File compression with LZO algorithm using NVIDIA CUDA architecture

L. Erdődi*

* Óbuda University, Faculty of John von Neumann, Budapest, Hungary
erdodi.laszlo@nik.uni-obuda.hu

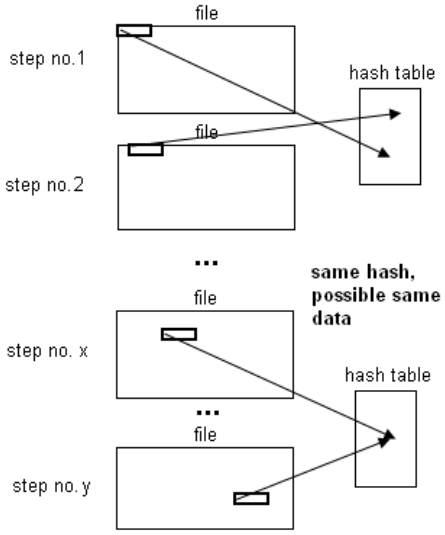
Abstract— **File compression in the case of large files can be time consuming and it is not even necessarily effective. Vast majority of the compression software use algorithms with implementations for CPU architecture. From the beginning of the 2000's the performance of graphic processing units (GPU) have been continuously increasing and at the present time in some cases the GPU exceeds the CPU in performance. However this high performance of the GPU is rarely exploited except in the case of some special tasks such as password cracking or linear algebra calculations. One of the most well-known compression algorithms is the LZO (Lempel-Ziv-Oberhumer). This study discusses the possible ways for the implementation of LZO for GPU Fermi architecture. Three different algorithms are provided and compared and finally it is also shown that the use of GPU can significantly decrease the time of the file compression.**

I. Introduction

Several algorithms and implementations have been developed for file compression in the past years. The most well-known methods of these are the LZW (Lempel-Ziv-Welch) and the LZO (Lempel-Ziv-Oberhumer) algorithms. The implementation of these algorithms runs on one thread without parallel procedures. In the case of the LZW algorithm a large dictionary file is built which is interpretable only when it is read together with the compressed file from the beginning of the decompressing process. So this way paralleling is impossible. The LZO algorithm manages the files to be compressed in blocks so theoretically the paralleling is possible when the parallel procedures are running in different file-blocks at the same time. However at the present time because of the efficient use of the CPU processor architecture (L1, L2, L3 cache) the implementation of the LZO algorithm uses only one thread for the compression.

From the 2000's GPU architectures have appeared. Initially GPU (graphics processing unit) was only used for the increasing of the speed of graphical procedures. However it turned out that GPU can be effective in the case of several well-parallelable procedures such as e.g. linear algebra calculations or password cracking. Nowadays GPU hardware can possess as good 1024 processors (e.g. NVIDIA GTX680) and this can greatly increase the calculation speed.

According to the above mentioned the possibility of the use of GPU for the acceleration of file compression arises. The present study discusses in details the implementation of LZO algorithm for NVIDIA CUDA architecture.



with lzo algorithm

II. The LZO compression algorithm

The LZO algorithm is introduced in the following [1]. The file to be compressed is cut into file-blocks which have the same size as the L2 cache of the processor. The compression of each file-block is done the following way:

During the processing of the file-blocks hash values are being established per each byte group of four. The value of the hash function is formed from the combination of the value of the actual and the three previous bytes (see Figure 1). During the compression a hash table is kept which is able to store one memory address for each hash value. The hash function is chosen in a way that the size of the hash table (memory address size * hash variations) equals to the size of L1 cache. This secures the quick run of the algorithm.

According to Figure 1 when from the combination of the zeroth, first, second and third bytes a hash is formed, the memory address of the pointer assigned to the third byte is written into the given place of the hash table. The process goes on like this, the fourth memory address is written into the place of the value from the hash of the 1st, 2nd, 3rd, 4th places of the hash table.

Suppose that a text file with the following data is compressed:

"SOMETHING IS A THING, THAT IS IMPORTANT"

978-1-4673-4519-4/12/$31.00 ©2012 IEEE

Let the pointer assigned to the beginning of the text be PTR0. Into the hash place ('S','O','M','E') of the hash table PTR0+3 is written, into the place of hash (O, M, E, T) PTR0+4 is written, etc.

In order to detect the recurrences the algorithm checks the already existing values when writing into the hash table. If the new memory address and the initial address (a random number at the beginning) are close values, the algorithm will check whether there is a real recurrence, so the byte groups of four are compared (this comparison is necessary because of the initial random hash values and the hash collision). In the case of the example above the hash (T, H, I, N) is calculated in the 16th step, so the value PTR0+15 should be written into the table. However there already exists a memory address which is the value PTR0+7 (it has been written into the table at the 5th step, at the word "something"). In order to decide whether there is a hash collision or not, the algorithm compares the values:

^PTR0+7 == ^PTR0+15 and ^PTR0+6 == ^PTR0+14 and ^PTR0+5 == ^PTR0+13 and ^PTR0+4 == ^PTR0+12

If the condition above is fulfilled, it means that the same byte group of four is at both places (in this case the „thin" part of the word). At this place the file can be compressed in that way that instead of the recurrence the initial position and length of the original word-part is written into the given position.

"SOMETHING IS A (RECURRENCE from the place back 11 bytes, length: 5 bytes), THAT IS IMPORTANT"

For the determination of the length of the recurrence the check of the coincidence goes in the memory from bytes to bytes. In the case above ^PTR0+8 equals to ^PTR0+16 but in the next step this is not fulfilled anymore (space and comma characters are not coincide). It means that in the compressed file only the length and the object of the coincidence are stored.

There are other several subtle details hiding in the algorithm which are not discussed here, such as e.g. the most compressed description of the recurrence, or the size of the memory addresses difference where it is worth to take into account the recurrence.

The algorithm is highly efficient and quick because of the matching of the sizes of the file blocks and hash table to the L1 and L2 caches. In the case of a text file with many recurrences (e.g. log files) 5% compression of the file is often possible.

III. GPU ARCHITECTURES

GPU architectures appeared first in the 2000's. At the beginning their performance was hardly higher than the CPU performance at that time. However the improvement of the graphic cards has been enormous in the last years. One of the fundamental factors of this improvement was the increasing of the number of the processors on the graphic card. The first GPU had a single chip processor (NVIDIA GeForce256) while the latest GTX 680 graphic card possesses 1024 processors. Besides increasing the number of the processors there are several other improvements that promoted the speed increment, e.g. the invention of the shared memory with almost as high performance as the registers, the increasing of the latency of the device memory of the GPU and the introduction of the Fermi architecture on the graphic cards from the GTX480 model. The performance of the GTX 680 graphic card exceeds 3000 GFLOP/s (see Figure 2 [2]).

The performance values of Figure 2 show extreme differences between the CPU and the GPU, but it is important to know that these values are only theoretic. When using the GPU several performance decreasing factors have to be considered. The difference in the characteristics can be experienced when running algorithms which are well parallelable. The 1024

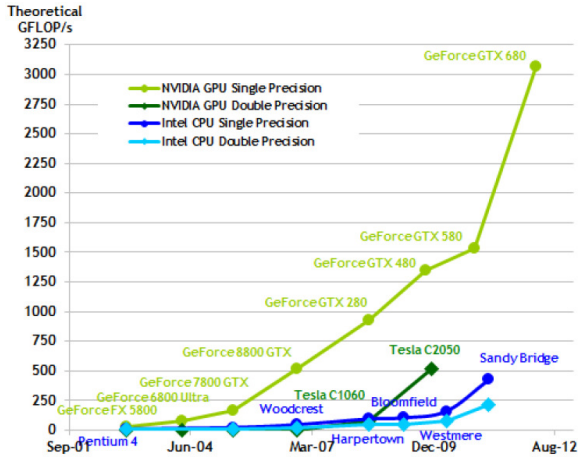


Figure 3. Performance comparison of CPU and GPU [2]

processors of the GTX680 card are working together in the groups of eight (multiprocessor), but if the reading of the data from the memory is slower than the calculation itself then the high performance of the graphic card will not reveal.

The architecture of the latest GPU is shown in Figure 3.

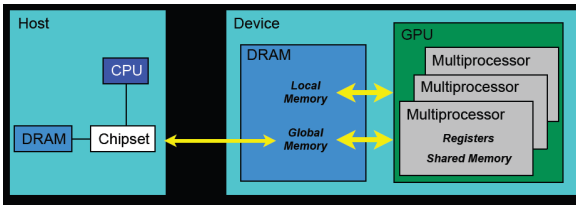


Figure 2. Architecture of NVIDIA GPUs [2]

The GPU device possesses own memory, where every data have to be copied into at the beginning of the calculation. After the completion of the calculation the data have to be copied again back to the host (CPU) memory. This can cause a huge overhead in the case of a compression algorithm since the copying of the file means loss of time.

The multiprocessors share the work between each other with computational blocks. The work-execution of the multiprocessors is uniform. This is determined by a general program code which is applied for its own block by each multiprocessor. When a multiprocessor is done with its work it starts to work on a new block. In the case of a GTX580 card which possesses 64 multiprocessors

(64x8 = 512 cores) it is worth distributing the task in a way that the number of the blocks is the multiple of 64. If the task comprises 65 blocks, the 64 multiprocessors will do the calculation, and then the first which finishes with its own block will start with the calculation of the 65th block. The other 63 multiprocessors will wait without work in this case.

The work-execution of the multiprocessors are done by the threads. In each block the work has to be divided into threadblocks. Similarly to the previously shown process the threadprocessors of the multiprocessor do the work of the threadblocks. When a threadprocessor is done with its work it starts to work on a new threadblock (so the number of the threadblocks should be chosen according to these). While the block-works of the multiprocessors are independent of eachother (and therefore it can be perfectly paralleled) the parallel work of the threads requires several limitations. Often the execution time is influenced by the appropriate application of the threadblocks. The threads are theoretically able to work independently, but the reading of the device memory is done in so-called non-coalesced way. It means if the threads read the data from the adjacent part of the memory then the reading will be parallel (coalesced access, see Figure 4 [3]), but if the reading is done from different memory parts then the multiprocessor will serialize the task so the execution can take even 8 times as slow as in other cases. This architectural solution plays a major role when implementing the LZO algorithm.

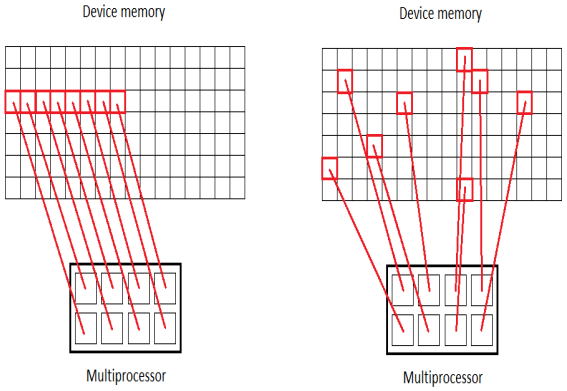


Figure 4. Coalesced and non-coalesced memory access [3]

IV. GPU LZO IMPLEMETATION POSSIBILITIES

The Lempel-Ziv-Oberhumer (LZO) is one of the fastest among the CPU compressing implementations. There is implementation for GPU but it uses and parallels the algorithm for zip files. In the following a new solution is presented where the possible GPU implementations of the LZO algorithm are introduced.

As it is previously pointed out the GPU multiprocessors can work independently. So for example in the case of a GTX580 graphic card (64x8 processors) the increase of the calculation speed can be theoretically 64 times. The first solution of the implementation is the following (Algorithm #1):

Let the blocksize be equal to the size of the L2 cache of the GPU card. Let us copy into the device memory data of the size L2 multiplied by the number of the multiprocessors. Run the compression process by blocks (one block for each multiprocessor). In one multiprocessor only one thread is enabled. Let the size of the hash table be equal to the size of the L1 cache of the multiprocessor (see Figure 5). Let us copy the compressed data back into the host memory.

Step 1: copying data of the size L2 multiplied by the number of the multiprocessors from the host memory into the device memory

Step 2: compression by blocks, one multiprocessor calculates one block, using one thread

Step 3: copying of the compressed data from the device memory back to the host memory

Step 4: repeating Steps 1-3 until it is required according to the file size

The time loss comes in this case from the followings:

- The file has to be copied into the device memory and then back to the host memory

- The reading of the device memory is slower during the compression process than the reading of the data in the L2 cache of the CPU

Regarding the GPU the following factors are increasing the calculation speed:

- One multiprocessor works with one thread so non-coalesced memory access has no possibility to occur

- The selected sizes of the hash table and the blocks are favorable for the Fermi architecture GPU

In the case of Algorithm #1 it is not taken into account that a multiprocessor possesses more than only one thread processor so the exploitage of the multiprocessor is small.

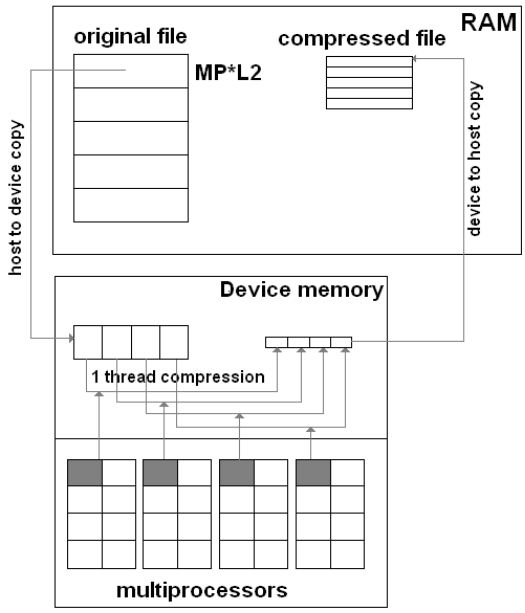


Figure 5. GPU compression Algorithm #1

In the case of Algorithm #2 this problem is about to be solved.

Algorithm #2: Let more than one thread processor work in one multiprocessor, so in this way a multiprocessor is able to compress more than one block at the same time.

Regarding the GPU the following factor is increasing the calculation speed:

- many blocks can be compressed at the same time (multiprocessor * thread count)

The time loss comes in this case from the followings:

- because of the continuous reading and writing of the hash table the execution of the threads within a multiprocessor is continuously being serialized so we get back to Algorithm #1 with the difference that the data size is bigger than the cache of the multiprocessor and this can cause time loss

In order to eliminate the drawbacks of the two previous algorithms a third has been worked out.

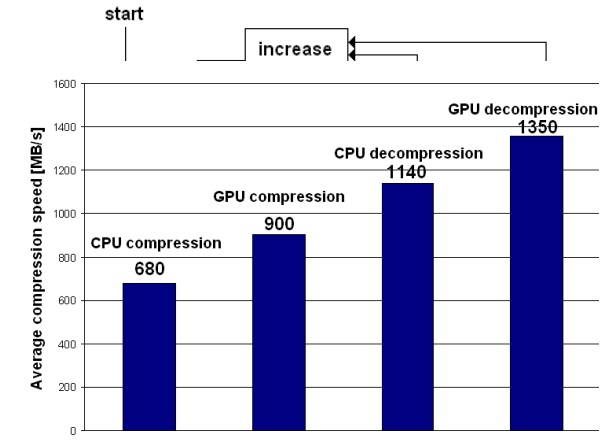


Figure 7. Measured performance of CPU and GPU (Algorithm #3) file compression using LZO algorithm

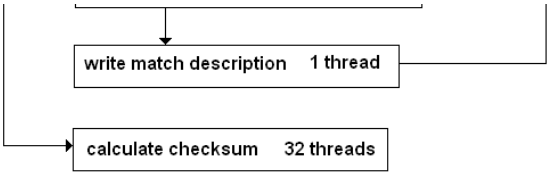


Figure 6. Algorithm #3 multiprocessor process flow

Algorithm #3 (see Figure 6): Only one multiprocessor is working on one block (similar to Algorithm #1) but there is a special thread of each multiprocessor which does the main calculations of the compression. In the case of the processes where the coalesced memory latency is possible (e.g. determination of the length of the recurrence or calculating the cheksum), the other thread processors start to work as well.

It results the increasing of the speed of Algorithm #1.

The previously introduced algorithms were tested on several files of different sizes and types. Figure 7 shows the compression and decompression speed of a large logfile (compression efficiency 6%).

V. Conclusion

Besides the introduction of GPU technology and parallel compressing algorithms, the present study introduced the implementation possibilities of the LZO algorithm on GPU architectures. GPU architectures are keeping developed but the main advantages which come from their architecture and the numbers of the processors can only be exploited when solving real parallel computing on them. Three new algorithms were introduced for the implementation of LZO algorithm on GPU. The calculation efficiency of Algorithm #3 exceeds the efficiency of the CPU compression implementation. Using NVIDIA GTX580 (512 cores) graphical processing unit with Intel Core i7-2600 CPU with Algorithm #3 the compression process is 20% faster with GPU than with CPU. The latest graphical processing unit nowadays is the NVIDIA GTX680 model. This unit possesses 1024 cores, this means that using this device with Algorithm #3 the calculation speed estimated to be even more than two times higher than with CPU.

The development of GPU devices is continuous so the performance is growing as well which means that the here presented speeds will increase (of course the CPU is being developed but according to the past and present developing processes the efficiency of the GPU for this special problem is expected to be better than the CPU efficiency). A great breakthrough for GPU compression will be if the thread processors are able to read the memory independently perhaps directly from the host memory (CPU RAM).

References

[1] M. F. X. J Oberhumer “LZO source code” www.oberhumer.com/opensource/lzo

[2] “NVIDIA CUDA Compute Unified Device Architecture – Programming Guide” *www.nvidia.com*, 2012.

[3] D. B. Kirk, W. W. Hwu “Programming Massively Parallel Processors – A Hands-on Approach” *NVidia, Morgan Kaufmann* Burlington, USA, 2010.