Report 2D Convolution
Parallel Programming for Machine Learning project

Chisci Marco
marco.chisci@edu.unifi.it

Abstract

This report details the parallelization of a sequential 2D convolution algorithm using CUDA. After a description of the image data used for testing and an overview of the 2D convolution process, the document presents both the sequential and CUDA-based implementations of 2D convolution. Various experiments were conducted to measure the runtime performance of the different implementations, analyzing speedup based on image size, block size, kernel size, constant memory kernel and use of tiling.

1. Introduction

This report presents the work I did to parallelize a previously sequential version of 2D convolution using CUDA.

The following sections will contain:

- Section 2: a description of the images used in the experiments.
- Section 3: an overview of 2D convolution, focusing specifically on image blurring.
- Section 4: detailed explanation of both the sequential and CUDA implementations of the 2D convolution.
- Section 5: a comprehensive analysis of the experiments to measure the runtime of the 2D convolution across different configurations.
  The main goal of this section is to evaluate the potential speedup achieved through the use of a GPU.

2. Used Data

A basic grayscale image of a lynx was used to visualize the results of the 2D convolution.

For performance testing of the different 2D convolution implementations (both sequential and CUDA), randomly generated images of varying sizes were utilized. These images, created using the *generateImage* function, are filled with random integers representing pixel intensities, based on the specified input width and height.

3. 2D Convolution, blurring

2D convolution involves taking an input matrix that represents an image and a smaller square matrix of weights, known as a kernel. An elementwise multiplication of the mutually overlapping pixels is performed, and the resulting values are summed. This process is repeated for each pixel in the input matrix, with the kernel being centered on each one.

When all the weights in the kernel are set to 1/(kernel size * kernel size), the convolution functions as a box blur. The effect of this type of convolution can be seen in Fig. 1 and Fig. 2.

4. Code

4.1. Code structure

The files listed below, available in the GitHub repository, will be utilized in the upcoming experiments.

- **Image.cu/cuh**: define and implement the Image and Kernel structures along with their associated methods.
- **SequentialConvolution.cu/cuh**: provides a simple sequential implementation of 2D convolution.
- **GPUConvolution.cu/cuh**: contains two CUDA-based versions of 2D convolution (both with or without the kernel in the constant memory).
- **main.cu**: the executable file that tests and compares the different convolution implementations using various image sizes, CUDA blocks, and kernels.

4.2. Image.cu/cuh

These files define and implement the Image and Kernel structures, along with their associated functions.

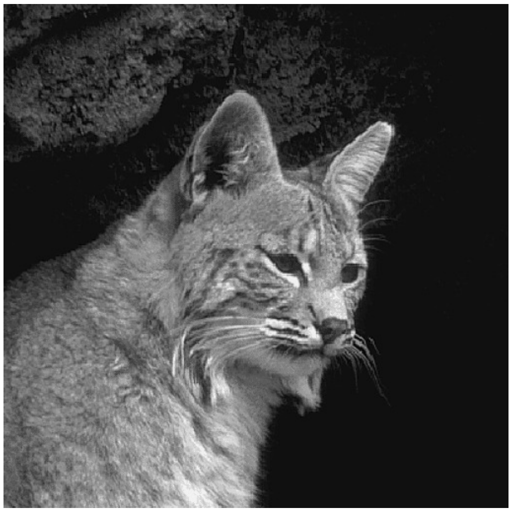


Figure 1. Grayscale lynx

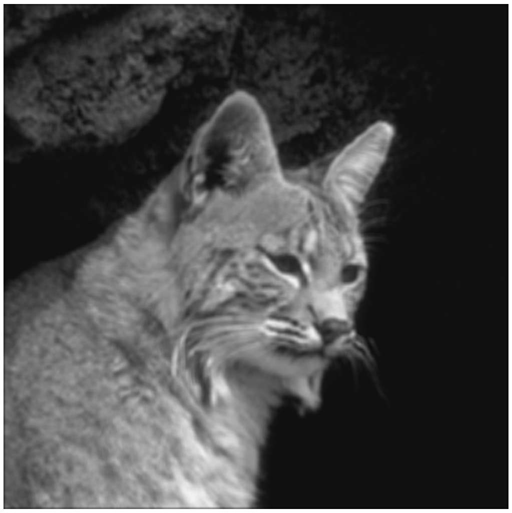
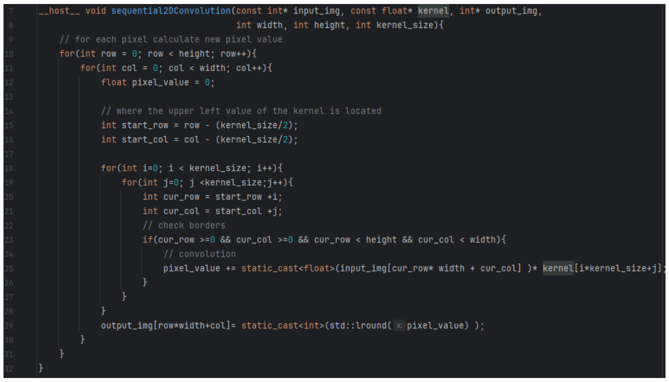


Figure 2. Blurred grayscale lynx

Each Image has its own width, height and array of integers representing the grayscale intensity of each pixel.
The function *getImageFromFile* creates an Image object by extracting data from a PNG file, while *saveImageToFile* does the opposite.
To ensure accuracy, *compareImages* is used during testing to check whether the output image of each implementation of 2D convolution is the same. It does that by simply comparing their height, width and pixels.
copyImage is used to create a copy of the input image. This copy will be used as the output image that will be modified in the 2D convolution.

A Kernel object consists of an integer representing its size and an array of floating-point values containing its weights.



```
__host__ void sequential2DConvolution(const int* input_img, const float* kernel, int* output_img,
                                      int width, int height, int kernel_size){
    // for each pixel calculate new pixel value
    for(int row = 0; row < height; row++){
        for(int col = 0; col < width; col++){
            float pixel_value = 0;

            // where the upper left value of the kernel is located
            int start_row = row - (kernel_size/2);
            int start_col = col - (kernel_size/2);

            for(int i=0; i < kernel_size; i++){
                for(int j=0; j <kernel_size;j++){
                    int cur_row = start_row +i;
                    int cur_col = start_col +j;
                    // check borders
                    if(cur_row >=0 && cur_col >=0 && cur_row < height && cur_col < width){
                        // convolution
                        pixel_value += static_cast<float>(input_img[cur_row* width + cur_col] )* kernel[i*kernel_size+j];
                    }
                }
            }
            output_img[row*width+col]= static_cast<int>(std::lround( pixel_value) );
        }
    }
}
```

Figure 3. sequential2DConvolution()

The function *generateBlurKernel* generates a Kernel object that, when used in a convolution, blurs the image, as described earlier.

The other functions are used to manage the host and device memory and are all pretty self-explanatory.

4.3. SequentialConvolution.cu/cuh

The main function in this file is *sequential2DConvolution* Fig. 3, which provides a straightforward sequential implementation of the 2D convolution, as described earlier.

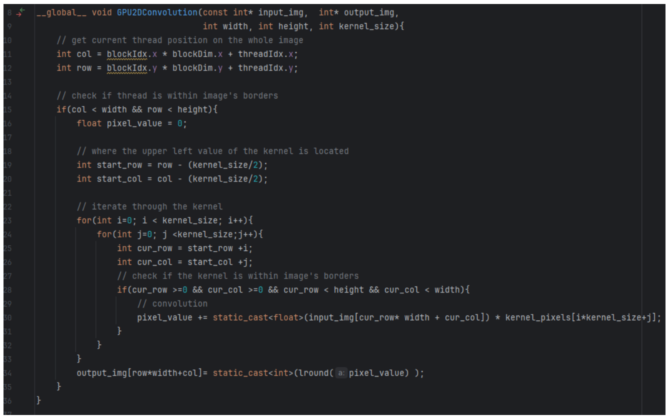
This function uses two for loops to iterate over every pixel of the input image. For each pixel, it calculates the intensity of the output pixel given the weights of the kernel, while being aware of the image's edges. When a weight of the kernel would be placed outside the input image, it is treated as zero.

4.4. GPUConvolution.cu/cuh

This file contains four functions: *GPU2DConvolution* and *GPU2DConvolutionTiling*, both with their respective versions where the kernel is not stored in constant memory.

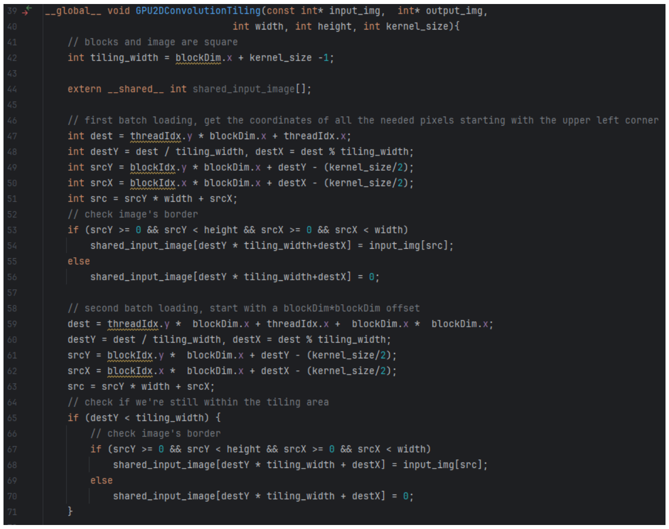
GPU2DConvolution Fig. 4 is a simple CUDA-based 2D convolution. Each thread gets his current pixel position (row and column in the image) and, after verifying it is within the image edges, calculates the output value of his pixel by iterating through the valid kernel values.
GPU2DConvolution_kernel is the same function, but the kernel values are passed as input, as they are not stored in constant memory.

GPU2DConvolutionTiling is a 2D CUDA-based convolution with tiling. Given a CUDA block, the pixels within that block share the need for a lot of the same neighborhood pixels. To reduce the execution time, the need to ac-



```
__global__ void GPU2DConvolution(const int* input_img,  int* output_img,
                                int width, int height, int kernel_size){
    // get current thread position on the whole image
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // check if thread is within image's borders
    if(col < width && row < height){
        float pixel_value = 0;

        // where the upper left value of the kernel is located
        int start_row = row - (kernel_size/2);
        int start_col = col - (kernel_size/2);

        // iterate through the kernel
        for(int i=0; i < kernel_size; i++){
            for(int j=0; j <kernel_size;j++){
                int cur_row = start_row +i;
                int cur_col = start_col +j;
                // check if the kernel is within image's borders
                if(cur_row >=0 && cur_col >=0 && cur_row < height && cur_col < width){
                    // convolution
                    pixel_value += static_cast<float>(input_img[cur_row* width + cur_col]) * kernel_pixels[i*kernel_size+j];
                }
            }
        }
        output_img[row*width+col]= static_cast<int>(lround( pixel_value) );
    }
}
```

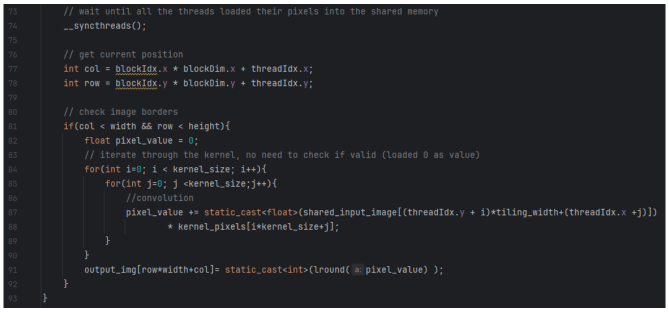
Figure 4. GPU2DConvolution()



```
__global__ void GPU2DConvolutionTiling(const int* input_img,  int* output_img,
                                int width, int height, int kernel_size){
    // blocks and image are square
    int tiling_width = blockDim.x + kernel_size -1;

    extern __shared__ int shared_input_image[];

    // first batch loading, get the coordinates of all the needed pixels starting with the upper left corner
    int dest = threadIdx.y * blockDim.x + threadIdx.x;
    int destY = dest / tiling_width, destX = dest % tiling_width;
    int srcY = blockIdx.y * blockDim.x + destY - (kernel_size/2);
    int srcX = blockIdx.x * blockDim.x + destX - (kernel_size/2);
    int src = srcY * width + srcX;
    // check image's border
    if (srcY >= 0 && srcY < height && srcX >= 0 && srcX < width)
        shared_input_image[destY * tiling_width+destX] = input_img[src];
    else
        shared_input_image[destY * tiling_width+destX] = 0;

    // second batch loading, start with a blockDim*blockDim offset
    dest = threadIdx.y *  blockDim.x + threadIdx.x +  blockDim.x *  blockDim.x;
    destY = dest / tiling_width, destX = dest % tiling_width;
    srcY = blockIdx.y *  blockDim.x + destY - (kernel_size/2);
    srcX = blockIdx.x *  blockDim.x + destX - (kernel_size/2);
    src = srcY * width + srcX;
    // check if we're still within the tiling area
    if (destY < tiling_width) {
        // check image's border
        if (srcY >= 0 && srcY < height && srcX >= 0 && srcX < width)
            shared_input_image[destY * tiling_width + destX] = input_img[src];
        else
            shared_input_image[destY * tiling_width + destX] = 0;
    }
```

Figure 5. GPU2DConvolutionTiling() loading steps

cess global memory can be limited if all the pixels within a block and the pixels that are reached by the kernel are in the shared memory.
Assuming that $(blocksize+kernelsize-1)*(blocksize+kernelsize-1)$ is smaller than $2*blocksize*blocksize$ all the pixel within kernel reach can be loaded in two steps. Each thread has a first loading step where the first $(blocksize*blocksize)$ pixels are loaded into shared memory and a second step where, after checking the "tiling zone" limits, the remaining pixels are loaded Fig. 5.
Once each pixel within the "tiling zone" is in the shared memory, the threads are synched and each thread calculates the output value of his assigned pixel using the values in the shared memory Fig. 6.
As before, *GPU2DConvolutionTiling_kernel* is the version where the kernel is not stored in constant memory.



```
    // wait until all the threads loaded their pixels into the shared memory
    __syncthreads();

    // get current position
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // check image borders
    if(col < width && row < height){
        float pixel_value = 0;
        // iterate through the kernel, no need to check if valid (loaded 0 as value)
        for(int i=0; i < kernel_size; i++){
            for(int j=0; j <kernel_size;j++){
                //convolution
                pixel_value += static_cast<float>(shared_input_image[(threadIdx.y + i)*tiling_width+(threadIdx.x +j)])
                        * kernel_pixels[i*kernel_size+j];
            }
        }
        output_img[row*width+col]= static_cast<int>(lround( pixel_value) );
    }
}
```

Figure 6. GPU2DConvolution() convolution step

4.5. main.cu

This is the executable file used to test the functions described earlier

The *time* functions are responsible for measuring the execution time of the different 2D convolution implementations, after managing the host and device memory correctly.

The *test* functions run the sequential and CUDA *time* functions multiple times, calculating the average execution times and speedups after allocating the required images and kernel.

The results are saved in a txt file.

If the CUDA-based 2D convolution produces the same blurred output as the sequential version when using the same kernel, it is considered correct.

5. Experiments and Results

All the following experiments were conducted on the SSH server 'papavero.dinfo.unifi' using CLion as the IDE and nvcc as the compiler.

Unless stated otherwise, in each experiment the default size of the image is 8192, the block size is 16, the kernel size is 5 and it is stored in the constant memory.

5.1. Images of different sizes

The first experiment examines the speedup (calculated as sequential execution time divided by GPU execution time) as the input image size increases.

Tab. 1 displays the speedups obtained using either *GPU2DConvolution* or *GPU2DConvolutionTiling* as the width of the input image grows. Since a larger image contains exponentially more pixels, each requiring its own convolution, the sequential version becomes slower. With CUDA each output pixel can be processed independently,

Image width	GPU	GPU with tiling
8	0.756137	0.786471
16	3.29623	3.33136
32	12.6697	13.1244
64	50.7918	54.0384
128	185.18	192.461
256	512.273	557.868
512	649.236	804.402
1024	827.704	900.158
2048	870.965	1012.16
4096	919.735	1027.49
8192	996.311	1086.65
16384	989.53	1090.95
32768	1021.57	1158.86

Table 1. Speedups obtained by changing the size of the randomly generated image

since 2D convolution is an embarrassingly parallel problem.
As the image's size increases, a higher percentage of GPU threads are utilized, and so does the speedup until each thread is used.

Loading the used pixels into the shared memory does improve the performances of the CUDA-based 2D convolution, as the tiling version consistently achieves better speedup than the non-tiling version.

5.2. Changing the size of the CUDA blocks

As previously stated, the block size in *GPU2DConvolutionTiling* is constrained by the assumption that $(blocksize + kernelsize - 1) * (blocksize + kernelsize - 1)$ is smaller than $2 * blocksize * blocksize$.
With a default kernel size of 5, the minimum allowable block size is 10.
The size of each grid is: $(imagesize + blocksize - 1)/blocksize$ x $(imagesize + blocksize - 1)/blocksize$ to avoid wasting threads, since if they are outside the image they do not compute any useful value.

Fig. 7 show how speedup changes with different block sizes, which affects the number of threads per block. As expected, blocks that are too small do not provide enough threads to fully utilize the GPU, and too many threads per block leads to fewer per-SM hardware resources available to each thread, not enough blocks fit in a single SM. The speedup peaks at block size 16, since enough threads are allocated at the same time, keeping the GPU fully utilized with no wasted thread since 8192 is a multiple of 16.

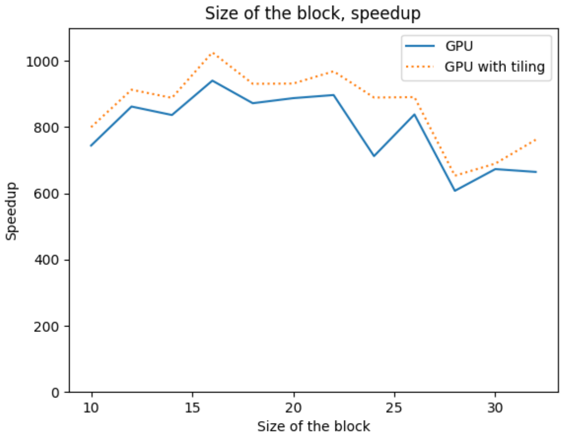


Figure 7. Speedup as the block size changes

Once again, the performances of the CUDA-based 2D convolution with tiling consistently outperforms the non-tiling version.

5.3. Kernels of different sizes

The assumption that $(blocksize + kernelsize - 1) * (blocksize + kernelsize - 1)$ is smaller than $2 * blocksize * blocksize$ also limits kernel size.
With a block size of 16 the maximum kernel size is 7.

Fig. 8 displays the speedups for varying kernel sizes in both CUDA-based 2D convolution methods with a block size of 16. As the kernel size grows, the number of multiplications needed for each pixel exponentially increases, making the sequential version slower. As expected, the speedup of both CUDA-based functions improves with larger kernel sizes, once again benefitting from tiling.

Fig. 9 presents similar results as the previous figure, but with a block size of 32, making it feasible to use bigger kernels. The trends remain the same, with an even higher speedup gap between the tiling and non tiling versions of the 2D CUDA convolutions at kernel size 9. This is because a larger kernel involves more pixels in the computation of each output pixel. Without tiling, the need for global memory access per operation increases with the kernel size.

5.4. Constant memory kernel vs global memory kernel

The final experiment evaluates the performance of the two CUDA-based 2D convolution methods compared to their versions where the kernel is stored in global memory.

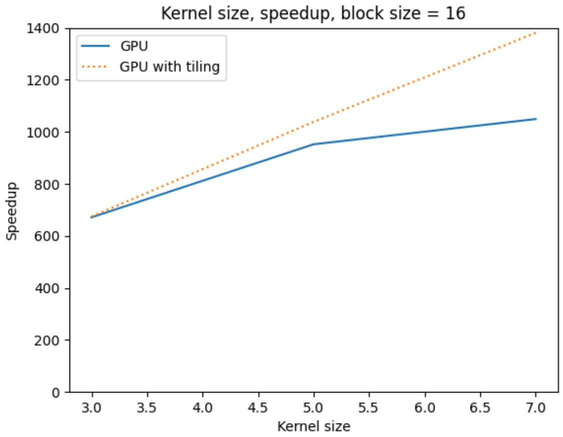


Figure 8. Speedup for varying kernel sizes with a block size of 16

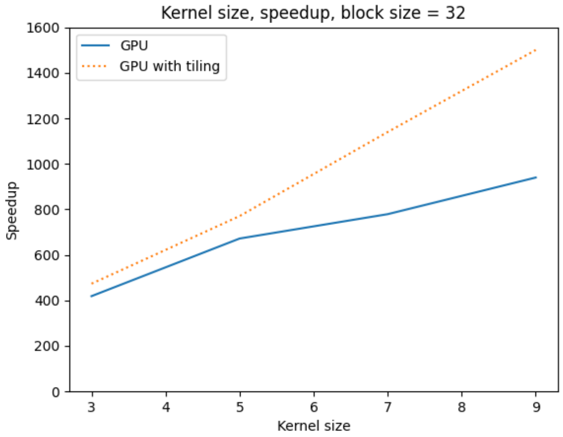


Figure 9. Speedup for varying kernel sizes with a block size of 32

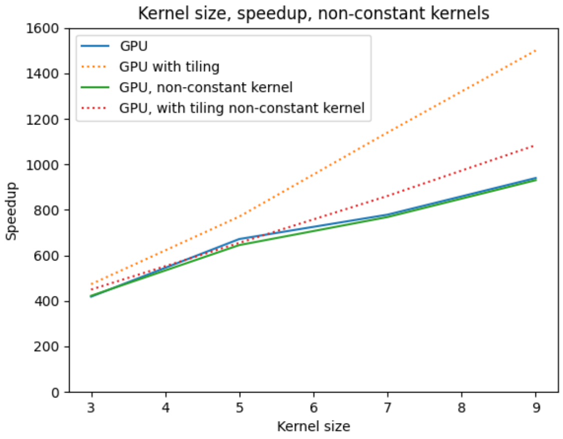


Figure 10

Fig. 10 show the speedups obtained by the four different 2D CUDA convolutions as the kernel size changes. As discussed earlier, tiling has the biggest impact on the speedup when the kernel size gets bigger.

When comparing the two convolutions with tiling, is it clear that storing the kernel in the global memory negatively affects performance, as the number of global memory accesses needed per operation increases significantly with larger kernel sizes.
For the non-tiling convolutions, the results are similar, since the number of global memory access is not greatly increased by the different kernel storage.