Design, Implementation and Evaluation of a Task-parallel JPEG Decoder for the Libjpeg-turbo Library

Jingun Hong[1], Wasuwee Sodsong[1], Seongwook Chung[1], Cheong Ghil Kim[2], Yeongkyu Lim[3], Shin-Dug Kim[1] and Bernd Burgstaller[1*]

[1]Yonsei University, [2]Namseoul University, [3]LG Electronics Inc.

Abstract

In this paper, we propose a task-parallel programming extension for the JPEG decoder of the libjpeg-turbo library. Efficient JPEG decoding is especially important for resource-constrained mobile devices such as smartphones, where decoding (e.g., browsing of web pages containing images, image search aso) is far more common than image encoding. The aim of our work is to utilize multiple CPU cores for JPEG decompression from a single client thread. Our method is orthogonal to libjpeg-turbo's support for data-parallelism (SIMD). Experimental evaluation of our approach on a 4-core Intel i7-2600K CPU shows speed-ups of up to 2.5x over the sequential, and up to 34% over the SIMD-version of the libjpeg-turbo JPEG decoder.

Keywords: *JPEG decoding, multicores, parallel programming patterns, fork/join task parallelism*

1. Introduction

The JPEG format is a de-facto standard for digital images on the world-wide web. Six out of the top-ten most popular websites [1], including Facebook, Youtube, Yahoo, Amazon and Twitter use JPEG images. W3Tech's web statistics [2] report JPGEG as the most popular image format for web sites, with a penetration of 72%. Virtually all digital cameras and smartphones employ JPEG compression, which results in large amounts of new user content on a day-to- day basis. E.g., more than 250 million photos are uploaded on Facebook every day [3].

Decoding JPEG images is a computationally expensive operation: compressed images undergo Huffan decompression, de-quantization, inverse DCT, upsampling, color space conversion (e.g., from YCbCr to RGB) and optional color quantization and precision reduction. Decoding of a color image of resolution 3072x2304 takes 112 ms on an Intel i7, and at resolution 720 x 960 the same image takes 22 ms to decompress (measurements where conducted on an instrumented version of the Google Chrome web browser). JPEG decoding thus constitutes a substantial part of webpage rendering.

To enhance the web user experience, efficient JPEG decoding should take advantage of multicore architectures. Libjpeg [4], the JPEG reference implementation from the Independent JPEG Group, is strictly sequential. Libjpeg-turbo [5] is a re-implementation of libjepg that utilizes MMX, SSE, SSE2, and NEON SIMD instructions on x86 and ARM platforms. Libjpeg-turbo found wide-spread use, e.g., with the Chrome and Firefox web-

[1]* Corresponding Author

browsers, WebKit [6], and the Ubuntu, Fedora and openSuse Linux distributions. From our experiments, libjpeg-turbo accelerates JPEG decoding by a factor of 2 on x86. Neither libjpeg nor libjpeg-turbo are capable of utilizing multiple CPU cores. Considerable computing power is thus un-utilized during JPEG decoding. To the best of our knowledge, this is the first approach to apply task parallelism with the libjpeg-turbo library, although thread parallel approaches to image processing algorithms exist [7, 8].

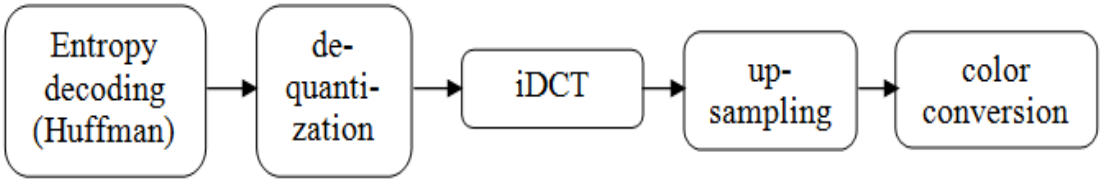

Figure 1.JPEG Decoder Path

To parallelize JPEG decoding on multicore architectures, we extend the libjpeg-turbo library with task-level parallelism; images are partitioned and decoded on the available cores of a shared-memory multicore computer.

2. Background and Related Work

JPEG image encoding consists of color space transformation, downsampling, block splitting, discrete cosine transform, quantization and entropy coding. After loading the raw image into the main memory, the image is broken down into blocks of 8x8 pixels. A 2D discrete cosine transform (DCT) is applied to each 8x8 block to separate it into its frequency components. Since the human eye is less sensitive to higher frequency information, the quantization step divides the DCT transform by a quantization table so that the higher frequency coefficients become 0. At this stage, the lossy compression takes place, meaning that the high frequency components have been discarded. After applying zig-zag reordering, as a final step, Huffman encoding is used to encode the whole picture by replacing the statistically higher occurring bits with the smallest symbols.

Decoding a JPEG image takes the above steps in reversed order as depicted in Figure 1. Entropy decoding, dequantization, inverse DCT, upsampling and color conversion are done during decoding. First we apply Huffman decoding in order to revert Huffman encoding. The JPEG decoder takes the DCT coefficient matrices and performs the entrywise product with the quantization matrix. Then the inverse DCT reconstructs a sequence from the DCT coefficients. The image can be displayed after color conversion from YCbCr to RGB color space finally.

A preliminary version of this work, excluding design, implementation and evaluation of the task-parallel JPEG decoder and excluding combined task/data-parallel JPEG decoding appeared at the IST conference [9].

3. Utilizing Task-parallelism with the Libjpeg-turbo Library

Figure 2 shows the overall software architecture of the libjpeg-turbo library. Libjpeg-turbo has been designed with the goal to conserve resources, esp. memory. Both encoding and decoding is done in units of 8x8 pixels. As depicted in Figure 2, a 2-tier controller and buffer hierarchy is used to manage decoding and storage of single 8x8 pixel rows in various stages of the decoding process. We had to re-engineer the library to store the whole image in memory, to facilitate partitioning and task-parallel image decoding. Storing the whole image is done after the Huffman decoding stage. All other phases of the JPEG decoding path are then conducted in parallel.

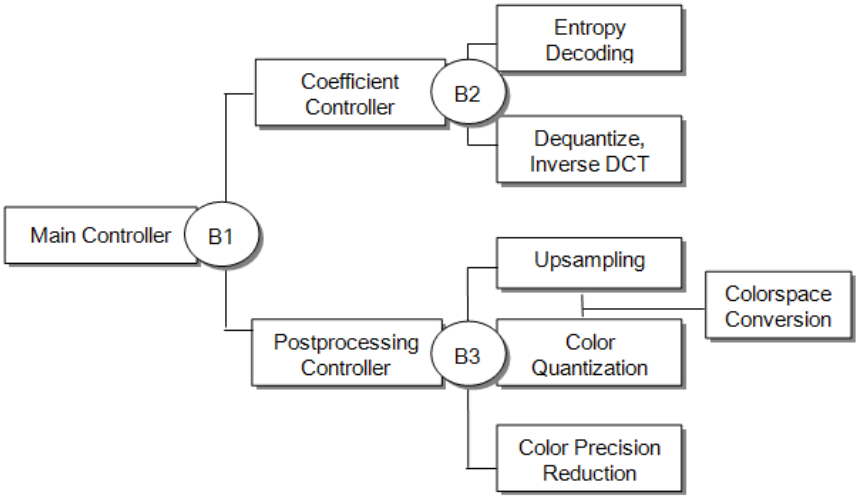

Figure 2. Libjpeg-turbo software architecture, B1-B3 are buffers which store each a single row of 8x8 blocks from the JPEG image

3.1. Design and Implementation of Fork/Join Task Parallelism with the Libjpeg-turbo JPEG decoder

To employ task parallelism with JPEG decoding, we applied fork/join parallelism [10]. In the execution context of the client, the library would create a worker for each available core and partition the image across those workers, who would then decode in parallel. After finishing decoding, workers where joined by the client. After joining all workers, the client would resume execution. Retro-fitting fork/join task-parallelism with the libjpeg-turbo library was possible once we had the whole image available. Because each worker induces an almost identical computation load, load-balance across cores was ensured.

Algorithm 1 shows the code executed by each of our workers. In the libjpeg-turbo library, decoding happens in rows of minimum coded units (MCUs), which consist of 8 or 16 pixel rows of the original image, depending on the upsampling factor of the image. Although we hold the whole compressed JPEG image in memory, we kept row-by-row decoding with the worker threads to exploit cache locality of image data.

Algorithm 1: ThreadKernel

```
1   foreach MCU_row in ImageSlice(tid) do
2     inverseDCT(MCU_row)
3     upsampling(MCU_row)
4     colorConversion(MCU_row)
5   end loop
```

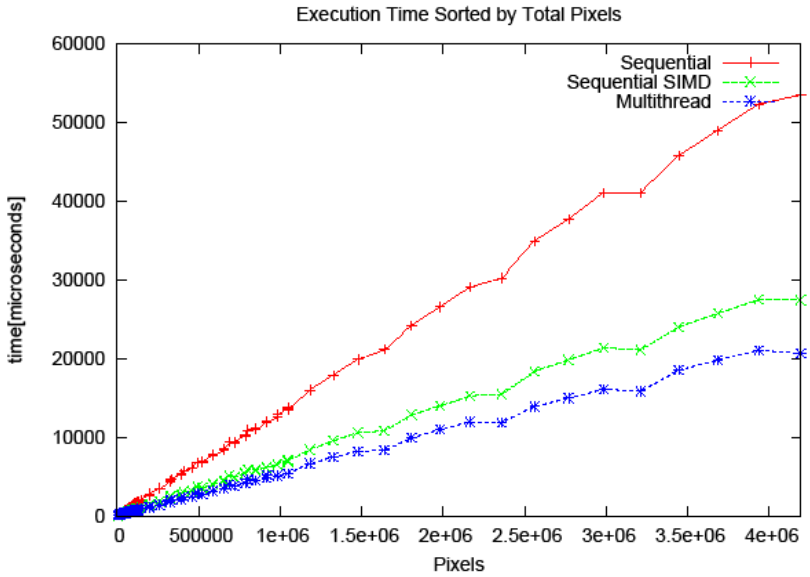

Figure 3. Execution times of the sequential version of libjpeg-turbo, the SSE-based SIMD version, and the SIMD/task-parallel combined version. The combined version is up to 2.52x faster than the sequential version, and 34% faster than the SIMD version

4. Experimental Results

We have implemented our task-parallel decoding extension with libjpeg-turbo [5] version 1.2.3. All experiments were conducted on a 4-core Intel i7-2600K 3.4GHz CPU running Ubuntu Linux 10.10, Linux kernel version 2.6.35-32. All source code was compiled with GCC version 4.4.5 and option -O3.

Experiments where conducted with a selection of 100 JPEG images, with resolutions ranging from 16x16 up to 2200x2200 pixels. Experiments utilized all 4 cores/8 hyperthreads of the i7 processor. Fig. 3 shows the speed-ups achieved with sequential, SSE-based SIMD and combined SIMD/fork-join task-parallel decoding. For the SIMD/fork-join decoding, each task-parallel worker thread would run the SIMD-version of the JPEG decoder. The combined version is up to 2.52 times faster than the sequential version. Adding task-parallelism to the SIMD version yields an improvement of 34%.

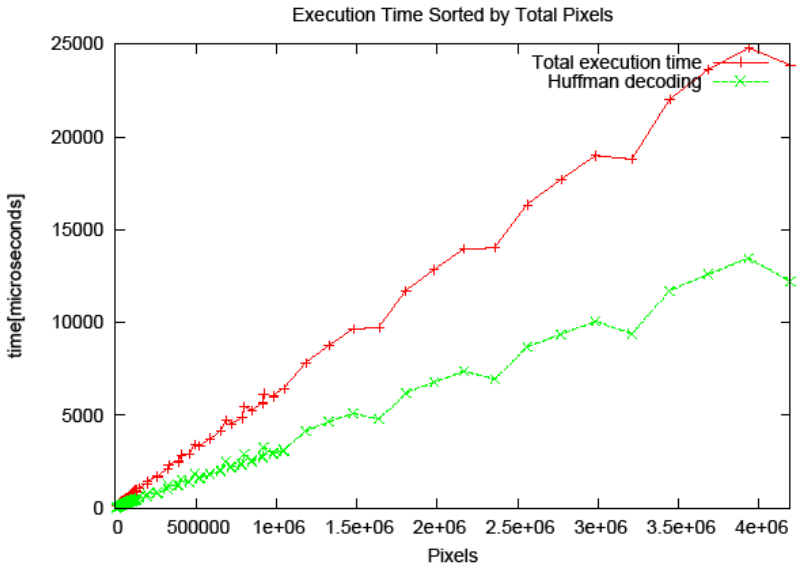

Figure 4. Proportions of Huffman-decoding (sequential) and the parallel part of the decoding path with the SIMD/task-parallel combined version. It follows that with the combined version, around 50% of the execution time is spent for Huffman decoding

It should be noted that both with the SIMD-version and the combined version, Huffman decoding is done sequentially; we investigated the ratio of Huffman-decoding vs. the later, parallel part of the decoding path from Fig. 1. The results are depicted in Fig. 4. As it turned out, with the SIMD/task-parallel combined version, Huffman decoding constitutes a serious, sequential bottleneck: around 50% of the overall execution time is spent for Huffman decoding. It will thus become necessary, to attempt to parallelize the Huffman decoder itself, as suggested in [11].

5. Conclusion and Future Work

We have proposed a task-parallel programming extension for decoding of JPEG images with the libjpeg-turbo library. Our method is orthogonal to libjpeg-turbo's support for data-parallelism (SIMD).

We have achieved speed-ups of up to 2.5x over the sequential version, and up to 34% over the SIMD-version of the libjpeg-turbo JPEG decoder on a 4-core Intel i7-2600K CPU.

As for future work, it should be noted that although the observed overhead from fork/join parallelism was low, different parallel programming patterns need to be considered yet to orchestrate JPEG decoding with multiple clients. Such a scenario would likely occur, e.g., with multiple threads from a web browser, or with several client programs performing JPEG decoding in parallel. With more parallelism exploited with JPEG decoding, Huffman decoding becomes a sequential bottleneck; it will thus become necessary to attempt to parallelize the Huffman decoder itself, as suggested in [11].

References

[1] Google Inc.: Most popular websites on the internet, http://mostpopularwebsites.net, retrieved **(2012)** January.
[2] W3Techs: http://w3techs.com, retrieved **(2012)** March.

[3] Facebook Inc.: Statistics, http://www.facebook.com/press/info.php?statistics, retrieved **(2012)** January.
[4] Libjpeg: http://libjpeg.sourceforge.net/, retrieved **(2012)** January.
[5] Libjpeg Turbo: http://sourceforge.net/projects/libjpeg-turbo, retrieved **(2012)** January.
[6] The WebKit Open Source Project: http://www.webkit.org/, retrieved **(2012)** March.
[7] J. K. Heo, "Distributed Image Preprocessing Using Object Activation", The Journal of The Institute of Webcasting, Internet and Telecommunication 11(1), pp. 87-92, **(2011)** February.
[8] I. S. Ahn, G. S. Choi and S. Y. Kim, "System Development of Iron Plate Defects Detection System Using Image Processing and Multi Thread Method. The Journal of The Institute of Webcasting, Internet and Telecommunication 9(3), pp. 145-153, **(2009)** June.
[9] J. Hong, W. Sodsong, S. Chung, C. G. Kim, Y. Lim, S. D. Kim and B. Burgstaller, "Task-parallel JPEG Decoding with the Libjpeg-turbo Library", In: Proc. of the International Conference on Information Science and Technology **(2012)**.
[10] T. Mattson, B. Sanders and B. Massingill, "Patterns for parallel programming", First edn. Addison-Wesley Professional **(2004)**.
[11] S. T. Klein and Y. Wiseman, "Parallel Huffman decoding with applications to JPEG files", Comput. J. 46(5), pp. 487-497, **(2003)**.