GPGPU: Image Convolution

Abstract

Convolution of two functions is an important mathematical operation that found heavy application in signal processing. In computer graphics and image processing we usually work with discrete functions (e.g. an image) and apply a discrete form of the convolution to remove high frequency noise, sharpen details, detect edges, or otherwise modulate the frequency domain of the image. In this assignment, we discuss an efficient implementation of image convolution filters on the GPU. A general 2D convolution has a high bandwidth requirement as the final value of a given pixel is determined by several neighboring pixels. Since memory bandwidth is usually the main limiting factor of algorithm's performance, our optimization techniques will focus on minimizing global memory accesses during the computations.

1 Image Convolution

1.1 Introduction

Convolution is a mathematical operation on two signals f and g, defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau.$$

$(f * g)(t)$ is frequently considered as the *filtered* variant of the $f(t)$ input signal, where $g(t)$ is the filtering *kernel*. One of the fundamental properties of this operator is defined by the *convolution theorem*, which states that

$$F\{f * g\} = kF\{f\}F\{g\}$$

Where F is the Fourier-transform of the signal. Therefore, convolution in the time / spatial domain is equivalent to multiplication in the frequency domain. This practically means that a properly designed kernel can be used to remove or amplify certain frequencies of a given signal. In digital image processing (DSP), we can use this property to blur or sharpen an image (low-pass vs. high-pass filtering).

If an image is represented as a 2D discrete signal $y[,]$, we can perform the discrete convolution in 2-dimensions using a discrete kernel $k[,]$ as:

$$(y * k)[i,j] = \sum_n \sum_m y[i-n, j-m]k[n,m].$$

As we always process an image with a finite resolution, the convolution is actually a scalar product of the filter weights and all pixels of the image within a window that is defined by the extent of the filter and a center pixel. Figure 1 illustrates the convolution using a small 3×3 kernel. The filter is defined as a matrix, where the central item weights the center pixel, and the other items define the weights of the neighbor pixels. We can also say that the *radius* of the 3×3 kernel is 1, since only the one-ring neighborhood is considered during the convolution. We also have to define the convolution's behavior at border of the image, where the kernel maps to undefined values outside the image. Generally, the filtered values outside the image boundaries are either treated as zeros (this is what we will do in this assignment) or clamped to the border pixels of the image.

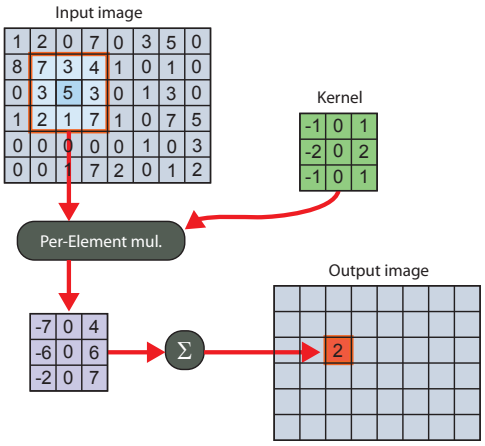

Figure 1: *Convolution using a 3×3 kernel.*

The design of the convolution filter requires a careful selection of kernel weights to achieve the desired effect. In the following, we introduce a few examples to demonstrate basic filtering kernels often used in image processing.

1.2 Convolution Kernels

1.2.1 Sharpness Filter

The aim of this filter is to emphasize details of the input image (Figure 2 B). The simplest sharpness filter is defined by a 3×3 kernel that can be described by any of the following matrices:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}; \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}; \begin{bmatrix} -k & -k & -k \\ -k & 8k+1 & -k \\ -k & -k & -k \end{bmatrix}.$$

Examining the matrices, we can see that for each source pixel, the filter takes its neighborhood and subtracts it from the pixel. The weight of the source pixel is always greater than the absolute sum of all other weights, meaning that this kernel tries to preserve the original color by scaling it and subtracting the neighborhood.

1.2.2 Edge Detection

In order to detect edges, we compute the gradient of the input image along a given direction. Convolving the image with one of the following matrices will result in large values where the pixel intensity changed significantly. Unfortunately these simple techniques are not very practical, as they greatly emphasize any noise in the image and only detect edges from one direction (Figure 2 C). Note

that all matrices sum up to zero.

$$\begin{bmatrix} -1/8 & -1/8 & -1/8 \\ -1/8 & 1 & -1/8 \\ -1/8 & -1/8 & -1/8 \end{bmatrix}; \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix};$$

1.2.3 Embossing Filter

A very interesting example is the embossing filter which makes the impression that the image is graved into stone and lit from a specific direction (Figure 2 D). The difference to the previous filters is that this filter is not symmetric. The filter is usually applied to gray scale images. As the resulting values can be negative, we should add a normalization offset that will shift the range of results into positive values (otherwise some viewers may not display them).

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix};$$

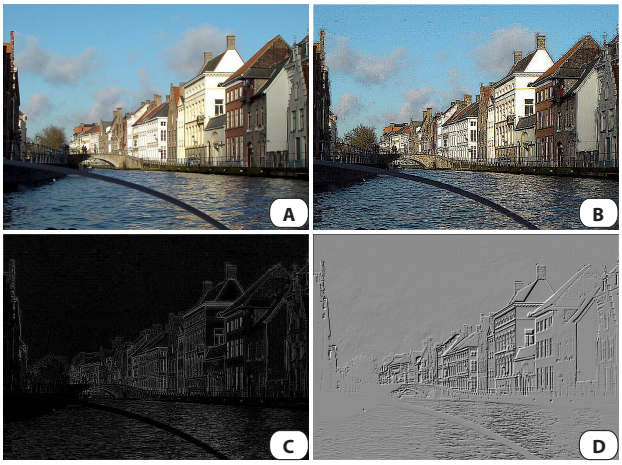

Figure 2: *Even small image convolution kernels can be powerful image processing operators. (A): The original image. (B): Sharpening filter. (C): Edge detection filter. (D): Embossing filter.*

1.3 Separable Kernels

Convolution is a useful, but computationally expensive operation. For a given kernel matrix with width k we need k^2wh multiplications and additions to convolve an image of size $w \times h$. Some 2D convolution kernels can be broken down to two 1D convolution kernels, one in the horizontal and one in the vertical direction. Applying these two kernels sequentially to the same image yields equivalent results, but with much lower complexity: only $2kwh$ multiplications and additions. We call kernels with such property *separable*. In practice, we want to determine if a given kernel is separable and if so, find its two 1D equivalents for separable convolution.

A convolution kernel is separable, if the convolution matrix K has the special property that it can be expressed as the *outer product* of two vectors u and v. For a 3x3 matrix:

$$K = v \otimes u = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} = \begin{bmatrix} v_1u_1 & v_1u_2 & v_1u_3 \\ v_2u_1 & v_2u_2 & v_2u_3 \\ v_3u_1 & v_3u_2 & v_3u_3 \end{bmatrix}$$

Having these vectors, we have already separated the convolution kernel: u is the horizontal, v is the vertical 1D kernel. Unfortunately, only a small fraction of the possible convolution kernels are separable (it is not difficult to see that the above decomposition is only possible, if the *rank* of the K matrix is 1), but there are still several practical image filters that can be implemented this way.

1.3.1 Box Filter (Averaging)

The simplest separable filter takes the neighborhood of each pixel in the filter area and computes their average (Figure 3).

$$K_{box} = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}; u = v = \begin{bmatrix} 1/3 & 1/3 & 1/3 \end{bmatrix}$$

Figure 3: *A 9 × 9 box-filter is applied to the image. As the filter is not smooth, the blocks are still visible after a relatively wide kernel.*

1.3.2 Gaussian Filter

Box filtering is simple, but does not result in a smoothly blurred image. Gaussian blur is widely used in graphics software to reduce image noise or remove details from the image before detecting relevant edges. Gaussian blur is a low-pass filter, attenuating high frequency components of the image. The 2D Gaussian function (Figure 4) is the product of two 1D Gaussian functions:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}; G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

An example of a Gaussian-kernel with radius 2 is shown in Figure 4.

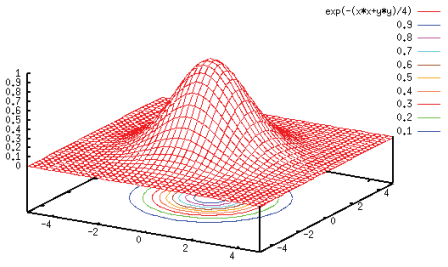

Figure 4: *The 2D Gaussian function.*

$$K_{G5} = \frac{1}{273} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix};$$

$$u = v = \begin{bmatrix} .061 & .242 & .383 & .242 & .061 \end{bmatrix}$$

As illustrated in Figure 5, the Gaussian filter gives a much smoother blurring result than the simple box filter.

Figure 5: *The Gaussian filter gives much smoother results compared to the box filter. We applied a 7×7 filter to the above image twice, one after the other. The result preserves important details of the original image while the noise is effectively eliminated.*

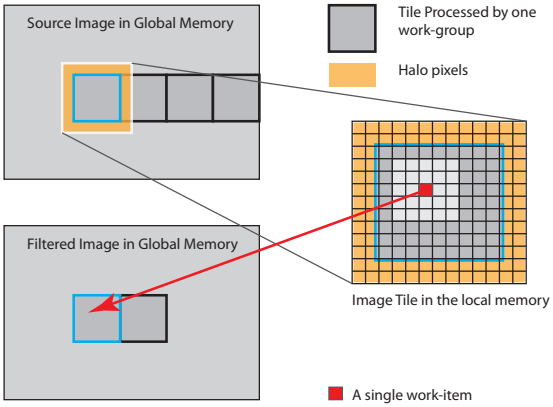

Figure 6: *In the parallel implementation of the convolution in OpenCL, each work-group must load a block of pixels to be filtered into the fast local memory. As the kernel extends over the processed tile at the border pixels, there is a halo of additional pixels of the width of the kernel radius that is required in order to filter the tile.*

2 Implementation Considerations

Image convolution can be efficiently implemented on massively parallel hardware, since the same operator gets executed independently for each image pixel. However, there are several practical problems that make the efficient implementation non-trivial. A naïve OpenCL implementation would simply execute a work-item for each pixel, read all values inside the kernel area from the global memory, and write the result of the convolution to the pixel. In this case, each work-item would issue nm reads for an $n \times m$-sized kernel. Even worse, these reads would be mostly unaligned in the global memory, resulting in several non-coalesced loads. This approach is very inefficient and we will not implement it in this assignment.

As the first improvement, we can divide the input image into small tiles, each of which gets processed by a single work-group. These work-groups can copy the pixels of the tile into the fast on-chip local memory in a coalesced manner (the same as in the matrix rotation task), then each work-item can quickly access neighboring pixels loaded by other work-items (Figure 6). This can already mean magnitudes of speedup without any further optimizations, especially for large kernels.

For any reasonable kernel size, the blocks of pixels read by neighboring work-groups will overlap, as the processing of each output block will also depend on pixels outside its boundary. To correctly compute the convolved result, the work-group will also need to load a *halo* of the kernel radius. This will make the efficient OpenCL implementation of the convolution more complicated as we will need to take special care of keeping the memory accesses aligned.

2.1 Constant Memory

During the implementation of the convolution we will also make use of the constant memory of the device for the first time. Constant memory is a special part of the OpenCL memory model containing data that are invariant during the kernel execution. The device can cache these values during the execution so they can be accessed with low latency. The size of the constant memory is limited due to the cache size, the maximum amount available is 64KB on the Fermi architecture. Note that the constant data is still allocated in the global memory, but cached using a dedicated cache for constants (2KB), which helps to hide most of the global memory latency.

In our implementation we will store the kernel weights in the constant memory.

2.2 Memory Alignment

We will store the processed image as a linear array in the global memory, but work-groups will operate on it in the two-dimensional domain. To keep data accesses coalesced, the base address of each warp (group of 32 threads) must align to 64 or 128-byte aligned segments. Now, if the width of the 2D image is not multiple of the coalesced segment size, the memory access pattern of 2D work-groups will be misaligned, as every row of the image gets shifted.

We can eliminate this problem by making sure that the width of the 2D array is always a multiple of the coalesced segment size. If this is not the case, we add a small padding at the end of each line, ensuring the proper alignment. The actual size of a line (i.e. length + padding) is called a *pitch*. When mapping a 2D index to the linear array we will use the pitch instead of the line length.

3 Task 1: Non-Separable Convolution

As the first task you will implement a convolution with an arbitrary 3×3 kernel on the device. In the general case, the convolution kernel is not separable, therefore, each pixel must consider its entire one-ring neighborhood.

3.1 Introducing the code

The reference solution on the CPU is already implemented in the `CConvolution3x3Task` class, which you can find in the startup project. This class takes an input PFM image (portable float map), applies the given convolution kernel, and saves the result in the same format. During the assignment you do not have to change the code of this class. You only need to implement the OpenCL kernel that performs the same operation on the device. In order to view the PFM files you can use *pfstools* (cross-platform open source, `http://pfstools.sourceforge.net/`), Adobe Photoshop, or some other viewer for HDR images. On the ATIS machines Image Magick is installed under Fedora so you can just type `display color.pfm` to display the file `color.pfm`. **Make sure that you have a PFM viewer handy for the evaluation!**

Before implementing, we recommend to take a closer look at the class to understand its behavior. Application of an emboss filter to the input image (`CAssignment3.cpp`):

```
size_t TileSize[2] = {32, 16};
float ConvKernel[3][3] = {
  {2, 0, 0},
  {0, -1, 0},
  {0, 0, -1},
};
CConvolution3x3Task convTask("Images/input.pfm",
  TileSize, ConvKernel, true, 0.5f));
RunComputeTask(convTask, TileSize);
```

The first parameter of the constructor is the input image, the second parameter is the size of the tile in pixels, which will be processed by a single work-group. The third parameter defines the 9 weights of the convolution kernel, while the last one is a constant offset that will be added to each pixel after the convolution. The boolean parameter simply defines if we want to convert the image to gray scale before processing or not. You can also try out the other 3x3 kernels introduced in Section 1.2.

3.2 Kernel Implementation

We store the image in separate arrays, one for each color channel, and perform the convolution individually on each channel. Hence, for an RGB floating point image, the filtering algorithm will be executed three times. This lowers the requirements on the local memory, as we only need to load data of one channel. More importantly, the accesses of a single warp to one row of pixels will match a 128-byte aligned segment (32 floats). We also do not need to implement different kernels for filtering colored and gray scale images.

Your OpenCL implementation should divide the image into tiles, which are small enough to fit into the local memory. The algorithm then processes each tile using a single work-group to reduce loads from the global memory. The kernel should consist of two main parts separated by a memory barrier. In the first part the work-items of the work-group should cooperatively load the relevant image region for the convolution of the tile. Each work-item will load one pixel in the active area of the convolution. Since the convolution requires also pixels lying in the halo ring, a subset of the work-items has to additionally load the halo pixels. Do not forget to allocate enough of shared memory to contain also the halo region!

We assume that the width of the work-group matches the coalesced segment size of the device, so the base addresses of the work-items are always aligned. The header of the kernel is already defined in `Convolution3x3.cl`:

```
__kernel __attribute__((reqd_work_group_size(TILE_X,
    TILE_Y, 1)))
void Convolution(
        __global float* d_Dst,
        __global const float* d_Src,
        __constant float* c_Kernel,
        uint Width,
        uint Height,
        uint Pitch
        )
{}
```

The input data is in a buffer referenced by `d_Src`, the convolved image should be stored in `d_Dst`. As you can see, `c_Kernel` is defined as a pointer to the constant memory, so all kernel weights will be cached in the on-chip constant cache during execution. `c_Kernel` contains 11 float values, `c_Kernel[0]` - `c_Kernel[8]` are the kernel weights, `c_Kernel[9]` is the normalization factor (with which you have to multiply the convolution result) and `c_Kernel[10]` is the offset that must be added to the normalized result.

It is important to mention that both `d_Dst` and `d_Src` are linearly aligned in the global memory as described in Sect. 2.2, therefore you should use the last attribute, `Pitch` to calculate row offsets on the memory:

```
// Access pixel [x, y]
// Use Pitch instead of Width!
// Width is only for boundary checks
float pix = d_Src[y * Pitch + x];
```

Finally, we can define strict conditions for the allowed work-group size. This feature of the OpenCL compiler can be useful if we need to know the dimensions of the work-group at the compilation time, and want to avoid run-time errors using the kernel with incorrect execution configuration. The `reqd_work_group_size()` attribute will prevent the kernel from running if the size of the work-group is not `TILE_X` $\times$ `TILE_Y`. For example, we can statically allocate local memory for the work-group in the kernel code (note that in all previous assignments we allocated the local memory dynamically, using an argument to the kernel):

```
//local memory for the convolution + the halo area
__local float tile[TILE_Y + 2][TILE_X + 2];
```

The reference solution is implemented in the `ConvolutionChannelCPU()` method of the `CConvolution3x3Task` class. To pass the evaluation test, your implementation should match the reference result. The reference test also computes a difference image which you can examine to clearly see regions of the GPU output that contain incorrect values. These difference images and also the result images are saved to the `Images` sub-folder on each run. We encountered a probable driver bug while preparing this assignment, which results in random NAN values on the last line for the first task. Because of this the last line is not checked in our automatic test. However, this does not mean you can skip any test for valid pixel positions.

Hint: Since some halo pixels will map outside the image, you should not forget to manually set them to zero.

3.3 Evaluation

The total amount of points reserved for this task is 8:

- Tile and halo pixels are loaded into the local memory without bank-conflicts. (4 points).
- The 3x3 convolution is performed for each pixel in the tile and the result is stored in the output image (4 points).

4 Task 2: Separable Convolution

Apart from being less computation-intensive in terms of arithmetic operations, a separable filter also allows us to employ further optimizations to improve the performance. Instead of executing a single kernel for the entire convolution, we can separate the convolution kernel into a horizontal and a vertical pass. Note that without any further steps, the memory bandwidth can already improve significantly. If the kernel radius is large, the non-separable implementation must load a large halo region for each processed tile. Having a 16×16-sized block and a kernel with radius 16, this would mean that each pixel must be loaded into the local memory of different work-groups 9 times (see Figure 7). The separable implementation of the same dimensions would only need to load the halo along one direction, thus the required bandwidth already drops by 33% (a pixel is loaded 3 times in each directions).

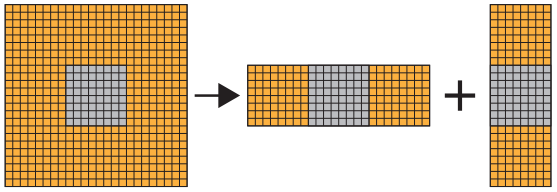

Figure 7: *If the kernel radius is large, the non-separable implementation must load a large halo region of pixels to the local memory. Besides its computational efficiency, a separable kernel also improves the memory bandwidth requirements of the algorithm.*

4.1 Horizontal Filter

We can further improve the bandwidth efficiency of the horizontal kernel by increasing the width of the image region processed by the same work-group. By omitting halo values in the vertical direction, we have enough local memory available for each work-group to handle more pixels per work-item. In this case we are more limited by the work-group size (maximum 1024 work-items on Fermi) than the local memory. The computational complexity of the kernel remains the same, of course, but now there will be several pixels that are only loaded once during the horizontal convolution pass.

By proper tiling of the image to work-group areas, it is simple to ensure that each work-group has a properly aligned base address for coalescing. The halo regions, however, make the algorithm a lot more complicated. In this task we allow the user to define an arbitrary kernel radius. The question is then how to load pixels in the halo area. If the work-items with `get_local_id(0) == 0` would load all the leftmost halo pixels as well, the memory accesses would be unaligned and we would lose the coalescing. The best solution to this problem is illustrated in Figure 8. By sacrificing a small amount of local memory, we make sure that the memory

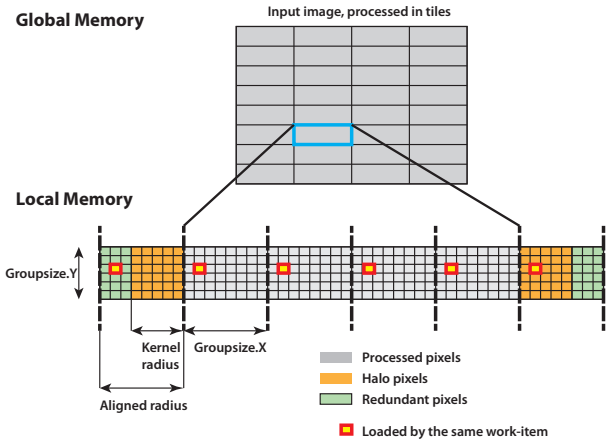

Figure 8: *The horizontal filter, as processed by a single work-group. Since we are more limited by the number of work-items than the local memory, a single work-item can load and process multiple pixels in the horizontal direction. To maintain coalescing, the loaded halo pixels are extended to match a 128-bit aligned segment.*

accesses of the work-items are always properly aligned: the entire work-group loads both the left and right halo pixels *inside the work-group width*. These redundant loads will not have any performance drawback as the load of the entire halo region will be coalesced into a single transaction, and it even makes the code simpler: as each work-item loads the same number of pixels to the local memory, no branching is necessary to check if the work-item is inside the halo or not.

4.2 Vertical Filter

The vertical filter uses the same approach, but this time the work-item indices are increasing perpendicularly to the filter direction rather than along it. The goal is now to maximize the height of the tile being filtered by a single work-group, so we should keep the tile as narrow as possible. To match coalescing requirements, it is the best to set the width to 32 (or 16 on pre-Fermi cards), so that each row of the tile can be loaded in a single transaction. Akin to the horizontal kernel, each thread loads multiple elements to the local memory, reducing the number of overlapped pixels of different tiles. Figure 9 depicts the layout of the kernel memory accesses in the vertical filtering pass.

4.3 Implementation

The `CConvolutionSeparableTask` class implements the reference solution to the separable convolution on the CPU. We recommend you to closely examine the CPU solution before proceeding with the implementation of the OpenCL kernel. The structure of this class is very similar to `CConvolution3x3Task`, but now two kernels have to be executed for the convolution, and the filtering function is given by two 1D arrays. The following code snippet uses a `CConvolutionSeparableTask` object to perform a box filter on the image with radius 4:

```
size_t HGroupSize[2] = {64, 8};
size_t VGroupSize[2] = {32, 16};
float ConvKernel[9];
for(int i = 0; i < 9; i++)
    ConvKernel[i] = 1.0f / 9.0f;
```

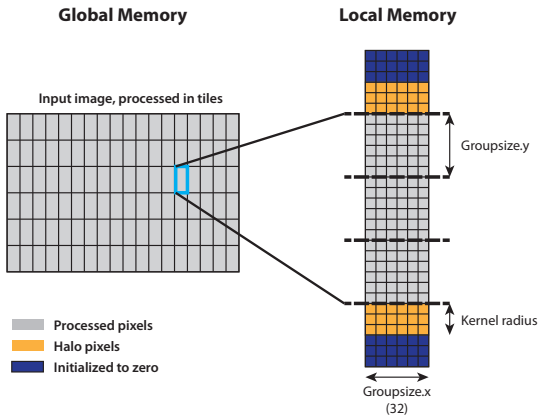

Figure 9: *In the vertical pass, the width of the tile processed by a work-group should be 32, so that the Fermi architecture can load a row of float values in a single transaction. The concept of implementation is the same as in the horizontal pass, but note that the redundant pixels in the local memory are initialized to zeros this time, as loading additional pixels would mean redundant memory transactions as well.*

```
CConvolutionSeparableTask convTask("input.bmp",
    HGroupSize, VGroupSize,
     3, 3, 4, ConvKernel, ConvKernel);
RunComputeTask(convTask, HGroupSize);
```

This time we should define the work-group dimensions for the horizontal and vertical passes separately, as the optimal configuration can be different in each case. The fourth attribute is the number of pixels a single thread computes in the horizontal pass (3), the fifth one is the same for the vertical pass, the next value (4) is the kernel radius.

Your task is to implement the body of the `ConvHorizontal` and `ConvVertical` kernel functions in the `ConvolutionSeparable.cl` file. Note that during the building of your OpenCL program, several macro definitions will be provided for the compiler, so it can optimize the code by unrolling loops and you can statically allocate the local memory, similarly to the previous task. You can find the description of these macros before the kernel headers. For example, in the horizontal kernel, each work-item processes `H_RESULT_STEPS` pixels. The static local memory for the workgroup can be allocated like this:

```
__local float tile[H_GROUPSIZE_Y][(H_RESULT_STEPS + 2)
    * H_GROUPSIZE_X];
```

If `H_GROUPSIZE_X` is the multiple of 32, there will be no bank conflicts during loading data to the local memory. Each work item has `H_RESULT_STEPS + 2` slots in the local memory, the two adds space for loading the halo pixels on the left and right side, respectively. For simplicity, we assume that the kernel radius is not greater than the dimension of the work-group along the convolution direction, so it is enough if each work-item loads exactly one halo pixel.

Some general advice for the implementation:

- Do not forget to use barriers before processing data from the local memory.
- Use the macro definitions whenever possible. If a value is known at compilation time, the compiler can optimize the code much better. For example, the innermost loop performing the 1D convolution can be automatically unrolled. You can enforce unrolling by adding `#pragma unroll` before the loop.
- Do not forget to check image boundaries. If the referenced pixel is outside the image boundaries set the corresponding value in the local memory to zero. Use the image pitch as the number of pixels allocated for a single row in the memory.
- As the convolution consists of two separate passes this time, it is not easy to see which kernel executed incorrectly, if the CPU reference test failed. In this case we recommend you to temporarily comment out one convolution pass in the CPU code, so you can have an intermediate evaluation for a single convolution kernel.The difference images between the reference and the OpenCL solution can also help revealing problems.
- Ask yourself a question: Is it necessary/beneficial to load the unused pixel (green in Figure 8 and blue in Figure 9)?

4.4 Evaluation

The total amount of points reserved for this task is 10:

- Implementation of the horizontal convolution kernel. (4 points).
- Implementation of the vertical convolution kernel (4 points).
- Performance experiments: change the number of pixels processed by a single work-group (for example: `H_RESULT_STEPS`) to see how does it influence the bandwidth requirements and the performance of your application. Summarize your experiences on a chart (2 point).

5 Task 3: Histogram

A common image operation is to compute a histogram over all pixels. In this task we will compute a histogram of a grey-scale floating point image by binning the values and incrementing a per-bin counter. The CPU reference solution is already implemented in `CHistogramTask.cpp`. Since incrementing a global counter from multiple threads introduces data hazards we need to rely on atomic functions.

5.1 Atomic Functions

Atomic functions implement uninterruptible read-modify-write memory operations. They can be used to enable co-ordination among multiple threads and to serialize contentious updates from multiple threads. Only a limited set of types and data sizes, as well as operations are supported. Please refer to `https://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/atomicFunctions.html` for a complete list of all atomic operations available in OpenCL 1.1.

The following pseudo-code shows how `atomic_add` works *semantically*:

```
int atomic_add(int *p, int v)
{
  int old;
  exclusive_single_thread // pseudo code
  {
    // atomically perform load, add, store operations
    old = *p; // Load from memory
    *p = old + v; // Store after adding v
  }
  return old;
}
```

Exclusive access to the memory is granted to one thread and the memory content is updated by this thread. However when several threads issue an atomic operation for the same memory location, the order of the atomic operations is not deterministic. Each of the atomic operations returns the value which was previously stored at the memory location.

On older GPUs, atomic operations where considered to be very slow and therefore its use was discouraged. In contrast, on current GPUs atomics are highly optimized: NVIDIA claims that on current GPUs (Maxwell), the costs for atomics that are accessing the global memory roughly correspond to the costs of a global memory read.

5.2 Computing the histogram

To compute the histogram on the GPU, we launch one thread per pixel and compute into which bin the pixel-value falls. Similar to the previous tasks, the image is padded to ensure a suitable alignment for coalesced memory accesses, and you therefore need to take the `pitch` into account. Refer to the CPU implementation for the binning computation. Use a suitable atomic operation to increment the counter for the corresponding bin. For this task you only need to change the kernel called `compute_histogram` in `histogram.cl`. The kernel `set_array_to_constant` is used to initialize the histogram bins with zero and does not need to be modified.

5.3 Use of local memory

When computing the histogram, many threads will issue atomic operations on the same memory address, which makes the atomic operations less efficient. To lessen this effect we will use counters that are stored in the local memory to compute a local histogram. The local histogram of one work-group is then added to the global histogram. Implement this approach in `compute_histogram_local_memory` in `histogram.cl`. You do not need to change anything besides this kernel. The local memory is already allocated with a size of `num_hist_bins`. First you need to initialize the local histogram to zero. In the next step load one pixel in each thread and increment the corresponding local counter using a suitable atomic operation. In the last step one thread is used per histogram bin to add the local histogram to the global histogram using `atomic_add`. How does the performance compare to the version without local memory?

5.4 Evaluation

In total there are 2 points reserved for this task:

- Compute the histogram using global memory atomics (1 point)
- Compute a partial histogram in local memory using atomics and compute the global histogram thereof (1 point)