of general data.
- Outperformed the serial LZSS by up to 18x, Pthread LZSS by up to 3x in compression time
- Compression ratios kept very similar with CPU version
Q & A