International Journal of Multimedia and Ubiquitous Engineering
Vol.11, No.2 (2016), pp. 9-16
http://dx.doi.org/10.14257/ijmue.2016.11.2.02

Parallel JPEG Color Conversion on Multi-Core Processor

Cheong Ghil Kim[1] and Yong-Ho Seo[2,*]

[1] *Department of Computer Science, Namseoul University*
91 Daehak-ro, Seobuk-gu, Cheonan, Chungnam, Republic of Korea
[2] *Department of Intelligent Robot Engineering, Mokwon University*
88 Doanbuk-ro, Seo-gu, Daejeon, Republic of Korea
cgkim@nsu.ac.kr, yhseo@mokwon.ac.kr

Abstract

Multi-core processors have become the dominant market trend because they provide a great opportunity in increasing processing performance by exploiting various parallelisms. In JPEG (Joint Photographic Experts Group) compression, color space conversion is one of the major kernels known as a computationally expensive module. This paper presents a fast solution for color space conversion with multi-core parallel computation. For this purpose, we utilize Threading Building Blocks (TBB), a runtime library based on C++, and OpenMP (Open Multi-processing), a shared programming language. A RGB image is transformed into a luminance-chrominance color space such as YCbCr. The implementation results show that parallel implementations achieve greater performance improvement regarding processing speed compared with the serial implementation.

Keywords*: JPEG, Color Conversion, Threading Building Blocks, OpenMP, Parallel Programming, Multi-core Processor*

1. Introduction

JPEG [1] images play a significant role as a still image compression standard in multimedia applications. JPEG is designed for compressing full-color or grayscale images of natural, real-world scenes. Being a popular lossy mode of image compression, JPEG has extensively been used in almost all sorts of digital device including the mobile phones, tablets, and handheld computers. There are several modes defined for JPEG such as baseline, lossless, progressive, and hierarchical. The baseline mode is the most popular one supporting lossy coding only. Although the popularly used Baseline JPEG Algorithm can easily be performed by the powerful processors, still the small devices of less capable processors suffer a lot from encoding or decoding a JPEG image because of some complex computations required by Baseline JPEG [2, 10].

In JPEG compression, color space conversion has become an important role in the image acquisition, display, and the transmission of the color information. However, this is known as a computationally expensive step. Figure 1 shows the basic JPEG compression method; it can be summarized into the following: (1) the image is separated into three color components; (2) each component is partitioned into 8-by-8 blocks; (3) each block is transformed using the two dimensional DCT (Discrete Cosine Transform); (4) each transformed block is quantized with respect to an 8-by-8 quantization matrix; (5) the resulting data is compressed, using Huffman or arithmetic coding.

Currently, chip multiprocessors (CMPs) architecture has become the dominant market in desktop PCs as well as mobile devices, in which there are two or more execution

* Corresponding Author

ISSN: 1975-0080 IJMUE

cores within a single processor. Execution cores have their own set of execution and architectural resources. Depending on design, these processors may or may not share a large on-chip cache. Under this circumstance, exploiting and managing parallelism has become a central problem in computer systems [3, 11].

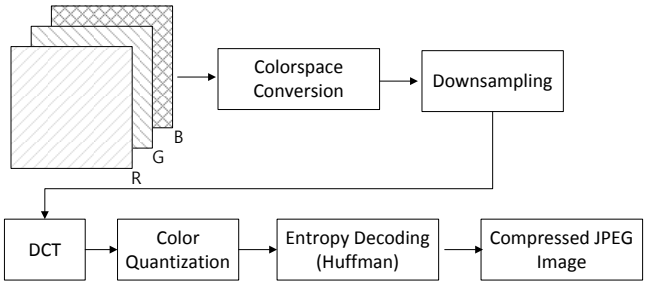

Figure 1. Basic JPEG Compression Method

In order to take advantage of multi-cores, programs could be just written to accomplish their tasks using multiple parallel threads of execution. Therefore, several parallel languages [4, 11] that allow programmers to write parallel code in a quick and efficient manner have been introduced. For example, Intel introduced TBB [5, 12], a C++ library, for desktop-shared memory parallel programming. TBB provides programmers with an API used to exploit parallelism through using tasks rather than parallel threads. Moreover, TBB uses the task stealing to reduce the load imbalance and improve the performance scalability significantly, allowing applications to exploit concurrency with little consideration of the underlying CMP characteristics (*i.e*. number of cores) [6]. Another parallel language is OpenMP [7, 13] which is an API (Application Programming Interface) for multi-platform shared-memory parallel programming in C/C++, in which all threads can access global and shared memories.

In this paper, we describe parallel implementations of color conversion, a computation intensive signal-processing algorithm that is widely used in JPEG compression standards using parallel programming technologies on multi-core CPU. The program for transforming RGB image into a luminance-chrominance color space such as YCbCr is implemented using Intel TBB and OpenMP. The performance evaluation is made by comparing the execution times with and without parallel programing technologies.

The organization of this paper is as follows. In Section 2, two parallel programming languages of TBB and OpenMP are reviewed. Section 3 introduces the background of color space conversion. In Section 4, the experimental results will be discussed; finally, the conclusion will be addressed in Section 5.

2. Background

2.1. TBB

The popularity of multi-core CPUs requires tools enabling easy and quick parallel coding with the form of parallel runtime systems and libraries that aim at improving application portability and programming efficiency. TBB is an open source runtime C++ library that targets desktop-shared memory parallel programming.

TBB provides programmers with APIs used to exploit parallelism through the use of tasks rather than parallel threads. Moreover, TBB is able to significantly reduce load imbalance and improve performance scalability through task stealing, allowing applications to exploit concurrency with little regard to the underlying CMP characteristics (*i.e*. number of cores) [8]. Because it is a library, not a new language or language extension, it integrates into existing programming environments with no

change to the compiler. An additional advantage compared to depending on a language or extension for parallelism is that most programmers can more readily modify a library than a compiler. Hence, a library permits more rapid evolution and customization [9].

```
const size_t L = 150;
const size_t M = 225;
const size_t N = 300;

void SerialMatrixMultiply( float c[M][N], float a[M][L], float
b[L][N] ) {
    for( size_t i=0; i<M; ++i ) {
        for( size_t j=0; j<N; ++j ) {
            float sum = 0;
            for( size_t k=0; k<L; ++k )
                sum += a[i][k]*b[k][j];
            c[i][j] = sum;
        }
    }
}
```

Figure 2. Serial Code Block of Matrix Multiplication

The code blocks in Figures 2 and 3 show the usage of TBB with matrix multiplication. The former shows its serial version and the latter the corresponding parallel version, which uses blocked-rabge2d to specify the iteration space. Header starts with *#include "tbb/blocked_range2d.h"* statement. The *blocked_range2d* enables the two outermost loops of the serial version to become parallel loops. The parallel for recursively splits the blocked_range2d until the pieces are no larger than 16 × 32. It invokes *MatrixMultiplyBody2D::operator()* on each piece.

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"
using namespace tbb;

const size_t L = 150;
const size_t M = 225;
const size_t N = 300;

class MatrixMultiplyBody2D {
    float (*my_a)[L];
    float (*my_b)[N];
    float (*my_c)[N];
public:
    void operator()( const blocked_range2d<size_t>& r ) const {
        float (*a)[L] = my_a;
        float (*b)[N] = my_b;
        float (*c)[N] = my_c;
        for( size_t i=r.rows().begin(); i!=r.rows().end(); ++i ){
            for( size_t j=r.cols().begin(); j!=r.cols().end(); ++j ) {
                float sum = 0;
                for( size_t k=0; k<L; ++k )
                    sum += a[i][k]*b[k][j];
                c[i][j] = sum;
            }
        }
    }

    MatrixMultiplyBody2D( float c[M][N], float a[M][L], float
        b[L][N] ) :
            my_a(a), my_b(b), my_c(c)
        {}
};

void ParallelMatrixMultiply(float c[M][N], float a[M][L], float b[L][N]){
    parallel_for( blocked_range2d<size_t>(0, M, 16, 0, N, 32),
                MatrixMultiplyBody2D(c,a,b) );
}
```

Figure 3. Parallel Implementation of Matrix Multiplication Using TBB

2.2. OpenMP

OpenMP is an API for multi-platform shared-memory parallel programming in C/C++, in which all threads can access global and shared memories. Figure 4 depicts the operational block diagram of OpenMP, in which a process can be divided into several threads and programmers can control the number of threads. Optimal performance occurs when the number of threads represents the number of processors. Here, a master thread exists to assign tasks to threads, *i.e.*, fork-join. Fork-join time increases when there are more threads than processors.

Moreover, data can be labeled either private or shared. In particular, private data are visible to one thread only, while all threads can spot shared data. In practical programs, local variables that are about to be parallelized should be private. Additionally, global variables must be assigned as shared data. OpenMP requires a compiler. Most IDEs today accommodate OpenMP. Numerous benefits exist to using OpenMP, *e.g.*, preservation of serial code, simplicity, flexibility and portability. Nevertheless, explicit synchronization remains an issue that needs to be addressed [8].

Recently, many multi-core processors with common L2 cache have been introduced. The advantage now is that we could execute different threads in these processing elements and the communication cost between the threads would be very less since they share the L2 cache. Another advantage of having multiple cores is that we could use these cores to extract thread level parallelism in a program and hence increase the performance of the single program.

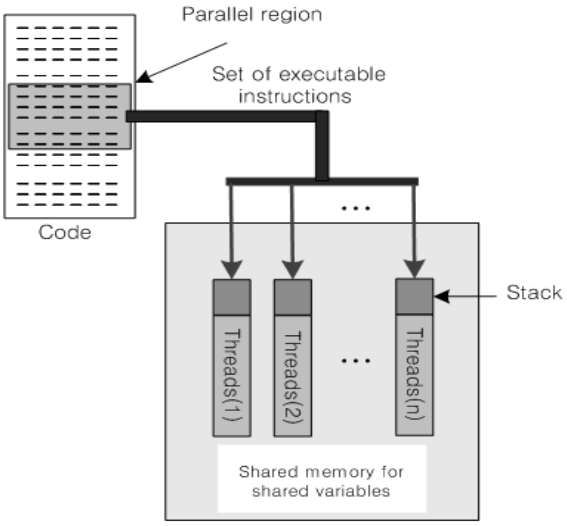

Figure 4. Operational Block Diagram of OpenMp

3. Color Conversion

Color spaces are three-dimensional and images are formed on a computer monitor or television by combining red, green, and blue, which is known as the most common kind of color space, the RGB space; while three- dimensional space of YUV is adopted in the system of JPEG compression.

In order to achieve good compression performance, correlation between the color components is first reduced by converting the RGB color space into a de-correlated color space. In baseline JPEG, a RGB image is first transformed into a luminance-chrominance color space such as YUV. Therefore, YUV signals are typically created from RGB source. Weighted values of R, G, and B are summed to produce Y, a measure of overall brightness or luminance. U and V are computed as scaled differences between

Y and the B and R values [4]. Defining the following constants: W_R=0.299, W_B=0.114, W_G= 1- W_R - W_B =0.587, U_{Max} =0.436, V_{Max} =0.615; YUV is computed from RGB as follows:

$$\begin{aligned} Y &= W_R R + W_G G + W_B B \\ U &= U_{Max}\frac{B-Y}{1-W_B} \approx 0.492(B-Y) \\ V &= V_{Max}\frac{R-Y}{1-W_R} \approx 0.877(R-Y) \end{aligned} \tag{1}$$

The resulting ranges of Y, U, and V, respectively are [0, 1], [-U_{Max}, U_{Max}], and [-V_{Max}, V_{Max}]. Inverting the above transformation converts YUV to RGB:

$$\begin{aligned} R &= Y + V\frac{1-W_R}{V_{Max}} = Y + \frac{V}{0.877} \\ G &= Y - U\frac{W_B(1-W_B)}{U_{Max}W_G} - \frac{W_R(1-W_R)}{V_{Max}W_G} \\ &= Y - \frac{0.232U}{0.587} - \frac{0.341V}{0.587} \\ &= Y - 0.395U - 0.581V \\ B &= Y + U\frac{1-W_B}{U_{Max}} = Y + \frac{U}{0.492} \end{aligned} \tag{2}$$

Equivalently, substituting values for the constants and expressing them as matrices gives:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.13983 \\ 1 & -0.39465 & -0.58060 \\ 1 & 2.03211 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \tag{4}$$

Another color conversion is the transformation from RGB to YCbCr, which is based on the following mathematical expression:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299000 & 0.587000 & 0.114000 \\ -0.168736 & -0.331264 & 0.5000002 \\ 0.500000 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{5}$$

The value Y = 0.299R + 0.587G + 0.114B is called the luminance. It is the value used by monochrome monitors to represent an RGB color. The formula is like a weighted-filter with different weights for each spectral component. Accordingly, the inverse transformation from YC_bC_r to RGB can be:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 1.40210 \\ 1.0 & -0.34414 & -0.71414 \\ 1.0 & 1.77180 & 0.0 \end{bmatrix} \begin{bmatrix} Y \\ C_b - 128 \\ C_r - 128 \end{bmatrix} \tag{6}$$

4. Simulation

This section describes the simulation environments and results in detail. For the performance evaluation, the processing time of TBB and OpenMP color space conversion over its serial one is measured and the unit time is seconds. The result is the average of a comparison of 300 times using system clock. As for the measured section,

the loading time is excluded because it is the common stage. Table 1 describes the system parameters.

The simulation result shows that the average processing time for RGB to YUV and YCbCr with and without TBB and OpenMP are shown in Table 2. These results are depicted in Figures 5 and 6, respectively. The performance gain using TBB is about 600 times on the processing speed compared with the serial implementation.

Table 1. Simulation Parameter

System Parameters	Value
CPU	Intel Core i5-3570@ 3.40GHz
RAM	16.00GB
System	Window 7 Ultimate SP1(64bit)
Compiler	Microsoft Visual Studio 2012 Intel Parallel Studio XE 2013
TBB version	TBB 4.1
OpenMP	OpenMP 4.0
Image Sizes	512 x 512 pixels
Image Type	BMP

Table 2. Simulation Results

Category		Times
RGB to YCbCr	w/o Parallel Programming	0.000763
	w/ OpenMP	0.000255
	w/ TBB	0.000341
YCbCr to RGB	w/o Parallel Programming	0.000722
	w/ OpenMP	0.000895
	w/TBB	0.000762

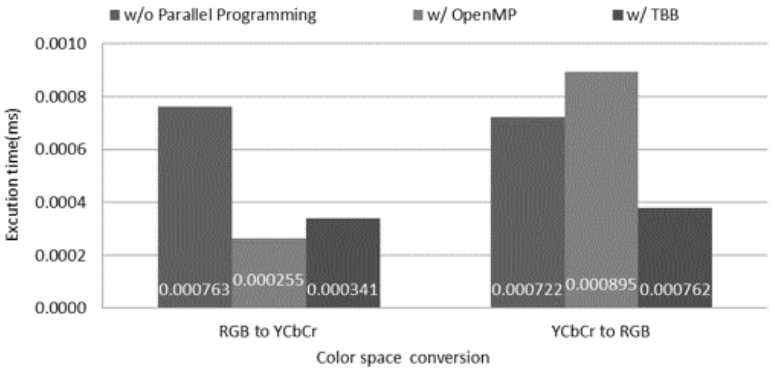

Figure 5. RGB to Ycbcr Color Conversion Average Time

Table 3. Simulation Results

Category		Times
RGB to YCbCr	w/o Parallel Programming	0.000763
	w/ OpenMP	0.000299
	w/ TBB	0.000301
YCbCr to RGB	w/o Parallel Programming	0.000722
	w/ OpenMP	0.000368
	w/TBB	0.000304

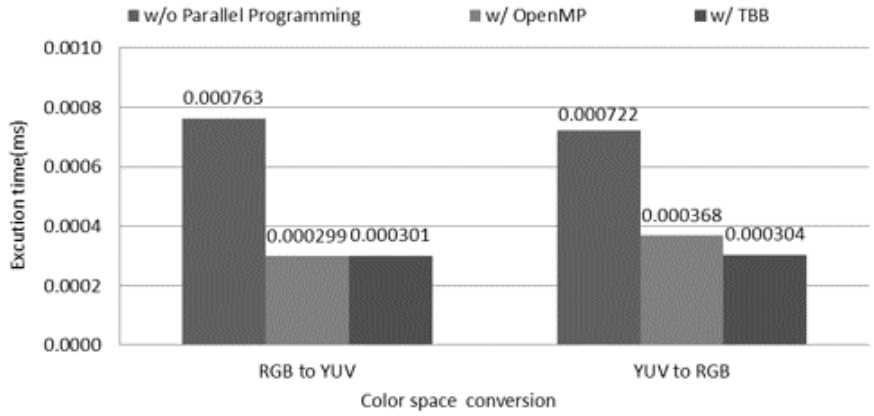

Figure 6. RGB to YUV Color Conversion Average Time

5. Conclusion

Nowadays, modern CPUs with multi-core architecture provide a great opportunity to increase processing performance using parallel programming. This paper presented an optimization of JPEG color conversions of transformation from RGB to YCbCr and YUV. For this purpose, the two parallel programming techniques of TBB and OpenMP were discussed in detail. The simulation results show that the parallel implementations of JPEG color conversion are performed considerably faster than the serial ones.

The parallel implementation results show the 2.5 times of performance improvements on processing speed compared with the serial implementation. Our future work will include the disparity map computations with real image pairs, which will help improve OpenMP better than the serial implementation.

References

[1] T. Acharya and P. Tsai, "JPEG2000 standard for image compression: concepts, algorithms and VLSl architectures", John Wiley & Sons, Inc. New Jersey, **(2005)**.
[2] C. G. Kim and B. J. Beak, "Fast JPEG Color Space Conversion on Shared Memory", Proceedings of the 2013 Int. Conf. on Information Science and Applications (ICISA), Pattaya, Bangkok, **(2013)** June 24-26.
[3] C. G. Kim, D. H. Lee and J. G. Kim, "Optimizing Image Processing on Multi-core CPUs with Intel Parallel Programming Technologies", Multimedia Tools and Applications, no. 68, **(2014)**, pp. 237-251.
[4] E. Ajkunic, H. Fatkic, E. Omerovic, K. Talic and N. Nosovic, "A Comparison of Five Parallel Programming Models for C++", Proceedings of the 35th Int'l Convention on Information and

Communication Technology Electronics and Microelectronics (MIPRO 2012), Opatija, Croatia, **(2012)**, May 21-25.
[5] J. Reinders, "Intel Threading Building Block", O'Reilly, Sebastopol CA, **(2007)**.
[6] S. Lu and Q. Li, "Improving The Task Stealing In Intel Threading Building Blocks", Proceedings of the Int'l Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC 2011), Beijing, China, **(2011)**, October 10-12.
[7] A. Marongiu and L. Benini, "An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs", Computers, IEEE Transactions on, vol. 2, no. 61, **(2012)**, pp. 222-236.
[8] W. Stallings, "Computer Organization and Architecture 8/E: Designing for Performance, Prentice Hall, **(2009)**.
[9] S. Akhter and J. Roberts, "Multi-Core Programming: Increasing Performance through Software Multi-threading", Intel Press, **(2006)**.
[10] D. Santa-Cruz, R. Grosbois and T. Ebrahimi, "JPEG 2000 performance evaluation and assessment", Signal Processing: Image Communication, vol. 17, no. 1, **(2002)**, pp. 113–130.
[11] D. T. C. Shekhar, K. Varaganti, R. Suresh, R. Garg and R. Ramamoorthy, "Comparison of Parallel Programming Models for Multicore Architectures", 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW), Shanghai, China, **(2011)**, May 16-20.
[12] S. Zhang, W. Zhang, and X. Wang, "Implementation of Multi-core Parallel Computation for Solving Large Dense Linear Equations Based on TBB", 2012 International Conference on Control Engineering and Communication Technology (ICCECT), Liaoning, China, **(2012)**, December 7- 9.
[13] C. Qian, Z. Ding and H. Sun, "A Performance Visualization Method for OpenMP Tasks", High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and 2013 IEEE 10th International Conference on Ubiquitous Computing (HPCC_EUC), Zhangjiajie, China, **(2013)**, November 13-15.

Author

Cheong Ghil Kim, He received his B.S. in Computer Science from the University of Redlands, CA, U.S.A. in 1987. He received his M.S. and Ph.D. degree in Computer Science from Yonsei University, Korea, in 2003 and 2006, respectively. Currently, he is a professor at the Department of Computer Science, Namseoul University, Korea. His research areas include Multimedia Embedded Systems and AR.

Yong-Ho Seo, He received his BS and MS degrees from the Department of Electrical Engineering and Computer Science, KAIST, in 1999 and 2001, respectively. He also received a PhD degree at the Artificial Intelligence and Media Laboratory, KAIST, in 2007. He was an Intern Researcher at the Robotics Group, Microsoft Research, Redmond, WA in 2007. He was also a consultant at Qualcomm CDMA Technologies, San Diego, CA in 2008. He is currently a Professor of the Department of Intelligent Robot Engineering, Mokwon University. His research interests include humanoid robot, human-robot interaction, robot vision and wearable computing.

