멀티코어와 매니코어 환경에서의 2 차원 DCT 가속

홍진건*, 정성욱*, 김정길**, Bernd Burgstaller*
*연세대학교 컴퓨터과학과
**남서울대학교 컴퓨터학과
e-mail : ginug@yonsei.ac.kr

Accelerating 2D DCT in Multi-core and Many-core Environments

Jingun Hong*, Sungwook Jung*, Cheong Ghil Kim**, Bernd Burgstaller*
*Dept. of Computer Science, Yonsei University
**Dept. of Computer Science, Namseoul University

요 약

Chip manufacture nowadays turned their attention from accelerating uniprocessors to integrating multiple cores on a chip. Moreover desktop graphic hardware is now starting to support general purpose computation. Desktop users are able to use multi-core CPU and GPU as a high performance computing resources these days. However exploiting parallel computing resources are still challenging because of lack of higher programming abstraction for parallel programming. The 2-dimensional discrete cosine transform (2D-DCT) algorithms are most computational intensive part of JPEG encoding. There are many fast 2D-DCT algorithms already studied. We implemented several algorithms and estimated its runtime on multi-core CPU and GPU environments. Experiments show that data parallelism can be fully exploited on CPU and GPU architecture. We expect parallelized DCT bring performance benefit towards its applications such as JPEG and MPEG.

1. Introduction

Because fundamental laws of physics prevent further performance increases with uniprocessor architectures, chip manufacturers are now utilizing Moore's law to integrate more and more processors on a single chip. In 2001, IBM released the world's first non-embedded dual-core processor. In 2008, Intel launched the Core i7 CPU with 8 logical cores with hyper threading. Today more than 90% of processors available have at least 2 cores.

In terms of the number of processors on a single chip, Graphical Processing Units (GPUs) already integrate several hundred compute units. People traditionally regarded GPUs as fast graphic renderers. However, with the provision of general purpose programing abstractions such as CUDA and OpenCL, general purpose GPUs (GPGPUs) have become a driving force in the server market for scientific computing. GPGPUs are nowadays an integral part of desktop computers, and they are about to enter the embedded market as well, esp. in the area of smartphones, tablets and internet TVs.

Because of a glaring lack of higher programming abstractions, GPGPU programming is still a challenging task. [6] The 2-dimensional discrete cosine transform (2D-DCT) algorithm is the most computationally intense part of JPEG encoding [5]. Many researchers have studied fast 2D DCT algorithms. In this paper we investigate the performance of various 2D DCT algorithms on multi-core CPUs and GPGPUs. We investigated specifically how algorithms can be adapted for GPGPU memory models.

2. Multi- and Many-core Programming Environments

2.1. Threading Building Blocks

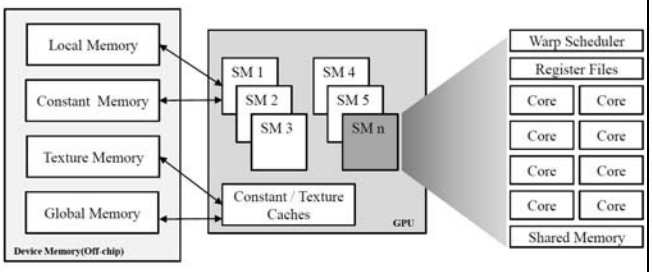
Intel Thread Building Blocks (TBBs, [4]) provide an API for programmers to exploit parallelism on the level of tasks rather than threads. A task is a code block that contains computations including procedure calls. Tasks have shorter code blocks than threads which allows the TBB run-time system to achieve load balanced program execution on multi-core hardware. The TBB run-time system dynamically allocates tasks to processors using global and local task queues. Whenever a local task queue for a processor runs out of tasks, it steals tasks from the global task queue to prevent load imbalance [3].

2.2. CUDA

NVIDIA, one of the major graphic hardware vendors, has developed a standard programming framework for GPGPU computing, called Compute Unified Device Architecture (CUDA, [2]). CUDA allows programmers to make use of GPGPUs for general purpose computing other than using openGL. CUDA-enabled graphics hardware has dozens of Stream Multi-Processors (SMs) and each SM provides 8 cores. Threads in the same SM are allowed to access fast on-chip memory called local memory. Because of its high memory bandwidth, programmers are recommended to use local memory to optimize programs. By exploiting multiple compute units and high memory bandwidth, programmers can achieve enormous data parallelism on GPGPUs.

2.3. OpenCL

OpenCL [1] is a new programming standard for various compute devices including GPGPUs. CUDA and openCL architecture models and programming APIs are very similar, apart from subtle differences wrt. terminology. All features of CUDA are mapped onto OpenCL programming primitives, including memory hierarchy and thread model. Already now, a variety of high performance computing devices support OpenCL, e.g., the IBM Cell BE architecture, GPGPUs from NVIDIA and ATI, and AMD CPUs. In this study we use OpenCL for our portable implementation of a 2D DCT algorithm.



(Figure 1) GPU architecture in CUDA

3. 2D 8x8 DCT algorithms

The DCT is a foundation of many image and video coding standards such as MPEG and JPEG. DCT enables efficient coding with respect to quality measurement and simplicity [5]. The 8x8 two-dimensional DCT is defined as follows:

$$f(x,y) = \sum_{i=0}^{7}\sum_{j=0}^{7} c(u)c(v)\cos\frac{(2i+1)\pi u}{16}\cos\frac{(2j+1)\pi v}{16}$$

$$c(k) = \begin{cases} 1/2\sqrt{2}, & k = 0 \\ 1/2, & otherwise \end{cases}$$

The formula can be converted to matrix form.

$$T = \begin{bmatrix} \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} \\ \frac{1}{2}\cos\frac{1}{16}\pi & \frac{1}{2}\cos\frac{3}{16}\pi & \frac{1}{2}\cos\frac{5}{16}\pi & \frac{1}{2}\cos\frac{7}{16}\pi & \frac{1}{2}\cos\frac{9}{16}\pi & \frac{1}{2}\cos\frac{11}{16}\pi & \frac{1}{2}\cos\frac{13}{16}\pi & \frac{1}{2}\cos\frac{15}{16}\pi \\ \frac{1}{2}\cos\frac{2}{16}\pi & \frac{1}{2}\cos\frac{6}{16}\pi & \frac{1}{2}\cos\frac{10}{16}\pi & \frac{1}{2}\cos\frac{14}{16}\pi & \frac{1}{2}\cos\frac{18}{16}\pi & \frac{1}{2}\cos\frac{22}{16}\pi & \frac{1}{2}\cos\frac{26}{16}\pi & \frac{1}{2}\cos\frac{30}{16}\pi \\ \frac{1}{2}\cos\frac{3}{16}\pi & \frac{1}{2}\cos\frac{9}{16}\pi & \frac{1}{2}\cos\frac{15}{16}\pi & \frac{1}{2}\cos\frac{21}{16}\pi & \frac{1}{2}\cos\frac{27}{16}\pi & \frac{1}{2}\cos\frac{33}{16}\pi & \frac{1}{2}\cos\frac{39}{16}\pi & \frac{1}{2}\cos\frac{45}{16}\pi \\ \frac{1}{2}\cos\frac{4}{16}\pi & \frac{1}{2}\cos\frac{12}{16}\pi & \frac{1}{2}\cos\frac{20}{16}\pi & \frac{1}{2}\cos\frac{28}{16}\pi & \frac{1}{2}\cos\frac{36}{16}\pi & \frac{1}{2}\cos\frac{44}{16}\pi & \frac{1}{2}\cos\frac{52}{16}\pi & \frac{1}{2}\cos\frac{60}{16}\pi \\ \frac{1}{2}\cos\frac{5}{16}\pi & \frac{1}{2}\cos\frac{15}{16}\pi & \frac{1}{2}\cos\frac{25}{16}\pi & \frac{1}{2}\cos\frac{35}{16}\pi & \frac{1}{2}\cos\frac{45}{16}\pi & \frac{1}{2}\cos\frac{55}{16}\pi & \frac{1}{2}\cos\frac{65}{16}\pi & \frac{1}{2}\cos\frac{75}{16}\pi \\ \frac{1}{2}\cos\frac{6}{16}\pi & \frac{1}{2}\cos\frac{18}{16}\pi & \frac{1}{2}\cos\frac{30}{16}\pi & \frac{1}{2}\cos\frac{42}{16}\pi & \frac{1}{2}\cos\frac{54}{16}\pi & \frac{1}{2}\cos\frac{66}{16}\pi & \frac{1}{2}\cos\frac{78}{16}\pi & \frac{1}{2}\cos\frac{90}{16}\pi \\ \frac{1}{2}\cos\frac{7}{16}\pi & \frac{1}{2}\cos\frac{21}{16}\pi & \frac{1}{2}\cos\frac{35}{16}\pi & \frac{1}{2}\cos\frac{49}{16}\pi & \frac{1}{2}\cos\frac{63}{16}\pi & \frac{1}{2}\cos\frac{77}{16}\pi & \frac{1}{2}\cos\frac{91}{16}\pi & \frac{1}{2}\cos\frac{105}{16}\pi \end{bmatrix}$$

$$D = TMT^T$$

Applying 1D DCT on every column of an 8x8 array and then applying 1D DCT on every row of the result is equivalent to an 8x8 2D DCT. Therefore some serial 8x8 DCT algorithms rely on fast 1D DCT. In this study, we investigate three 2D DCT algorithms adapted for different hardware architectures.

4. Implementation

We implemented 3 different 2D 8x8 DCT using TBB and OpenCL. First we directly mapped the formula to naïve version implementation. Naïve version has mathematical function call such as cosine and square root function as described in Fig. 2.

```
:   Naive8x8DCT(In, Out)
1:  for x in 1..SIZE loop
2:    for y in 1..SIZE loop
3:      if x mod 8=0 then c(x)←sqrt(1/8) else c(u):=1/2
4:      if y mod 8=0 then c(y)←sqrt(1/8) else c(u):=1/2
5:        for i in 0..7 loop
6:          for j in 0..7 loop
7:            Out(x,y)←In(x,y) * cosf(i) * cosf(j)
8:          end loop
9:        end loop
10:     Out(x,y)←Out(x,y) * c(x) * c(y)
11:   end loop
12: end loop
```

(Figure 2) Pseudo code for naïve implementation

To reduce mathematical function cost and improve performance, we used matrix form of the formula for second implementation. Every cosine function calls are replaced by constant element of matrix T. In this version of implementation 8x8 DCT is done by performing two matrix multiplication operations as Fig. 3 shows. However, unlike previous algorithm, if we execute Matrix8x8DCT in parallel using TBB or OpenCL, each thread will compute 8x8 pixels. If target hardware has a higher number of cores than the number of 8x8 blocks in input, for example, GPU, we will get some loss in parallelism.

```
:   Matrix8x8DCT(In, Out)
1:  for x in 1..SIZE loop
2:    for y in 1..SIZE loop
3:      /* Perform two matrixes multiplication op */
4:      Tmp←MatrixMultiply(In, transpose(M));
5:      Out←MatrixMultiply(M, Tmp);
6:    end loop
7:  end loop
```

(Figure 3) Pseudo code for matrix multiplication version implementation

```
:   Matrix8x8DCT2(In, Out)
1:  for x in 1..SIZE loop
2:    for y in 1..SIZE loop
3:      for z in 1..8 loop
4:        Tmp(x,y)←Tmp(x,y)+M(x,z)*In(z,y)
5:      end loop
6:      for z in 1..8 loop
7:        Out(x,y)←Out(x,y)+Tmp(x,z)*MT(z,y)
```

```
8:          end loop
9:        end loop
10:     end loop
```

(Figure 4) Pseudo code for revised version of Fig.3

Figure. 4 describes revised implementation which performs fine grained parallelism, that is each thread compute DCT for single pixel. To ensure first matrix multiplication operation is done, we need a barrier between line 5 and line 6 in Fig. 4 in concurrent programming. In OpenCL implementation, we used local memory for temporary variable *Tmp* (line 4 and 7 in Fig. 4) to mitigate global memory access overhead.

In practice many 8x8 DCT implementations use row and column scanning algorithm using fast 1D DCT. Fast 1D DCT algorithm commonly are unrolled and has less addition and multiplication operations compared to its original algorithm.

```
:       RowColScanningDCT(In, Out)
1:      for x in 1..SIZE loop
2:        for y in 1..SIZE loop
3:          /* process 1D DCT row wise */
4:          for k in 1..8 loop
5:            1D_DCT8(Out(x+k, y), In(x+k ,y));
6:          end loop
7:          /* process 1D DCT column wise */
8:          for k in 1..8 loop
9:            ID_DCT8(Out(x,y+k), In(x, y+k))
10:         end loop
11:       end loop
12:     end loop
```

(Figure 5) Pseudo code for row and column scanning DCT algorithm implementation

Fig. 5 shows how 2D 8x8 DCT can be implemented using fast 1D DCT. However, Fig. 5 performs DCT on 8x8 block each iteration so in parallel version of this implementation using TBB or OpenCL each thread will cover a 8x8 block.

5. Experimental Results

All experiment was done with Intel Core i5 CPU which has 4 processors, 2.67GHz clock speed and NVIDIA GTX 275 GPU which contains 240 CUDA cores (1404MHz) and supports device compute capability 1.3 and OpenCL 1.0. All implementations were compiled by gcc 4.1.2.

Because of a number of expensive mathematical function call, naïve version implementation showed very slow performance on serial program on CPU. The algorithm consists of independent computation blocks so that TBB and OpenCL are able to exploit data parallelism. In OpenCL implantation, each thread access 8x8 local block 64 times. Therefore each pixel of the block is accessed 64 times. This frequent accessing pattern can be optimized using on chip local memory. We could achieve 5x more speed up using local memory which is faster than global memory access.

<Table 1> Execution time for naïve DCT algorithm implementation

	1024x1024	*2048x2048*	*4096x4096*
Serial	5.809	23.177	92.650
TBB	1.722	6.888	27.555
Speedup	3.373	3.364	3.362
OpenCL(g.mem)	0.014	0.058	0.140
Speedup	264.045	257.522	256.648
OpenCL(l.mem)	0.001	0.005	0.019
Speedup	5811	4634.4	4876.7

OpenCL version of naïve DCT algorithm showed about 4000x speed up using local memory compared to serial version. GPUs support powerful transcendental operation and native mathematical procedure call so that we could get speed up more than the number of cores GPU has. [7]

<Table 2> Execution time for DCT algorithm implementation using matrix multiplication

	1024x1024	*2048x2048*	*4096x4096*
Serial	0.157	0.576	2.258
TBB	0.032	0.130	0.522
Speedup	4.906	4.430	4.325
OpenCL(g.mem)	0.016	0.066	0.265
Speedup	10.18	8.77	8.49
OpenCL(l.mem)	0.001	0.005	0.015
Speedup	130	122	125.7

Table. 2 shows that serial version of matrix multiplication version implementation shows better performance than naïve version so that overall speedup decreases. We could get 100x more speedup replacing global memory access with local memory access. Moreover local memory version could use local barrier that enables to implement fine grained version of implementation in Fig. 4.

<Table 3> Execution time for row and column scanning DCT algorithm implementation

	1024x1024	*2048x2048*	*4096x4096*
Serial	0.007	0.029	0.115
TBB	0.004	0.017	0.068
Speedup	1.75	1.71	1.69
OpenCL	0.00024	0.00068	0.00217
Speedup	10.18	8.77	8.49

Table 3. describes that row and column scanning for DCT is fastest algorithm. Although algorithm itself has coarse grained tasks, because of highly optimized 1D DCT overall runtime is reduced.

The sequential implementation suffers from high overheads due to the transcendental functions. DCT using matrix multiplication was about 40x faster. With TBB, our speedups were a factor of 3x over the sequential version. On the GPGPU, we achieved a speedup of 4000x, 100x, 10x for naïve, matrix multiplication and row and column scanning DCT respectively. Facilitating GPGPU local memory yielded an additional about 10x speed up.

6. Conclusion

We have presented several implementations of 2D 8x8 DCT algorithms in multi-core CPU and GPU hardware environments. Because of its task independent characteristic like common image processing algorithms, we focused on accelerating DCT algorithm using data parallelism. By experiments we found that accelerating DCT is promising both in multi-core CPUs and GPUs. We expect that parallel DCT can speed up real world applications such as JPEG encoding.

Acknowledgment

This research was supported partly by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (KRF 2010-0 0028047, KRF 2010-0 005234).

References

[1] Khronos OpenCL Working Group, The OpenCL Specification, Version 1.1, 2010.

[2] John Nickolls, Ian Buck, Michael Garland and Kevin Skadron, Scalable Parallel Programming with CUDA, ACM Queue, Vol. 6 (2), ACM, 2008.

[3] Gilberto Contreras, Margaret Martonosi. Characterizing and Impoving the Performance of Intel Threading Building Blocks, IEEE International Symposium on Workload Characterization, 2008.

[4] James Reinders. Intel Threading Building Blocks. O'Reilly, 2007.

[5] William B. Pennebaker, Joan L. Mitchell, JPEG Still Image Data Compression Standard, Van Nostrand Reinhold, 1993.

[6] Jongtae Park, Beorn Faccini, Jingun Hong, Bernd Burgstaller, Implementing Efficient Camera ISP Filters on GPGPUs Using OpenCL, Korea Information Processing Society, 2010. 11.

[7] Victor W Lee et al, Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, ACM SIGRACH Computer Architecture News, 2010.