Gaussian blur by parallel and concurrency (CUDA)

Author: Code_Monkey

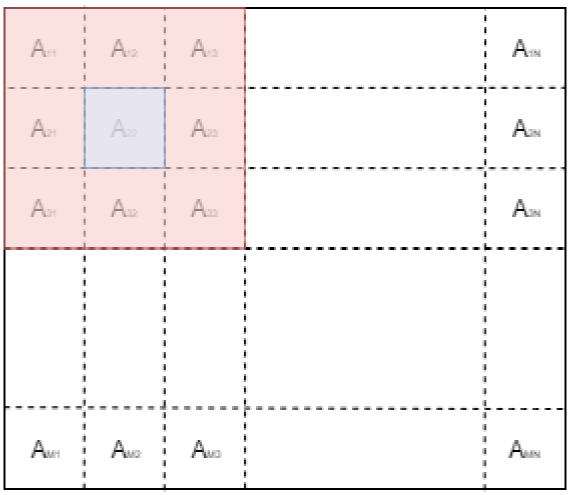
The conception of Gaussian blur:

Many algorithms that operate over images are well-suited for execution on the GPU. An image can be thought of as simply a rectangular array of pixels (image formats are a different matter, but for this coursework you will be assuming that an image is a 2D array of pixels). Your task is to compute an image blurring kernel.

Mathematically, an image blurring function calculates the value of an output image pixel as a weighted sum of a patch of pixels encompassing the pixel in the input image. For this assignment, we will be taking a simplified approach for this blurring operation: each pixel is simply the average of a patch of pixels that surrounds it (i.e. we will not place a weight on the value of each pixel, which is typical in Gaussian blur).

For an MxN input image, an example can be using a 3x3 patch as shown below. In this case, the value of the highlighted pixel A2,2 is equal to the average of the nine pixels outlined in red. Similarly, for every pixel at position (row, col), we would average a 3x3 patch spanning three rows (row-1, row, row+1) and three columns (col-1, col, col+1).

M x N input image



implement this algorithm using CUDA, and to optimize implementation with efficient use of the GPU memory model.

Briefly explain general approach in terms of dividing input into thread blocks:

For keeping the validity and correctness of image data, the best way to insert the data is by passing “source. img” (a pointer pointing address space loading all RGB value of image in CPU)as a parameter to cudaMalloc((void**)&d_img,size) and use cudaMemcpy(d_img,source.img,size,cudaMemcpyHostToDevice) to get the data from host to new address space in GPU. After that, pass the new pointer pointing this new address space which covers all image data in GPU to __global__ void blurShared(unsigned char* in, unsigned char* out, int w, int h) so that image data can be shunted into R, G, B by offset in this method:

```
pixValR += in[rgbOffset];
pixValG += in[rgbOffset+1];
pixValB += in[rgbOffset+2];
```

show the kernel run-time for 3 different grid sizes including GPU computing capability and number of SMs.

Because the limitation of thread block of the Linux in school computer (DEC 10 Lab), the size of thread block cannot become more larger than 32*32(1024).

Grid size	GPU computing capability(time) in common	GPU computing capability(time) in shared memory	Number of SMs
10	0.849824	0.545120	48*64(480/10,640/10)
20	0.618176	0.472768	24*32(480/20,640/20)
30	0.618112	0.496448	16*22(480/30,640/30+1)

Report any steps that have taken to optimise the code, and why you expect your design decisions to improve performance beyond a naive solution.

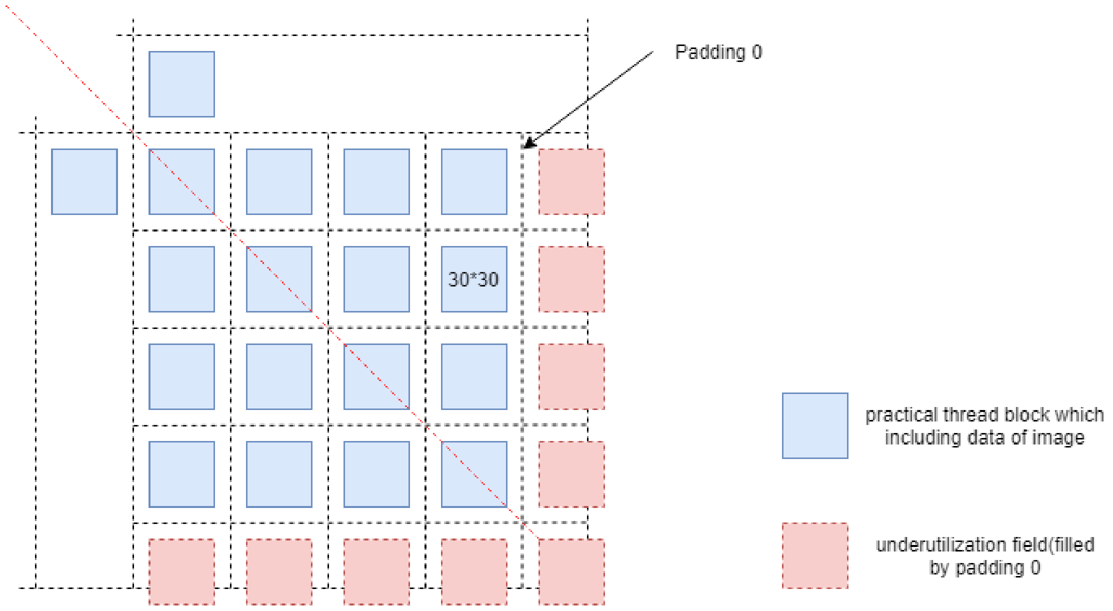
1. load all image data into shared memory and operate the image blur algorithm without any edge detection here because the speed of shared memory is faster than other type.

__global__ void blurShared (unsigned char* in, unsigned char* out, int w, int h)

2. add padding "0" in every thread block so that periphery interspace will be filed by "0" adequately.

Explanatory diagram

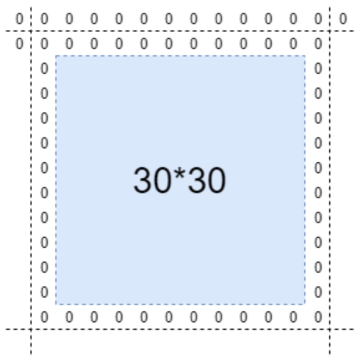
1. The diagram below is general picture about the segmentation pattern of storage resource.


Padding 0
30*30
practical thread block which including data of image
underutilization field(filled by padding 0

General picture

There are four different case for padding 0 in segmentation pattern of storage resource above.

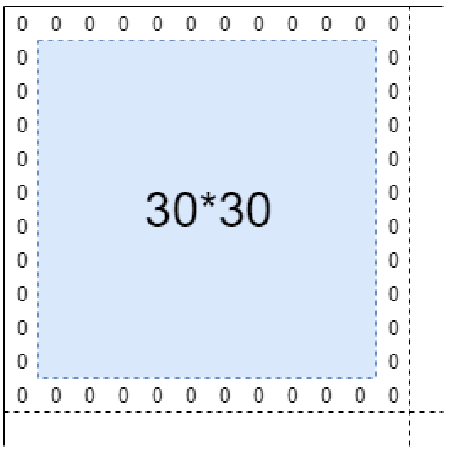
1. Center.



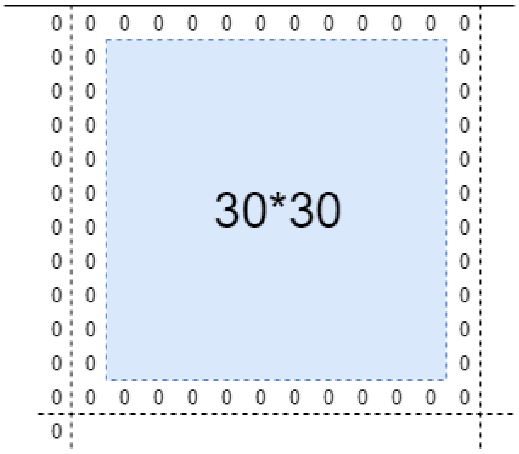
center

2. **Corner.**

3. **Edge.**

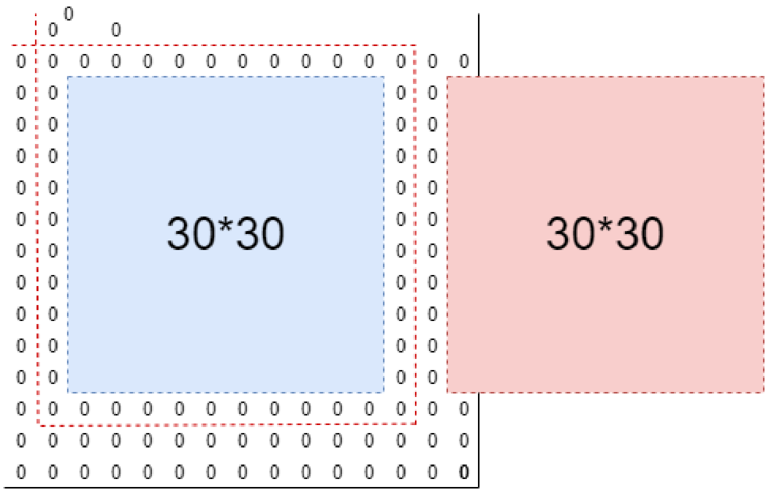


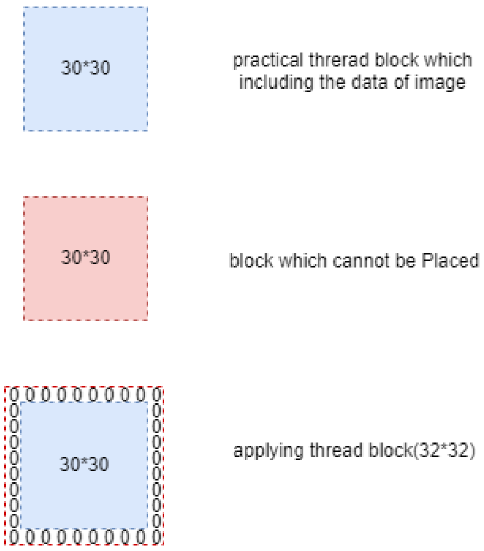
Corner



edge

3. **Underutilization in right part.**




30*30
practical threrad block which including the data of image
30*30
block which cannot be Placed
30*30
applying thread block(32*32)

underutilization

But, because the image size is 480*640, the wide and height cannot be divided exactly by block

Dim.x and Dim.y. It means there are underutilizations in SMs space which is filed by a mass of padding 0. But for the image blur algorithm, it can operate faster because doesn’t have to consider the particular case such as the situation of corner and edge of image.