DCT8x8

Authors
Anton Obukhov
anton.obukhov@gmail.com
Alexander Kharlamov
akharlamov@nvidia.com

March 2008

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

Abstract

The Discrete Cosine Transform (DCT) is a Fourier-like transform, which is widely used in digital signal processing. It was first proposed by Ahmed et al. (1974) and has become the key feature in lossy compression and other DSP-related areas. DCT is known for its high “energy compaction” and decorrelation properties, meaning that the transformed signal can be easily analyzed using few low-frequency components.

There are several types of DCT [2]. The most popular is a symmetric presentation and its inverse; these are utilized in JPEG compression routines. We will focus on this application of DCT to signal processing.

This whitepaper illustrates the concept of highly-parallelized DCT without going deep into mathematical details and focuses on the CUDA-based implementation. Performing DCT computations on a GPU gives a significant performance boost even compared to a modern CPU. To illustrate our approach, the sample code implements part of JPEG compression routine: performs forward DCT on 8x8 blocks, quantizes coefficients, and performs inverse DCT. The performance testing is done using Barbara image from Marco Schmidt's standard test images database (Figure. 1).

Figure. 1. “Barbara” test image

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

Motivation

Discrete cosine transform is a powerful tool widely utilized in many applications, especially related to digital signal processing. It has become a de-facto standard in image and video coding algorithms. While the Fourier Transform represents a signal as the mixture of sines and cosines, the Cosine Transform operates only with cosines. The purpose of DCT is to perform "energy compaction" of the input signal and to present the output in the frequency domain. DCT turns out to be a reasonable balance of optimality of input decorrelation (approaching Karhunen-Loève transform) and the computational complexity.

A lot of effort has been put into optimizing DCT routines on existing hardware. However there is no limit for improvement. DCT coupled with quantizer are commonly used to decrease the amount of redundant information in different compression algorithms. GPU acceleration of DCT computation has been possible since appearance of shader languages. However this required a specific setup to utilize common graphics API such as Open GL or D3D. CUDA, on the other hand, provides a natural extension of C language that allows a transparent implementation of GPU accelerated algorithms. Also DCT can greatly benefit from CUDA-specific features, such as shared memory, explicit synchronization points.

This whitepaper is organized as follows:

- In section 1 DCT is described to provide some theoretical background.
- Several CUDA implementations of DCT are discussed in section 2.
- Additional references can be found in section 3.

1. DCT Theory

The formal definitions for a discrete cosine transform (DCT) of a 1-D sequence of length N [1]:

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

for u = 0, 1, 2, … N-1. The inverse transformation is defined as

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

for x = 0, 1, 2, … N-1. For both equations **(1)** and **(2)**

$$\alpha = \begin{cases} \sqrt{\dfrac{1}{N}}, u = 0 \\ \sqrt{\dfrac{2}{N}} \quad u \neq 0 \end{cases} \qquad \textbf{(3)}$$

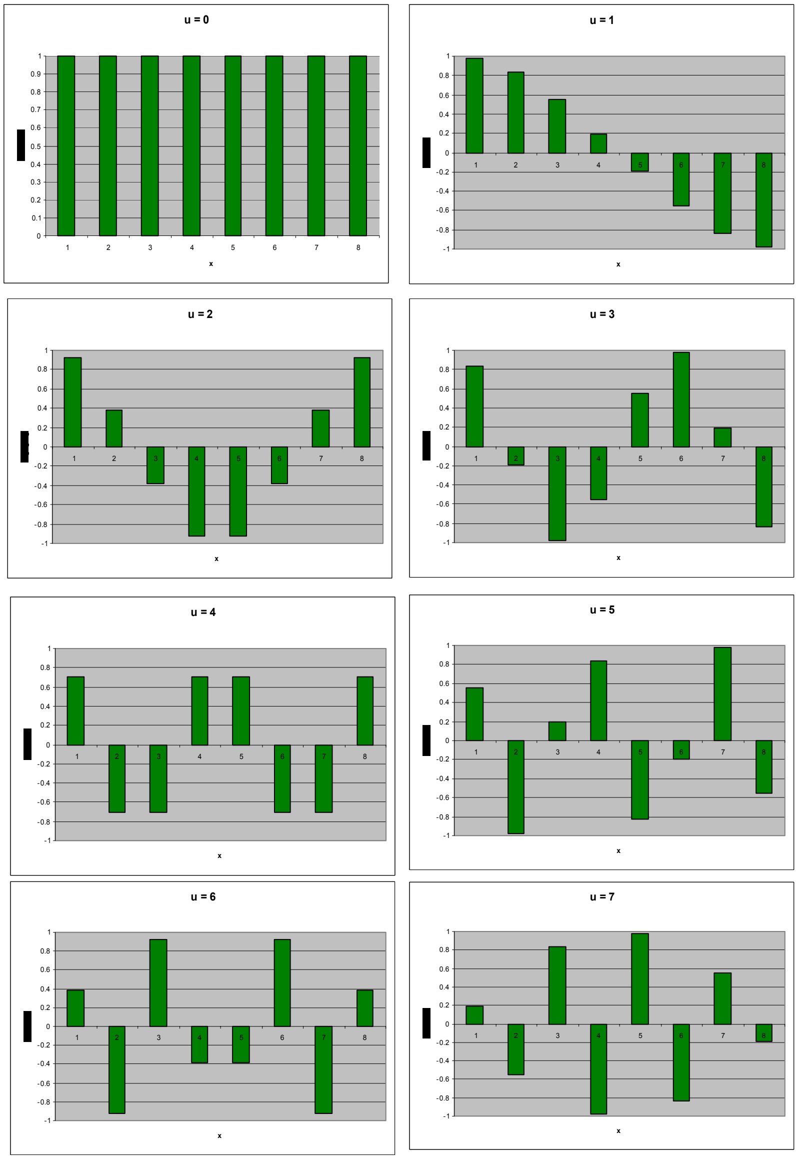


Figure 2. 1-D basis functions for N=8.

For every value of u = 0 …N-1, transform coefficients correspond to a certain waveform. The first waveform renders a constant value, whereas all other waveforms (u = 1, 2… N-1) give a cosine function at increasing frequencies. Figure 2 shows these functions for a sequence of 8 samples.

A 2-D DCT is defined as follows

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

And inverse transform is defined as

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

Separability is an important feature of 2-D DCT, and allows expressing equation **(4)** as

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

This property allows an easy representation for the basis functions: multiply the horizontally oriented 1-D basis functions (shown in Figure 3) with vertically oriented set of the same functions. Figure 3 shows the basis functions for N = 8. Note, that the basis functions exhibit a progressive increase in frequency both in the vertical and horizontal direction.

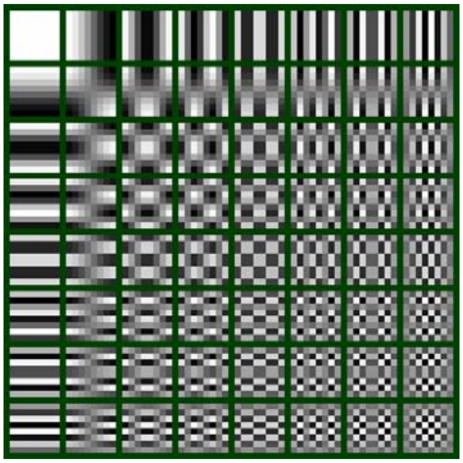


Figure 3. 2-D basis functions for N = 8.

To perform a DCT of length N effectively the cosine values are usually pre-computed offline. A 1-D DCT of size N will require N vectors of N elements to store cosine values (matrix *A*). 1-D cosine transform can be then represented as a sequence of dot products between the signal samples vector (**x**) and cosine values vectors ($A\mathbf{x}^T$). A 2-D approach performs DCT by subsequently applying DCT to rows and columns of the input signal. In matrix form this can be expressed as

$$C(u,v)=A^T SA$$

Where *A* is the matrix for forward DCT.

2. Implementation Details

Implementing DCT by definition (1st implementation)

DCT can be easily programmed using CUDA. In fact, implementing DCT by definition yields great results in performance, since the algorithm maps nicely to CUDA programming model.

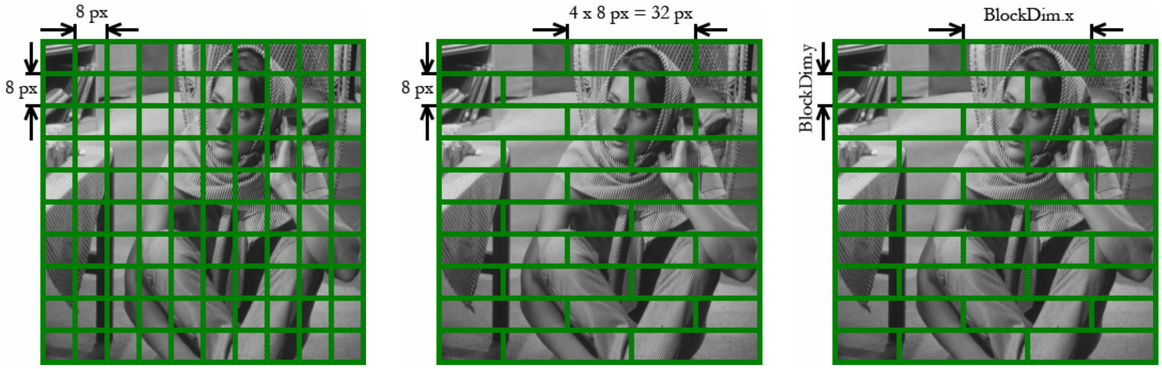
In order to avoid confusion some notations need to be introduced:

- Block of pixels of size 8x8 will be further referred to as simply block.
- CUDA threads grouped into execution block will be referred to as CUDA-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 of size 8x8 pixels.

Image is split into a set of blocks. Each CUDA-block runs 64 threads that perform DCT for a single block. Cosine values are pre-computed beforehand and stored in array located in constant memory. This array and can be viewed as a two dimensional array containing values of C(u, x) function. Rows contain values of C (u, x) per every value of u. Columns contain values of C (u, x) per every value of x.

2-D DCT is performed in two steps. Every thread in a CUDA-block computes a single DCT coefficient. Each thread loads input from a texture to shared memory. After they sync, each thread computes a dot product between two vectors: thread with (ThreadIdx.x, ThreadIdx.y) computes a dot product between ThreadIdx.x row of input signal and ThreadIdx.y column of cosine coefficients. This dot product is stored in shared memory as well.

The second step requires performing the same matrix multiplication with a transposed matrix of cosine coefficients. Results are stored in shared memory.



Barbara image with 8x8 blocks

Barbara image split by macroblocks. Each macroblock contains 4 blocks of 8x8 pixels

Barbara image split into CUDA blocks (each CUDA block maps to one macroblock)

Figure 4. Barbara image split into different blocks.

The described approach is implemented in the first kernel (dct8x8_kernel1.cu) and first reference (gold) function (DCT8x8_gold.cpp).

After DCT 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 (75% 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 DCT an additional step for quantizing DCT coefficients and running inverse DCT on them is performed. The resulting image is dumped to HDD.

Traditional DCT Implementation (2nd implementation)

DCT is not usually computed by definition. In fact the most common practice is to optimize computation by avoiding redundant multiplications. This approach has been implemented on GPU in [3] and shows 300 Hz for a 512x512 Lena image [4].

Image is split into a number of macroblocks. To calculate the DCT coefficients for a single block (Fig. 4, left) only 8 threads are needed, so in order to create enough workload for the GPU the size of a macroblock is defined by the number of threads within the warp. In case of GeForce 8x series the number of threads in a warp is 32, which maps nicely to a macroblock of size 4 (Fig. 4, center, right). Each thread performs a DCT for the row and column corresponding to its (ThreadIdx.x mod 8) number inside (ThreadIdx.x div 8) block of macroblock.

Since image doesn't necessarily contain an integer number of macroblocks additional launching of specific kernels may needed. This kernel will use the implementation of DCT by definition, to perform DCT over any remaining blocks. This won't happen in case when image size in blocks is multiple of 4.

Hardcore Optimizations that are common in CPU optimizations aren't needed here since floating point math is native to GPUs and MUL, ADD and MAD operations are executed with the same speed. This implementation is optimized by the total amount of described low-cost arithmetic operations.

The reduced computation of 8-point DCT is based on the approach from [3]. It takes into account the structure of matrix *A* that exhibits high redundancy of matrix elements. It can be presented in the symbolic form (7), where a, b, c, d, e, f stand for (8).

$$. A=\frac{1}{\sqrt{8}}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ a & c & d & f & -f & -d & -c & -a \\ b & e & -e & -b & -b & -e & e & b \\ c & -f & -a & -d & d & a & f & -c \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ d & -a & f & c & -c & -f & a & d \\ e & -b & b & -e & -e & b & -b & e \\ f & -d & c & -a & a & -c & d & -f \end{bmatrix} \qquad \textbf{(7)}$$

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

Thus, the DCT 8-point equation Y = AX can be decomposed as in (9):

$$
\begin{bmatrix} Y(0) \\ Y(2) \\ Y(4) \\ Y(6) \end{bmatrix} = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ b & e & -e & -b \\ 1 & -1 & -1 & 1 \\ e & -b & b & -e \end{bmatrix} \begin{bmatrix} x(0)+x(7) \\ x(1)+x(6) \\ x(2)+x(5) \\ x(3)+x(4) \end{bmatrix}
$$

$$
\begin{bmatrix} Y(1) \\ Y(3) \\ Y(5) \\ Y(7) \end{bmatrix} = \frac{1}{\sqrt{8}} \begin{bmatrix} a & -c & d & -f \\ c & f & -a & d \\ d & a & f & -c \\ f & d & c & a \end{bmatrix} \begin{bmatrix} x(0)-x(7) \\ -x(1)+x(6) \\ x(2)-x(5) \\ -x(3)+x(4) \end{bmatrix} \qquad \textbf{(9)}
$$

The described approach is implemented in the second kernel (dct8x8_kernel2.cu) and second reference (gold) function (DCT8x8_gold.cpp).

PSNR Evaluation

In order to assure that GPU implementation is consistent, PSNR comparison is done.

We have chosen PSNR because it is commonly used to evaluate image reconstruction quality. PSNR stands for Peak Signal to Noise Ratio and is defined for two images I and K of size MxN as:

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

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:

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

PSNR is expressed in decibel scale and PSNR for identical values is an INF. In data compression typical values for PSNR lie in the interval [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. Any differences in PSNR are related to FPU differences.

Comparing CUDA and CPU implementation of DCT has been done in two steps:

1. PSNR between the original image and compressed results is calculated. It is natural to expect that for each implementation these values should be the same.
2. PSNR between compressed images is calculated. Any values above 50 dB are considered as identical results.

For Barbara test image the obtained results are shown in the table below. PSNR values are calculated between the original image and the compressed.

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

Table.1 PSNR values

Comparison between compressed images on CPU and GPU is given in table 2.

	Barbara_512x512	Barbara_1Kx1K	Barbara_2Kx2K	Barbara_4Kx4K
GPU vs CPU (1)	58.613663	62.188042	62.834183	63.089985
GPU vs CPU (2)	63.903233	66.192337	66.547394	68.100159

Table 2. PSNR between different methods

Despite the differences in floating point computations, PSNR between GPU computed and CPU computed images is easily acceptable.

Source Code Details

The sample is organized as follows:

- Dct8x8.cu is the main file. It is compiled with nvcc compiler. All CUDA kernels are described separately and included into Dct8x8.cu
- Dct8x8_Gold.cpp and Dct8x8_Gold.h contains a gold version of DCT implemented on the CPU.
- Dct8x8_kernel1.cu contains the implementation by definition of DCT.
- Dct8x8_kernel2.cu contains a traditional implementation of DCT.
- BmpUtil.cpp and BmpUtil.h contain a simple code for loading/saving bmp images.

Running the Sample

Running the sample doesn't require any additional actions. Sample loads Barbara.bmp from \data folder. It checks that image sizes allow splitting it into blocks of 64 pixels, and launches different DCT implementations. DCT, quantizer and IDCT are executed in separate kernels. Compressed images are dumped to HDD.

Conclusion

DCT is used in so many algorithms it is impossible to underestimate its value. DCT can be accelerated using the latest NVIDIA hardware. CUDA provides a natural way of shifting DCT computation to GPU.

3. 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

Trademarks

Copyright