Image Convolution with CUDA

Victor Podlozhnyuk
sdkfeedback@nvidia.com

September 2013

Document Change History

Version	Date	Responsible	Reason for Change
0.1	10/25/2006	Lee Howes	Initial version
0.2	2/09/2007	Mark Harris	Revised Lee's original document
0.3	2/26/2007	Eric Young	Revised document to match new SDK document format
0.8	3/21/2007	Mark Harris	First release version.
1.0	06/1/2007	Victor Podlozhnyuk	Adapted the whitepaper to new convolutionSeparable project.

Table of Contents

Abstract

Convolution filtering is a technique that can be used for a wide array of image processing tasks, some of which may include smoothing and edge detection. In this document we show how a separable convolution filter can be implemented in NVIDIA CUDA and provide some guidelines for performance optimizations.

Motivation

Convolutions are used by many applications for engineering and mathematics. Many types of blur filters or edge detection use convolutions. This example illustrates how using CUDA can be used for an efficient and high performance implementation of a separable convolution filter. Figure 1(b) shows the effect of a convolution filter.

Figure 1(a) Original Image

Figure 1(b) Blur convolution filter applied to the source image from Figure 1(a)

These two images show a comparison of an image convolution applied to an original source image.

How Does It Work?

Mathematically, a convolution measures the amount of overlap between two functions [1]. It can be thought of as a blending operation that integrates the point-wise multiplication of one dataset with another.

$$r(i) = (s * k)(i) = \int s(i-n)k(n)dn$$

In discrete terms this can be written as:

$$r(i) = (s * k)(i) = \sum_n s(i-n)k(n).$$

Convolution can be extended into two dimensions by adding indices for the second dimension:

$$r(i) = (s * k)(i, j) = \sum_n \sum_m s(i-n, j-m)k(n, m)$$

In the context of image processing a convolution filter is just the scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels. This scalar product is a parallel operation that is well suited to computation on highly parallel hardware such as the GPU. This is demonstrated below in Figure 2.

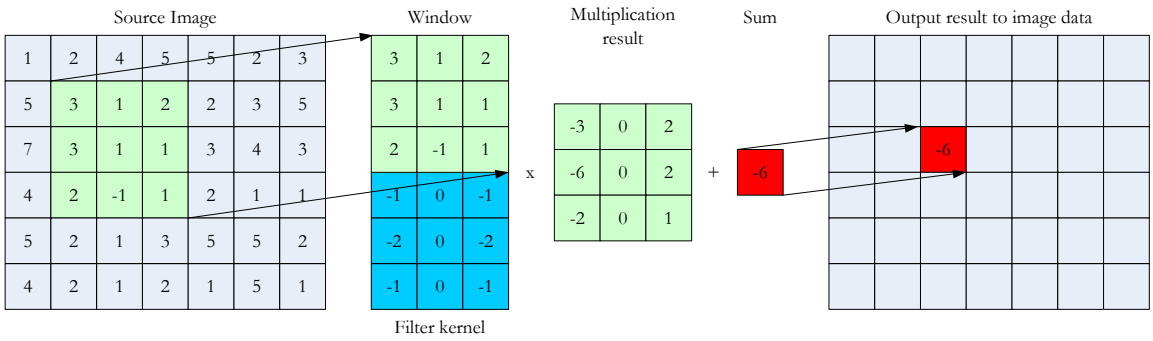

Figure 2: The basic convolution method.

Separable Filters

Generally, a two-dimensional convolution filter requires *n*m* multiplications for each output pixel, where *n* and *m* are the width and height of the filter kernel. Separable filters are a special type of filter that can be expressed as the composition of two one-dimensional filters, one on the rows on the image, and one on the columns. A separable filter can be divided into two consecutive one-dimensional convolution operations on the data, and therefore requires only *n* + *m* multiplications for each output pixel. For example, the filter in the diagram of Figure 2 is a separable Sobel [2] edge detection filter.

Applying $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ to the data is the same as applying $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ followed by $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$.

Separable filters have the benefit of offering more flexibility in the implementation, and in addition reducing the arithmetic complexity and bandwidth usage of the computation for each data point.

The `convolutionSeparable` code sample in the NVIDIA CUDA SDK uses a separable Gaussian [3] blur filter.

A Naïve Implementation

The simplest approach to implement convolution in CUDA is to load a block of the image into a shared memory array, do a point-wise multiplication of a filter-size portion of the block, and then write this sum into the output image in device memory. Each thread block processes one block in the image. Each thread generates a single output pixel. An illustration of this is shown in Figure 3.

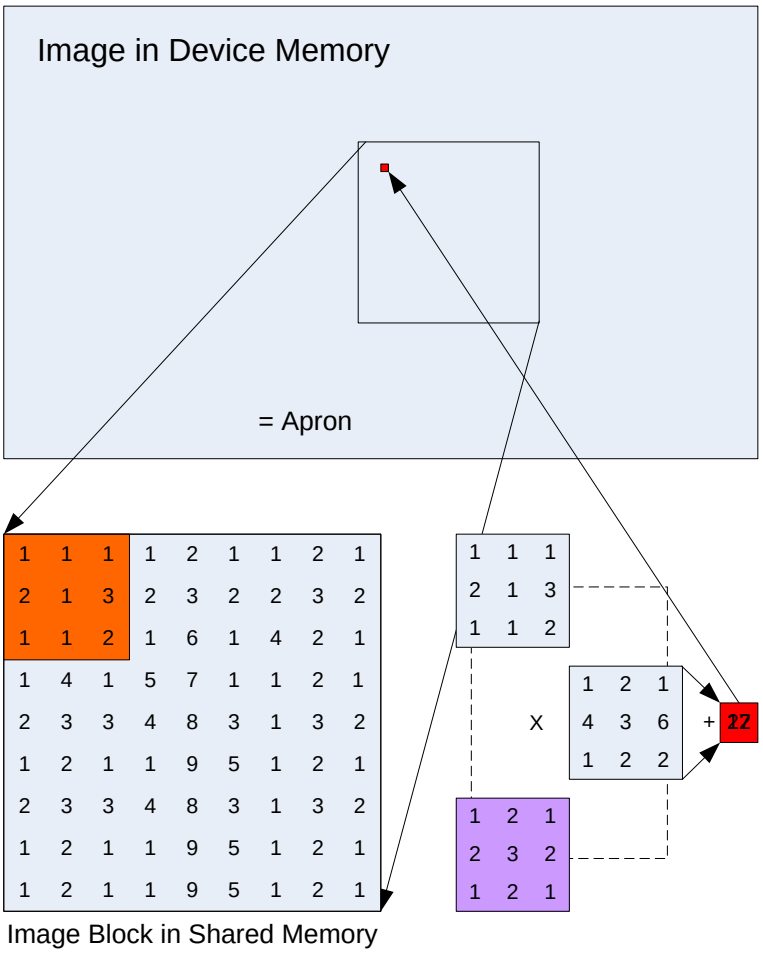

Figure 3: A naïve convolution algorithm. A block of pixels from the image is loaded into an array in shared memory. To process and compute an output pixel (red), a region of the input image (orange) is multiplied element-wise with the filter kernel (purple) and then the results are summed. The resulting output pixel is then written back into the image.

Shared Memory and the Apron

The algorithm itself is somewhat complex. For any reasonable filter kernel size, the pixels at the edge of the shared memory array will depend on pixels not in shared memory. Around the image block within a thread block, there is an *apron* of pixels of the width of the kernel radius that is required in order to filter the image block. Thus, each thread block must load into shared memory the pixels to be filtered and the apron pixels. This is shown in Figure 4. Note: The apron of one block overlaps with adjacent blocks. The aprons of the blocks on the edges of the image extend outside the image – these pixels can either be clamped to the color of pixels at the image edge, or they can be set to zero.

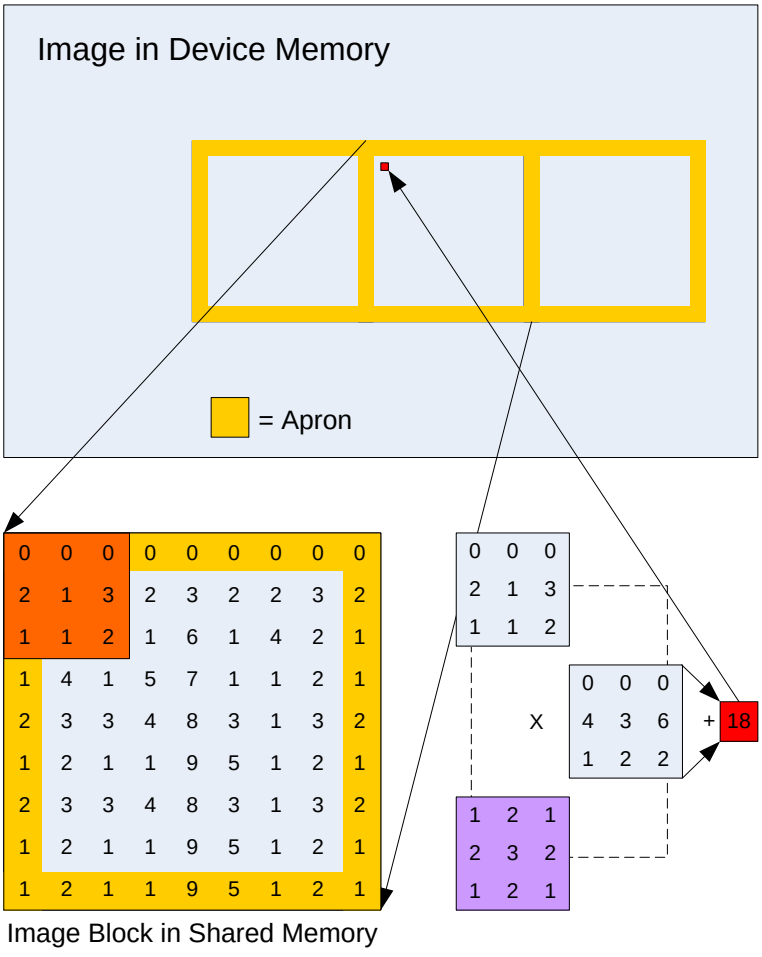

Figure 4: Modification of the naive algorithm of Figure 3 to include the image block apron region.

Avoid Idle Threads

If one thread is used for each pixel loaded into shared memory, then the threads loading the apron pixels will be idle during the filter computation. As the radius of the filter increases, the percentage of idle threads increases. This wastes much of the available parallelism, and with the limited amount of shared memory available, the waste for large radius kernels can be quite high.

As an example, consider a 16x16 image block and a kernel of radius 16. This only allows one active block per multiprocessor. Assuming 4 bytes per pixel, a block will use 9216 bytes. This is more than half of the available 16KB shared memory per multiprocessor on the G80 GPU. In this case, only 1/9 of the threads will be active after the load stage shown in Figure 5.

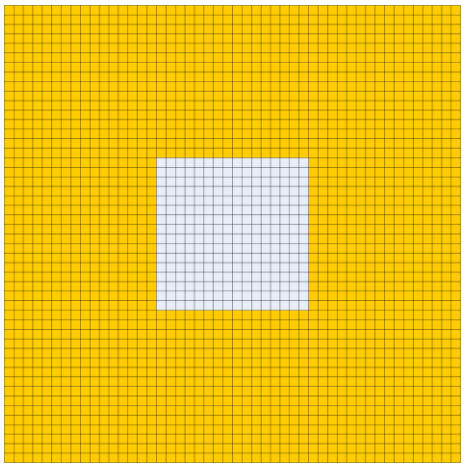

Figure 5: If the radius of the filter kernel is large relative to the image block size, there will be many idle threads during filter computation.

We can reduce the number of idle threads by reducing the total number of threads per block and also using each thread to load multiple pixels into shared memory. For example, if we use a vertical column of threads with the same width as the image block we are processing, we have a 48x48 pixel region in shared memory, but only 16x48 threads. We read the data in three columns, as shown in Figure 6.

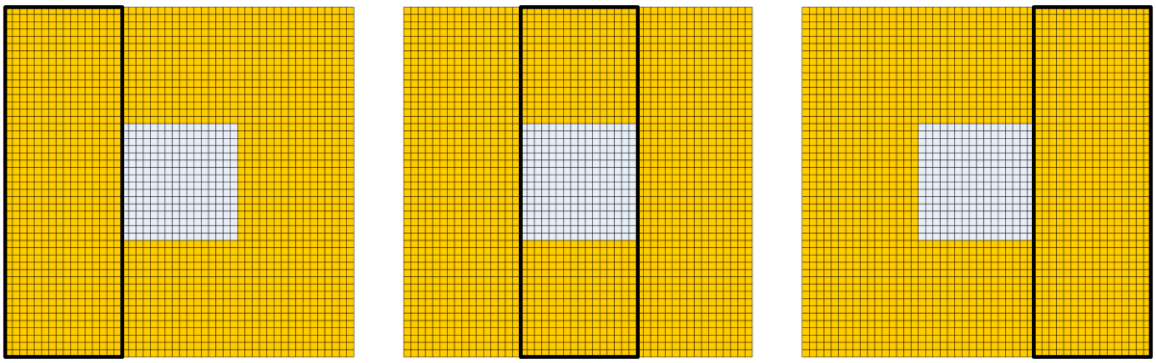

Figure 6: Reduce idle threads by loading multiple pixels per thread.

During filter computation, one third of the threads are active rather than one ninth. Because the load stage is bandwidth limited, performance should not suffer. This can be taken one step further by reducing the thread block to 16x16 threads and dividing the image block into 9 squares of pixels. This ensures that all threads are active during the computation stage. Note that the number of threads in a block must be a multiple of the warp size (32 threads on G80 GPUs) for optimal efficiency. If the apron is not as wide as the thread block, we will end up with some threads that are inactive during the loading stage, as shown in Figure 7. Fortunately, device memory loads will be coalesced as long as the first thread of each half-warp is properly aligned. This is also the case even if some threads conditionally skip their loads (See the CUDA Programming Guide for more information on coalesced device memory accesses).

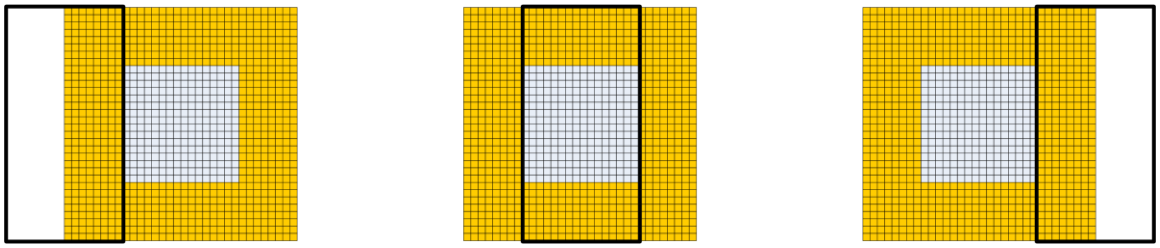

Figure 7: When the apron is narrower than the thread block, some threads are inactive during loading.

These changes may not consistently improve performance because the extra complexity can outweigh the advantages. In the next section we take advantage of separable filters to further reduce wasted or idle threads.

Separable Filters Increase Efficiency

In addition to reducing the number of idle threads through tiling, we can reduce the number of unnecessary data loads by dividing the processing into two passes. One pass is performed for each of the two dimensions in a separable image filter. In the last technique of the previous section, a 48x48 region includes a 16-pixel apron. Each pixel within the apron-width area on the outside of the image will be loaded 9 times because of the overlap between neighboring blocks!

If we separate the computation into horizontal (row) and vertical (column) passes, with a write to global memory between each pass, each pixel will be loaded six times at most. With a small tile (16x16), this does not gain anything. The real benefit is seen because it is no longer necessary to load the top and bottom apron regions (for the horizontal pass) of pixels. This allows more pixels to be loaded for processing in each thread block. We are limited by thread block size rather than shared memory size. To achieve higher efficiency, each thread must process more than one pixel. We can increase the width of the image block processed by a thread block more flexibly using this separable approach, as shown in Figure 8. This leads to significant performance improvements.

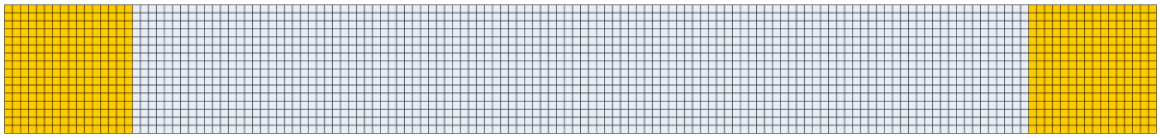

Figure 8: A separable filter allows multiple pixels to be processed for each thread, achieving higher efficiency.

Optimizing for Memory Coalescence

Bandwidth to off-chip ("device") DRAM is much higher than on a host CPU memory. However, in order to achieve high memory throughput, the GPU attempts to *coalesce* accesses from multiple threads into a single memory transaction. If all threads within a warp (32 threads) simultaneously read consecutive words then single large read of the 32 values can be performed at optimum speed. If 32 random addresses are read, then only a fraction of the total DRAM bandwidth can be achieved, and performance will be much lower.

Base read/write addresses of the warps of 32 threads also must meet half-warp alignment requirement in order to be coalesced. If four-byte values are read, then the base address for the warp must be 64-byte aligned, and threads within the warp must read sequential 4-byte addresses. If the dataset with apron does not align in this way, then we must fix it so that it does.

The approach used in the row filter is to have additional threads on the leading edge of the processing tile, in order to make threadIdx.x == 0 always reading properly aligned address and thus to meet global memory alignment constraints for all warps. This may seem like a waste of threads, but it is of little importance when the data block, processed by a single thread block is large enough, which decreases the ratio of apron pixels to output pixels.

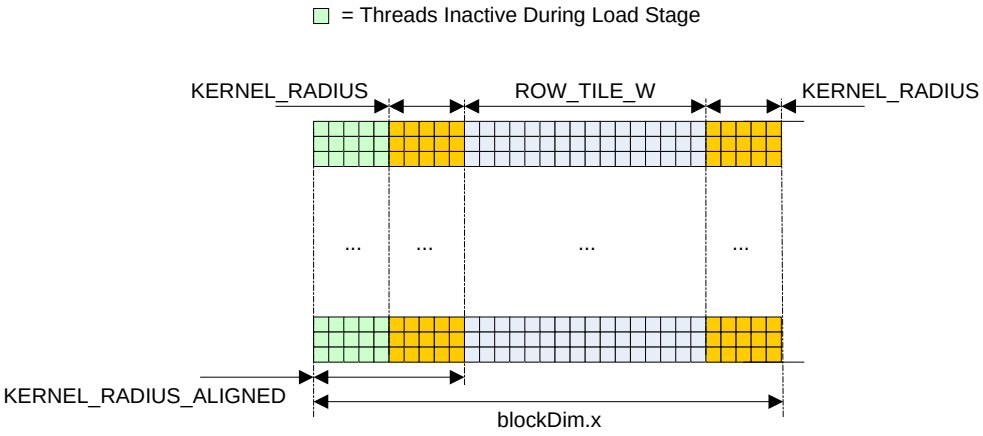

Figure 9: Padding thread block with inactive threads to achieve the alignment required for coalesced loads in the row filtering pass.

During the column convolution pass, the apron does not affect the coalescing alignment, as long as the image tile width is a multiple of 16.

Each image convolution pass in both row and column pass is separated into two sub stages within corresponding CUDA kernels. The first stage loads the data from global memory into shared memory, and the second stage performs the filtering and writes the results back to global memory. We mustn't forget about the cases when row or column

processing tile becomes clamped by image borders, and initialize clamped shared memory array indices with correct values. Indices not lying within input image borders are usually initialized either with zeroes or with values, corresponding to clamped image coordinates. In this sample we opt for the former.

In between the two stages there is a __syncthreads( ) call to ensure that all threads have written to shared memory before any processing begins. This is necessary because threads are dependent on data loaded by other threads.

The Row Filter

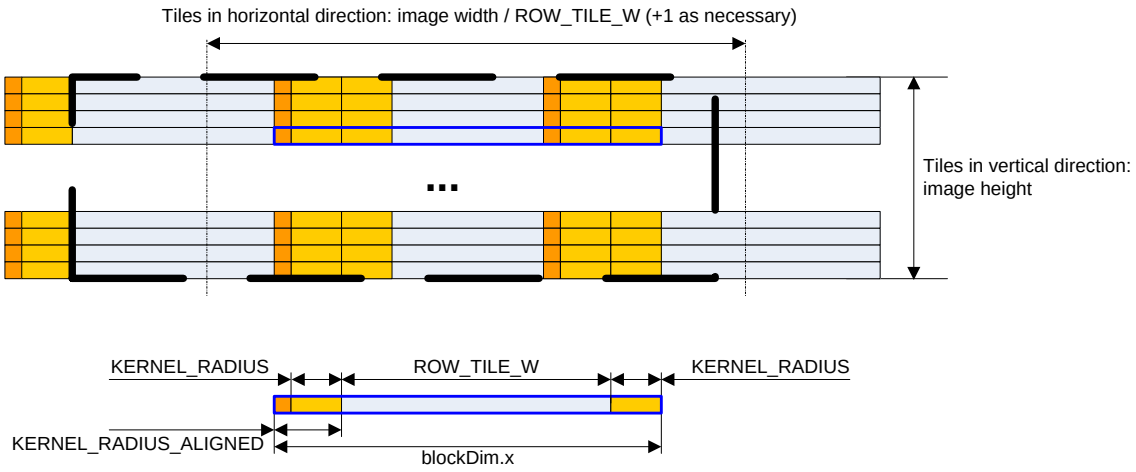

Figure 10: Layout of the thread block grid for the row filtering pass.

For both the loading and processing stages each active thread loads/outputs one pixel. In the computation stage each thread loops over a width of twice the filter radius plus 1, multiplying each pixel by the corresponding filter coefficient stored in constant memory. Each thread in a half-warp accesses the same constant address and hence there is no penalty due to constant memory bank conflicts. Also, consecutive threads always access consecutive shared memory addresses so no shared memory bank conflicts occur as well.

The Column Filter

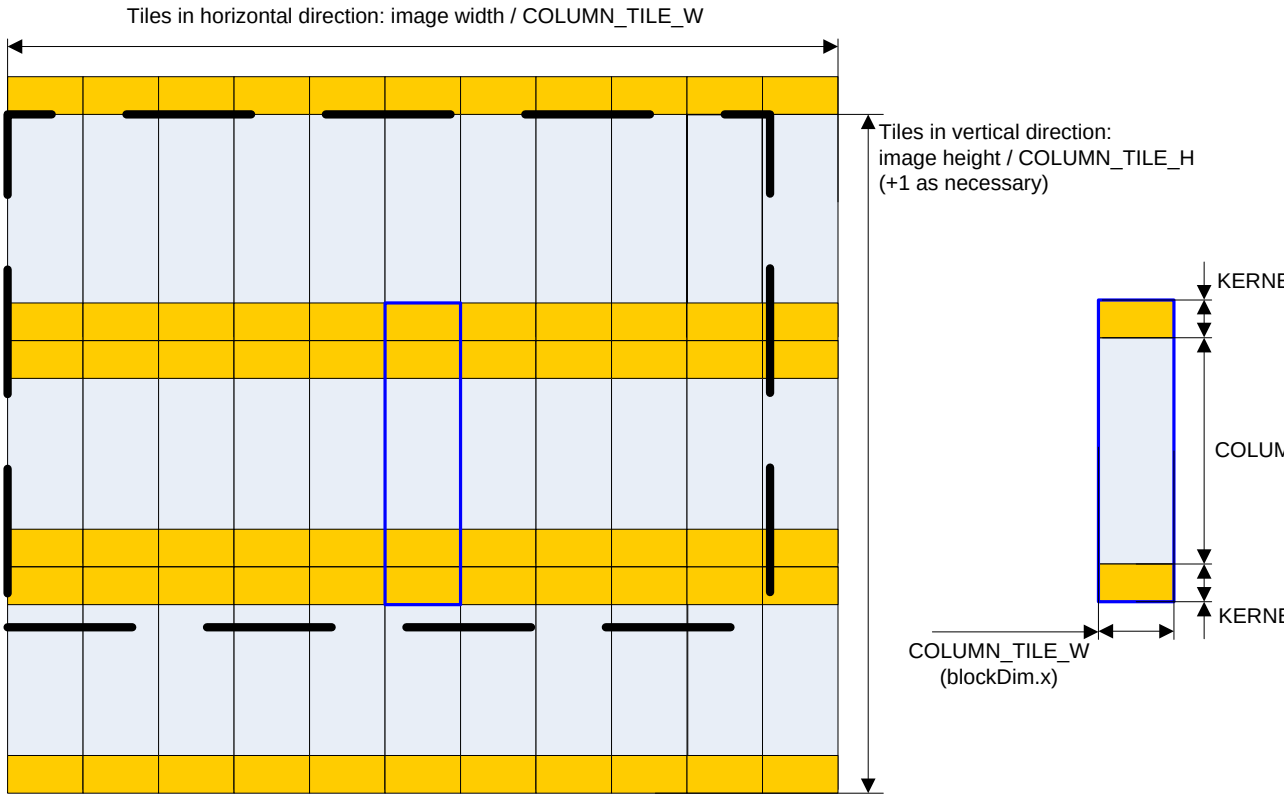

Figure 12: Layout of the thread block grid for the column filtering pass.

The column filter pass operates much like the row filter pass. The major difference is that thread IDs increase across the filter region rather than along it. As in the row filter pass, threads in a single half-warp always access different shared memory banks, but the calculation of the next/previous addresses involves increment/decrement by COLUMN_TILE_W, rather than simply 1. In the column filter pass we do not have inactive "coalescing alignment" threads during the load stage, because we assume that the tile width is a multiple of the coalesced read size. In order to decrease the ratio of apron to output pixels we want image tile to be as tall as possible, so to have reasonable shared memory utilization we shoot for as thin image tiles as possible: 16 columns.

Implementations Details

Source code is divided among 3 `source files (*.cu, *.cpp)`:

- **`convolutionSeparable.cu`**: main program, allocating host and device memory, generating input data, issuing CUDA computations and validating the obtained results.

- **convolutionSeparable_kernel.cu**: CUDA convolution kernels.
- **convolutionSeparable_gold.cpp**: reference CPU separable convolution implementation, which is used for CUDA results validation.

The following are the steps, performed `convolutionSeparable` by main() function.

1) For testing purposes input data is generated using libc rand() function.

2) Gaussian convolution kernel is calculated and copied to CUDA constant array. The Gaussian is a symmetric function, so the row and column filters are identical.

3) CUDA computation grid is configured for requested image and filter parameters.

4) Row and column filters are applied onto the input data.

5) The resulting image is copied back to the CPU and checked for correctness.

Unrolling Loops

```
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
    sum += data[smemPos + k] * d_Kernel[KERNEL_RADIUS - k];
```

Listing 1: Innermost convolution loop for row filter.

Since the innermost processing loop of both row and column filter performs very few computations per iteration, the loop/branching overhead is very big, so in order to improve performance we unroll the loop, which gains 2+ performance improvement for `convolutionSeparable`.

```
#define CONVOLUTION_ROW1(sum, data, smemPos) {sum = \
    data[smemPos - 1] * d_Kernel[2] + \
    data[smemPos + 0] * d_Kernel[1] + \
    data[smemPos + 1] * d_Kernel[0];  \
}
```

Listing 2: Row innermost loop unrolling macro for kernel radius of 1.

Unrolling macro for column filter looks similar.

Running the Sample

The SDK sample can be parameterized at compile time using a set of options available in the main `convolutionSeparable.cu`. These parameters affect the performance of the code and hence should be tweaked to reach the optimum:

UNROLL_INNER: Enable innermost loop unrolling. This greatly improves performance, at the cost of principal filter size fixing at compile time.

WARMUP: Perform a warm-up computation outside of the timed computation to remove the CUDA startup overhead from performance measurements.

Performance

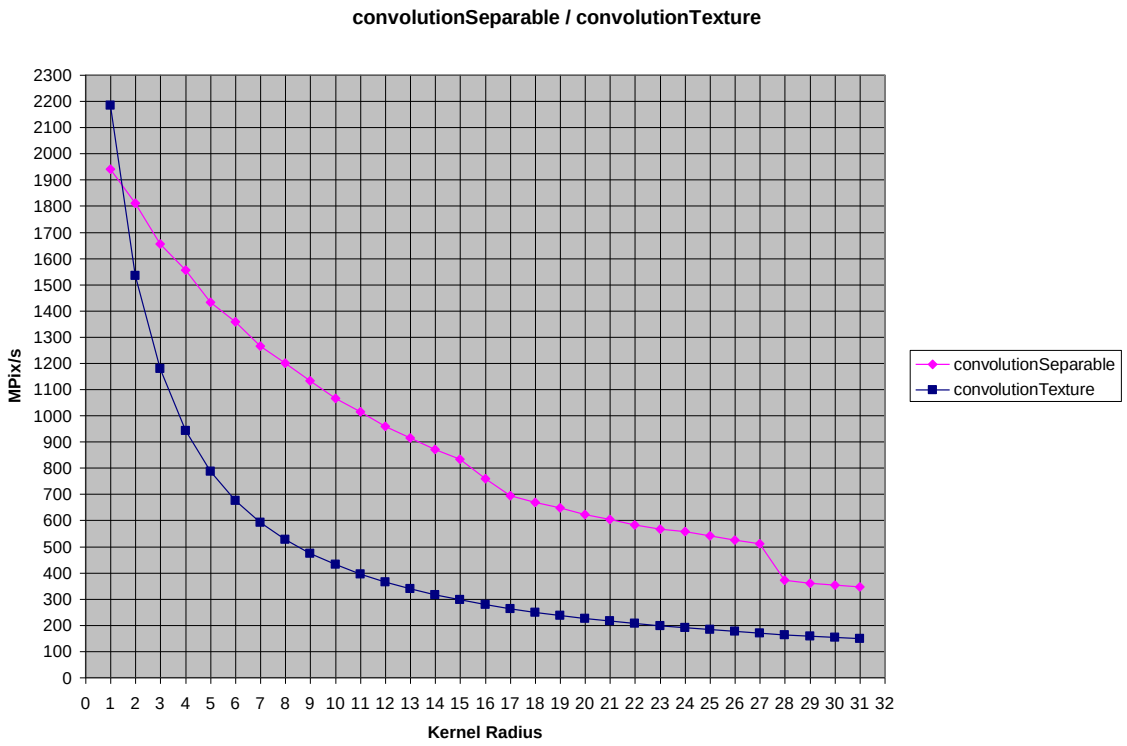

Figure 13. Combined row + column filter performance of convolutionSeparable and convolutionTexture.

Figure 13 shows the execution rate we obtain with convolutionSeparable and the convolutionTexture SDK sample.

The convolutionTexture SDK sample implements the same algorithm as convolutionSeparable, but without using the shared memory at all. Instead, it uses textures in exactly the same way an OpenGL-based implementation would do. This approach is more straightforward and leads to simpler code, but as illustrated by Figure 13, it performs more

than two times slower than the shared-memory approach on a significant range of kernel radius values.

Conclusion

CUDA enables great flexibility in the implementation of image processing algorithms. The `convolutionSeparable` code sample demonstrates a number of performance techniques and tradeoffs for separable image filtering. The image convolution algorithm is a good match to the banked structure of shared memory and the coalescing requirements for high device memory throughput. The efficiency of the GPU implementation is limited by the amount of available shared memory, much like the efficiency of the CPU implementation is limited by cache size and policy. Finally, the technique described in this paper is a good illustration of the advantage provided by shared memory since it outperforms more than twice a purely texture-based implementation of the same algorithm running on the same hardware.

Bibliography

1. Wolfram Mathworld. "Convolution" http://mathworld.wolfram.com/Convolution.html
2. Generation5. "An Introduction to Edge Detection: The Sobel Edge Detector" http://www.generation5.org/content/2002/im01.asp
3. Wolfram Mathworld. "Normal Distribution" http://mathworld.wolfram.com/NormalDistribution.html

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