FFT-based 2D convolution

Victor Podlozhnyuk
vpodlozhnyuk@nvidia.com

June 2007

Document Change History

Version	Date	Responsible	Reason for Change
1.0	2007/06/01	vpodlozhnyuk	Initial release

Abstract

This sample demonstrates how general (non-separable) 2D convolution with large convolution kernel sizes can be efficiently implemented in CUDA using CUFFT library.

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

Introduction

The whitepaper of the `convolutionSeparable CUDA SDK` sample introduces convolution and shows how separable convolution of a 2D data array can be efficiently implemented using the CUDA programming model. However, the approach doesn't extend very well to general 2D convolution kernels. In such cases, a better approach is through Discrete Fourier Transformation. This latter approach is based on the theorem, central to Digital Signal Processing applications, that convolution in geometric space amounts to point-wise multiplication (modulation) in frequency space. Therefore, given an efficient direct/inverse Discrete Fourier Transformation implementation, Fourier-based convolution can be more efficient than a straightforward implementation.

Implementation Details

Fast Fourier Transformation (FFT) is a highly parallel "divide and conquer" algorithm for the calculation of Discrete Fourier Transformation of single-, or multidimensional signals. It can be efficiently implemented using the CUDA programming model and the CUDA distribution package includes CUFFT, a CUDA-based FFT library, whose API is modeled after the widely used CPU-based "FFTW" library.

The basic outline of Fourier-based convolution is:

- Apply direct FFT to the *convolution kernel*,
- Apply direct FFT to the *input data array* (or *image*),
- Perform the point-wise multiplication of the two preceding results,
- Apply inverse FFT to the result of the multiplication.

However, there are some issues that need to be taken into account:

1) The DSP theorem applies to input data (both the image and the convolution kernel) of same size. Therefore, assuming the image is bigger than the convolution kernel, which is usually the case in practice, the convolution kernel needs to be expanded to the image size and padded according to Figure 1. As can be seen on figures 2 and 3 (see below), cyclic convolution with the expanded kernel is equivalent to cyclic convolution with initial convolution kernel.

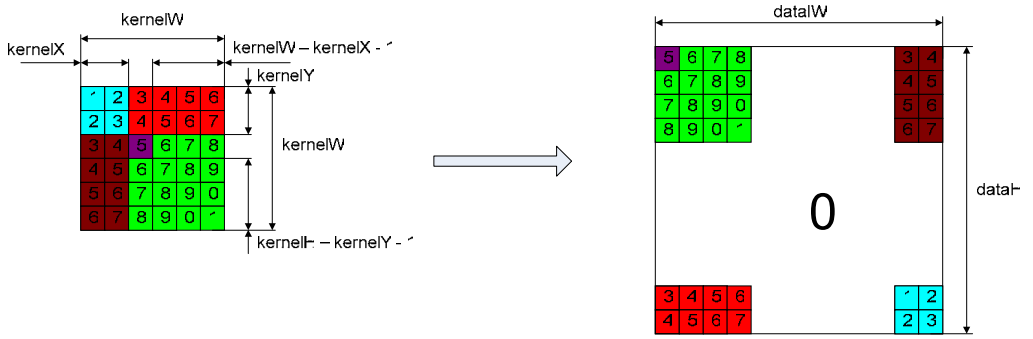

Figure 1. Expansion of the convolution kernel to the image size: cyclically shift the original convolution kernel, so that the central element of the kernel is at (0, 0)

2) The FFT "performs" *cyclic* convolution: The convolution kernel wraps around image borders in both dimensions. Figure 2 illustrates the convolution computation in the non-border case – that is when the kernel does not cross any image borders – and Figure 3 illustrates the same computation in the border case for a cyclic convolution.

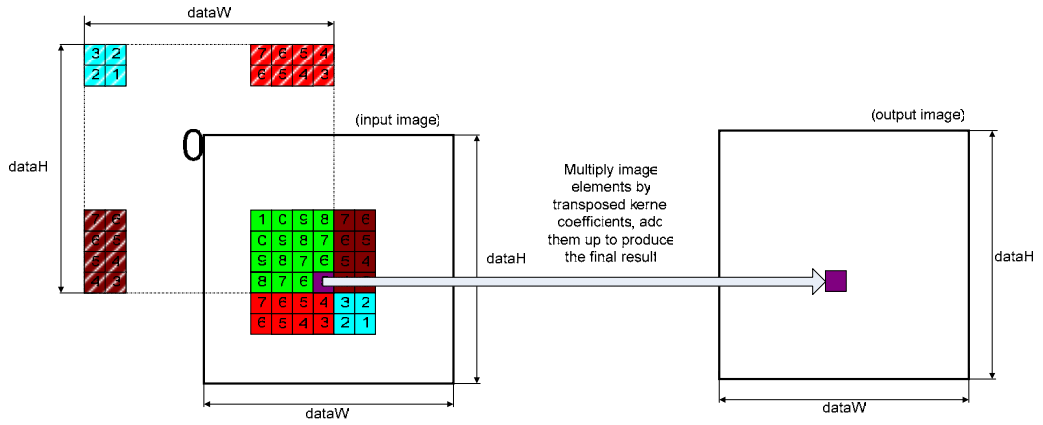

Figure 2. Cyclic convolution: non-border case

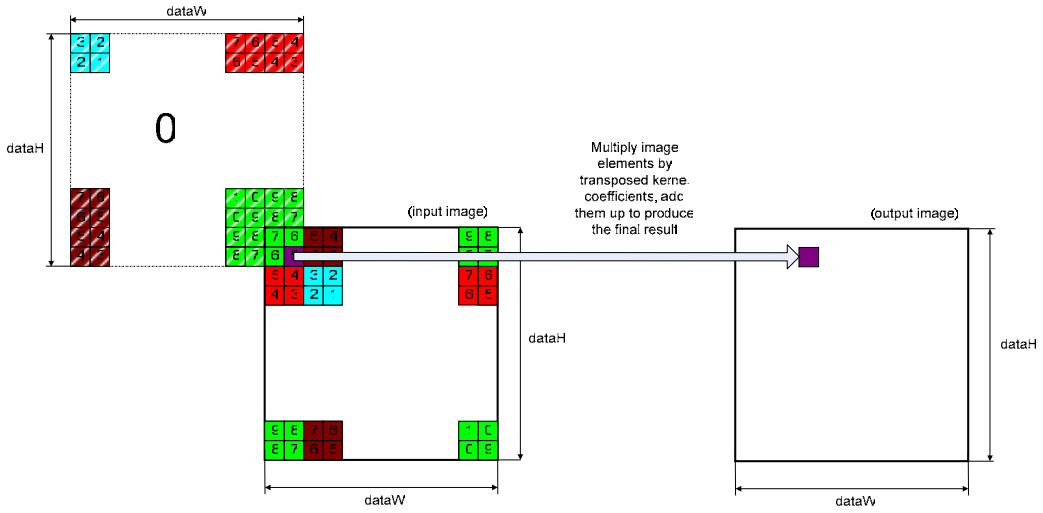

Figure 3. Cyclic convolution: border case.

However, most image processing applications require a different behavior in the border case: Instead of wrapping around image borders the convolution kernel should *clamp to zero* or *clamp to border* when going past a border. For the Fourier-based convolution to exhibit a *clamp to border* behavior, the image needs to be expanded and padded in both dimensions as illustrated in Figures 4, 5, 6, and 7. The convolution kernel just needs to be adjusted to the new padded sizes `(fftW, fftH)` as illustrated in Figure 1.

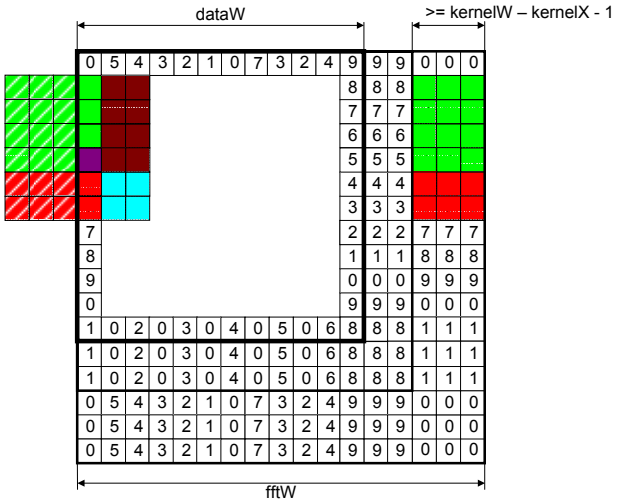

Figure 4. Cyclic convolution of the padded image: left border case. Wrapped around kernel elements get multiplied by correct data value (pixels).

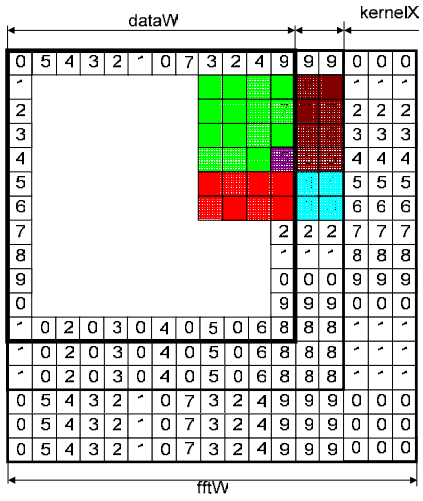

Figure 5. Cyclic convolution of the padded image: right border case. Wrapped around kernel elements get multiplied by correct data values (pixels).

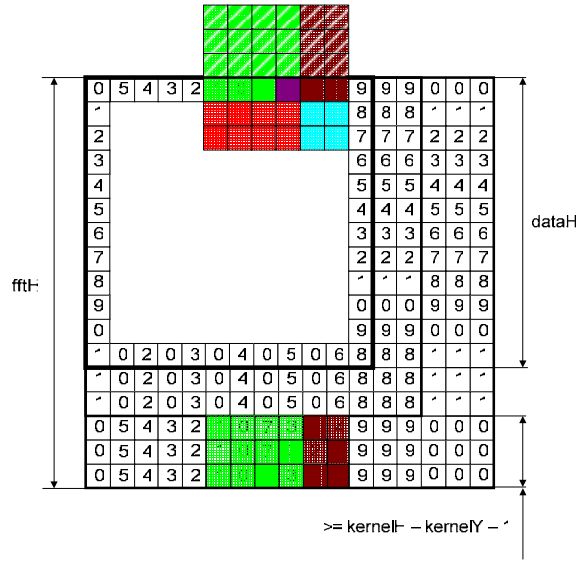

Figure 6. Cyclic convolution of the padded image: top border case. Wrapped around kernel elements are multiplied by correct data values (pixels).

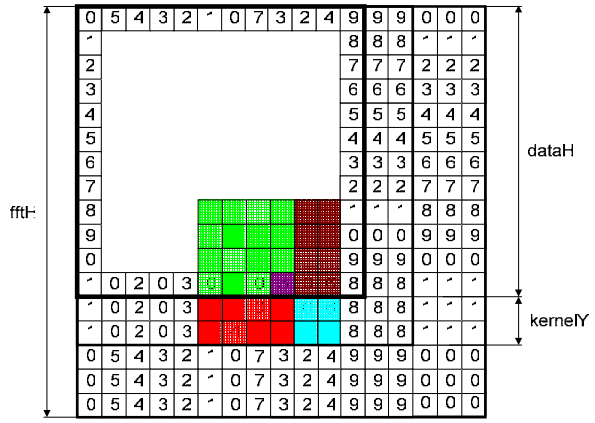

Figure 7. Cyclic convolution of the padded image: bottom border case. Wrapped around kernel elements are multiplied by correct data values (pixels).

Required padding is only kernelW – 1 elements in width and kernelH – 1 elements in height, but for different data sizes CUFFT operates with different speeds and different precision. In particular, if FFT dimensions are small multiples of powers of N, where N varies from 2 to 8 for CUFFT, the performance and precision are best. For this reason, we round up each padded dimension (dataW + kernelW – 1) and (dataH + kernelH – 1) to fftW and fftH respectively as follows: We round up to the nearest power of two if the padded dimension is less than or equal to 1024, or only to the nearest multiple of 512 otherwise, in order to save computations and memory bandwidth, since the per-element computation complexity of the FFT algorithms grows as the sizes of the Fourier transformation increase.

Performance

Benchmarking FFT-based convolution is somewhat ambiguous issue to be expressed in a single rate of filtered pixels per second, since per-element computation complexity of the underlying FFT algorithms depends only on the padded image size, and in its turn padded image size depends both on the image and convolution kernel sizes. For power of two padded image size (`fftW, fftH),` effective per-element computation complexity is:

$$\frac{O(fftW \cdot fftH \cdot \log(fftW \cdot ftH))}{dataW \cdot dataH} \approx O(\log(fftW) + \log(fftH))$$

against `O(kernelW * kernelH)` of a straightforward convolution implementation. Increasing the convolution kernel size, still assuming it to be much smaller than of the image (as usually observed on practice), we can see that the value of the former expression remains approximately the same, but the value of the latter expression increases proportionally to the number of elements in the convolution kernel. So, for each "big enough" input image size, starting from certain convolution kernel size, FFT-based convolution becomes more advantageous than a straightforward implementation in terms of performance.

Table below gives performance rates

FFT size	256x256	512x512	1024x1024	1536x1536	2048x2048	2560x2560	3072x3072	3584x3584
Execution time, ms	0.75	2.08	6.73	28	42	89	146	178
FFT convolution rate, MPix/s	87	125	155	85	98	73	64	71

So, performance depends on FFT size in a non linear way. On average, FFT convolution execution rate is 94 MPix/s (including padding).

The 2D FFT-based approach described in this paper does not take advantage of separable filters, which are effectively 1D. Therefore, it should not come as a surprise that for separable convolutions, the approach used in convolutionSeparable performs at much higher rates. The 2D FFT-based approach is however the better choice for large non-separable 2D convolutions, for which straightforward shared memory-based approaches either do not perform as well because they require a big apron that introduces too much memory read overhead, or are simply not applicable because they require more shared memory than is available

Bibliography

1. Y. A. Romanyuk (2005). "Discrete Signal Transformations". *Basics of Digital Signal processing. Part 1*. Chapter 3.
2. Wolfram Mathworld. "Convolution" http://mathworld.wolfram.com/Convolution.html

3. Wolfram Mathworld. “Fast Fourier Transform” http://mathworld.wolfram.com/FastFourierTransform.html