Discrete Cosine Transform for 8x8 Blocks with CUDA

Anton Obukhov
aobukhov@nvidia.com
Alexander Kharlamov
akharlamov@nvidia.com

July 2012

Document Change History

Version	Date	Responsible	Reason for Change
0.8	24.03.2008	Alexander Kharlamov	Initial release
0.9	25.03.2008	Anton Obukhov	Added algorithm-specific parts, fixed some issues
1.0	17.10.2008	Anton Obukhov	Revised document structure

Abstract

In this whitepaper the Discrete Cosine Transform (DCT) is discussed. The two-dimensional variation of the transform that operates on 8x8 blocks (DCT8x8) is widely used in image and video coding because it exhibits high signal decorrelation rates and can be easily implemented on the majority of contemporary computing architectures. The key feature of the DCT8x8 is that any pair of 8x8 blocks can be processed independently. This makes possible fully parallel implementation of DCT8x8 by definition. Most of CPU-based implementations of DCT8x8 are firmly adjusted for operating using fixed point arithmetic but still appear to be rather costly as soon as blocks are processed in the sequential order by the single ALU. Performing DCT8x8 computation on GPU using NVIDIA CUDA technology gives significant performance boost even compared to a modern CPU. The proposed approach is accompanied with the sample code "DCT8x8" in the NVIDIA CUDA SDK.

1. Introduction

The Discrete Cosine Transform (DCT) is a Fourier-like transform, which was first proposed by Ahmed *et al.* (1974). While the Fourier Transform represents a signal as the mixture of sines and cosines, the Cosine Transform performs only the cosine-series expansion. The purpose of DCT is to perform decorrelation of the input signal and to present the output in the frequency domain. The DCT is known for its high "energy compaction" property, meaning that the transformed signal can be easily analyzed using few low-frequency components. It turns out to be that the DCT is a reasonable balance of optimality of the input decorrelation (approaching the Karhunen-Loève transform) and the computational complexity. This fact made it widely used in digital signal processing.

There are several types of DCT [2]. The most popular is two-dimensional symmetric variation of the transform that operates on 8x8 blocks (DCT8x8) and its inverse. The DCT8x8 is utilized in JPEG compression routines and has become a de-facto standard in image and video coding algorithms and other DSP-related areas. The two-dimensional input signal is divided into the set of nonoverlapping 8x8 blocks and each block is processed independently. This makes it possible to perform the block-wise transform in parallel, which is the key feature of the DCT8x8.

A lot of effort has been put into optimizing DCT routines on existing hardware. Most of the CPU implementations of DCT8x8 are well-optimized, which includes the transform separability utilization on high-level and fixed point arithmetic, cache-targeted optimizations on low-level. On the other hand, the key feature of the DCT8x8 is not utilized in any implementation due to architecture limits. However, there is no limit for improvement.

GPU acceleration of DCT8x8 computation has been possible since appearance of shader languages. Nevertheless, this required a specific setup to utilize common graphics API such as OpenGL or Direct3D. CUDA, on the other hand, provides a natural extension of C language that allows a transparent implementation of GPU accelerated algorithms. Also, DCT8x8 greatly benefits from CUDA-specific features, such as shared memory and explicit synchronization points.

Figure 1. "Barbara" test image.

This paper illustrates the concept of highly-parallelized DCT8x8 implementation using CUDA. Performing DCT8x8 computations on a GPU gives the significant performance boost even compared to a modern CPU. The proposed approach is illustrated using the sample code that performs part of JPEG routine: forward DCT8x8, quantization of each 8x8 block followed by the inverse DCT8x8. Finally, the comparison of execution speed is performed for CPU and GPU implementations. The performance testing is done using Barbara image from Marco Schmidt's standard test images database (Figure 1). The quality assurance is done by means of PSNR, the objective visual quality metric.

This paper is organized as follows. Section 2 gives some theoretical background of DCT and DCT8x8. The proposed implementations are described in Section 3. The evaluation of the proposed approaches and quality assurance issues are presented in Section 4. Optimization issues can be found in Section 5, followed by the source code details in Section 6 and the conclusion in Section 7.

2. DCT Theory

Formally, the discrete cosine transform is an invertible function $F : \Re^N \to \Re^N$ or equivalently an invertible square $N \times N$ matrix [1]. The formal definition for the DCT of one-dimensional sequence of length N is given by the following formula:

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) \cos\left[\frac{\pi(2x+1)u}{2N}\right], \quad u = 0,1,\ldots,N-1 \tag{1}$$

The inverse transformation is defined as

$$f(x) = \sum_{u=0}^{N-1} \alpha(u) C(u) \cos\left[\frac{\pi(2x+1)u}{2N}\right], \quad x = 0,1,\ldots,N-1 \tag{2}$$

The coefficients at the beginnings of formulae make the transform matrix orthogonal. For both equations (1) and (2) the coefficients are given by the following notation:

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}}, u = 0 \\ \sqrt{\frac{2}{N}}, u \neq 0 \end{cases} \tag{3}$$

As can be seen from (1), the substitution of $u = 0$ yields $C(0) = \alpha(0) \sum_{x=0}^{N-1} f(x)$, which is the mean of the sample. By convention, this value is called the DC coefficient of the transform and the others are referred to as AC coefficients.

For every value $u = 0,1,\ldots,N-1$, transform coefficients correspond to a certain waveform. The first waveform renders a constant value, whereas all other waveforms ($u = 1,2,\ldots,N-1$) produce a cosine function at increasing frequencies. The output of the transform for each u is the convolution of the input signal with the corresponding waveform.

Figure 2 shows plots of waveforms mentioned above.

Figure 2. 1D basis functions for *N=8*.

The two-dimensional DCT for a sample of size $N \times N$ is defined as follows:

$$C(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right] \quad (4)$$

The inverse of two-dimensional DCT for a sample of size $N \times N$:

$$f(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1}\alpha(u)\alpha(v)C(u,v)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right] \quad (5)$$

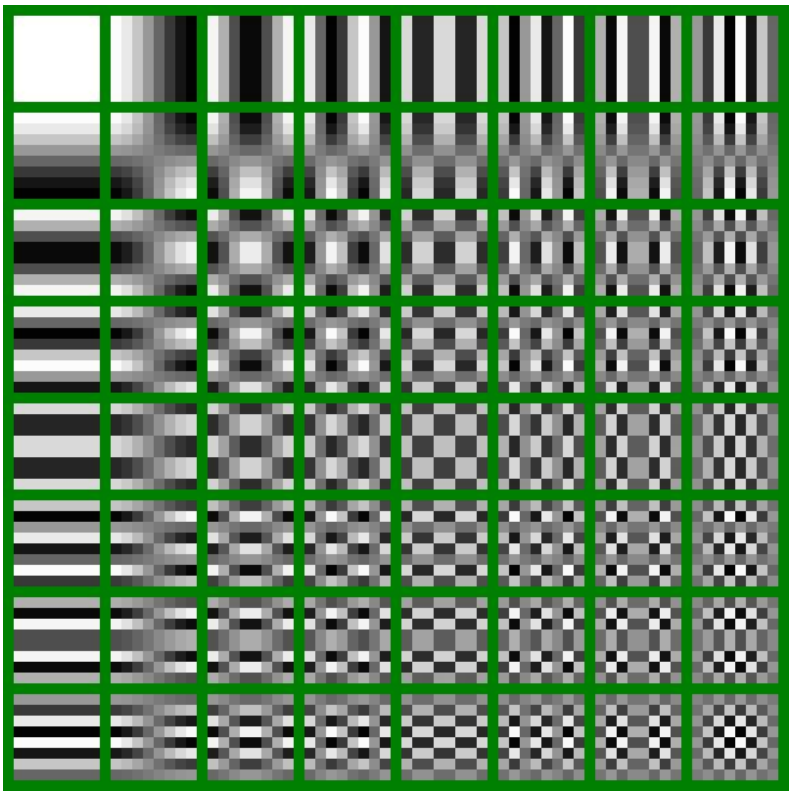


Figure 3. 2D basis functions for N = 8.

Separability is an important feature of 2D DCT, and allows expressing equation (4) in the following form:

$$C(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{N-1}\cos\left[\frac{\pi(2x+1)u}{2N}\right]\left\{\sum_{y=0}^{N-1} f(x,y)\cos\left[\frac{\pi(2x+1)v}{2N}\right]\right\} \quad (6)$$

This property yields the simple representation for basis functions of 2D transform; they can be calculated by multiplication of vertically oriented 1D basis functions (shown in Figure 2 for the case $N = 8$) with their horizontal representations. The visualization of such representation is plotted on Figure 3. As can be seen from the plot, the basis functions exhibit a progressive increase in frequency both in the vertical and horizontal directions.

To perform the DCT of length N effectively the cosine values are usually pre-computed offline. A 1D DCT of size N will require N vectors of N elements to store cosine values (matrix A). 1D cosine transform can be then represented as a sequence of dot products between the signal sample (vector x) and cosine values vectors (A^T), resulting in transformed vector $A^T x$.

A 2D approach performs DCT on input sample X by subsequently applying DCT to rows and columns of the input signal, utilizing the separability property of the transform. In matrix notation this can be expressed using the following formula:

$$C(u,v) = A^T XA \quad (7)$$

3. Implementation Details

With advent of CUDA technology it has become possible to perform high-level program parallelization. Generally, DCT8x8 is a high-level parallelizable algorithm and thus can be easily programmed with CUDA.

This section presents two different approaches to implementing DCT8x8 using CUDA. The first one is used to demonstrate CUDA programming model benefits, while the second allows creating really fast highly optimized kernels. The SDK sample includes 2 kernels based on the second approach: for the floating point data and for short integer data types (the routine is similar to that is used in LibJPEG).

In order to avoid confusion some notations need to be introduced:

- Block of pixels of size 8x8 will be further referred to as simply block;
- A set of blocks will be called a macroblock. The number of blocks in a macroblock denotes the size of a macroblock. The common size of a macroblock is 4 or 16 blocks. The layout of blocks in a macroblock should be given before each usage;
- CUDA threads grouped into execution block will be referred to as CUDA-block.

Implementing DCT by definition

The implementation of DCT8x8 by definition is performed using (7). To convert input 8x8 sample into the transform domain, two matrix multiplications need to be performed.

This solution is never used in practice when calculating DCT8x8 on CPU because it exhibits high computational complexity relatively to some separable methods. Things are different with CUDA; the described approach maps nicely to CUDA programming model and architecture specificity.

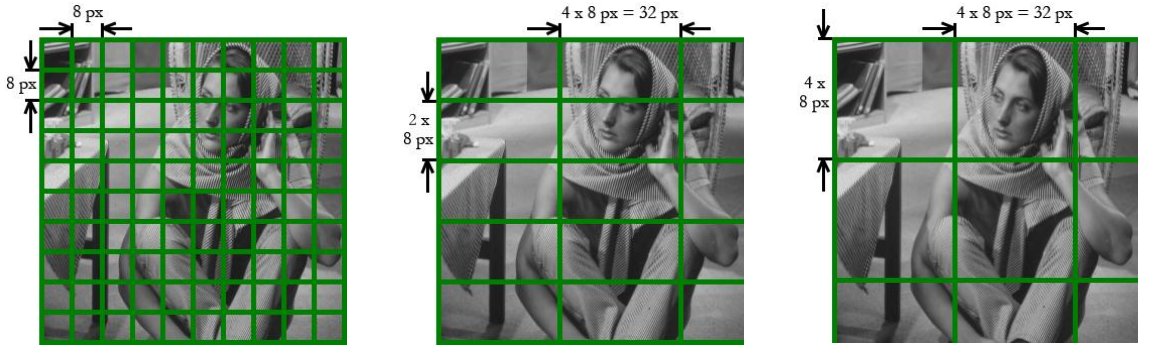
Image is split into a set of blocks as shown on Figure 4, left. Each CUDA-block runs 64 threads that perform DCT for a single block. Every thread in a CUDA-block computes a single DCT coefficient. All waveforms are pre-computed beforehand and stored in the array located in constant memory. This array can be viewed as a two dimensional array containing values of basis functions $A(x,u)$ one per column (shown in Figure 2).

Two-dimensional DCT is performed in four steps (considering thread-level):

1. A thread with coordinates (ThreadIdx.x, ThreadIdx.y) loads one pixel from a texture to shared memory. In order to make sure the whole block is loaded to the moment , all threads pass synchronization point;
2. The thread computes a dot product between two vectors: ThreadIdx.y column of cosine coefficients (which is actually the row of A^T with the same number) and ThreadIdx.x column of the input block. To ensure all coefficients of $A^T X$ are calculated, the synchronization must be passed;
3. The thread computes $(A^T X)A$ in the same manner as in 2;

4. The whole block is copied from shared memory to the output in global memory. Each thread works with the single pixel.

Note: The proposed implementation requires only two 8x8 blocks in shared memory and one 8x8 block in constant memory. The CUDA and the CPU versions can be found in the first kernel (dct8x8_kernel1.cu) and first reference (gold) functions (DCT8x8_gold.cpp).



Barbara image with 8x8 blocks (1st kernel)

Barbara image split into macroblocks. Each macroblock contains 8 blocks (2nd kernel)

Barbara image split into macroblocks. Each macroblock contains 16 blocks (short kernel)

Figure 4. Barbara image split with different grids.

Traditional DCT Implementation

DCT is not usually computed by definition. In fact the most common practice is to optimize computation by avoiding redundant multiplications and utilizing the separability of 2D case. Such approach has been implemented on GPU in [4] and shows processing speed of 300 Hz for 512x512 Barbara image.

The proposed approach uses DCT8x8 separability on all levels of detail. An image is split into a number of macroblocks. To calculate the DCT coefficients for a single block (Figure 4, left) only 8 threads are needed (the whole 1D 8-tap DCT is performed by a thread), so in order to create enough workload for the GPU, the size of a macroblock must be multiple of the number of threads within the warp. In case of NVIDIA GeForce 8x series the number of threads in a warp is 32, which maps to a macroblock of sizes 4x (Figure 4, center, right). Each thread performs DCT for the row and column corresponding to its ThreadIdx.x number inside the block with coordinates (ThreadIdx.y, ThreadIdx.z) inside the macroblock.

The reduced computation of 8-point DCT is based on the approach from [3]. The majority of low-level optimizations that are common in CPU implementations aren't needed here since floating point math is native to GPUs and MUL, ADD and MAD operations are executed with the same speed. The proposed implementation is optimized by the total amount of described floating point operations. The floating point divisions by constants are replaced with multiplications by reciprocals or arithmetic shifts.

The described approach takes into account the structure of matrix A^T that exhibits high redundancy and symmetry of matrix elements. It can be presented in the following symbolic form (the axes of symmetry are drawn with dash line):

$$A^T = \frac{1}{\sqrt{8}} \cdot \left[\begin{array}{cccc:cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ a & c & d & f & -f & -d & -c & -a \\ b & e & -e & -b & -b & -e & e & b \\ c & -f & -a & -d & d & a & f & -c \\ \hdashline 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ d & -a & f & c & -c & -f & a & -d \\ e & -b & b & -e & -e & b & -b & e \\ f & -d & c & -a & a & -c & d & -f \end{array}\right], \quad (8)$$

where a, b, c, d, e, f stand for (9):

$$\begin{aligned} a &= \sqrt{2}\cos\left(\frac{\pi}{16}\right) & d &= \sqrt{2}\cos\left(\frac{5\pi}{16}\right) \\ b &= \sqrt{2}\cos\left(\frac{\pi}{8}\right) & e &= \sqrt{2}\cos\left(\frac{3\pi}{8}\right) \\ c &= \sqrt{2}\cos\left(\frac{3\pi}{16}\right) & f &= \sqrt{2}\cos\left(\frac{7\pi}{16}\right) \end{aligned} \quad (9)$$

Thus, the 8-point DCT equation $Y = A^T X$ can be decomposed in (10):

$$\begin{aligned} \begin{bmatrix} Y(0) \\ Y(2) \\ Y(4) \\ Y(6) \end{bmatrix} &= \frac{1}{\sqrt{8}} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ b & e & -e & -b \\ 1 & -1 & -1 & 1 \\ e & -b & b & -e \end{bmatrix} \cdot \begin{bmatrix} X(0)+X(7) \\ X(1)+X(6) \\ X(2)+X(5) \\ X(3)+X(4) \end{bmatrix} \\ \begin{bmatrix} Y(1) \\ Y(3) \\ Y(5) \\ Y(7) \end{bmatrix} &= \frac{1}{\sqrt{8}} \cdot \begin{bmatrix} a & -c & d & -f \\ c & f & -a & d \\ d & a & f & -c \\ f & d & c & a \end{bmatrix} \cdot \begin{bmatrix} X(0)-X(7) \\ X(6)-X(1) \\ X(2)-X(5) \\ X(4)-X(3) \end{bmatrix} \end{aligned} \quad (10)$$

As can be seen from (10), there is still some clear symmetry in the matrix corresponding to even elements of vector Y, which can be properly utilized.

Note: The proposed implementation requires single macroblock allocated in shared memory and few floats in constant memory. The CUDA and the CPU versions can be found in the second kernel (dct8x8_kernel2.cu), short kernel (dct8x8_kernel_short.cu) and second reference (gold) functions (DCT8x8_gold.cpp).

4. Evaluation

After DCT8x8 is computed, it is possible to perform all sorts of analysis based on its values. In JPEG compression DCT coefficients are quantized to reduce the amount of information

that cannot be perceived by the human eye. The compression rate depends on the quantity of coefficients that are non-zero after quantization has been performed. Roughly speaking to achieve compression rate of 75 percent (of the initial size), 25 percent of least valuable coefficients should be zero after quantization step.

This sample is not dedicated to JPEG compression, however to illustrate the use of DCT8x8 an additional step for quantizing DCT coefficients and running inverse DCT8x8 on them is performed. The resulting image that contains blocking artifact due to quantization is stored to hard drive.

The evaluation of the sample can be done in several ways: the speedup rate analysis and consistency checking of CPU and CUDA implementations of the same approach. As for the first, each implementation outputs the pure processing timing to the console window. The speedup rate can be measured as the ratio of timings of reference CPU (Gold) implementation and of CUDA implementation. The consistency checking is the assurance that both CPU and CUDA implementations of the same approach produce the same output given the same input. The bitwise check of results may fail here because of possible differences in floating point operations sequences in both implementations or due to differences in floating point units. Therefore the consistency checking is performed using the objective image similarity metric PSNR.

We have chosen PSNR because it is commonly used to evaluate image degradation or reconstruction quality. PSNR stands for Peak Signal to Noise Ratio and is defined for two images I and K of size $M \times N$ as:

$$PSNR(I,K) = 20\log_{10}\frac{MAX_I}{\sqrt{MSE(I,K)}} \tag{11}$$

Where I is the original image, K is a reconstructed or noisy approximation, MAX_I is the maximum pixel value in image I and MSE is a mean square error between I and K:

$$MSE(I,K) = \frac{1}{M}\frac{1}{N}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\|I(i,j) - K(i,j)\|^2 \tag{12}$$

PSNR is expressed in decibel scale and takes on positive infinity for identical images. In image reconstruction typical values for PSNR vary within the range $[30,50]$. PSNR of 50 and higher calculated from two images that were processed on diverse devices with the same algorithm says the results are practically identical.

Orig.–Impl.	Barbara_512x512	Barbara_1Kx1K	Barbara_2Kx2K	Barbara_4Kx4K
Gold 1	32.777092	34.612900	36.814545	39.721603
CUDA 1	32.777027	34.612907	36.814545	39.721588
Gold 2	32.777050	34.612888	36.814545	39.721588
CUDA 2	32.777039	34.612885	36.814545	39.721592

Table 1. PSNR between Original and Processed images.

The consistency checking of CPU and CUDA implementations of both approaches to DCT8x8 implementation is performed in two steps:

1. PSNR values between the original image and the processed (blocky) results are calculated. It is natural to expect that these values should be similar for both implementations. Given Barbara image of size 512x512 as an input, the expected PSNR values approximate to 32.777.
2. PSNR between images compressed by CPU and CUDA implementations of the same approach is calculated. Any values above 50 dB are considered as consistent result, which yields that CUDA implementation works properly.

GPU–CPU	Barbara_512x512	Barbara_1Kx1K	Barbara_2Kx2K	Barbara_4Kx4K
Impl. 1.	58.613663	62.188042	62.834183	63.089985
Impl. 2.	63.903233	66.192337	66.547394	68.100159

Table 2. PSNR between images processed on diverse architectures with the same algorithm.

For Barbara test image the results obtained on step 1 are shown in the Table 1. Comparison between images processed on CPU and GPU is given in Table 2.

5. Optimization issues

There are several ways to get rid of G8x architecture while developing CUDA kernel. Here are some of the short paths:

- **Eliminate bank conflicts while working with shared memory.** This issue was resolved by padding each row of the macroblock stored in shared memory with one element. The resulting amount of shared memory used per CUDA-block is (MACROBLOCK_WIDTH + 1) x MACROBLOCK_HEIGHT. Such configuration allows simultaneous accessing rows and columns without bank conflicts. This hack is also used in Transpose SDK sample.
- **Access global memory in coalesced manner.** In second kernel the copying from global to shared memory is performed by the same threads that perform 8-tap DCT. However, this approach doesn't work for short kernel (each element is 2 bytes long). It causes 2-way bank conflict (in shared memory) and uncoalesced global memory access. This issue can be resolved if only half of threads in each block perform moving of 2 short elements as a single 4-byte element.
- **Eliminate all other reasons of warp serialization** (non-unified constant memory access, *etc.*)
- **Maximize GPU occupancy**. The value can be calculated using Occupancy Calculator. Briefly, the value depends on the following parameters:
    - *Amount of shared memory usage.* The more memory is used by CUDA block – the less amount of CTAs can be launched simultaneously.

- *Number of registers used per-thread*. The occupancy values close to 1.0 can be reached in case that each cuda thread uses not more than 10 registers per thread. Note that nvcc has several optimization options (refer nvcc guide). One can adjust maximum register count per thread using ptxas switch –maxrregcount=N that forces the compiler to use not more than N registers per thread. This usually increases slow local storage usage, which can drastically degrade performance when no local storage had been used. There are still few ways to decrease registers usage: either by rewriting portions of the code directly in PTX assembler, or by manually declaring N variables and assigning all intermediate calculation results to them (this approach was partially adopted in 2nd and short kernels).
- *Amount of threads per block*. This parameter often conflicts with shared memory usage. They have to be adjusted simultaneously.

6. Source Code Details

The whole sample code is documented using Doxygen comments. In order to generate full project documentation it is necessary to include *.cu files in the set of filetypes containing code on the generation step.

The sample project is organized as follows:

- Dct8x8.cu is the main file. It should be compiled with nvcc compiler. All CUDA kernels are described separately and included into Dct8x8.cu
- Dct8x8_kernel1.cu contains the implementation of DCT8x8 by definition.
- Dct8x8_kernel2.cu contains optimized traditional implementation of DCT8x8 for floats.
- Dct8x8_kernel_short.cu contains optimized traditional implementation of DCT8x8 for short integers.
- Dct8x8_kernel_quantization.cu contains unoptimized routines that perform quantization of the DCT coefficients.
- Dct8x8_Gold.cpp and Dct8x8_Gold.h contains a gold version of DCT8x8 implemented on the CPU.
- BmpUtil.cpp and BmpUtil.h contain a simple code for loading/saving bitmaps.

Running the sample doesn't require any additional actions. The program loads Barbara image from \data folder. It checks that image dimensions are multiples of 8 and launches different DCT8x8 implementations. DCT8x8, quantizer and inverse DCT8x8 are executed in separate kernels. Finally, forward DCT8x8 kernel timings are shown both with PSNR values. All processed images are stored to the hard drive.

Since any image does not necessarily contain an integer number of macroblocks, additional launching of specific kernels may be needed to process the remaining blocks.

7. Conclusion

In this paper we proposed two new approaches to calculation of the discrete cosine transform for 8x8 blocks with NVIDIA CUDA technology. CUDA provides a natural way of shifting DCT8x8 computation to GPU. Both approaches were implemented for CPU and GPU. The GPU implementations utilize DCT8x8 separability on high-level, which yields the significant performance boost even compared to a modern CPU. The proposed approaches are illustrated using the sample code that performs parts of JPEG compression and decompression routines. The performance testing was held for both approaches and they exhibited good speedup rates while keeping objective result quality constant.

References

[1] Syed Ali Khayam. "*The Discrete Cosine Transform (DCT): Theory and Application*". ECE 802 – 602: Information Theory and Coding, March 10th 2003.

[2] R. Kresch and N. Merhav, *"Fast DCT domain filtering using the DCT and the DST"*. HPL Technical Report #HPL-95-140, December 1995.

[3] Tze-Yun Sung, Yaw-Shih Shieh, Chun-Wang Yu, Hsi-Chin Hsin. *"High-Efficiency and Low-Power Architectures for 2-D DCT and IDCT Based on CORDIC Rotation"*. Proceedings of the 7th ICPDC, pp. 191-196, 2006.

[4] Simon Green. *Discrete Cosine Transform GPU implementation.* http://developer.download.nvidia.com/SDK/9.5/Samples/vidimaging_samples.html#gpgpu_dct

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, GeForce, NVIDIA Quadro, and NVIDIA CUDA are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2008-2012 NVIDIA Corporation. All rights reserved.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com