“Gaussian blur algorithm” of image blurring program by CUDA

Author: Code Monkey

Purpose:

Implement Gaussian blur algorithm for image blur processing by CUDA. Try to design different implementations, basis on the architecture of Nvidia graphics cards and the parallel and concurrency theories, of this algorithm and test the running time of them for exploring how to improve the program further.

Theory:

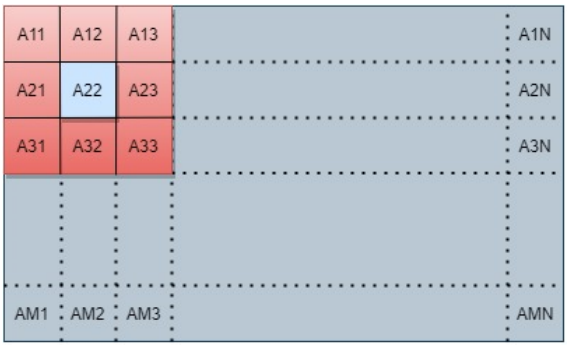
As we known, there are many different types of memory on the GPU(Nvidia):

1. Register.
2. Share memory.
3. Local memory.
4. Constant memory.
5. Texture memory.
6. Global memory.

Although they can all store data, different types of memory have different data manipulation speeds. In the code of this image blur program, we design different two different implementations of Gaussian blur algorithm by using two different types of memory” Global memory” and “Share memory”. The first implementation is general method and implemented by “Edge detection”. The second one is special method and implemented by “0 padding”. By recording and comparing the running time of program using different implementations, we can find which one is faster.

Gaussian blur algorithm:

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 (2D array of pixels). 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. A simplified approach for this blurring operation: each pixel is simply the average of a patch of pixels that surrounds it and not place a weight on the value of each pixel, which is typical in Gaussian blur. For an $M \times N$ input image, an example can be using a 3×3 patch as shown above. 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).



Implementations

1. Implementations of Gaussian blur algorithm by using “Global memory” and “Edge detection” idea.

1.1 “Edge detection” idea.

For simple and convenient, we just take small image(5×9) as an example and show the idea by chart. We get $(row, column)$ in Cartesian coordinate system for each pixel(5×9 in total) of image and calculate the $(row, column)$ of other 8 pixels in pink 9-pixel block according to the row, column of centre pixel(red one of pink 9-pixel block) of it. According to “the Gaussian blur algorithm”, we only accumulate the pixels of which: $0 \leq row \leq height$ and $0 \leq column \leq width$ of $(row, column)$ in pink 9-pixel block.

1.2 Code.

```
__global__ void blur(unsigned char * in,unsigned char* out,int w,int h){
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (col < w && row < h){
        int pixValR = 0;
        int pixValG = 0;
        int pixValB = 0;
        int count = 0;
        for(int dr = -1;dr <=1;dr++){
            for(int dc = -1;dc <=1;dc++){
                int curRow = row + dr;
                int curCol = col + dc;
                if (curRow >=0 && curRow < h && curCol >=0 && curCol < w){
                    int rgbOffset = (curRow*w + curCol)*3;
                    pixValR += in[rgbOffset];
                    pixValG += in[rgbOffset+1];
                    pixValB += in[rgbOffset+2];
                    count++;
                }
            }
        }
        int offset = (row*w+col)*3;
        out[offset] = (unsigned char)(pixValR/count);
        out[offset+1] = (unsigned char)(pixValG/count);
        out[offset+2] = (unsigned char)(pixValB/count);
    }
}
```

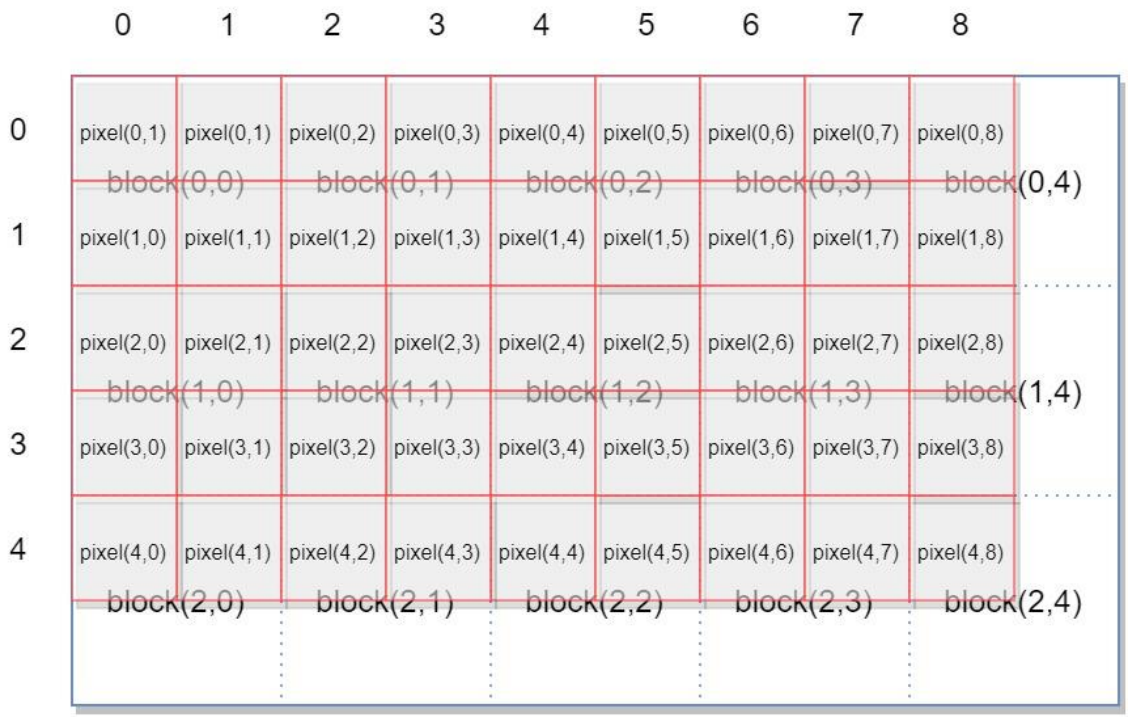
As the code above, the code in red loop calculates the $(row, column)$ of 8 pixels in pink 9-pixel block according to the row, column of centre pixel(red one of pink 9-pixel block) in Cartesian coordinate system and the code in blue loop just does the Edge detection.

2. Implementations of Gaussian blur algorithm by using “Share memory” and “0 padding” idea.

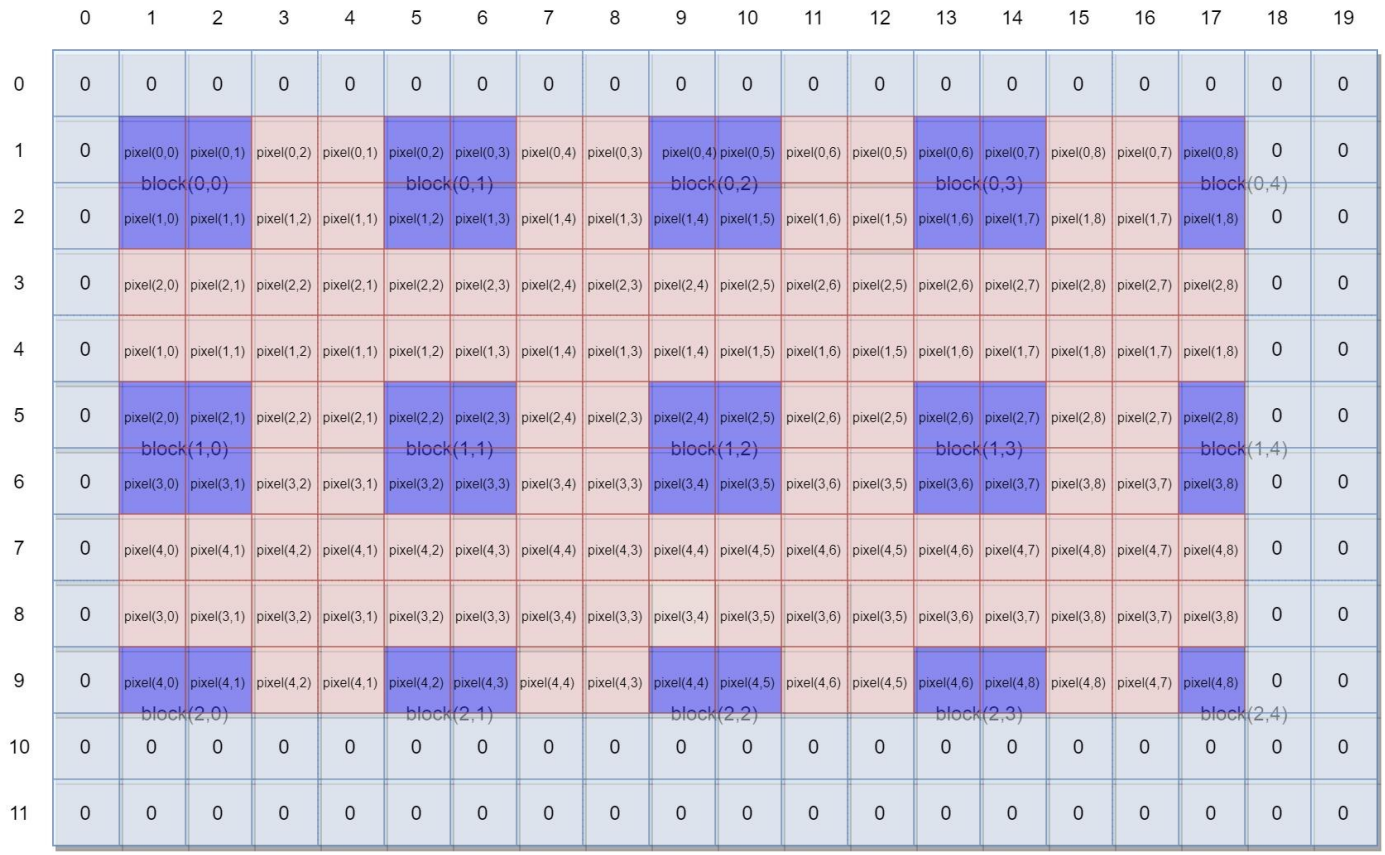
2.1 “0 padding” idea.

Before introducing the idea of “0 padding”, we need to discuss the distribution of 2D pixels in image. For simple and convenient, we just take small image(5×9) as an example and show the idea by chart.

In the code about “Edge detection” Implementations of Gaussian blur algorithm, distribution of 2D pixels of image in GPU(Nvidia) is as below:



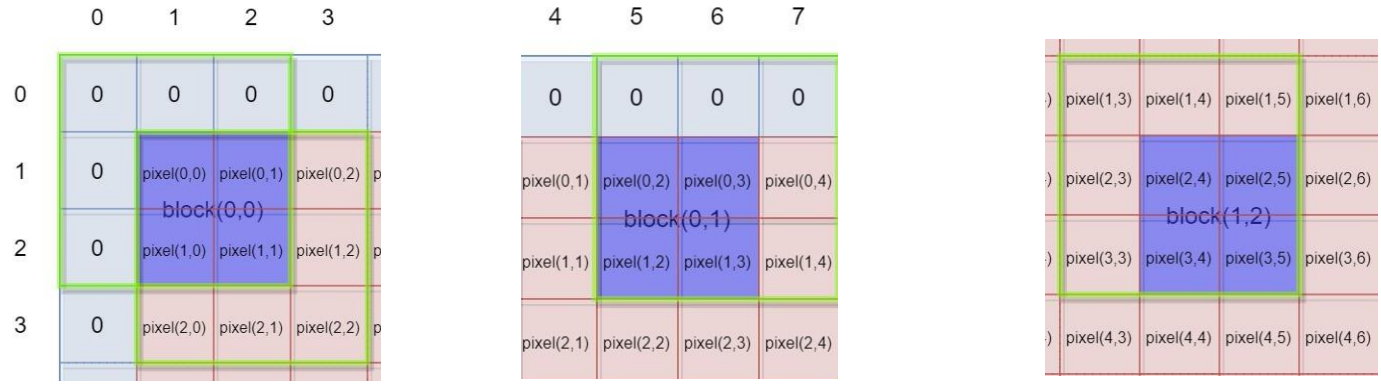
But in the code about “0 padding” Implementations of Gaussian blur algorithm, distribution of 2D pixels of image in GPU(Nvidia) is as below:



As we can see, there are A large number of “0” filling in blocks. The pixels with purple colour are the real image pixels and each of them will be treated as centre pixel and execute the Gaussian blur algorithm on them. At that time, we don’t need to detect whether pixel is on

the edge of image or not because image pixels surrounded by “0”. We will prove it as shown below.

Corner: Edge: Inside:



As we can see, the calculation of 9-pixel block(inside green loop) when each real pixel of image is treated as centre pixel is right because adding “0” means to add nothing.

2.2 Code.

```
__global__ void blurShared(unsigned char* in, unsigned char* out, int w, int h) {
    int filterRow, filterCol;
    int cornerRow, cornerCol;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int bx = blockIdx.x; int by = blockIdx.y;
    int bdx = blockDim.x; int bdy = blockDim.y;
    int row = by * (bdy - 2) + ty;
    int col = bx * (bdx - 2) + tx;
    if ((row < h + 1) && (col < w + 1)) {
        __shared__ unsigned char tile[AUGMENTED_Y][AUGMENTED_X][3];
        int imgRow = row - 1;
        int imgCol = col - 1;
        if ((imgRow < h) && (imgCol < w) && (imgRow >= 0) && (imgCol >= 0)) {
            int rgbOffset = (imgRow*w + imgCol)*3;
            tile[ty][tx][0] = in[rgbOffset];
            tile[ty][tx][1] = in[rgbOffset+1];
            tile[ty][tx][2] = in[rgbOffset+2];
        }
        else {
            tile[ty][tx][0] = 0;
            tile[ty][tx][1] = 0;
            tile[ty][tx][2] = 0;
        }
        __syncthreads();
        int pixValR = 0;
        int pixValG = 0;
        int pixValB = 0;
        int count = 0;
        if ((tx >= 1) && (ty >= 1) && (ty < bdy - 1) && (tx < bdx - 1)) {
            cornerRow = ty - 1;
            cornerCol = tx - 1;
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    filterRow = cornerRow + i;
                    filterCol = cornerCol + j;
                    if ((filterRow >= 0) && (filterRow <= h) && (filterCol >= 0) && (filterCol <= w)) {
                        pixValR += tile[filterRow][filterCol][0];
                        pixValG += tile[filterRow][filterCol][1];
                        pixValB += tile[filterRow][filterCol][2];
                        count++;
                    }
                }
            }
            int offset = (imgRow*w + imgCol)*3;
            out[offset] = (unsigned char)(pixValR/count);
            out[offset+1] = (unsigned char)(pixValG/count);
            out[offset+2] = (unsigned char)(pixValB/count);
        }
    }
}
```

As code above, the codes marked by read loop mean they are the Important part of “0 padding” implementation of Gaussian blur algorithm.

CUDA technology

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 blur(unsigned char * in,unsigned char* out,int w,int h) or __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];

Test

Our image size is 480×640. 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)

Conclusion

“0” padding implementation of Gaussian blur algorithm is faster than “Edge detection” implementation of Gaussian blur algorithm.