POLITECNICO DI TORINO

Master degree
in Computer engineering

GPU programming - project report

Parallelization of a JPEG decompression library on a CUDA GPGPU

Candidate
Lorenzo Chiola s287911

Academic Year 2022-2023

Summary

Application of CUDA Parallel Computing to the JPEG decompression algorithm, in the form of the already existing NanoJPEG library. The resulting program is tested on an NVidia Jetson Nano GPU board, showing a significant performance increase on images bigger than one megapixel.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

JPEG image decompression happens very often in modern consumer software. Small images such as icons on web pages may be decoded in general purpose CPUs with sufficient performance, however handling big images on dedicated hardware could at least offload the CPU, but also speed up quite substantially.

This document describes the effort of exploiting **NVIDIA CUDA** parallel computation inside **NanoJPEG**, a JPEG decoding library. The sources of NanoJPEG are freely available from its writer's website [1]. The document consists of:

- chapter 2 essential JPEG notions needed for decoding
- chapter 3 details about the CUDA implementation
- chapter 4 where results are presented illustrating the benefits obtained with the parallel implementation and some conclusions on the use of a parallel approach for JPEG encoding are drawn.

Chapter 2

Background

2.1 JPEG

It is not the purpose of this document to give a comprehensive description of the JPEG/JFIF format, or the algorithm. Previous work about compression of JPEG images using CUDA parallel acceleration was done by Simone Pistilli: his paper [2] also gives a more in-depth description of the file format and compression algorithm.

JPEG is a standard for the compression of digital images. It takes the name of its creators, the Joint Photographic Experts Group. JPEG images are commonly stored in JFIF (JPEG File Interchange Format) files.

2.1.1 JPEG encoding algorithm

The JPEG compression exploits the relatively poor sensitivity of the eye to variations of high-frequency content in images. Images are converted from spacial information to frequency information in the horizontal and vertical directions, by applying the Dual Cosine Transform on blocks of 8x8 pixels, simply called "blocks". This generates an equivalent amount of data (8x8 cells) which represents the frequency content of the original block as shown in Figure 2.1. Cells representing small high frequency content are then dropped (i.e. set to zero) by integer multiplication with a weighting table (Figure 2.2) which can be more or less aggressive depending on desired quality. Then the block is serialized in a zig-zag order from low to high frequency (see Figure 2.3, and Huffman-encoded, finally achieving compression.

The first cell (0, 0) of the frequency-domain block contains the zero-frequency content, or simply the average brightness of the block. It is called "DC coefficient" of the block. This cell is compressed differently from the other 63 (which compose the AC coefficient). The DC component is stored as DPCM (Differential Pulse Code Modulation), which practically means saving only the difference from the last data point (DC component of the last block). The sequence of differences restarts periodically as defined by the Restart Interval.

This is enough to compress single channel (i.e. black and white) images. To represent color, each channel is represented separately. While the RGB representation can be used,

it is most common to represent color images in the YCbCr format (sometimes improperly called YUV in industry). The YCbCr color space separates luminance (Y) from chrominance (Cb, Cr) components, allowing then to subsample the chrominance component (for example 2:1 reduction in both axes is labeled "4:2:0") which is perfectly acceptable to the eye, allowing further data compression without perceived loss of quality.

2.1.2 JPEG File Format (JFIF)

The container format of interest is JFIF, because it is the most widespread one (any file with `.jpg` extension is a JFIF container).
JFIF files are made of sections which can be distinguished by the tags that precede them. Below is a list of the most important ones:

- SOF (Start Of Frame): contains general information like width, height, color space.
- DHT (Define Huffman Table): describes a Huffman compression tree. For YCbCr there are usually four DHTs: two for the Y channel (one for AC components, the other for DC components) and two different ones for the Cb and Cr channels (again separated for DC and AC components).
- DQT (Define Quantization Table): describes how the low-weight, high-frequency cells are dropped. Usually there are two, one for the Y channel and another one, usually more aggressive, for Cb and Cr.
- DRI (Define Restart Interval): specifies how often the DC component encoding restarts from zero.
- (Start of) Scan: the main section, which contains the actual data. For every block

Thankfully the format is already handled by NanoJPEG, so only a superficial understanding of the format is needed. For the purpose of porting NanoJPEG to CUDA, the most important section is the Scan section.
The Scan section contains the actual data. Single blocks are saved as a DC difference coefficient, then the Huffman-coded AC coefficients. If no subsampling is used, blocks from each component simply follow each other (for example {R,G,B,R,G,B...} or {Y,Cb,Cr,Y,Cb,Cr...}). If chroma subsampling is used, there are less Cb and Cr blocks than Y blocks, so the blocks from different components are grouped in integer proportions into Minimum Coded Units (MCU) or Minimum Coded Blocks (MCB). For example, the scan of a 4:2:0 YCbCr JPEG will be composed of a sequence of MCUs with the structure {Y,Y,Y,Y,Cb,Cr}.

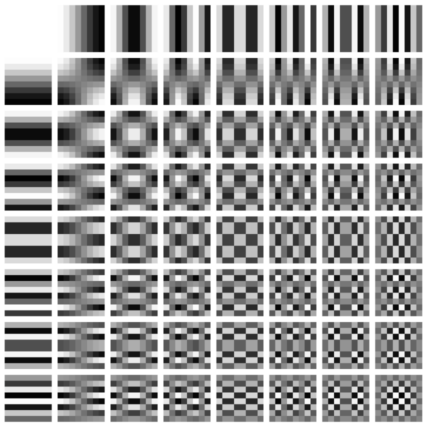

Figure 2.1. DCT representation of an MCB. Each cell of the block here contains an image of the pattern it represents.

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Figure 2.2. Example Quantization Table. Every cell of a 8x8 frequency-domain block is divided by the value in the corresponding cell of the quantization table

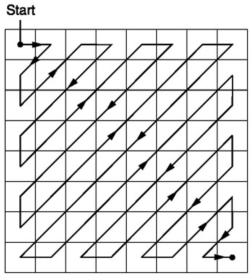

Figure 2.3. Zig-Zag order for the serialization of the frequency-domain blocks

Chapter 3

Implementation

3.1 Encoding algorithm

The original NanoJPEG library, written in C, has an API composed of two main functions: `njDecode()`, which receives raw JPEG data in a byte array and does the actual decoding, and `njGetImage()`, which copies the decoded image from an internal buffer back to the caller. The whole library uses a static state structure, so it is not thread-safe, however passing the state structure as parameter would allow multithreading.

Decompressing a JPEG image is simpler than creating it for three reasons: the input size is already known (the output size is fixed by width and height), there is no quality setting (the quantization table is already known), and there are no choices to be made about the compression (the Huffman table us already known).
Decompressing JPEG images could also be complicated by the myriad legal combinations of color spaces and representation formats and advanced features. NanoJPEG deals with this complexity by supporting only a core set of features, that in practice cover most use cases: only a few color spaces (YCbCr, RGB and single channel) are supported with any integer subsampling ratio on any channel, and no progressive scan (which is the most immediate limit of the library).

Although the order in which the specific functions are called is slightly different, the data flow can be summarized as:

- Read 8x8 data blocks from stream
  - Read DC differential coefficient and compute DC coefficient (sum of last DC coef and difference)
  - Read Huffman Variable Length Codes until end of block (function `njGetVLC()`)
  - Convert from Huffman code to numeric value using the correct code for the component
  - De-quantize or rescale AC coefficients: multiply each by the corresponding value in the correct Quantization Table for the component
- Inverse Discrete Cosine Transform (IDCT) (functions `njRowIDCT()` and `njColIDCT()`)

- rescaling of components which are downsampled (functions `njUpsampleH()` and `njUpsampleV()` called inside `njConvert()`)
- Convert to RGB color space if different (inside function `njConvert()`)
- Copy back to user buffer (separate user API call, function `njGetImage()`)

Of these phases, the read and write phases cannot be processed on a GPGPU because they are inherently serial.
The other phases can exploit parallel computing since any pixel depends at most from a few neighboring pixels.

3.2 NanoJPEG CUDA Implementation Details

NanoJPEG CUDA compiles with the standard `nvcc` compiler.
The library allows the user of the API to disable all CUDA features, by setting to zero the parameter `use_cuda` of function `njInit()`.

The block diagram of the CUDA version of the library is shown in Figure 3.1.

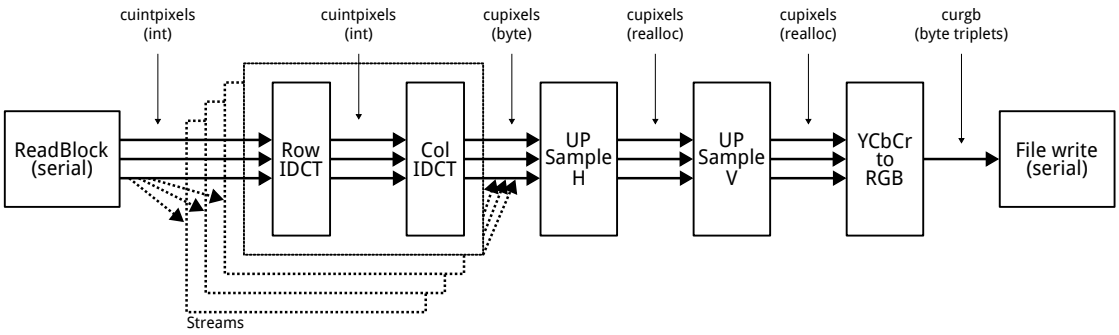

Figure 3.1. Data flow of NanoJPEG CUDA

NanoJPEG CUDA is divided into two main parts: function **`njCudaDecodeScan()`** handles from input data up to the IDCT, while function **`njCudaConvert()`** handles upsampling and RGB color space conversion.

Read from stream, Huffman decoding and de-quantization are implemented in function `njReadBlock()`, which is CPU-only.

The bi-dimensional IDCT is implemented as two separate functions, `njCudaRowIDCT()` and `njCudaRowIDCT()`.

The image is divided into four horizontal slices. When `njReadBlock()` finishes reading one slice, `njCudaRowIDCT()` and `njCudaRowIDCT()` are executed for that slice. The IDCTs for the four slices are executed in four separate CUDA streams so that an initial `CudaMemcpyAsync()` and the IDCT kernels can run on the GPU while the CPU reads the next slice from file. As a result, the starting times of the IDCTs in the four streams are staggered. This allows to reduce non-parallelizable task time at the beginning of the algorithm, since only a quarter of the file needs to be read before the GPU can start.

The program is disk-bound anyway, so a lower number of threads (as low as 2) could in theory be used, alternating the execution of the IDCT kernels between the first and the second, optionally subdividing the IDCT task in more than 4 slices. However the speed gain would be very small, most probably negated by the kernel setup time. Also, the creation of streams is comparatively cheap, so the solution using four streams was chosen. Figure 3.2 shows the sequence of kernel executions.

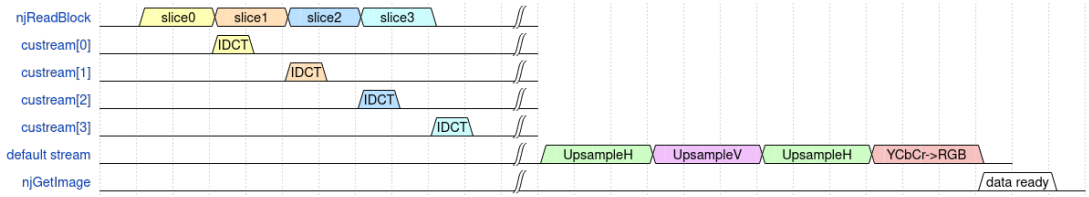

Figure 3.2. Execution timeline

Upsampling is optional, depending on the content of the file. In general functions `njCudaUpsampleH()` and `njCudaUpsampleV()` can be executed many times on any component, each doubles the size of the image in the horizontal or vertical axis respectively.

Eventually the image is converted back to the RGB format, which is a simple linear function implemented in integer arithmetic.

3.3 Test application nanoex.cu

The test application provided with the library (`nanoex.cu`) simply reads a JPEG file, decodes it using NanoJPEG, and writes the uncompressed result to a PPM file.

The test program received minor modifications to test NanoJPEG CUDA.

A new command-line option `"-nocuda"` was added to test the non-CUDA version of the library.

Chapter 4

Results and Conclusions

The execution time of the original and modified libraries were compared on an NVidia Jetson Nano ARM board. As was already observed by Simone Pistilli in [2] for the JPEG compression algorithm, NanoJPEG CUDA also shows better performance than the original NanoJPEG only above a certain image size, which is roughly 1 Megapixel, YCbCr 4:2:0.

YCbCr 4:4:4 moves the convenience threashold higher, because no upsampling is needed, reducing the opportunities to use GPU time.

Halving of the processing time was observed on a 6000x4000 image on the CUDA version compred to the original.

The process is substantially bound by the disk I/O bandwidth. Using four streams for IDCT allows to copy data to the GPU and to execute the IDCT kernels while the CPU is still reading data from disk, reducing serial time of the operation. The reduction is however far smaller than the theoretical 75% because in this phase of the algorithm the GPU is under-utilized. An intermediate version of the program with a single CUDA stream for the IDCT showed a time penalty of 23% compared to the final version with four streams (image: 4272x2848 YCbCr 4:2:0 shown in much smaller form in Figure 4.1).

The image size lower threshold discourages the use of this library for consumer products, however future applications could arise that would benefit from the use of dedicated processing power to decompress very high resolution images in real-time, embedded applications, in which case the use of GPGPUs for the purpose would be second only to application specific processors.

Figure 4.1. Thumbnail of a 4272x2848 pixel, YCbCr 4:2:0 image which in the decompression tests showed a speedup from 3.5s (with the original NanoJPEG), to 2.7s (with one-stream NanoJPEG CUDA), to 2.23s (with final NanoJPEG CUDA)

Bibliography

[1] Martin Fiedler. NanoJPEG: a compact JPEG decoder, June 2019. URL `https://keyj.emphy.de/nanojpeg/`. Last visited: 2023/02/12.

[2] Simone Pistilli. Parallelization of the JPEG compression algorithm on a CUDA GPGPU, January 2023.