Non-Separable 2D, 3D and 4D Filtering with CUDA

Anders Eklund and Paul Dufort

1.1 Introduction

Filtering is an important step in many image processing applications such as image denoising (where the goal is to suppress noise, see Figure 1.1), image registration (where the goal is to align two images or volumes, see Figure 1.2) and image segmentation (where the goal is to extract certain parts of an image or volume, see Figure 1.3). In medical imaging, the datasets generated are often three or four dimensional and contain a large number of samples, making filtering a computationally demanding operation. A high resolution magnetic resonance (MR) scan of a human head normally contains on the order of 256 x 256 x 200 voxels (a voxel is the 3D equivalent of a pixel). Functional magnetic resonance imaging (fMRI) is used for studying brain function, and the generated 4D datasets can easily contain 300 volumes over time with 64 x 64 x 30 voxels each. Ultrasound machines are increasingly affordable and can output volume data at 20-30 Hz. Computed tomography (CT) scanners can yield even higher spatial resolution than MR scanners, at the cost of ionizing radiation. A 4D CT dataset of a beating heart can be of the size 512 x 512 x 445 x 20 samples [Eklund et al. 11]. Reducing the amount of radiation in CT leads to higher noise levels, but this can be remedied by applying image denoising algorithms. However, to apply 11 non-separable denoising filters with 11 x 11 x 11 x 11 coefficients to a dataset of size 512 x 512 x 445 x 20, for example, requires approximately 375,000 billion multiply-add operations using a convolution approach. Fortunately, graphics processing units (GPUs) can now easily be used to speed up a large variety of parallel operations [Owens et al. 07].

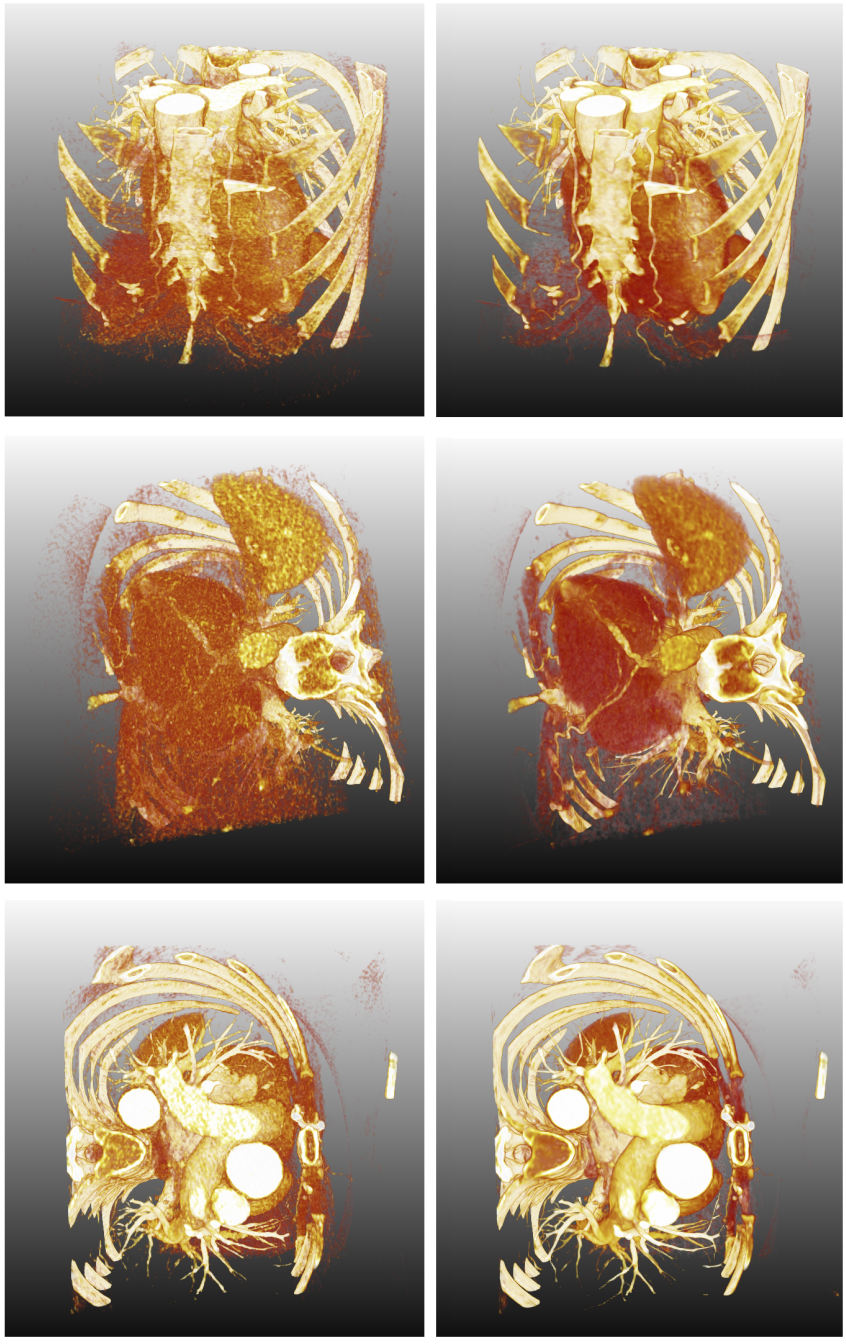

Figure 1.1. Results of denoising a 4D CT dataset, requiring convolution with 11 non-separable 4D filters of size 11 x 11 x 11 x 11. See [Eklund et al. 11] for further information about the denoising algorithm. **Left:** Original data. **Right:** Denoised data.

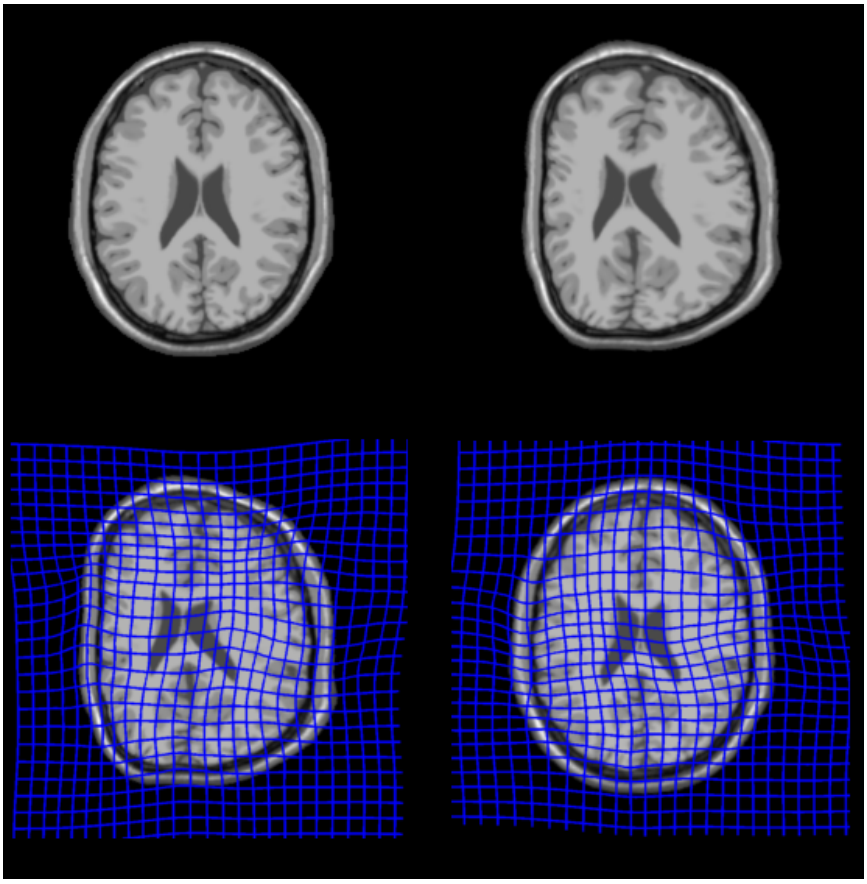

Figure 1.2. An example of image registration where the goal is to align two images or volumes. The registration algorithm used here is called the Morphon; it takes advantage of quadrature filters to detect edges and lines to estimate the displacement between two images or volumes. Quadrature filters are non-separable and can for example be of size 9 x 9 x 9 voxels for volume registration. **Top left:** Original image. **Top right:** A modified version of the original image. The task for the registration algorithm is to align the two images by finding an optimal field deforming one into the other. **Bottom left:** The deformed image with the applied deformation field on top. **Bottom right:** The registration algorithm has successfully aligned the two images, and the computed deformation field is presented on top of the image (this deformation field is the inverse of the applied deformation field). This example was kindly provided by Daniel Forsberg.

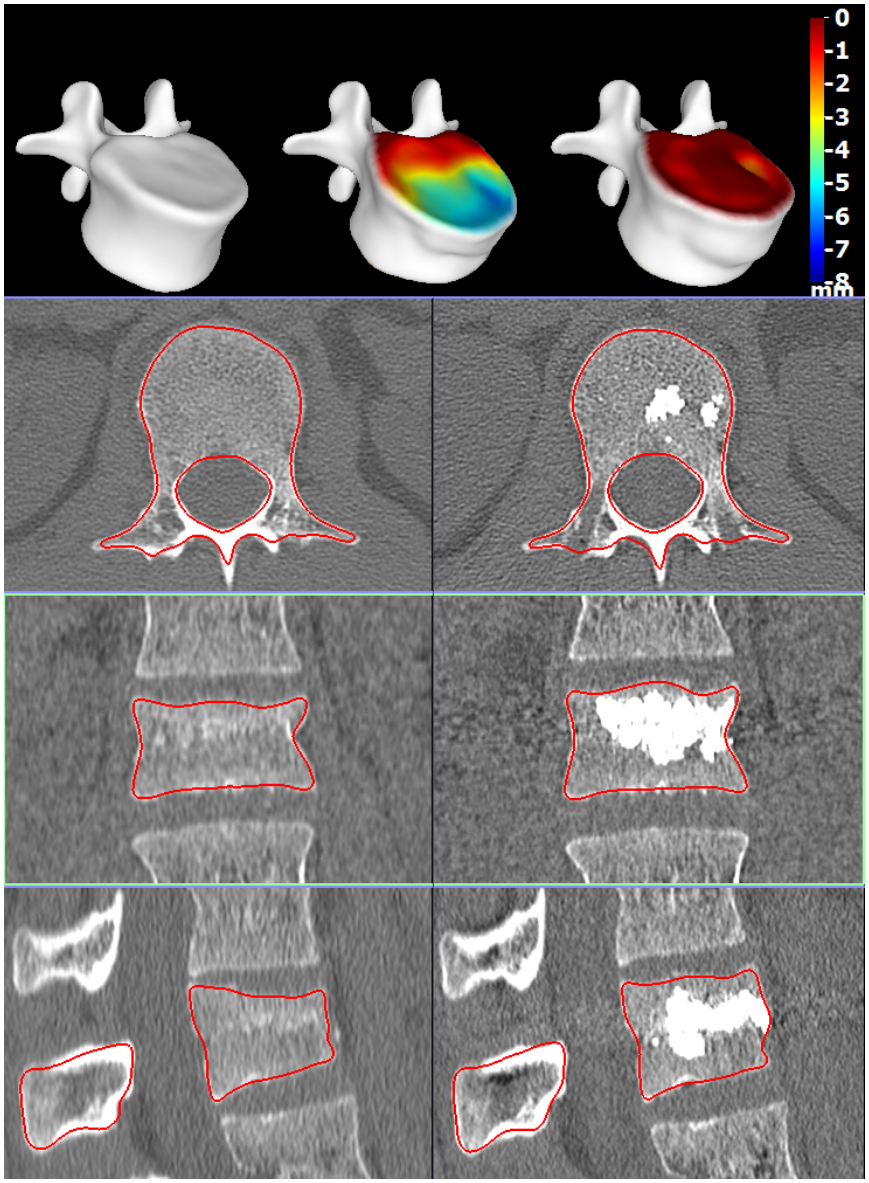

Figure 1.3. Three-dimensional segmentation of a fractured vertebra before and after a surgical procedure to restore its height. The bottom three rows show before (left) and after (right) images overlaid with the contours of the 3D segmentations. The top row shows 3D renderings of the segmentations, colour-coded to indicate the change in height pre- and post-surgery. The reference (leftmost) 3D vertebra is a pre-trauma reconstruction - a prediction of the vertebra's shape before it was fractured. Filtering is used here to detect the edges of the vertebra.

1.2 Non-separable filters

Filtering can be divided into separable and non-separable variants. Popular separable filters are Gaussian filters (used for smoothing/blurring to reduce noise and details) and Sobel filters (used for detection of edges). Filter kernels for Gaussian (G) smoothing and edge detection along x and y using Sobel (S) filters can be written as

$$G = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /16 \,,\; S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} ,\; S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} . \quad (1.1)$$

Separability means that these filters can be decomposed as one 1D filter along x and one 1D filter along y. The Sobel filter used for edge detection along x can for example be decomposed as

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} . \quad (1.2)$$

A separable filter of size 9 x 9 requires 18 multiplications per pixel and can be applied in two passes. One pass performs convolution along the rows, and the other pass performs convolution along the columns. A non-separable filter of size 9 x 9, on the other hand, requires 81 multiplications per pixel and is applied in a single pass.

While separable filters are less computationally demanding, there are a number of image processing operations that can only be performed using non-separable filters. The best-known non-separable filter is perhaps the Laplace (L) filter, which can be used for edge detection. In contrast to Gaussian and Sobel filters, it cannot be decomposed into two 1D filters. Laplace filters of size 3 x 3 and 5 x 5 can for example be written as

$$L_{3x3} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} ,\; L_{5x5} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -24 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} . \quad (1.3)$$

The quadrature filter is another popular non-separable filter, which is complex valued in the spatial domain. The real part of the filter is a line detector and the imaginary part is an edge detector. A one-dimensional quadrature filter is given in Figure 1.4, but quadrature filters of any dimension can be created. The name quadrature comes from electronics and describes the relation between two signals having the same frequency and a phase difference of 90 degrees. An edge detector is an odd function similar

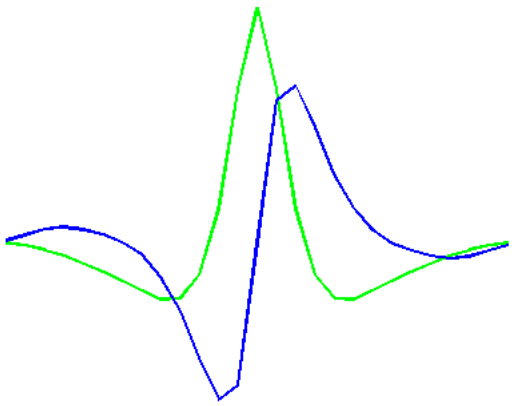

Figure 1.4. A one-dimensional quadrature filter; the real part (green) is a line detector (a cosine modulated with a Gaussian window) and the imaginary part (blue) is an edge detector (a sine modulated with a Gaussian window).

to a sine wave, while a line detector is an even function similar to a cosine wave. A sine and a cosine of the same frequency always differ in phase by 90 degrees and a filter that can be described as one sine wave and one cosine wave is therefore called a quadrature filter. The interested reader is referred to [Granlund and Knutsson 95, Knutsson et al. 99] for further information about quadrature filters and filter design. Quadrature filters can be applied for a wide range of applications, such as image registration [Knutsson and Andersson 05, Eklund et al. 10, Forsberg et al. 11], image segmentation [Läthen et al. 10] and image denoising [Knutsson et al. 83, Knutsson 89, Granlund and Knutsson 95, Westin et al. 01]. Quadrature filters are very similar to Gabor filters [Granlund 78, Jain and Farrokhnia 91], which are also complex valued in the spatial domain.

For most algorithms using Gabor or quadrature filters, several filters are applied along different directions. For example, estimation of a structure tensor [Knutsson 89, Knutsson et al. 11] in 3D requires filtering with at least 6 complex valued quadrature filters, i.e. a total of 12 filters (6 line detectors and 6 edge detectors). The Morphon [Knutsson and Andersson 05] is an image registration algorithm that uses quadrature filters to estimate the displacement between two images or volumes. To improve the registration, estimation of a structure tensor is performed in each iteration, thus requiring an efficient implementation of filtering. Monomial filters [Knutsson et al. 11, Eklund et al. 11] are a good example of filters appropriate for non-separable filtering in 4D.

1.3 Convolution vs FFT

Images and filters can be viewed directly in the image domain (also called the spatial domain) or in the frequency domain (also denoted Fourier space) after the application of a Fourier transform. Filtering can be performed as a convolution in the spatial domain or as a multiplication in the frequency domain, according to the convolution theorem

$$F[s * f] = F[s] \cdot F[f] \tag{1.4}$$

where $F[\,]$ denotes the Fourier transform, s denotes the signal (image), f denotes the filter, $*$ denotes convolution and $\cdot$ denotes pointwise multiplication. For large non-separable filters, filtering performed as a multiplication in the frequency domain can often be faster than convolution in the spatial domain. The transformation to the frequency domain is normally performed using the fast Fourier transform (FFT), for which very optimized implementations exist. However, FFT-based approaches for 4D data require huge amounts of memory and current GPUs have only 1-6 GB of global memory. Spatial approaches can therefore be advantageous for large datasets. Bilateral filtering [Tomasi and Manduchi 98], which is a method for image denoising, requires that a range function is evaluated for each filter coefficient during the convolution. Such an operation is hard to do with a FFT approach, since bilateral filtering in its original form is a non-linear operation and the Fourier transform is linear. For optimal performance, FFTs often also require that the dimensions of the data are a power of 2. Additionally, there is no direct support for 4D FFTs in the Nvidia CUFFT library. Instead, one has to apply two batches of 2D FFTs and change the order of the data between these (since the 2D FFTs are applied along the two first dimensions).

1.4 Previous work

A substantial body of work has addressed the acceleration of filtering using GPUs. Two of the first examples are the work by [Rost 96], who used OpenGL for 2D convolution and [Hopf and Ertl 99] who used a GPU for separable 3D convolution. For game programming, filtering can for example be used for texture animation [James 01]. A more recent example is a white paper from Nvidia [Podlozhnyuk 07] discussing separable 2D convolution. See our recent review about GPUs in medical imaging for a more extensive overview of GPU based filtering [Eklund et al. 13]. GPU implementations of *non-separable* filtering in 3D, and especially 4D, are less common. For example, the NPP (Nvidia performance primitives) library contains functions for image processing, but for convolution only supports

2D data and filters stored as integers. The CUDA SDK contains two examples of separable 2D convolution, one example of FFT based filtering in 2D and a single example of separable 3D convolution.

The main purpose of this chapter is therefore to present optimized solutions for non-separable 2D, 3D and 4D convolution with the CUDA programming language, using floats and the fast shared memory. Our code has already been successfully applied to a number of applications [Eklund et al. 10, Forsberg et al. 11, Eklund et al. 11, Eklund et al. 12]. The implementations presented here have been made with CUDA 5.0 and are optimized for the Nvidia GTX 680 graphics card. Readers are assumed to be familiar with CUDA programming, and may avail themselves of the many books available on this topic if not (e.g. [Sanders and Kandrot 11]). All the code for this chapter is available under GNU GPL 3 at

`https://github.com/wanderine/NonSeparableFilteringCUDA`

1.5 Non-separable 2D convolution

Two-dimensional convolution between a signal s and a filter f can be written for position $[x, y]$ as

$$(s * f)[x, y] = \sum_{f_x=-N/2}^{f_x=N/2} \sum_{f_y=-N/2}^{f_y=N/2} s[x - f_x, y - f_y] \cdot f[f_x, f_y], \qquad (1.5)$$

where N is the filter size. The most important aspect for a GPU implementation is that the convolution can be done independently for each pixel. To obtain high performance, it is also important to take advantage of the fact that filter responses for neighbouring pixels are calculated from a largely overlapping set of pixels. We will begin with a CUDA implementation for non-separable 2D convolution that uses texture memory, as the texture memory cache can speedup local reads. Threads needing pixel values already accessed by other threads can thus read the values from the fast cache located at each multiprocessor (MP), rather than from the slow global memory. The filter kernel is put in the constant memory (64 KB) as it is used by all the threads. For Nvidia GPUs the constant memory cache is 8 KB per MP, and 2D filters can thus easily reside in the fast on-chip cache during the whole execution. The device code for texture based 2D convolution is given in Listing 1.1.

The main problem with using texture memory is that such an implementation is limited by the memory bandwidth, rather than by the computational performance. A better idea is to instead take advantage of the shared memory available at each MP, which makes it possible for the threads in a thread block to cooperate very efficiently. Nvidia GPUs from the Fermi

```
__global__ void Convolution_2D_Texture(float* Filter_Response,
int DATA_W, int DATA_H)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;

   if (x >= DATA_W || y >= DATA_H)
        return;

   float sum = 0.0f;
   float y_off = -(FILTER_H - 1)/2 + 0.5f;
   for (int f_y = FILTER_H - 1; f_y >= 0; f_y--)
   {
        float x_off = -(FILTER_W - 1)/2 + 0.5f;
        for (int f_x = FILTER_W - 1; f_x >= 0; f_x--)
        {
             sum += tex2D(texture,x + x_off,y + y_off) *
             c_Filter[f_y][f_x];
             x_off += 1.0f;
        }
        y_off += 1.0f;
    }

    Filter_Response[Get2DIndex(x,y,DATA_W)] = sum;
}
```

Listing 1.1. Non-separable 2D convolution using texture memory: each thread calculates the filter response for one pixel. The filter kernel is stored in cached constant memory and the image is stored in cached texture memory. If the filter size is known at compile time, the inner loop can be unrolled by the compiler. The addition of 0.5 to each coordinate is due to the fact that the original pixel values for textures are actually stored between the integer coordinates.

and Kepler architectures have 48 KB of shared memory per MP. If one only considers the number of valid filter responses generated per thread block, the optimal solution is to use all the shared memory for a single thread block, since this would waste a minimum of memory on the halo of invalid filter responses at the outer edges. According to the CUDA programming guide, GPUs with compute capability 3.0 (e.g. the Nvidia GTX 680) can maximally handle 1024 threads per thread block and 2048 concurrent threads per MP. Using all the shared memory for one thread block would therefore lead to 50% of the possible computational performance, as only 1024 threads can be used in one thread block. Full occupancy can be achieved by instead dividing the 48 KB of shared memory into two thread blocks. For floating point convolution, 96 x 64 pixel values can be fitted into 24 KB of shared memory. The 1024 threads per thread block are arranged as 32 threads along x and 32 threads along y, to achieve coalesced

reads from global memory and to fit the number of banks in shared memory (32). Each thread starts by reading 6 values from global memory into shared memory (96 x 64 / 1024 = 6). For a maximum filter size of 17 x 17 pixels, 80 x 48 valid filter responses can then be calculated from the 96 x 64 values in shared memory, since a halo of size 8 on all sides is required. All threads start by first calculating 2 filter responses, yielding 64 x 32 values. Half of the threads then calculate an additional 3 filter responses, giving a total of 48 x 32 filter responses. Finally, a quarter of the threads are used to calculate the filter responses for the last 16 x 16 pixels. The division of the 80 x 48 values into six blocks is illustrated in Figure 1.5. The first part of the code for non-separable 2D convolution using shared memory is given in Listing 1.2 and the second part is given in Listing 1.3. The device function that performs the 2D convolution is very similar to the kernel for texture based convolution, interested readers are therefore referred to the repository.

If more than one filter is to be applied, e.g. four complex valued quadrature filters oriented along 0, 45, 90 and 135 degrees, all the filter responses can be calculated very efficiently by simply performing several multiplications and additions each time a pixel value has been loaded from shared memory to a register. This results in a better ratio between the number of memory accesses and floating point operations. By reducing the maximum filter size to 9 x 9, the number of valid filter responses increases to 88 x 56 since the halo size shrinks to 4. This will also result in a higher occupancy during the convolution. For the first case yielding 80 x 48 valid filter responses, the mean occupancy for the six blocks is $(32 \cdot 32 \cdot 2 + 32 \cdot 32 \cdot 2 + 32 \cdot 16 \cdot 2 + 32 \cdot 16 \cdot 2 + 16 \cdot 32 \cdot 2 + 16 \cdot 16 \cdot 2)/(6 \cdot 2048) = 62.5\%$ and for the second case yielding 88 x 56 valid filter responses the mean occupancy increases to $(32 \cdot 32 \cdot 2 + 32 \cdot 32 \cdot 2 + 32 \cdot 24 \cdot 2 + 32 \cdot 24 \cdot 2 + 24 \cdot 32 \cdot 2 + 24 \cdot 24 \cdot 2)/(6 \cdot 2048) = 80.2\%$.

The required number of thread blocks in the x- and y-direction are for the shared memory implementation *not* calculated by dividing the image width and height with the number of threads in each direction (32). The width and height should instead be divided by the number of valid filter responses generated in each direction, since each thread block generates more than one valid filter response per thread. The calculation of the x- and y-indices inside the kernel also needs to be changed from the conventional

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

to

```
int x = blockIdx.x * VALID_RESPONSES_X + threadIdx.x;
int y = blockIdx.y * VALID_RESPONSES_Y + threadIdx.y;
```

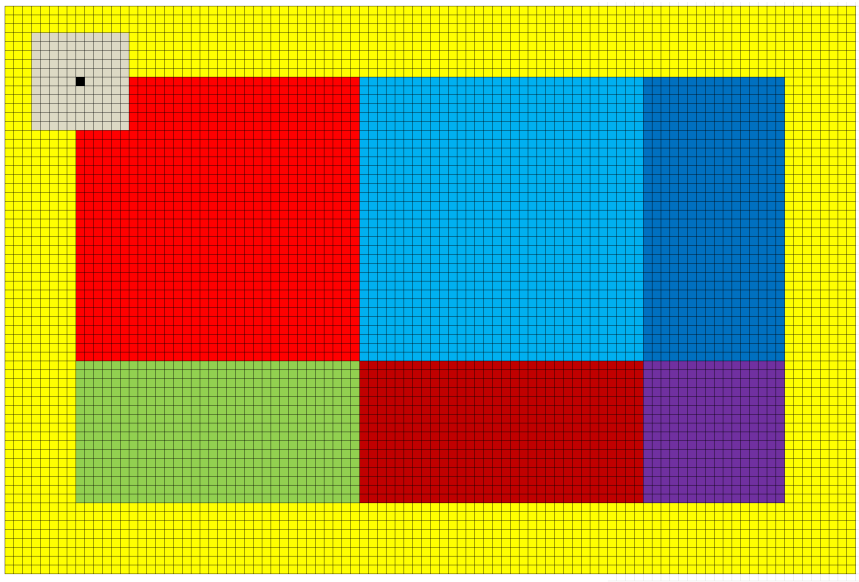

Figure 1.5. The grid represents 96 x 64 pixels in shared memory. As 32 x 32 threads are used per thread block, each thread needs to read 6 values from global memory into shared memory. The gray pixels represent the filter kernel and the black pixel represents where the current filter response is saved. A yellow halo needs to be loaded into shared memory to be able to calculate all the filter responses. In this case 80 x 48 valid filter responses are calculated, making it possible to apply at most a filter of size 17 x 17. The 80 x 48 filter responses are calculated as 6 runs, the first 2 consisting of 32 x 32 pixels (marked light red and light blue). Half of the threads calculate 3 additional filter responses in blocks of 32 x 16 or 16 x 32 pixels (marked green, dark blue and dark red). A quarter of the threads calculates the filter response for a last block of 16 x 16 pixels (marked purple). If the halo is reduced from 8 to 4 pixels, 88 x 56 valid filter responses can instead be calculated as two 32 x 32 blocks, one 24 x 32 block, two 32 x 24 blocks and one 24 x 24 block. In addition to increasing the number of valid filter responses, such an implementation will also increase the mean occupancy during convolution from 62.5% to 80.2%. The only drawback is that the largest filter that can be applied drops from 17 x 17 to 9 x 9.

```
#define HALO 8

__global__ void Convolution_2D_Shared(float* Filter_Response,
float* Image, int DATA_W, int DATA_H)
{
   int x = blockIdx.x * VALID_RESPONSES_X + threadIdx.x;
   int y = blockIdx.y * VALID_RESPONSES_Y + threadIdx.y;

   __shared__ float s_Image[64][96]; // y, x

   // Reset shared memory
   s_Image[threadIdx.y][threadIdx.x]      = 0.0f;
   s_Image[threadIdx.y][threadIdx.x + 32] = 0.0f;
   s_Image[threadIdx.y][threadIdx.x + 64] = 0.0f;
   s_Image[threadIdx.y + 32][threadIdx.x]      = 0.0f;
   s_Image[threadIdx.y + 32][threadIdx.x + 32] = 0.0f;
   s_Image[threadIdx.y + 32][threadIdx.x + 64] = 0.0f;

   // Read data into shared memory

   if ( ((x-HALO) >= 0) && ((x-HALO) < DATA_W)
     && ((y-HALO) >= 0) && ((y-HALO) < DATA_H) )
       s_Image[threadIdx.y][threadIdx.x] =
       Image[Get2DIndex(x-HALO,y-HALO,DATA_W)];

   if ( ((x+32-HALO) < DATA_W)
     && ((y-HALO) >= 0) && ((y-HALO) < DATA_H) )
       s_Image[threadIdx.y][threadIdx.x + 32] =
       Image[Get2DIndex(x+32-HALO,y-HALO,DATA_W)];

   if ( ((x+64-HALO) < DATA_W)
     && ((y-HALO) >= 0) && ((y-HALO) < DATA_H) )
       s_Image[threadIdx.y][threadIdx.x + 64] =
       Image[Get2DIndex(x+64-HALO,y-HALO,DATA_W)];

   if ( ((x-HALO) >= 0)
     && ((x-HALO) < DATA_W) && ((y+32-HALO) < DATA_H) )
       s_Image[threadIdx.y + 32][threadIdx.x] =
       Image[Get2DIndex(x-HALO,y+32-HALO,DATA_W)];

   if ( ((x+32-HALO) < DATA_W) && ((y+32-HALO) < DATA_H) )
      s_Image[threadIdx.y + 32][threadIdx.x + 32] =
      Image[Get2DIndex(x+32-HALO,y+32-HALO, DATA_W)];

   if ( ((x+64-HALO) < DATA_W) && ((y+32-HALO) < DATA_H) )
      s_Image[threadIdx.y + 32][threadIdx.x + 64] =
      Image[Get2DIndex(x+64-HALO,y+32-HALO,DATA_W)];

   __syncthreads();
```

Listing 1.2. Non-separable 2D convolution using shared memory. This listing represents the first part of the kernel, where data is loaded into shared memory. Each thread block consists of 32 x 32 threads, such that each thread has to read 6 values into shared memory (storing 96 x 64 values). The parameter HALO can be changed to control the size of the largest filter that can be applied (HALO*2 + 1). Before the actual convolution is started, synchronization of the threads is required to guarantee that all values have been loaded into shared memory.

```
  if ( (x < DATA_W) && (y < DATA_H) )
    Filter_Response[Get2DIndex(x,y,DATA_W)] =
    Conv2D(s_Image, threadIdx.y+HALO, threadIdx.x+HALO);

  if ( ((x + 32) < DATA_W) && (y < DATA_H) )
    Filter_Response[Get2DIndex(x+32,y,DATA_W)] =
    Conv2D(s_Image, threadIdx.y+HALO, threadIdx.x+32+HALO);

  if (threadIdx.x < (32 - HALO*2))
  {
    if ( ((x + 64) < DATA_W) && (y < DATA_H) )
      Filter_Response[Get2DIndex(x+64,y,DATA_W)] =
      Conv2D(s_Image, threadIdx.y+HALO, threadIdx.x+64+HALO);
  }

  if (threadIdx.y < (32 - HALO*2))
  {
    if ( (x < DATA_W) && ((y + 32) < DATA_H) )
      Filter_Response[Get2DIndex(x,y+32,DATA_W)] =
      Conv2D(s_Image, threadIdx.y+32+HALO, threadIdx.x+HALO);
  }

  if (threadIdx.y < (32 - HALO*2))
  {
    if ( ((x + 32) < DATA_W) && ((y + 32) < DATA_H) )
      Filter_Response[Get2DIndex(x+32,y+32,DATA_W)] =
      Conv2D(s_Image, threadIdx.y+32+HALO, threadIdx.x+32+HALO);
  }

  if ( (threadIdx.x < (32 - HALO*2)) &&
       (threadIdx.y < (32 - HALO*2)) )
  {
    if ( ((x + 64) < DATA_W) && ((y + 32) < DATA_H) )
      Filter_Response[Get2DIndex(x+64,y+32,DATA_W)] =
      Conv2D(s_Image, threadIdx.y+32+HALO, threadIdx.x+64+HALO);
  }
}
```

Listing 1.3. Non-separable 2D convolution using shared memory. This listing represents the second part of the kernel where the convolutions are performed, by calling a device function for each block of filter responses. For a filter size of 17 x 17, HALO is 8 and the first two blocks yield filter responses for 32 x 32 pixels, the third block yields 16 x 32 filter responses, the fourth and fifth block yield 32 x 16 filter responses and the sixth block yields the last 16 x 16 filter responses. The parameter HALO can easily be changed to optimize the code for different filter sizes. The code for the device function Conv2D is given in the repository.

1.6 Non-separable 3D convolution

Three-dimensional convolution between a signal s and a filter f for position $[x, y, z]$ is defined as

$$(s*f)[x,y,z] = \sum_{f_x=-N/2}^{f_x=N/2} \sum_{f_y=-N/2}^{f_y=N/2} \sum_{f_z=-N/2}^{f_z=N/2} s[x-f_x, y-f_y, z-f_z] \cdot f[f_x, f_y, f_z]. \tag{1.6}$$

This weighted summation can be easily implemented by using texture memory just as for 2D convolution. The differences between 3D and 2D are that an additional for-loop is added and that a 3D texture is used instead of a 2D texture. The code will therefore not be given here (but is available in the github repository). Using shared memory for non-separable 3D convolution is more difficult, however. A natural extension of the 2D implementation would be, for example, to load 24 x 16 x 16 voxels into shared memory. This would make it possible to calculate 16 x 8 x 8 valid filter responses per thread block for a filter of size 9 x 9 x 9. But the optimal division of the 1024 threads per thread block along x, y, and z is not obvious. One solution is to use 24 x 16 x 2 threads per thread block, giving a total of 1536 threads per MP and 75% occupancy. However, only 16 x 8 x 2 threads per thread block will be active during the actual convolution, giving a low occupancy of 25%. To use 24 threads along x will also result in shared memory bank conflicts, as there are 32 banks. To avoid these conflicts, one can instead load 32 x 16 x 12 values into shared memory and for example use thread blocks of size 32 x 16 x 2. The number of valid filter responses drops to 24 x 8 x 4 = 768 per thread block (compared to 1024 for the first case) while the occupancy during convolution will increase to 37.5%. It is not obvious how the different compromises will affect the total runtime, making it necessary to actually make all the different implementations and test them. We will leave 3D convolution for now and instead move on to 4D convolution.

1.7 Non-separable 4D convolution

There are no 4D textures in the CUDA programming language, so a simple texture implementation is impossible for non-separable 4D convolution. Possible solutions are to use one huge 1D texture or several 2D or 3D textures. One-dimensional textures can speedup reads that are local in the first dimension (e.g. x), but not reads that are local in the other dimensions (y,z,t). A solution involving many 2D or 3D textures would be rather hard to implement, as one cannot use pointers for texture objects. The use of shared memory is even more complicated than for 3D convolution, and here

```
// Loop over slices in filter
for (int zz = FILTER_D - 1; zz >= 0; zz--)
{
  // Copy current filter coefficients to constant memory
  CopyFilterCoefficients(zz);

  // Perform 2D convolution and
  // accumulate the filter responses inside the kernel,
  // launch kernel for several slices simultaneously
  Convolution_2D_Shared<<<dG, dB>>>(d_FR);
}
```

Listing 1.4. Host code for non-separable 3D convolution, by performing non-separable 2D convolution on the GPU and accumulating the filter responses inside the kernel (dG stands for dimGrid, dB stands for dimBlock and FR stands for filter responses). Just as for the non-separable 4D convolution, the 2D convolution is launched for all slices at the same time to increase the occupancy.

it becomes obvious that it is impossible to continue on the same path. The shared memory cannot even store 11 x 11 x 11 x 11 values required for a filter of size 11 x 11 x 11 x 11.

Let us take a step back and think about what a 4D convolution involves. To perform non-separable 4D convolution requires eight for-loops, four to loop through all the filter coefficients and four to loop through all voxels and time points. To do all the for-loops on the GPU is not necessarily optimal. A better way to divide the for-loops is to do four on the CPU and four on the GPU, such that the four loops on the GPU correspond to a non-separable 2D convolution. Thus, four loops on the CPU are used to call the 2D convolution for two image dimensions and two filter dimensions (e.g. z and t). During each call to the 2D convolution kernel, the filter responses are accumulated inside the kernel. Before each 2D convolution is started, the corresponding 2D values of the 4D filter are copied to constant memory.

A small problem remains, for a 4D dataset of size 128 x 128 x 128 x 128 the 2D convolution will be applied to images of size 128 x 128 pixels. If 80 x 48 valid filter responses are calculated per thread block, only 5 thread blocks will be launched. The Nvidia GTX 680 has 8 MPs and each MP can concurrently handle two thread blocks with 1024 threads each. At least 16 thread blocks are thus required to achieve full occupancy. To solve this problem one can launch the 2D convolution for all slices simultaneously, by using three-dimensional thread blocks, to increase the number of thread blocks and thereby the occupancy. This removes one loop on the CPU, such that three loops are taken care of by the CPU and five by the GPU. As some of the slices in the filter response will be invalid due to border effects,

```
// Loop over time points in data
for (int t = 0; t < DATA_T; t++)
{
  // Reset filter response for current volume
  cudaMemset(d_FR, 0, DATA_W * DATA_H * DATA_D * sizeof(float));

  // Loop over time points in filter
  for (int tt = FILTER_T - 1; tt >= 0; tt--)
  {
    // Loop over slices in filter
    for (int zz = FILTER_D - 1; zz >= 0; zz--)
    {
      // Copy current filter coefficients to constant memory
      CopyFilterCoefficients(zz,tt);

      // Perform 2D convolution and
      // accumulate the filter responses inside the kernel,
      // launch kernel for several slices simultaneously
      Convolution_2D_Shared<<<dG, dB>>>(d_FR);
    }
  }
}
```

Listing 1.5. Host code for non-separable 4D convolution, by performing non-separable 2D convolution on the GPU and accumulating the filter responses inside the kernel (dG stands for dimGrid, dB stands for dimBlock and FR stands for filter responses). The CPU takes care of 3 for-loops and the GPU 5 for-loops.

some additional time can be saved by only performing the convolution for the valid slices. The host code for non-separable 4D convolution is given in Listing 1.5 and the complete code is available in the github repository.

1.8 Non-separable 3D convolution, revisited

Now that an implementation for non-separable 4D convolution has been provided, 3D convolution is very easy. The host code is given in Listing 1.4 and the complete code is available in the repository. The only difference compared to 4D convolution is that the CPU for 4D also loops over time points (for data and filters). Just as for 4D convolution, the 2D convolution is launched for all slices at the same time to increase the occupancy.

1.9 Performance

We will now list some performance measures for our implementations. All the testing has been done with an Nvidia GTX 680 graphics card with 4 GB of memory.

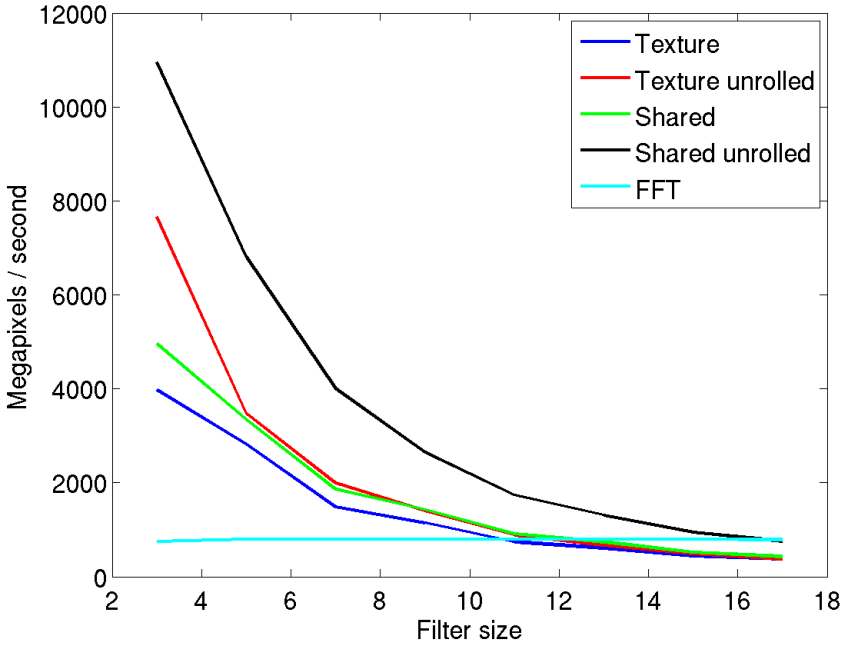

Figure 1.6. Performance, measured in megapixels per second, for the different implementations of 2D filtering, for an image of size 2048 x 2048 and filter sizes ranging from 3 x 3 to 17 x 17. The processing time for FFT based filtering is independent of the filter size, and is the fastest approach for non-separable filters larger than 17 x 17.

1.9.1 Performance, 2D filtering

Performance estimates for non-separable 2D filtering are given in Figures 1.6-1.7. Time for transferring the data to and from the GPU is not included. The first plot is for a fixed image size of 2048 x 2048 pixels and filter sizes ranging from 3 x 3 to 17 x 17. The second and third plots are for fixed filter sizes of 9 x 9 and 17 x 17, respectively. The image sizes for these plots range from 128 x 128 to 4096 x 4096 in steps of 128 pixels. All plots contain the processing time for spatial convolution using texture memory (with and without loop unrolling), spatial convolution using shared memory (with and without loop unrolling) and FFT based filtering using the CUFFT library (involving two forward FFT's, complex valued multiplication and one inverse FFT).

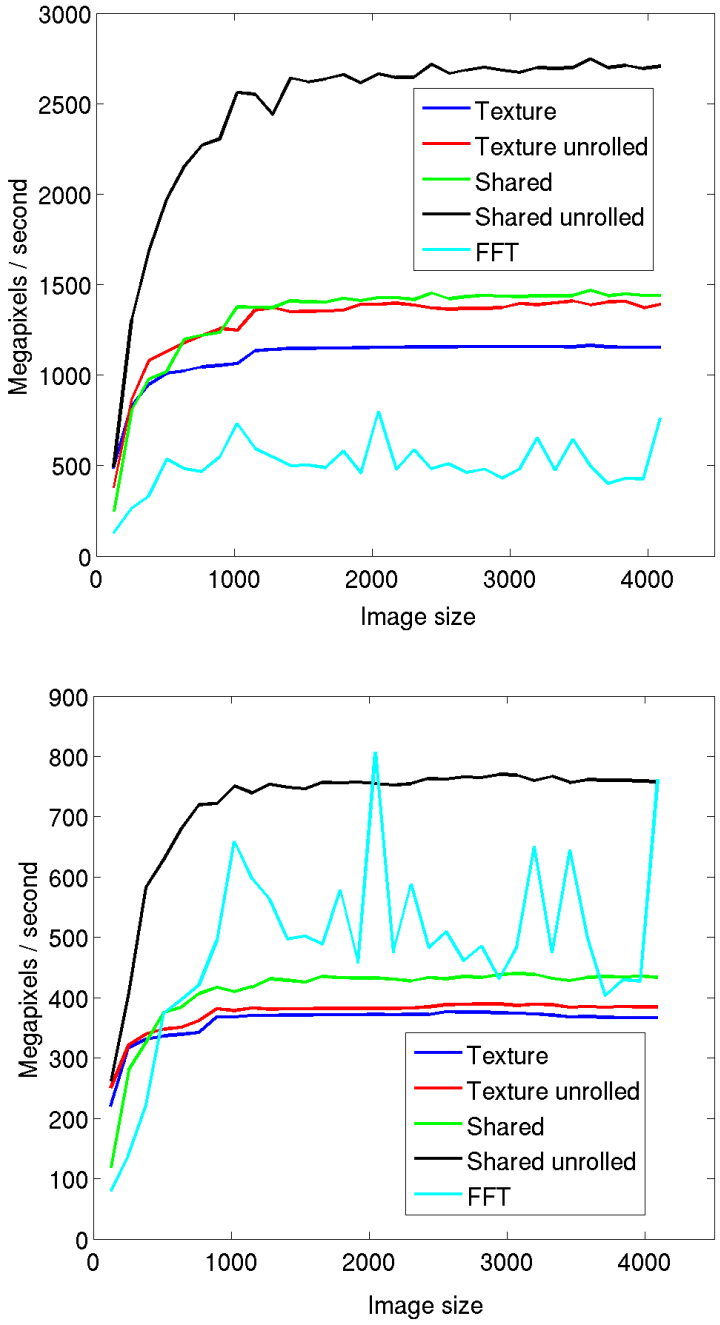

Figure 1.7. Performance, measured in megapixels per second, for the different implementations of 2D filtering and image sizes ranging from 128 x 128 to 4096 x 4096. Note the performance spikes for the FFT based filtering for image sizes that are a power of 2. **Top:** Results for a filter size of 9 x 9. **Bottom:** Results for a filter size of 17 x 17.

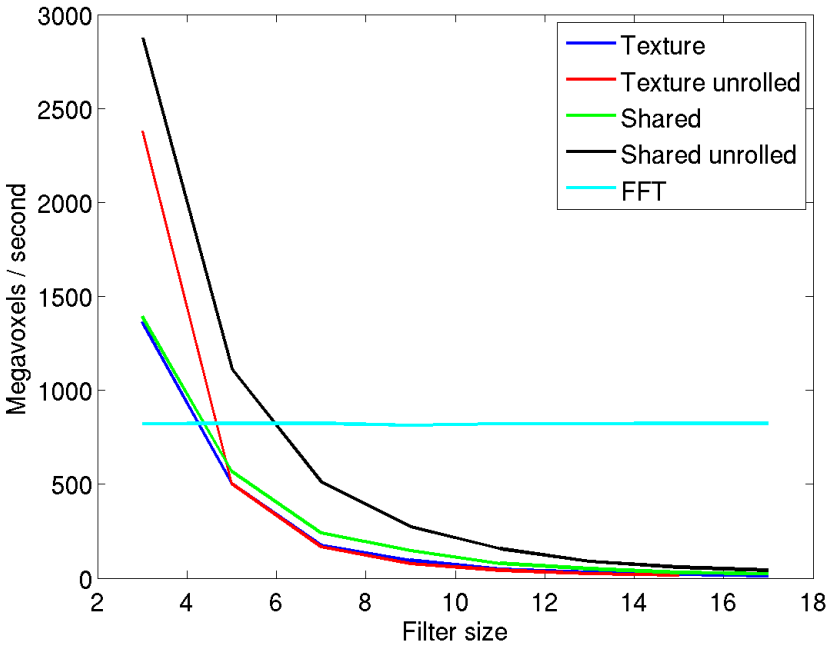

Figure 1.8. Performance, measured in megavoxels per second, for the different implementations of 3D filtering, for a volume of size 256 x 256 x 256 and filter sizes ranging from 3 x 3 x 3 to 17 x 17 x 17. FFT based filtering is clearly faster than the other approaches for large filters.

1.9.2 Performance, 3D filtering

Performance estimates for non-separable 3D filtering are given in Figures 1.8-1.9. Again, time for transferring the data to and from the GPU is not included. The first plot is for a fixed volume size of 256 x 256 x 256 voxels and filter sizes ranging from 3 x 3 x 3 to 17 x 17 x 17. The second and third plots are for fixed filter sizes of 7 x 7 x 7 and 13 x 13 x 13, respectively. The volume sizes for these plots range from 64 x 64 x 64 to 512 x 512 x 512 in steps of 32 voxels. All plots contain the processing time for spatial convolution using texture memory (with and without loop unrolling), spatial convolution using shared memory (with and without loop unrolling) and FFT based filtering using the CUFFT library (involving two forward FFT's, complex valued multiplication and one inverse FFT).

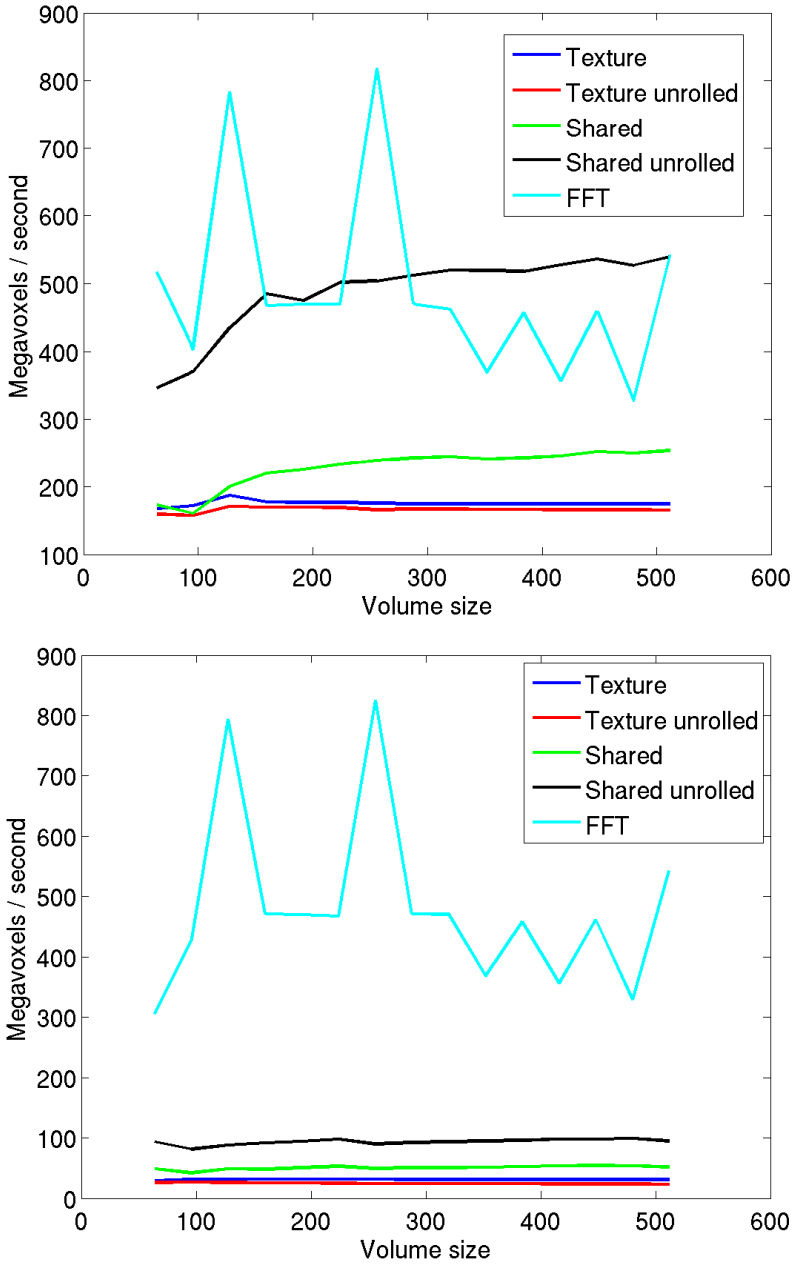

Figure 1.9. Performance, measured in megavoxels per second, for the different implementations of 3D filtering and volume sizes ranging from 64 x 64 x 64 to 512 x 512 x 512. FFT based filtering is the fastest approach for large filters. The texture based approach is actually slower for 3D with loop unrolling, which is explained by an increase in the number of registers used. **Top:** Results for a filter size of 7 x 7 x 7. **Bottom:** Results for a filter size of 13 x 13 x 13.

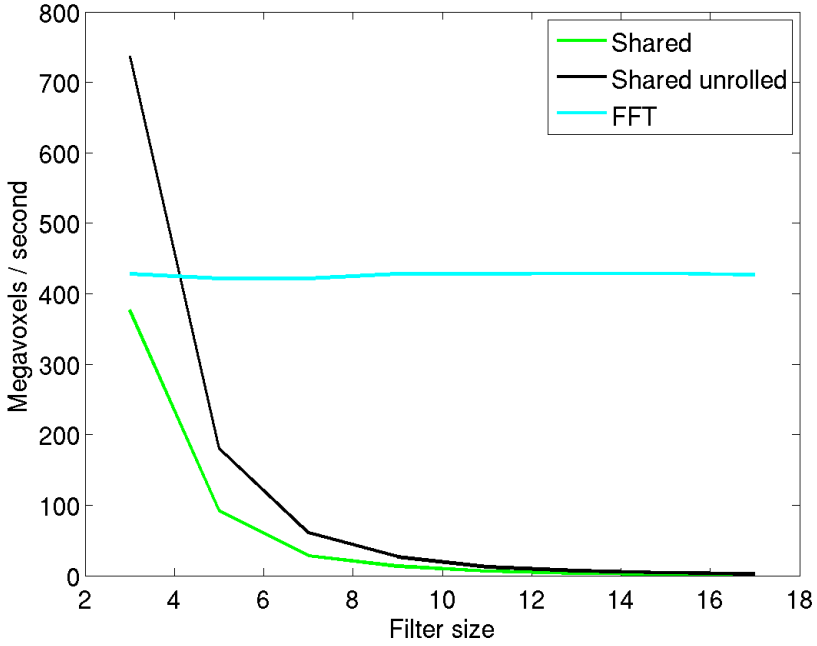

Figure 1.10. Performance, measured in megavoxels per second, for the different implementations of 4D filtering, for a dataset of size 128 x 128 x 128 x 32 and filter sizes ranging from 3 x 3 x 3 x 3 to 17 x 17 x 17 x 17.

1.9.3 Performance, 4D filtering

Performance estimates for non-separable 4D filtering are given in Figures 1.10-1.11. Again, time for transferring the data to and from the GPU is not included. The first plot is for a fixed data size of 128 x 128 x 128 x 32 elements and filter sizes ranging from 3 x 3 x 3 x 3 to 17 x 17 x 17 x 17. The second and third plots are for fixed filter sizes of 7 x 7 x 7 x 7 and 11 x 11 x 11 x 11, respectively. The data sizes range from 128 x 128 x 64 x 16 to 128 x 128 x 64 x 128 in steps of 8 time points. All plots contain the processing time for spatial convolution using shared memory (with and without loop unrolling) and FFT based filtering using CUFFT. The CUFFT library does not directly support 4D FFTs, it was performed by running two batches of 2D FFTs and changing the order of the data between them from (x,y,z,t) to (z,t,x,y). A 4D FFT developed by Nvidia would, however, probably be more efficient.

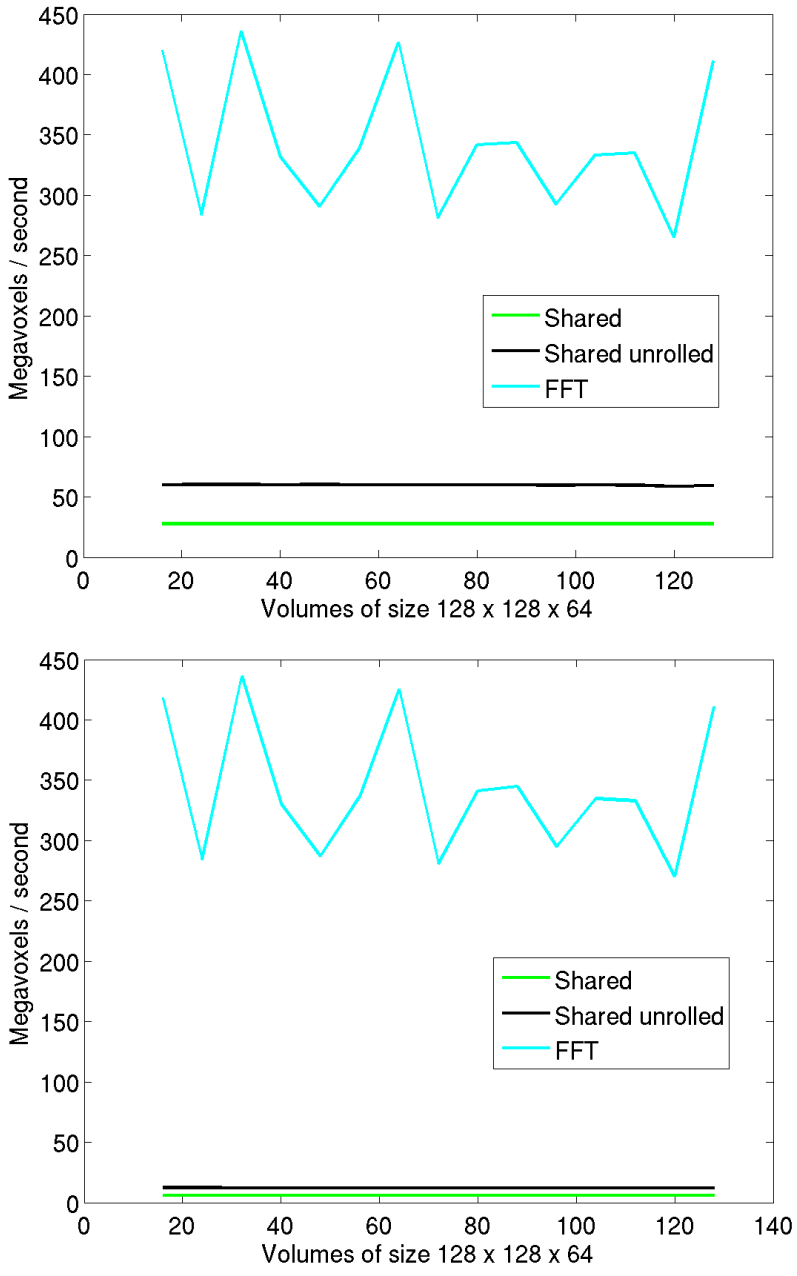

Figure 1.11. Performance, measured in megavoxels per second, for the different implementations of 4D filtering and data sizes ranging from 128 x 128 x 64 x 16 to 128 x 128 x 64 x 128. The FFT based approach clearly outperforms the convolution approaches. **Top:** Results for a filter size of 7 x 7 x 7 x 7. **Bottom:** Results for a filter size of 11 x 11 x 11 x 11.

1.10 Conclusions

We have presented solutions for fast non-separable floating point convolution in 2, 3 and 4 dimensions, using the CUDA programming language. We believe that these implementations will serve as a complement to the NPP library, which currently only supports 2D filters and images stored as integers. The shared memory implementation with loop unrolling is approximately twice as fast as the simple texture memory implementation, which is similar to results obtained by Nvidia for separable 2D convolution [Podlozhnyuk 07]. For 3D and 4D data it might seem strange to use convolution instead of an FFT, but the convolution approach can for example handle larger datasets. In our work on 4D image denoising [Eklund et al. 11], the FFT based approach was on average only three times faster (compared to about 30 times faster in the benchmarks given here). The main reason for this was the high resolution nature of the data (512 x 512 x 445 x 20 elements), making it impossible to load all the data into global memory. Due to its higher memory consumption, the FFT based approach was forced to load a smaller number of slices into global memory compared to the spatial approach. As only a subset of the slices (and time points) is valid after the filtering, the FFT based approach required a larger number of runs to process all the slices.

Finally, we close by noting two additional topics that readers may wish to consider for more advanced study. First, applications in which several filters are applied simultaneously to the same data (for example six complex valued quadrature filters to estimate a local structure tensor in 3D) can lead to different conclusions regarding performance using spatial convolution versus FFT based filtering. Second, filter networks can be used to speedup spatial convolution by combining the result of many small filter kernels, resulting in a proportionally higher gain for 3D and 4D than for 2D convolution [Andersson et al. 99, Svensson et al. 05]. All the code for this chapter is available under GNU GPL 3 at

`https://github.com/wanderine/NonSeparableFilteringCUDA`

Bibliography

[Andersson et al. 99] Mats Andersson, Johan Wiklund, and Hans Knutsson. "Filter networks." In *Proceedings of Signal and Image Processing (SIP)*, pp. 213–217, 1999.

[Eklund et al. 10] Anders Eklund, Mats Andersson, and Hans Knutsson. "Phase based volume registration using CUDA." In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 658–661, 2010.

[Eklund et al. 11] Anders Eklund, Mats Andersson, and Hans Knutsson. "True 4D image denoising on the GPU." *International Journal of Biomedical Imaging, Article ID 952819*.

[Eklund et al. 12] Anders Eklund, Mats Andersson, and Hans Knutsson. "fMRI analysis on the GPU - Possibilities and challenges." *Computer Methods and Programs in Biomedicine* 105 (2012), 145–161.

[Eklund et al. 13] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen LaConte. "Medical image processing on the GPU - Past, present and future." *Medical Image Analysis* 17 (2013), 1073–1094.

[Forsberg et al. 11] Daniel Forsberg, Anders Eklund, Mats Andersson, and Hans Knutsson. "Phase-based non-Rigid 3D image registration - From minutes to seconds using CUDA." In *Joint MICCAI Workshop on High Performance and Distributed Computing for Medical Imaging*, 2011.

[Granlund and Knutsson 95] Gösta Granlund and Hans Knutsson. *Signal processing for computer vision*. Kluwer Academic Publishers, 1995. ISBN 0-7923-9530-1.

[Granlund 78] Gösta Granlund. "In search of a general picture processing operator." *Computer Graphics and Image Processing* 8 (1978), 155–173.

[Hopf and Ertl 99] Matthias Hopf and Thomas Ertl. "Accelerating 3D convolution using graphics hardware." In *IEEE Visualization Conference*, pp. 471–475, 1999.

[Jain and Farrokhnia 91] Anil Jain and Farshid Farrokhnia. "Unsupervised texture segmentation using Gabor filters." *Pattern Recognition* 24 (1991), 1167–1186.

[James 01] Greg James. "Operations for hardware-accelerated procedural texture animation." In *Game Programming Gems 2, Charles River Media*, pp. 497–509, 2001.

[Knutsson and Andersson 05] Hans Knutsson and Mats Andersson. "Morphons: segmentation using elastic canvas and paint on priors." In *IEEE International Conference on Image Processing (ICIP)*, pp. 1226–1229, 2005.

[Knutsson et al. 83] Hans Knutsson, Roland Wilson, and Gösta Granlund. "Anisotropic non-stationary image estimation and its applications - Part I: Restoration of noisy images." *IEEE Transactions on Communications* 31 (1983), 388–397.

[Knutsson et al. 99] Hans Knutsson, Mats Andersson, and Johan Wiklund. "Advanced filter design." In *Scandinavian Conference on Image Analysis (SCIA)*, pp. 185–193, 1999.

[Knutsson et al. 11] Hans Knutsson, Carl-Fredrik Westin, and Mats Andersson. "Representing local structure using tensors II." *Lecture Notes in Computer Science, Proceedings of the Scandinavian Conference on Image Analysis (SCIA)* 6688 (2011), 545–556.

[Knutsson 89] Hans Knutsson. "Representing local structure using tensors." In *Scandinavian Conference on Image Analysis (SCIA)*, pp. 244–251, 1989.

[Läthen et al. 10] Gunnar Läthen, Jimmy Jonasson, and Magnus Borga. "Blood vessel segmentation using multi-scale quadrature filtering." *Pattern Recognition Letters* 31 (2010), 762–767.

[Owens et al. 07] John Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron Lefohn, and Timothy Purcell. "A survey of general-purpose computation on graphics hardware." *Computer Graphics Forum* 26 (2007), 80–113.

[Podlozhnyuk 07] Victor Podlozhnyuk. "Image convolution with CUDA, Nvidia white paper.", 2007.

[Rost 96] Randi Rost. "Using OpenGL for imaging." *SPIE Medical Imaging, Image Display Conference* 2707 (1996), 473–484.

[Sanders and Kandrot 11] Jason Sanders and Edward Kandrot. *CUDA by example - An introduction to general-purpose GPU programming.* Addison-Wesley, 2011. ISBN 978-0-13-138768-3.

[Svensson et al. 05] Björn Svensson, Mats Andersson, and Hans Knutsson. "Filter networks for efficient estimation of local 3D structure." In *IEEE International Conference on Image Processing (ICIP)*, pp. 573–576, 2005.

[Tomasi and Manduchi 98] Carlo Tomasi and Roberto Manduchi. "Bilateral filtering for gray and color images." In *Proceedings International Conference on Computer Vision*, pp. 839–846, 1998.

[Westin et al. 01] Carl-Fredrik Westin, Lars Wigström, Tomas Loock, Lars Sjöqvist, Ron Kikinis, and Hans Knutsson. "Three-dimensional adaptive filtering in magnetic resonance angiography." *Journal of Magnetic Resonance Imaging* 14 (2001), 63–71.