Convolution: sequential version vs parallel version

Bruno Martelli
E-mail address
bruno.martelli@edu.unifi.it

Abstract

This report presents a comprehensive analysis of the benefits and challenges of parallelizing the convolution algorithm. It examines the transition from sequential to parallel implementation, evaluating its impact on computational efficiency. Key performance metrics such as execution time and speedup are analyzed, with particular attention to the effects of tile dimensions and convolution mask size.

1. Introduction

Convolution is one of the fundamental operations in image processing and deep learning, used to apply filters that can detect edges, blur, or other effects. The basic algorithm requires that for each pixel of the image, a mask (or kernel) is applied that combines the values of nearby pixels according to specific weights.

1.1. Two-dimensional convolution

The two-dimensional convolution is defined by the following formula:

$$(I * K)(x, y) = \sum_{i,j} I(x - i, y - j) \cdot K(i, j)$$

where I represents the input image, K is the convolution mask, and (x, y) are the coordinates of the pixel in the output image.

1.2. Sequential approach

In the sequential implementation, each pixel is processed one after another, using a triple nested loop to go through all the pixels of the image and another double loop to apply the convolution mask to each pixel.

1.3. Parallel approach

The parallel implementation uses CUDA to take advantage of the computing power of GPUs. Tiles represent blocks of pixels in the image that are processed simultaneously by parallel threads, using shared memory to optimize data access.

2. Code Implementation

Different versions of the convolution algorithm have been implemented, as described in the following subsections.

2.1. Sequential Implementation

The sequential implementation processes each pixel of the image one by one and applies the convolution mask using nested loops.

Listing 1. Sequental Update with SoA and Fast

```
void sequential_convolution(const float* input,
    float* output,const float* mask,int width,int
     height,int channels,int mask_width) {
 int mask_radius = mask_width / 2;
 for (int c = 0; c < channels; c++){
   for (int y = 0; y < height; y++){
     for (int x = 0; x < width; x++){
       float sum = 0.0f;
       for (int i = 0; i < mask_width; i++){
         for (int j = 0; j < mask_width; j++){
           int img_y = y + i - mask_radius;
           int img_x = x + j - mask_radius;
             if(img_y >= 0 &&
               img_y < height &&
```

```
                img_x >= 0 &&
                img_x < width) {
                    sum += input[(img_y * width +
                        img_x) * channels + c] *
                        mask[i * mask_width + j];
                }
            }
        }
        output[(y * width + x) * channels + c
            ] = sum;
    }
  }
 }
}
```

2.2. Parallel Implementation with Shared Memory

The parallel implementation uses CUDA and shared memory to optimize data access. This version adopts a tile-based approach to process blocks of pixels simultaneously.

Listing 2. Sequental Update with SoA and Fast

```
__global__ void parallel_convolutionV2(const
    float* input, float* output, const float*
    __restrict__ mask, int width, int height, int
     channels, int Mask_width, int TILE_WIDTH) {

    extern __shared__ float N_ds[];

    int Mask_radius = Mask_width / 2;
    int w = TILE_WIDTH + Mask_width - 1;
    int k = blockIdx.z;
    int numThreadsNeeded = w * w;
    int threadsPerBlock = blockDim.x * blockDim.y
        ;
    int iterations = (numThreadsNeeded +
        threadsPerBlock - 1) / threadsPerBlock;
    for (int iter = 0; iter < iterations; iter++)
         {
        int loadIndex = iter * threadsPerBlock +
            threadIdx.y * blockDim.x + threadIdx.
            x;
        if (loadIndex < numThreadsNeeded) {
            int destY = loadIndex / w;
            int destX = loadIndex % w;
            int destIdx = destY * w + destX;

            int srcY = blockIdx.y * TILE_WIDTH +
                destY - Mask_radius;
            int srcX = blockIdx.x * TILE_WIDTH +
                destX - Mask_radius;
            int src = (srcY * width + srcX) *
                channels + k;

            if (srcY >= 0 && srcY < height &&
                srcX >= 0 && srcX < width) {
                N_ds[destIdx] = input[src];
            } else {
                N_ds[destIdx] = 0;
            }
        }
    }

    __syncthreads();

    int row = blockIdx.y * TILE_WIDTH + threadIdx
        .y;
    int col = blockIdx.x * TILE_WIDTH + threadIdx
        .x;

    if (row < height && col < width) {
        float sum = 0.0f;
        for (int i = 0; i < Mask_width; i++) {
            for (int j = 0; j < Mask_width; j++)
                {
                int y = threadIdx.y + i;
                int x = threadIdx.x + j;


                if (y < w && x < w) {
                    sum += N_ds[y * w + x] * mask
                        [i * Mask_width + j];
                }
            }
        }
        output[(row * width + col) * channels + k
            ] = sum;
    }
}
```

3. Experimental Results

In this section, we present a series of experiments conducted on the implemented code, varying the size of the mask and tiles.

3.1. Setup

The tests were conducted on a PC equipped with a Ryzen 7 3800X 8-core processor running Windows 11 Pro 24H2, 32GB of RAM, and an NVIDIA GeForce RTX 3070 GPU. For each configuration, the execution time was measured using the CUDA timer for the parallel version and the standard C++ clock for the sequential version. All tests are done on a jpeg image of 1333X2000 pixel.

3.2. Time Measurement

For parallel execution, we observe that the measured speedup can vary significantly depending on which time intervals are considered. As shown in tab.1, if we only take into account the time the GPU spends computing the convolution, the speed-up appears much larger. However, if we also consider the time required to transfer data between the CPU and GPU, the speedup is notably lower.

Since, in the sequential implementation, the final result is already available in RAM at the end of execution, we decided to include the data transfer time (from GPU to RAM and vice versa) in the total execution time for a fair comparison in the subsequent analysis.

Method	Sequential	Parallel	Speedup
Data transfer	259.133	10.379	24.967
No data transfer	259.133	0.688	376.598

Table 1. Comparison of Sequential and Parallel Time with and without Data transfer with mask 5x5 and tile 16x16

3.3. Impact of Tile Size

To evaluate the impact of tile size on performance, tests were conducted with a fixed mask size of 5×5 and varying tile sizes (8×8, 12×12, 16×16, 20×20, 24×24, 28×28, 32×32). Each test was repeated 30 times, and the average execution time was recorded.

Tile Size	Sequential (ms)	Parallel (ms)	Speedup
8	243.900	9.817	24.844
12	243.900	9.621	25.349
16	243.900	9.594	25.423
20	243.900	9.715	25.105
24	243.900	9.660	25.250
28	243.900	10.021	24.337
32	243.900	9.802	24.882

Table 2. Comparison of Sequential and Parallel Execution Times for Different Tile Sizes

The results indicate that optimal performance was achieved with a tile size of 16×16, which yielded the highest speedup factor of 25.423. However, the variation in performance across different tile sizes was relatively minimal. This small difference in performance suggests that the primary bottleneck in this implementation is data transfer overhead, not computational complexity.

3.4. Impact of Mask Size

To evaluate the impact of mask size, tests were conducted with fixed tile size of 16×16 and varying mask sizes (3×3, 5×5, 7×7, 9×9, 11×11, 13×13, 15×15). Each test was repeated 30 times, and the average execution time was recorded.

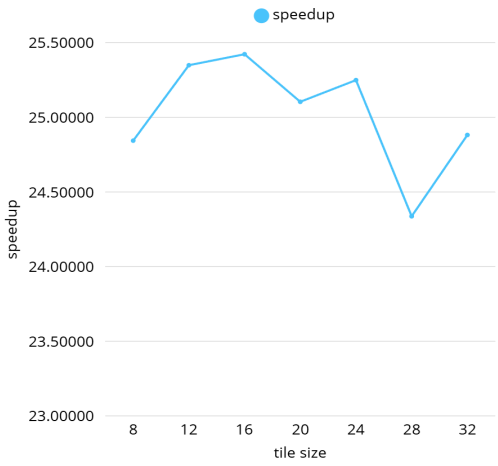

Figure 1. Effect of tile size

Mask Size	Sequential (ms)	Parallel (ms)	Speedup
3	111.600	9.812	11.374
5	244.600	14.886	16.432
7	446.000	20.043	22.252
9	633.633	17.247	36.738
11	923.600	19.738	46.793
13	1216.933	19.427	62.641
15	1595.233	17.522	91.042

Table 3. Comparison of Sequential and Parallel Execution Times for Different Mask Sizes

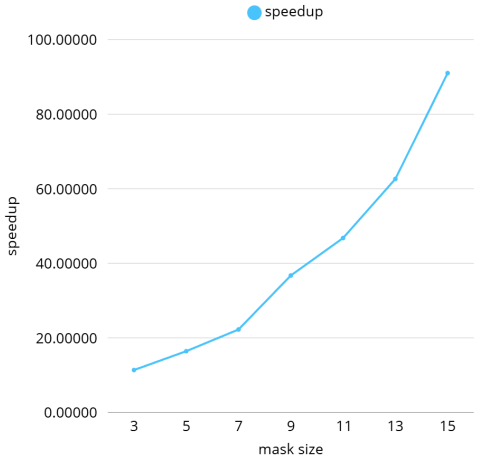

Figure 2. Effect of Mask size

The results show a strong correlation between mask size and achieved speedup. As the mask size increases, there is a notable improvement in speedup performance, with the largest mask size (15×15) achieving an 91.042× acceleration

over the sequential implementation.This performance gain is due to the efficient use of the GPU's computational capabilities in the parallel version, which becomes increasingly advantageous as the computational complexity rises with larger mask sizes. While the sequential execution time grows quadratically with mask size, the parallel implementation maintains relatively stable execution times, indicating effective parallelization of the convolution operations.

4. Final discussion

The experiments reveal two simple but important lessons about the use of GPUs for image processing. First, the way we divide the work into blocks (tile sizes) makes surprisingly little difference in speed. Whether using small 8x8 blocks or larger 32x32 blocks, the results stayed nearly the same. This happens because the biggest slowdown is not the calculations themselves; it is the time spent moving data between the computer's memory and the GPU.

Seconds, the advantages of GPUs become particularly evident when handling computationally intensive tasks. Smaller filters, such as the masks 3×3, provide modest speed-up (approximately 11 times faster), while larger masks, such as 15×15, result in significantly higher performance gains (up to 91 times faster).