Image Processing: 2D-Convolution with CUDA and OpenMP

Dragos Tanasa
E-mail address
dragos.tanasa99@stud.unifi.it

Abstract

Convolution stands as a cornerstone operation within signal processing, image analysis and deep learning. Specifically, discrete convolution finds its application in image processing, where it is employed to wield filters upon images or to preprocess them prior to executing machine learning tasks. Within the domain of deep learning, convolution emerges as a fundamental operation within CNNs proving its significance during both training and inference phases. As the demand for real-time processing of extensive datasets continues to surge, the efficient parallelization of convolution algorithms has grown to an indispensable necessity.

Future Distribution Permission

1. Introduction

This paper presents an exploration of the 2D convolution operation, focusing on its fundamentals and the strategies employed to parallelize its computations. We delve into the utilization of both CUDA and OpenMP, two powerful parallel computing frameworks, to leverage the computational resources of modern GPUs and multi-core CPUs, respectively.

2. Convolution

Given two functions f and g their convolution is defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) \cdot g(t - \tau)\, d\tau \qquad (1)$$

We are however focusing on 2-dimensional discrete convolution operation in which both function has finite support, in this case the convolution has the following form

$$(f * g)[m, n] = \sum_{i=-a}^{a} \sum_{j=-b}^{b} f[m - i, n - j] \cdot g[i, j] \qquad (2)$$

In the context of image processing the f function reppresents the input image and the g function the so called *kernel*. The visualization of this equation can be seen in Figure 1.

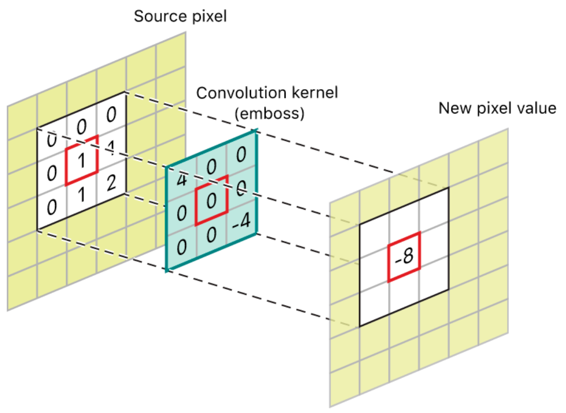

Figure 1. Visualization of 2D-convolution

When provided with an input image of dimensions $N \times M$ and a kernel measuring $k \times k$, the computational complexity of the 2D Convolution operation is $O(NMkk)$, when using separable kernels this complexity can be reduced to $O(2NMk)$. Despite using such kernels, the optimized algorithm will not be employed, as the primary objective is to assess the speed-up achieved by the parallel version in comparison to the naive sequential approach.

3. The naive serial algorithm

The following serial implementation, as well as all the parallel ones that will follow, work only on single-channel images. Regarding the typical edge problem that arises when dealing with convolution, no padding was added to the image. This approach was chosen considering that the application of the convolution is for image processing. The convolution operation is typically applied only once, and the kernel dimensions are generally much smaller than the input image. This means that the number of pixels lost in the process is negligible.

Following this important premisies the 2D-convolution

```
void host_convolution(const uchar *image,
    const float *ker, uchar *out, const int
    H, const int W){
int ker_r = KER/2;
for(int i = ker_r; i < W-ker_r; i++){
   for(int j = ker_r; j < H-ker_r; j++){
      for(int k = 0; k < KER; k++){
         for(int l = 0; l < KER; l++){
            out[i*W+j] += (uchar)image[
                (i-ker_r+k)*W +
                (j-ker_r+l)] * ker[k*KER+l];
         }
      }
   }
}
```

In order to solve the 2D convolution problem, the simplest approach is to loop over all the image pixels and all the kernel elements in one go. This algorithm is the most naive and slow one. It uses 4 nested loops: the 2 outer loops on the rows and columns of the image, and the 2 inner loops on the rows and columns of the kernel.

4. Algorithm correctness and images manipulation

To check the algorithm correctenss, in particular the one of the parallel implementations, two strategy were followed: comparing the result of the parallel implementation with the result of the serial one and showing the result of the operation on some test images (in particular Lenna grayscale 512x512). For I/O operations and visualization OpenCV framework was used. The result of a convolution with gaussian kernel che be seen in Figure 2 and Figure 3.

Figure 2. Input image

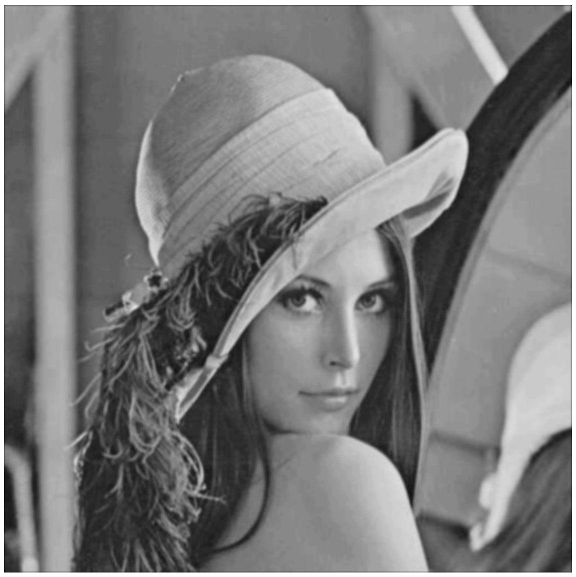
Figure 3. Output image

5. Parallelization strategy

It's worth mentioning that the Convolution operation belongs to the class of embarrassingly parallel problems. In fact, each pixel can be computed knowing only the kernel's value and the value of the pixel that surrounds it. There is no need for previous computation or dependencies between pixels. With that in mind, the convolution operation was parallelized using two different libraries: OpenMP and CUDA.

5.1. Parallelization with OpenMP

The OpenMP parallelization is straightforward: the outermost loop was parallelized using a `#pragma omp for` directive and the loop responsible for iterating through each element of the kernel was vectorized using `#pragma omp simd`. The result is that each thread works independently on a horizontal chunk of the image as we can see in Figure 4 where we consider the case with 4 threads.

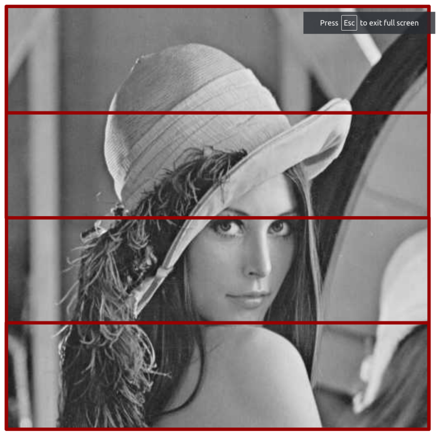

Figure 4. Workload distribution among threads

The core core function implementation is given below.

```
int ker_r = KER/2;
int thread_num = omp_get_max_threads();
int chunk = ceil(H / (float) thread_num);
#pragma omp parallel for
    num_threads(thread_num) shared(image,
    ker, out, ker_r), schedule(static,
    chunk)
for(int i = ker_r; i < W - ker_r; i++){
   for(int j = ker_r; j < H - ker_r; j++){
      float temp = 0;
      #pragma omp simd
      for(int k = 0; k < KER; k++){
         for(int l = 0; l < KER; l++){
            temp += image[(i-ker_r+k)*W +
                (j-ker_r+l)] * ker[k*KER+l];
         }
      }
      out[i*W+j] = (uchar)temp;
      }
   }
```

Other strategies were considered but none of them seem to improve the perfomance with respect to the one given.

5.2. Parallelization with CUDA

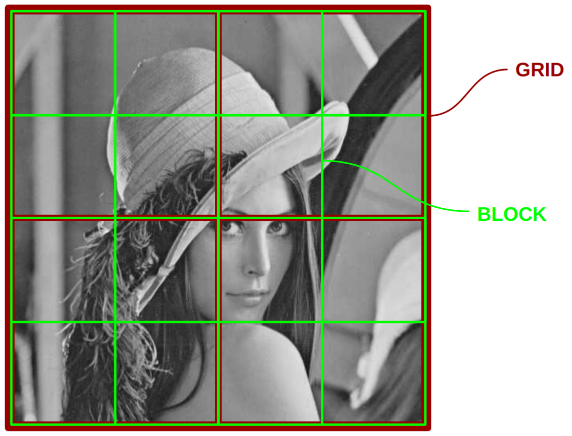

Figure 5. Workload distribution among threads

The CUDA implementation of 2D convolution utilizing constant memory and shared memory showcases a distinct parallelization strategy. In this approach, the input image is convolved with a kernel using optimized memory management techniques. The kernel itself is stored in constant memory, enhancing memory access efficiency. Additionally, shared memory is employed to cooperatively load data and facilitate communication among threads within a block, further optimizing data sharing during computation. The convolution process is divided into grid and block-based parallelism.

The CUDA kernel is executed across multiple blocks and threads. Each thread within a block loads a portion of the image into shared memory. Thread collaboration is harnessed to load image data from global memory into shared memory, taking advantage of the shared memory's low-latency and high-bandwidth characteristics. This cooperative loading strategy mitigates the need for redundant global memory access, resulting in improved memory access patterns.

```
__shared__ float
    N_ds[INPUT_TILE_Y][INPUT_TILE_X];
int ker_r = KER / 2;
int dest = threadIdx.y * TILE_WIDTH_X +
    threadIdx.x;
int dest_y = dest / INPUT_TILE_X;
int dest_x = dest % INPUT_TILE_X;
int src_x = blockIdx.x * TILE_WIDTH_X +
    dest_x - ker_r;
int src_y = blockIdx.y * TILE_WIDTH_Y +
    dest_y - ker_r;
if (src_y >= 0 && src_y < height && src_x
    >= 0 && src_x < width)
   N_ds[dest_y][dest_x] = image[(src_y *
       width + src_x)];
else
   N_ds[dest_y][dest_x] = 0;
   dest = threadIdx.y * TILE_WIDTH_X +
       threadIdx.x + TILE_WIDTH_X *
       TILE_WIDTH_Y;
   dest_y = dest / INPUT_TILE_X;
   dest_x = dest % INPUT_TILE_X;
   src_y = blockIdx.y * TILE_WIDTH_Y +
       dest_y - ker_r;
   src_x = blockIdx.x * TILE_WIDTH_X +
       dest_x - ker_r;
if (dest_y < INPUT_TILE_Y)
{
   if (src_y >= 0 && src_y < height &&
       src_x >= 0 && src_x < width)
      N_ds[dest_y][dest_x] = image[(src_y
          * width + src_x)];
   else
      N_ds[dest_y][dest_x] = 0;
}
__syncthreads();
```

Once data is efficiently loaded into shared memory, each thread computes the convolution operation by iterating over the kernel dimensions. The computed results are then aggregated into a local accumulator variable. To ensure synchronization and proper accumulation, thread synchronization points are strategically placed using the `__syncthreads()` function.

```
float accum = 0;
int y, x;
for (y = 0; y < KER; y++)
   for (x = 0; x < KER; x++)
      accum += (N_ds[threadIdx.y +
          y][threadIdx.x + x] * ker[y * KER
          + x]);
      y = blockIdx.y * TILE_WIDTH_Y +
          threadIdx.y;
      x = blockIdx.x * TILE_WIDTH_X +
          threadIdx.x;
if (y < height && x < width)
   out[(y * width + x)] = (uchar)accum;
```

After the convolution computations have concluded, the results are written back to global memory. Threads work together to store the computed pixel values back to the output image. This grid-based approach ensures that all pixels of the output image are processed with the appropriate convolution operation.

In summary, the CUDA implementation of 2D convolution leverages both constant memory for the kernel and shared memory. This approach optimizes memory access patterns ultimately resulting in accelerated convolution processing.

6. Setup

Experiments were performed on AMD Ryzen7 4800H (8 CPU cores and 16 threads) and NVIDIA GeForce RTX 2060 Mobile. The code was compiled using nvcc V11.7.64 on Ubuntu 22.04. Host code was compiled using g++ 11.4.0.

Test were performed on random generated single channel opencv Mat objects of varying dimensions. The kernel are all gaussian.

Functions were timed using `omp_get_wtime()` function. When testing CUDA implementation the memory allocation time associated with `cudaMalloc` was not considered.

No hypothesis testing was performed. To evaluate whether one implementation performs better than another only the mean values were considered.

7. Results

7.1. Results of OpenMP

7.1.1 Effect of vectorization

To test whether vectorization has an impact on the OpenMP implementation, both a vectorized and a non-vectorized implementation were run 100

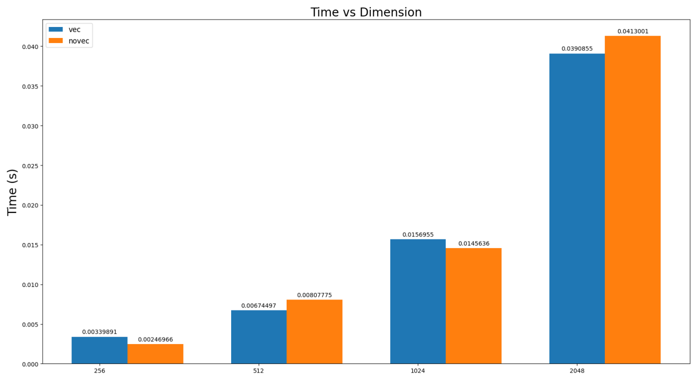

Figure 6. Result of speed up analysis

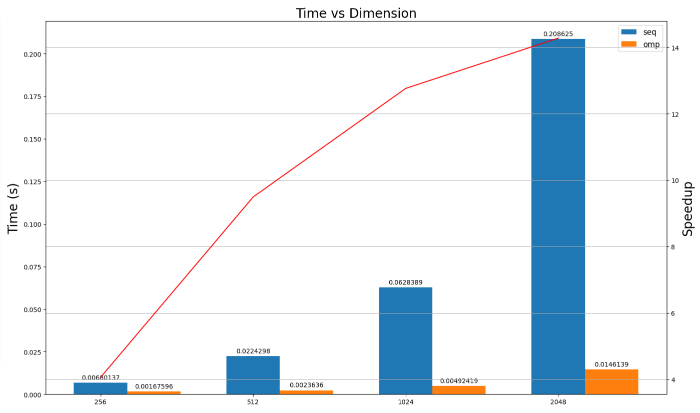

Figure 7. Result of speed up analysis

times. The results were averaged across different image sizes. As shown in Figure 5, vectorizing with `#pragma omp simd` doesn't deliver the desired improvements.

One reason why the pragma directive doesn't improve our code may be related to the non-optimal memory access pattern of the input pixels. Furthermore, the kernels tend to have an odd number of elements (3^2, 5^2, 7^2, etc.), leading to inefficient use of AVX2 registers, which are 256 bits long and can contain 8 float elements.

7.1.2 Speed up analysis

256x256	512x512	1024x1024	2048x2048
4.06	9.47	12.76	**14.28**

Table 1. Result of Speed up analysis

The speed-up analysis was performed by repeating the convolution operation 100 times and averaging the results. Experiments were conducted on square images with dimensions 256, 512, 1024, and 2048. The OpenMP version was always executed with the maximum number of threads available (16 in our specific setup).

Results are presented in Table 1 and Figure 5. As we can observe, the OpenMP version consistently outperforms the serial one. The speed-up tends to increase with the dimension of the image, which aligns with expectations considering the constant cost of spawning and managing threads relative to the image size. When dealing with larger images, this cost becomes proportionally lower compared to the computation cost, resulting in higher speed-ups.

It's important to emphasize the role of the `temp` variable. By creating such private variable instead of working directly on the output pixel we dramatically increase the performance of our function. Instead of accessing and refreshing the value of the output matrix k^2 times, we do it only once when copying the results from the `temp` variable into the proper pixel.

7.2. Results of CUDA

7.2.1 The ”overhead” of allocating memory

The actual speed-up achieved by the CUDA convolution is heavily reliant on what aspects are timed. We have the option to time solely the kernel, both the kernel and the `cudaMemcpy()` function, or the collective timing of the kernel, `cudaMemcpy()`, and `cudaMalloc()`. The most significant bottleneck is associated with the allocation carried out by `cudaMalloc()`

The profiling carried out with the Nvidia Visual Profiler tool shows exactly this. As we can see in Figure 8 the allocation time is order of magnitude slower than the data tranfer from the host to the device and the computation combined.

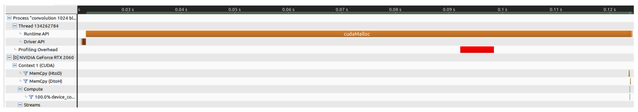

Figure 8. Result of Nvidia Visual Profiler

As far as the time required to copy the data and the computation time the results of profiling are shown in figure 9.

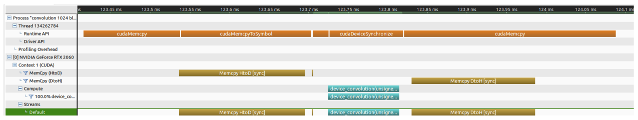

Figure 9. Result of Nvidia Visual Profiler

With that in mind only the kernel and the `cudaMemcpy()` function are timed when calculating the speedup.

Somehow we are not launching enough parallel work to hide the latency. The question is why are we exposed to this latency.

7.2.2 Optimal TILE dimensions

For aiding in the selection of the most suitable TILE dimensions, the Nvidia Nsight Compute tool was employed. Within our implementation, the ability to choose both horizontal and vertical widths is available. In order to evaluate performance, a comparison was conducted using tile dimensions of 16×16, 32×32, 32×16, and 16×32. This entailed the execution of blocks comprising 256, 512, and 1024 threads.

	32 x 32	16 x 16	32 x 16
Time	80.90 μs	77.28 μs	69.06 μs
Throughput (Compute of SM)	42.22%	46.37 %	51.86 %

Table 2. Tile dimensions

The block dimension 32×16 demonstrates a significant enhancement in kernel performance, resulting in a substantial improvement of 23% compared to the 32×32 block dimensions. Intrestingly the 16×32 dimension performed as poorly as the 32×32. The reason may be related to coalesced memory access.

7.2.3 Speed up analysis

Slight enhancement to the kernel function has minimal impact on the speed-up calculation, as explained in Paragraph 7.2.1. This is primarily due to the inherent limitations in improving the transfer of the input image from the host to the device, as well as the transfer of the output image from the device back to the host.

The speed-up, in comparison to the serial version, was evaluated under conditions similar to the OpenMP test setup. Changes were made not only to the image dimensions but also to the kernel size, including testing 3x3, 5x5, and 7x7 kernels. Remarkably, we observed a substantial enhancement in speed-up with regards to the kernel dimensions, maintaining consistency with the image dimensions.

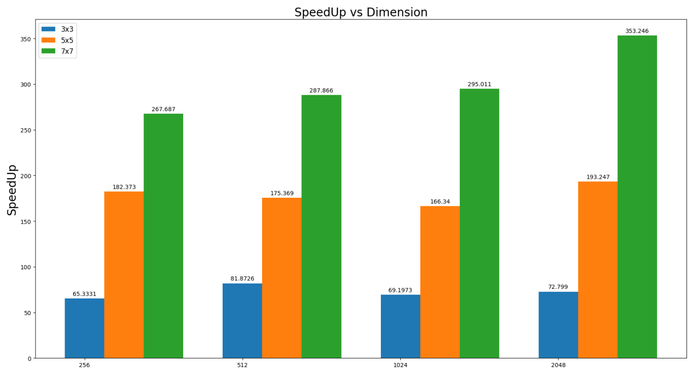

Figure 10. Result of speed up analysis

The observed phenomenon can be illuminated by delving into how the serial and parallel implementations scale concerning the kernel dimensions. As previously mentioned, the computational complexity of convolution is denoted as $O(N \times N \times k \times k)$, whereas the parallel complexity can be represented as $O\left(\frac{N\times N}{P} \times k \times k\right)$, with P signifying the number of threads. Similarly, we can express this as:

$$O(N \times N \times k \times k) = O(const_{serial} \times k^2)$$

$$O(\frac{N \times N}{P} \times k \times k) = O(const_{parallel} \times k^2)$$

Given that $const_{parallel} << const_{serial}$, we can account for the observed phenomenon in this manner.

7.3. Overall comparison

In the end, the acceleration achieved by my OpenMP and CUDA implementations is compared with respect to my sequential implementation. However, it's important to note that comparing the CUDA and OpenCV implementations might not be meaningful due to their distinct execution environments—one running on the CPU and the other on the GPU. Notably, the OpenCV implementation appears to exhibit a performance that is twice as effective as the OpenMP one. However, it's worth noticing that the OpenCV library was built using -O3 optimization.

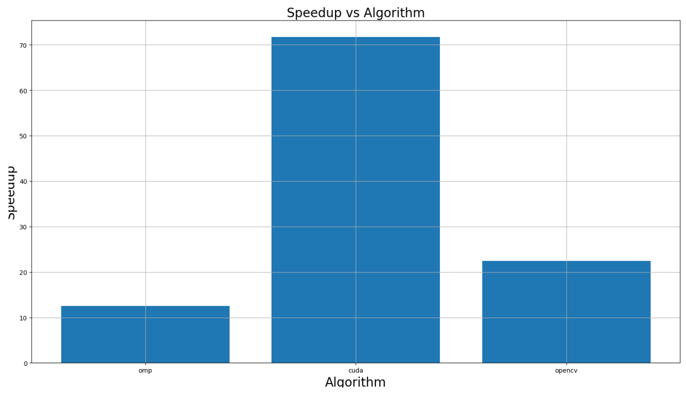

Figure 11. Result of speed up analysis

8. Conclusion

In summary, our study delved into 2D convolution fundamentals and parallelization strategies. OpenMP and CUDA implementations showcased significant speed-ups over the naive approach, with OpenMP consistently improving performance. Challenges were identified in fully mitigating latency in CUDA. This exploration underscores convolution's importance and the potential of parallel processing for efficiency gains.