JPEG-GPU:: a GPGPU Implementation of JPEG Core Coding Systems

Ang Li
Dept. of Electrical and Computer Engineering, University of Wisconsin-Madison

Abstract

JPEG is a commonly used method of lossy compression for digital photography (image). This work targets on accelerating JPEG’s compressor and decompressor with GPU. Though the final results are not promising, I would like to introduce the lessons I have learned in accelerating a general system with GPGPU.

1 Introduction

In computing, JPEG is a commonly used method of lossy compression for digital photography (image). The degree of compression can be adjusted, allowing a selectable tradeoff between storage size and image quality. JPEG typically achieves 10:1 compression with little perceptible loss in image quality. [3]

JPEG compression is used in a number of image file formats. JPEG/Exif is the most common image format used by digital cameras and other photographic image capture devices; along with JPEG/JFIF, it is the most common format for storing and transmitting photographic images on the World Wide Web (WWW). These format variations are often not distinguished, and are simply called JPEG.

Fig. 1 shows the JPEG encoding and decoding system. For encoding, an image should first be partitioned into 8 × 8 blocks, then go through a color conversion pre-processing, a DCT transformation, quantization, a Zig-zag scan and then the final encoding, which are separated to DC/AC encoding. The decoding processes are reversed. This work targeted on the GPGPU implementation of this encoding/decoding system.

2 Implementation Details

In this section, the methodology and implementation details regarding to the GPU version of JPEG will be introduced. The basic idea of the implementation is: first profiling on compressing and decompressing a picture with a certain size (in my case 720 × 450) and find the largest time consumers; then for hotspot functions, checking for each function the parallelism; then I implemented both a GPGPU version and OpenMP version for these functions; and in the end I sought for possibilities of doing optimizations. I will partition this section into three parts: in subsection 2.1, I’ll report the methodology profiling was done and the profiling results; in subsection 2.2, some pre-work before GPU implementation and some caveats related to parallelisms are described; in subsection 2.3, I’ll introduce the method the baseline implementation was achieved from the profiling results; and in subsection 2.4, I’ll briefly go through possibilities of optimizations and the final optimizations I applied.

2.1 Profiling

Before the actual GPU implementation, I carried out profiling to figure out in the JPEG code, which functions are the hotspot functions and which took most of compressing and decompressing time.

A relatively small figure was used for profiling. The figure was the test image in JPEG library and it was of size 720×450 (shown in Fig. 2). There are two versions of the image: a bitmap(.bmp) version and a JPEG(.jpg) version, used in compressing and decompressing, correspondingly. I regard a figure with this

Figure 2: the image used for profiling
size as suitable for profiling in the sense that (i) for large parts of the algorithm are blockwised and images of these size already contain non-trivial number of blocks; and (ii) the profiling results can already be clearly observed for this image.

I used GNU’s profiler (gprof) to profile the JPEG’s compressing and decompressing code. gprof is a performance analyzing tool in Unix. It uses a hybrid of instrumentation and sampling and is an extension of the older “prof” Unix tool. Unlike prof, gprof is capable of limited call graph printing.[1]

The profiling results of top 20 hotspot functions in JPEG’s compression and decompression are shown in Fig. 3. Observations from the profiling results are: (i) For both compression and decompression, the largest timer consumers come from the coding system, i.e. the routines which are used for encoding and decoding the graphs; (ii) To my surprise in some sense, two functions which are related to color conversion rgb_ycc_convert and ycc_rgb_convert, which do not seem to me quite time-consuming, are at least called for a large number of times, and for ycc_rgb_convert in decompression stage, it takes significant amount of experimental time; (iii) Functions related to DCT transformation, quantization and Zigzag scan are seemingly to be ignorable in terms of timing results.

These profiling results gave me a first impression and direction of which functions are to be focused on when doing GPU implementation.

2.2 Pre-work

Before the actual GPGPU implementation, works to be done are as follows: (i) for each hotspot function, seeking the function for the possibilities of being parallelized; and (ii) for each selected function, first writing OpenMP pre-compiling headers to make sure that: (1) the compressing/decompressing processes can still function correctly, and (2) at least with OpenMP pre-compiling headers, these functions can gain some (even trivial) speedups. There are also some caveats in selecting the functions to be parallelized, which will be described along with seeking for the parallelism.

To select the functions to be parallelized, I went through each function to see whether it is parallelizable or not. In this stage, two kinds of functions are to be discarded: (i) functions without any parallelism, and (ii) functions which are called or calling by a function which has already been decided to be parallelized. Fig. 4 shows us an example which violates point (i). Observe that in this code snippet inside function encode_mcu_DC_refine, in the for-loop it calls another function emit_bits_s for three times. The problem is, the function emit_bits_s changes the state of the coding system once it is called (i.e. the function is not stateless and idempotent). The for-loop is actually not parallelizable. As a matter of fact, the DC encoding is sequential and no parallelisms can be dug out. The examples for violating point (ii) are functions which call the two routines rgb_ycc_convert and ycc_rgb_convert. As these two functions contain potentials to be parallelized.
and I have decided to make a GPU implementation, if the function callers do not contain other parallelisms, I will not bother making a GPU implementation of them. This makes sense in the sense of the granularities of parallelism, which will be reported in detail in subsection 2.3.

After selecting those functions to be parallelized, an OpenMP implementation was first applied on these functions. This is for: (i) making sure a correct understanding of JPEG’s compression and decompression codes, and (ii) make sure that some speedup can be achieved. Fig. 5 shows an example of the OpenMP implementation inside function rgb → ycc convert. There is no significant change for this version with the exception of adding a pre-compiling header.

Fig. 6 shows the image used for experiments, which is of size 1280 × 768. For simulation environments, I used Intel® Nehalem Xeon E5520 2.26GHz processor as the CPU node, while using Tesla K20c as the GPU node.

Table 1 shows the comparison of timing results of the sequential version and the OpenMP version with declaring four OpenMP threads. We can observe that trivial though it is, I have grabbed the parallelism and GPGPU implementation has the potential to help improve the performance.
<table>
<thead>
<tr>
<th>Type</th>
<th>Seq</th>
<th>OpenMP</th>
<th>GPGPU-base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>2.886</td>
<td>2.648</td>
<td></td>
</tr>
<tr>
<td>Decompress</td>
<td>2.420</td>
<td>2.200</td>
<td>11.32</td>
</tr>
</tbody>
</table>

Table 1: the timing comparison between sequential version and OpenMP version. All timings were measured in seconds.

2.3 Baseline Implementation

After the preliminary work stated in 2.2, the GPGPU implementation of each selected function was planted on GPU. GPGPU implementations need to (i) copying data to GPU global memory, (ii) doing calculations and (iii) transferring data back to host memory. Fig. 7 shows an example of transferring data from host to device. Note that for saving execution time, the GPU memory allocation functions such as `cudaMalloc` should be called only once while we do need to call `cudaMemcpy` each time an input block comes in unless this is a coefficient table, in which case `cudaMemcpy` is called only once. In subsection 2.4, the optimization on dealing with the coefficient table will be covered.

2.4 Optimizations

In this subsection, I’ll introduce the one point of optimization in this work: using constant memory. Actually there are other methods in GPGPU which can be leveraged to optimize the code: adjusting the configuration parameters (block size and grid size), using coalesced memory accessing, avoid branch warp divergence, etc. Nonetheless, the kernels I’ve written have already been configured to be the optimal, and for other optimization methods, those functions selected do not really contain these characteristics. For example, in most cases when we need to address with the input contents, as there is no way we can predict the those contents, it is thus not possible to shuffle the coefficient table somehow to make the access coalesced. And the for-loops selected almost do not contain `if-else` statements at all s.t. there is no chance for avoiding branch warp divergence. Thus the only point of optimization I’ve got in this work is leveraging constant memory.

Constant memory, as its name suggests, is for read-only memory. This is memory that is either declared at compile time as read only or defined at runtime as read only by the host (in this work, the latter case suffices). It is, therefore, constant only in respect of the GPU’s view onto memory. The size of constant memory is restricted to 64 K, or can be configured to only 16K to trade for larger cache sizes.[2]

Fig. 8 shows the kernel code of the for-loop in function `rgb_ycc_convert`. The observation is that: all entries accessed in this function are independent and is in a sense suitable to be implemented on GPU.

Table 2 shows the comparison of timing results of GPGPU-based version v.s. the sequential version and the OpenMP version with declaring four OpenMP threads. Unfortunately, instead of observing any performance gain, a great performance deduction was observed. The discussion and lessons learned will be covered in section 3.

<table>
<thead>
<tr>
<th>Type</th>
<th>Seq</th>
<th>OpenMP</th>
<th>GPGPU-base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>2.886</td>
<td>2.648</td>
<td>11.32</td>
</tr>
<tr>
<td>Decompress</td>
<td>2.420</td>
<td>2.200</td>
<td>11.23</td>
</tr>
</tbody>
</table>

Table 2: the timing comparison between sequential version, OpenMP version and GPGPU-base version. All timings were measured in seconds.

3 Discussions of Results, Conclusions and Future Works

In this section, I’ll give brief discussions on the results and reach conclusions. I’ll also give instructions on potential future works.

In this work, extremely negative results which go against GPU were observed. These are contributed by several reasons: (i) The major reason lies in the
Figure 8: an example to show the kernel called by function rgb_ycc_convert

<table>
<thead>
<tr>
<th>Type</th>
<th>Seq</th>
<th>OpenMP</th>
<th>GPGPU-base</th>
<th>GPGPU-opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>2.886</td>
<td>2.648</td>
<td>11.32</td>
<td>19.05</td>
</tr>
<tr>
<td>Decompress</td>
<td>2.420</td>
<td>2.200</td>
<td>11.23</td>
<td>17.72</td>
</tr>
</tbody>
</table>

Table 3: the timing comparison between sequential version, OpenMP version and both GPGPU versions. All timings were measured in seconds.

fact that the parallelism found was too fine-grained, but GPGPU acceleration was coarse-grain targeted. For most (if not all) re-written functions, they will only deal with one block, or at most data size to the order of magnitude of thousand bytes. In this case, even a call to memory allocation or memory copy or a kernel launch will cost time to the order of magnitude of milliseconds even though the actual calculation inside the kernel is way faster. If there existed a coarser-grained implementation, on the other hand, compared with the calculations, memory copy or kernel launch time will be only small overheads; (ii) There are non-trivial number of times where the global memory is addressed by the input data, which in most cases have negative impact on coalesced memory accesses. An observation is this: coalesced memory access has a theoretical maximum speedup over non-coalesced memory access of 16, in the case where all threads in a half-warp (a warp contains 32 threads) diverge. (iii) Regarding to the performance deduction after applying constant memory, there is a caveat to constant memory which has probably been violated. The caveat is this: unlike global memory, accesses to constant memory will get serialized (i.e. split into multiple transactions) if they are not uniform (i.e. all threads of a (half-)warp access the same address). Due to the problem of addressing by contents (most (if not all) of the constant memories are coefficient tables), these accesses to constant memories can easily get serialized and thus we are worse off using constant memory.

I conclude that in this work, I got an OpenMP and two GPGPU implementations of JPEG compressor and decompressor. Though achieving modest speedups with OpenMP version with four OpenMP threads, rather than achieving any speedup, performance deductions were observed. I learn the lesson that GPGPU is only coarse-grain targeted and should not be used to speedup fine-grained functions. This in turn casts doubts on the profiling methods before GPU implementation: gprof gives us only the hotspot functions, but does not provide many hints on what functions are the suitable level for parallelism, especially for GPU. Thus probably instead of accelerating the hotspots function, it might be a good idea to expand the hotspot functions on its callers and parallelize the caller functions.

For future work instructions, the work to the highest priority is: reviewing the JPEG code and find functions which deal with several independent blocks. Then probably the callees inside this function can be expanded somehow for parallelizing.

Acknowledgement

I would like to give my thanks to Professor Yu Hen Hu for providing the JPEG code and give me the chance to know more about the JPEG coding system as well as some principles in using GPU to accelerate some systems.
References

