MaxCompiler
Dataflow Programming Assignment

Stanford EE392Q

MAXELER
Technologies
MAXIMUM PERFORMANCE COMPUTING

Contents

1 Section 1: Getting Started

1.1 Goals

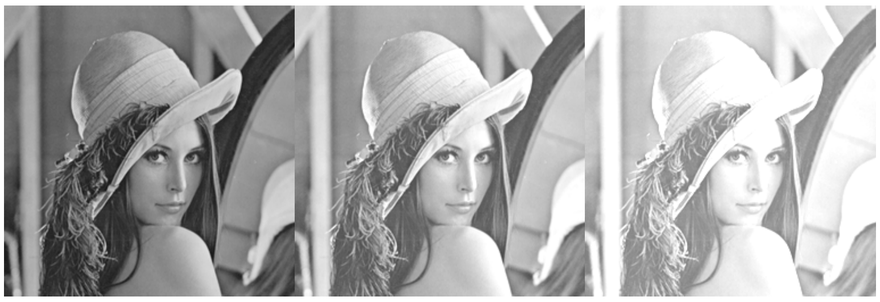
In this section, we will cover the basics of MaxCompiler using a simple image processing application as an example. You will create a Kernel that will read a gray-scale image as an input and adjust its brightness via different methods. We will start with a simple addition of a constant and move on to a more complicated gamma-correction implementation.

1.2 Topics Covered

- Basic Kernel implementation
- Basic CPU I/O
- Inputs and outputs to Kernels
- Simple arithmetic in a Kernel
- Scalar inputs to a Kernel
- Implementing look-up tables in a Kernel

1.3 Exercise 1: Brightness Adjustment

Figure 1 shows the test image we will use and the same image adjusted by adding 50 and 100 to each pixel to adjust its brightness.



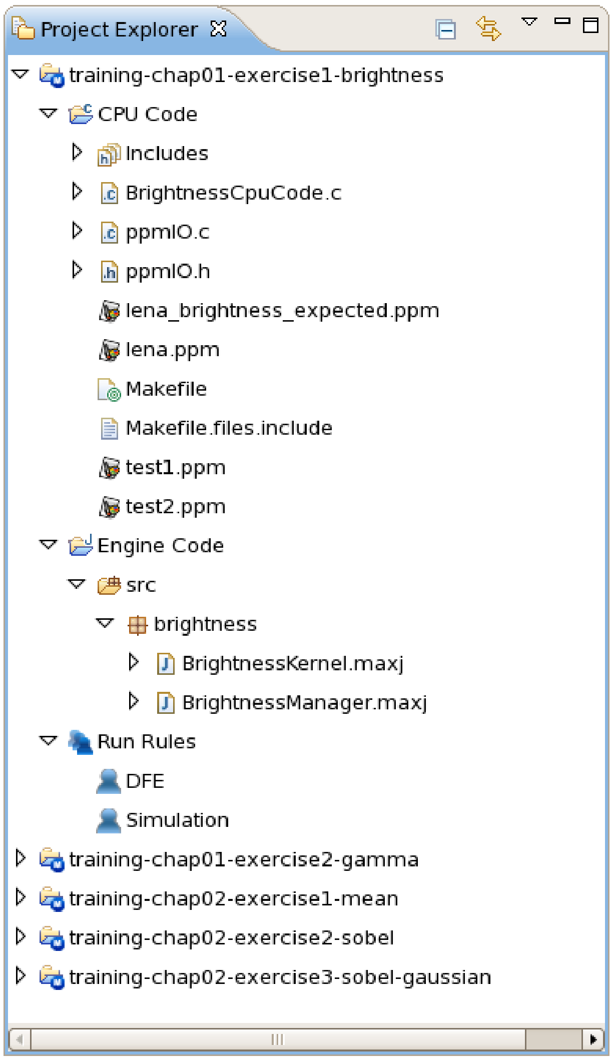
(a) Original Gray-scale *(b)* Brightness +50 *(c)* Brightness +100

Figure 1: Original and brightness-adjusted images.

A project and source code are provided to load an input image Portable Pixel Map (PPM) file (a simple text file with strings for the dimensions, color depth and for each pixel value). The default image of Lena is 256x256 pixels and is loaded as 8-bit gray-scale (256 shades of gray).

1.3.1 Your first Kernel

Task: Start MaxIDE by double-clicking on the desktop shortcut: this will load MaxIDE and the exercise workspace.



```
Project Explorer
training-chap01-exercise1-brightness
  CPU Code
    Includes
    BrightnessCpuCode.c
    ppmIO.c
    ppmIO.h
    lena_brightness_expected.ppm
    lena.ppm
    Makefile
    Makefile.files.include
    test1.ppm
    test2.ppm
  Engine Code
    src
      brightness
        BrightnessKernel.maxj
        BrightnessManager.maxj
  Run Rules
    DFE
    Simulation
training-chap01-exercise2-gamma
training-chap02-exercise1-mean
training-chap02-exercise2-sobel
training-chap02-exercise3-sobel-gaussian
```

Figure 2: Project Explorer window showing the training projects.

The MaxIDE workspace is set up with a project for each of the exercises, as shown in *Figure 2*. Each project has three sub-components:

- CPU Code: C source code that loads the input image, sends it to the Kernel for processing and saves the output image that it receives back.
- Engine Code: Java source code for the Manager and Kernel of the Dataflow Engine.
- Run Rules: Settings for running a simulation or DFE build.

An empty Kernel `BrightnessKernel.maxj` is provided as a starting point for this exercise.

All Kernels have at least one input and one or more outputs. These are added using the methods `io.input` and `io.output`:

```
DFEVar io.input(String name, KernelType type)
void io.output(String name, KernelObject output, KernelType type)
```

The `name` argument is a string to identify this stream. The `type` argument specifies the type of data that will be coming through this stream.

There are more complex stream input and output methods available which can stop and start the inputs based on a condition in the Kernel, but we don't need those for this exercise.

We are going to use an input stream for the input image and an output stream for our image once we have done some processing on it. Each Kernel tick, We will read one pixel value from the input stream and write out one processed pixel. The input image is already set up in the CPU code to send in the image in row order (i.e. you get pixels (0,0), (1,0), (2,0) ... (width-1,0), (0,1), (1,1) ...).

Task: Add a 32-bit signed integer input (by specifying the type as `dfeInt(32)`) with the string name `"inImage"` and an output called `"outImage"` to the Kernel. Note the type 32-bit signed integer is specified using `dfeInt(32)`.

You can connect your input and output directly together to make your first working Kernel.

Task: Build and run your design by selecting the project and simulation run rule from the drop-down in the toolbar at the top of MaxIDE, as shown in *Figure 3*, and clicking on the *Run* button.

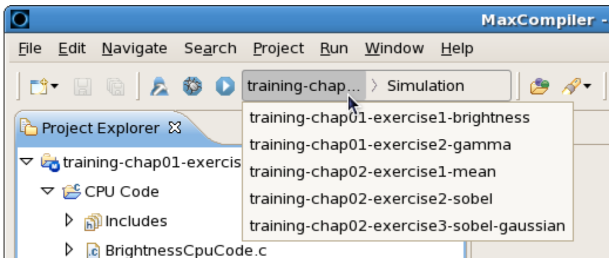

MaxCompiler -
File Edit Navigate Search Project Run Window Help
training-chap...
Simulation
Project Explorer
training-chap01-exercis
CPU Code
Includes
BrightnessCpuCode.c
training-chap01-exercise1-brightness
training-chap01-exercise2-gamma
training-chap02-exercise1-mean
training-chap02-exercise2-sobel
training-chap02-exercise3-sobel-gaussian

Figure 3: Buttons for building and running designs.

1.3.2 Brightness adjustment

Arithmetic in MaxCompiler is very intuitive. An `DFEVar` stream has overloaded Java operators for arithmetic and other operations (this is an extension to standard Java). You can describe a sequence of arithmetic operations on a combination of `DFEVars` and **construction-time constant** Java types and MaxCompiler will build a graph to implement this, scheduling the data to arrive at the correct time in the resultant pipeline.

MaxCompiler is implemented as Java libraries (with some exceptions, such as the operator-overloading support). **Construction time** is the phase after Java compilation when MaxCompiler runs and compiles the graph from your Java description of the Kernel.

For example, to add the square root of a number that is known at construction-time to a stream, we can do the following:

```
DFEVar input = io.input("input", dfeInt(32));
int a = 10;
DFEVar result = input + Math.sqrt(a);
```

Task: Add a constant to each pixel in the input image stream and write the result to the output of your Kernel.

Figure 1 shows some example results when the image is adjusted by adding 50 and 100 to each pixel to adjust its brightness.

This constant is built into the dataflow engine at compilation time, so to adjust the brightness by a different amount, we would have to rebuild the design.

Scalar inputs allow us to pass individual values (as opposed to stream inputs and outputs) from the CPU at run-time:

DFEVar ***io****.scalarInput(****String*** *name,* ***KernelType*** *type)*

These take a name and a type the same as a stream input, but they will output the same value, set by the CPU, every tick.

MaxCompiler will automatically add an argument for the scalar input to the SLiC function for running the DFE when the `.max` file is rebuilt.

Task: Use a scalar input to pass the brightness adjustment value from the CPU. This will involve changing both Engine Code and the CPU Code to account for the extra argument to the SLiC function. Use MaxIDE to examine the header-file generated by MaxCompiler after build to see how the function prototype for the SLiC function changes.

Tip: View your image by refreshing the project in MaxIDE after the simulation is complete (right-click on the project and select *Refresh* or select the project and press F5) and double-clicking on the image file that appears.

1.4 Exercise 2: Gamma correction

Adjusting the brightness in this fashion is a simple computation. A common, more complex, processing problem for correctly displaying images on computer screens is **gamma correction**. An image is usually stored with a linear intensity for each pixel (or color component thereof): we expect 0 to be black and 1 to be white, with the intensity distributed linearly in between.

However, most (CRT) monitors are imperfect and have a non-linear **transfer function** (the term for the function describing the relationship of the output image to the linear input image). The transfer function is a power function where γ (gamma) is the exponent:

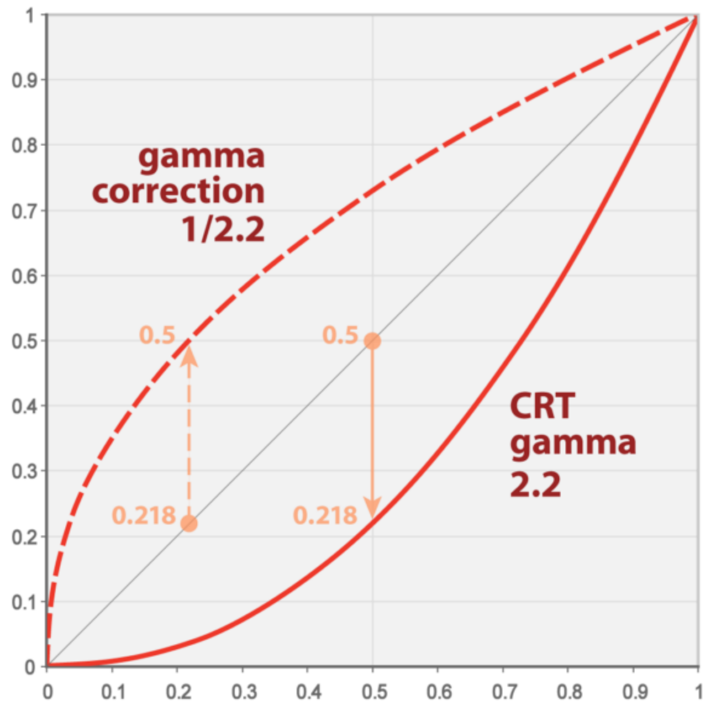


Figure 4: Graph showing the transfer function for the gamma of a CRT display and the inverse function to correct it (Source: Wikimedia Commons).

$$I(o) = I(i)^{\gamma}$$

Where $I(o)$ is the output intensity, $I(i)$ is the input intensity. A typical gamma value for a CRT display is 2.2.

We can correct for this by applying the inverse function to the image before sending it to the display:

$$I(c) = I(i)^{\frac{1}{\gamma}}$$

To get the the corrected intensity $I(c)$. *Figure 4* shows the graph for the two functions and the input image intensity.

To implement a power function in a DFE is very expensive, but we don't always need to implement a function in the DFE: we can use a **look-up table** instead, which is far more efficient. To implement an efficient look-up table, we can take advantage of the built-in memory blocks using the `mem.alloc` method:

*Memory<DFEVar> **mem**.alloc(DFEVar type, **int** depth)*

A memory can also be **mapped** to the CPU, which allows it to be accessed from the CPU code. This is enabled by using the `mapToCPU` method on a `Memory` object:

*Memory.mapToCPU(**String** name)*

This approach will create a look-up table that we can then populate with values from the CPU. The address we will use to read from the memory in the Kernel will be our pixel input itself and the size of the memory will be the color depth of our image (256 as we are using 8 bits per pixel gray-scale input).

You can read values from a memory in a Kernel using the `read()` method on the `Memory` object:

DFEVar Memory.read(DFEVar address)

The returned stream from this method contains the value addressed by the `address` argument.

As an example, let's say we needed to implement a square root function of our image. We could implement a square root function in our Kernel, but that would be expensive in resource usage terms. Instead, we can create a look up table using a ROM:

```
DFEVar inImage = io.input("inImage", dfeInt(32)) ;

Memory<DFEVar> sqrtLUT = mem.alloc(dfeInt(32), 256);
sqrtLUT.mapToCPU("sqrtLUT");

DFEVar sqrt = sqrtLUT.read(inImage.cast(dfeUInt(8)));

io.output("outImage", sqrt, sqrt.getType()) ;
```

MaxCompiler will automatically generate a new version of the SLiC Interface function for running the design, which will include an extra argument to set the contents of the ROM to the values specified via a pointer in the CPU code.

✎ **Task**: Create a look-up table in your Kernel to implement the gamma correction function and set the values in the table from CPU code. Experiment with different gamma values to see the effect. As the intensity is actually a measure from 0 to 1, you will need to divide the integer value by 255 to get the intensity to use in the power function and multiply the result by 255. Also, check that the resultant intensity is not greater 255 and set it to 255 if it is.

☞ **Tip**: The 32-bit signed integer input image stream needs to be cast to an 8-bit, unsigned integer stream when used as the address for the ROM as there are only 256 elements to address. This is done using `.cast(dfeUInt(8))` on the stream.

A test image (`test1.ppm`) is provided to more easily see the effect of the gamma correction. *Figure 12* shows Lena and the test image with and without gamma correction applied.

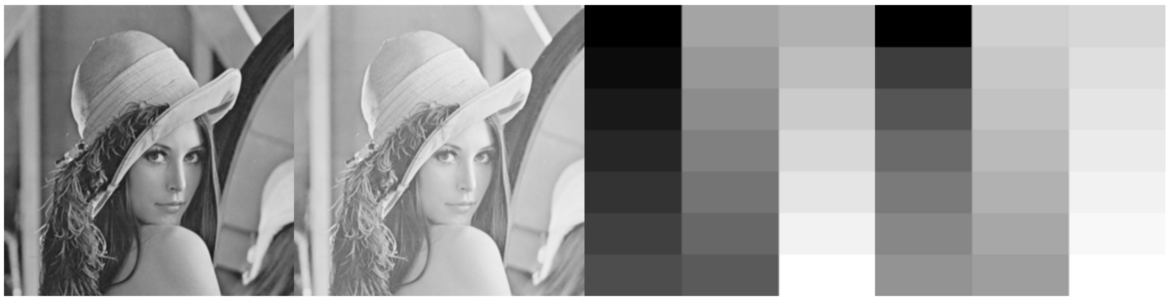

(a) Original Gray-scale Lena *(b)* Gamma-corrected Lena *(c)* Original test image *(d)* Gamma-corrected test image

Figure 5: Original and gamma-corrected images.

2 Section 2: Stream Offsets

2.1 Goals

In this section, we will use **stream offsets** which are a key feature of MaxCompiler for reading data from different locations within a stream. You will first create a Kernel that will perform a mean filter to smooth out an image. The second part of the exercise involves implementing Sobel edge detection on a gray-scale image. These image processing functions are good examples of simple convolution in MaxCompiler.

2.2 Topics Covered

- Stream offsets
- Convolution in MaxCompiler
- More complex arithmetic
- Counters
- Conditions in MaxCompiler
- Boundary conditions on convolution operations

2.3 Background

A core concept of stream computing is operating on windows into data streams. The data window is held in on-chip memory on the DFE, and data items are held on-chip for exactly the amount of time required.

Stream offsets allow us to access data elements within a stream relative to the current point. The distance from the largest to the smallest offset forms the window of data that must be held on the DFE.

A stream offset in declared using `stream.offset`:

```
DFEVar stream.offset(DFEVar src, int offset)
```

The `offset` argument is an integer where a positive value means an offset into the *future* of the stream (i.e. points that have not yet been read) and a negative value means an offset into the *past* of the stream (i.e. points that have already been read).

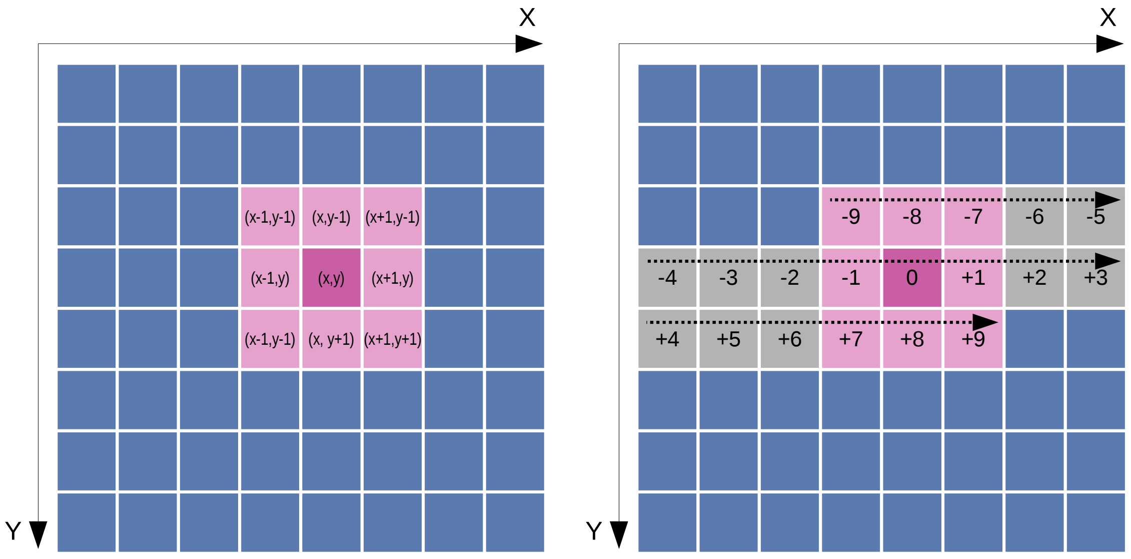
A 2D convolution operation, like the smoothing operation we are going to perform in the first part of this exercise, requires a window of the 8 points surrounding the current pixel in the image.

Figure 6 shows how the collection of points for a 2D convolution operation can be expressed in terms of offsets. The element at (x+1,y+1) is the point furthest into the *past* of the stream at an offset of −9 and the element at (x-1,y-1) is the point furthest into the *future* at an offset of +9.

The total size of the window that will be stored in the DFE in *Figure 6* amounts to 19 data items (all the items highlighted in gray or pink). The dotted arrows show the order of the arrival of the data in the incoming stream.

2.4 Exercise 1: Mean filter

A smoothing convolution operation can be carried out on an input image by simply taking the mean of the 9 points. The effective coefficients for this operation are shown in *Figure 7*.



(a) Coordinates

(b) Stream offsets

Figure 6: 2D coordinates for a convolution operation expressed as coordinates and stream offsets.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Figure 7: Effective coefficients for a mean filter.

The starting point for this exercise is a Kernel which takes an input image as a stream and connects it directly to an output.

Task: Using stream offsets, extract the 8 neighbors for each pixel in the image and output the result of a mean filter.

Figure 8 shows the result of a mean filter being applied to the gray-scale image of Lena.

You may notice that the edge of the image has a border of sorts of one pixel in width. This is because the the mean filter is operating on a 3x3 array of pixels, but at the borders of the image, some of those pixels are not valid. *Figure 9* shows the 3x3 array as applied at the edges of the image. In this instance, the offsets -9, +1 and +7 are actually read from the other side of the image, so the result of the mean filter will be incorrect.

To correct for boundary conditions such as these, we can conditionally output either the result of the

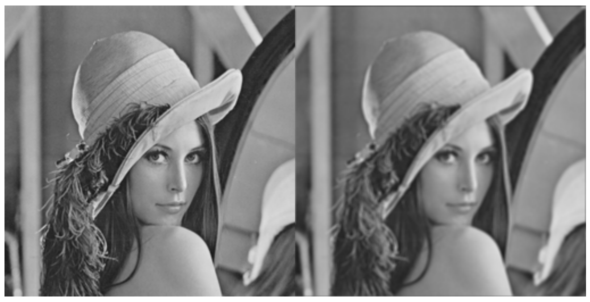


Figure 8: Original image and result of mean-filter smoothing.

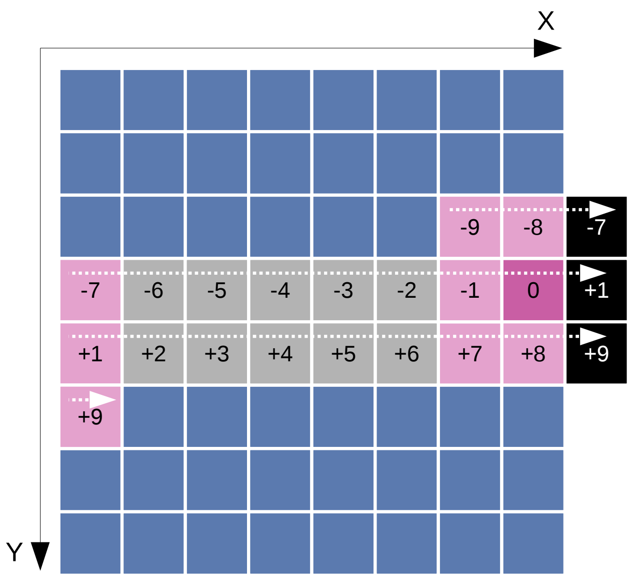


Figure 9: A convolution operation at the edges of the image showing invalid inputs.

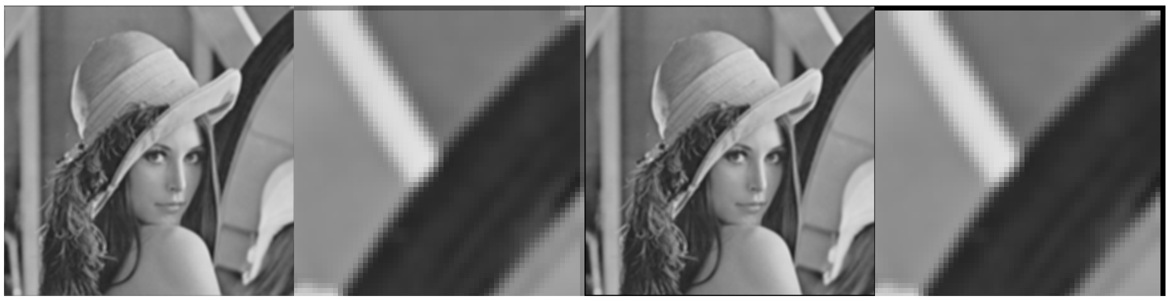
mean filter when the 3x3 input is all valid or 0 when one or more of the inputs are invalid. This will give us a one-pixel-wide border of black around our image.

Task: Fix the borders of the image by making appropriate use of counters, conditional logic, and predicates. For this exercise you may assume all images are 256*256 pixels.

2.4.1 Exercise 2: Sobel edge detection

Edge detection is an important step in feature extraction from an image. Gradient edge detection methods detect the edges by looking for the maximum and minimum in the first derivative of the image. The Sobel operator performs a 2-D spatial gradient measurement on an image.

The operator is implemented by performing 3x3 convolution using two sets of coefficients: one



(a) Smoothed Lena without boundary condition *(b)* Smoothed Lena without boundary condition (zoomed) *(c)* Smoothed Lena with boundary condition *(d)* Smoothed Lena with boundary condition (zoomed)

Figure 10: Smoothing implemented with and without boundary conditions.

for estimating the x gradient and one for estimating the y gradient. These coefficients are shown in *Figure 11*.

-1	0	+1
-2	0	+2
-1	0	+1

(a) Gx

+1	+2	+1
0	0	0
-1	-2	-1

(b) Gy

Figure 11: Sobel coefficients for calculating the gradient in x (*Figure 11a*) and y (*Figure 11b*).

The magnitude of the gradient (G) is then calculated using the formula:

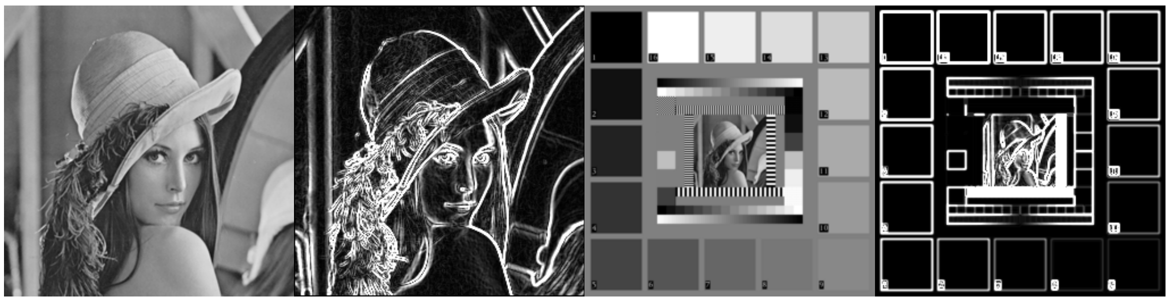
$$|G| = \sqrt{Gx^2 + Gy^2}$$

This can be approximated using:

$$|G| = |Gx| + |Gy|$$

Figure 11 shows the result of Sobel edge detection on Lena and another test image. The output image shows the magnitude of the gradient for each pixel.

Task: Implement the Sobel operator on the input image and output the magnitude of the gradient for each pixel. As before, handle the boundary cases by outputting 0 for pixels at the edge of the image.



(a) Original Gray-scale Lena *(b)* Lena with Sobel edge detection applied *(c)* Original test image *(d)* Test image with Sobel edge detection applied

Figure 12: Original images and images with Sobel edge detection applied.

2.4.2 Exercise 3: Sobel edge detection + Gaussian blur

As part of the Canny edge detection algorithm, a Gaussian blur is performed on the input image before Sobel edge detection is executed. An isotropic (i.e. circularly symmetric) 2-D Gaussian distribution has the form:

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

Where σ (sigma) is the standard deviation of the distribution (the blur strength). A larger standard deviation requires a larger number of points in the discrete implementation.

To reduce the computation complexity, a Gaussian blur can be implemented by first applying a linear Gaussian function in the x direction and then applying the same function in the y direction (to the output of the Gaussian function in x). A linear Gaussian function with a mean of 0 (centered around the x axis) is described by:

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

We can perform a 7-point linear Gaussian blur with the approximate coefficients given in *Figure 13*.

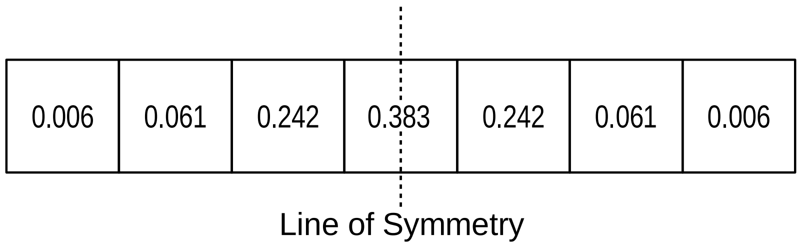


Figure 13: Coefficients for a 1D Gaussian blur.

Task: Write a single Kernel which implements a Gaussian blur followed by the Sobel edge detector. As we are using integer data in this exercise, implement the Gaussian blur by multiplying the coefficients by 1000 and dividing the result of the addition by 1000. Given the symmetrical nature of the coefficients around the central point, minimize the number of multiplies by adding the inputs before applying the coefficients.