Image Processing Accelerator

Team 36
311551007 江智端
311551144 王昇暉
311553046 楊宗泰

2022/12/29

Outline

01. Project Motivation

02. Background Introduction

03. Method

04. Assumption

05. Experiment Result

06. Conclusion

Motivation

Motivation

The main problem is that image processing is generally a time-consuming process.

The validation machine is bulky and unportable, it also takes a lot of time due to its old system and serial programming.

Parallel Computing provides an efficient and convenient way to address this issue.

Our main purpose of this research is to analyze different parallel programming ways such as SIMD, CUDA, OpenCL, OpenMP ...etc.

Implementation Guidelines for Image Processing with Convolutional Neural Networks

Florian Bordes
University of Vienna
Währingerstr. 29
Vienna, Austria
florian.bordes@florens.fr

Erich Schikuta
University of Vienna
Währingerstr. 29
Vienna, Austria
erich.schikuta@univie.ac.at

ABSTRACT

The domain of image processing technologies comprises many methods and algorithms for the analysis of signals, representing data sets, as photos or videos. In this paper we present a discussion and analysis, on the one hand, of classical image processing methods, as Fourier transformation, and, on the other hand, of neural networks. Specifically we focus on multi-layer and convolutional neural networks and give guidelines how images can be analyzed effectively and efficiently. To speed up the performance we identify various parallel software and hardware environments and evaluate, how parallelism can be used to improve performance of neural network operations. Based on our findings we derive several guidelines for applying different parallelization approaches on various sequential and parallel hardware infrastructure.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Image Processing, Convolutional Neural Networks, FFT, Sequential and Parallel Implementation

1. INTRODUCTION

In this paper we present a discussion and analysis of classical image processing methods, as Fourier transformation, and of neural networks. Specifically we focus on multi-layer and convolutional neural networks and show how effectively and efficiently images can be analyzed.

Hereby we explore the features of CUDA and OpenMP on multicore CPUs and GPUs for the parallelized simulation of a neural network based image processing. We analyze the application for different configurations of neural networks and give recommendations for their effective parallel simulation on both GPU and OpenMP versions.

MoMM '16, November 28 - 30, 2016, Singapore, Singapore

ISBN 978-1-4503-4806-5/16/11 ... $15.00

DOI: http://dx.doi.org/10.1145/3007120.3007165

2. IMAGE PROCESSING METHODS

2.1 Convolution

An image can be considered as a matrix consisting of values between 0 and 255. During a convolution, we apply a filter (also called kernel, which is a matrix too) on each pixel of this image. In fact a matrix product to compute a convolution is performed. The result of this operation is a new picture. For example, if we apply a Gaussian filter on the picture of Lena [1] Figure 1, we get Figure 2. We see, one goal of a convolution operation is to increase (sharpen) or decrease (blur) the details of a picture.

Figure 1: Original Image of Lena [1] **Figure 2: Convolution Gaussian filter**

An algorithm realizing this method needs 4 loops, two for the picture and two for the kernel. If we have a picture with many layers like in a RGB picture, we must add another loop for each layer. However, in Algorithm 1, we consider the image has only one layer.

We implemented this algorithm in C++, which can be used with bitmap images of 8 bits or 24 bits. We performed some performance test on varying image sizes which are shown in Table 1.

COROLLARY 1. *So it can be concluded that the processing effort is acceptable for a small kernel, but with large pictures and large kernels, the execution times grow dramatically.*

2.2 Fourier Transformation

The Fourier transform allows to translate a function in a certain domain to the frequency domain, in fact, we can translate a numerical function into a frequency. The Fourier transform is given by the following formula

Background Introduction

Add & Sub

Gamma Correction

$$s = c\,r^{\gamma}$$

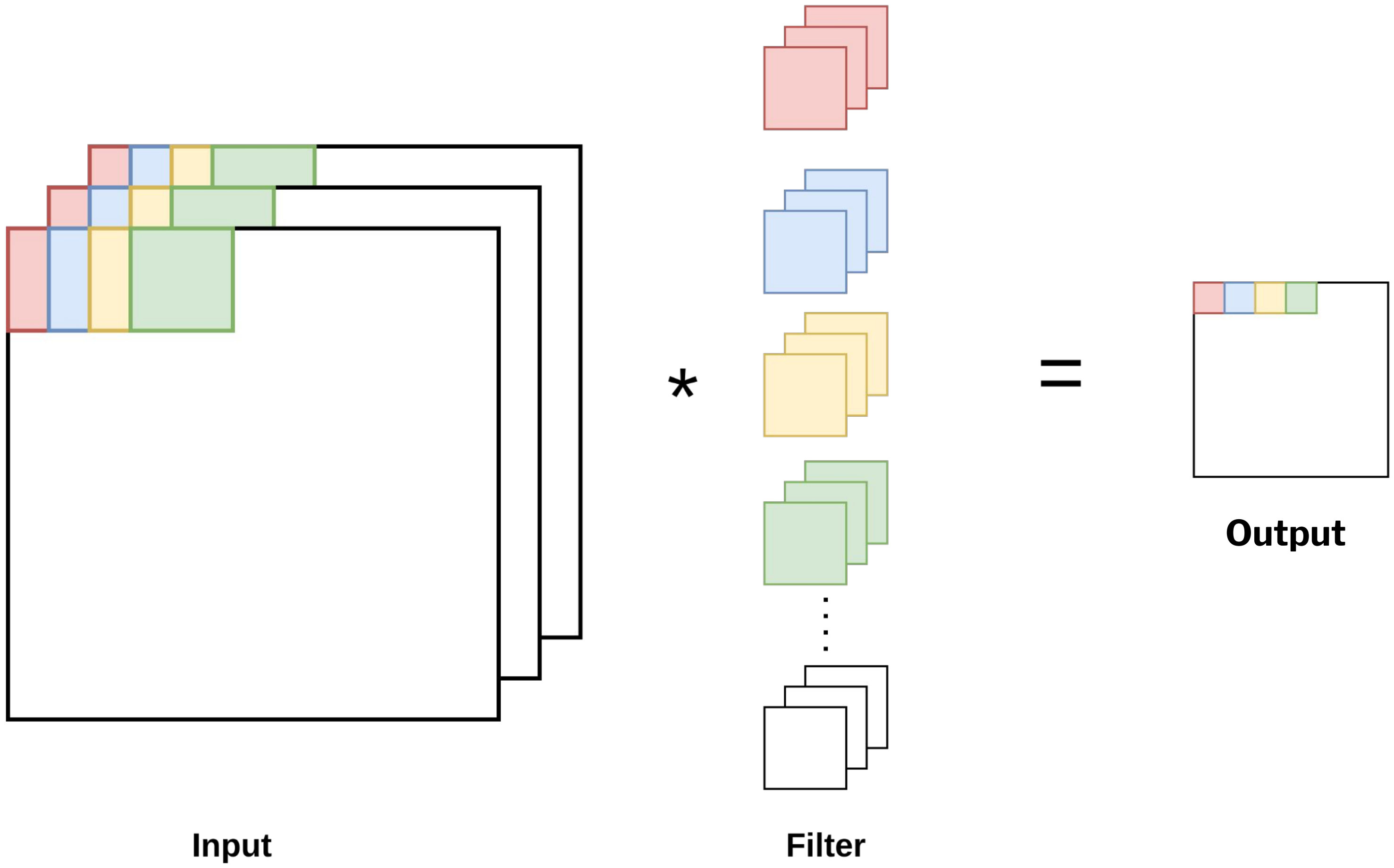
C=1, r is original pixel value

Normalization

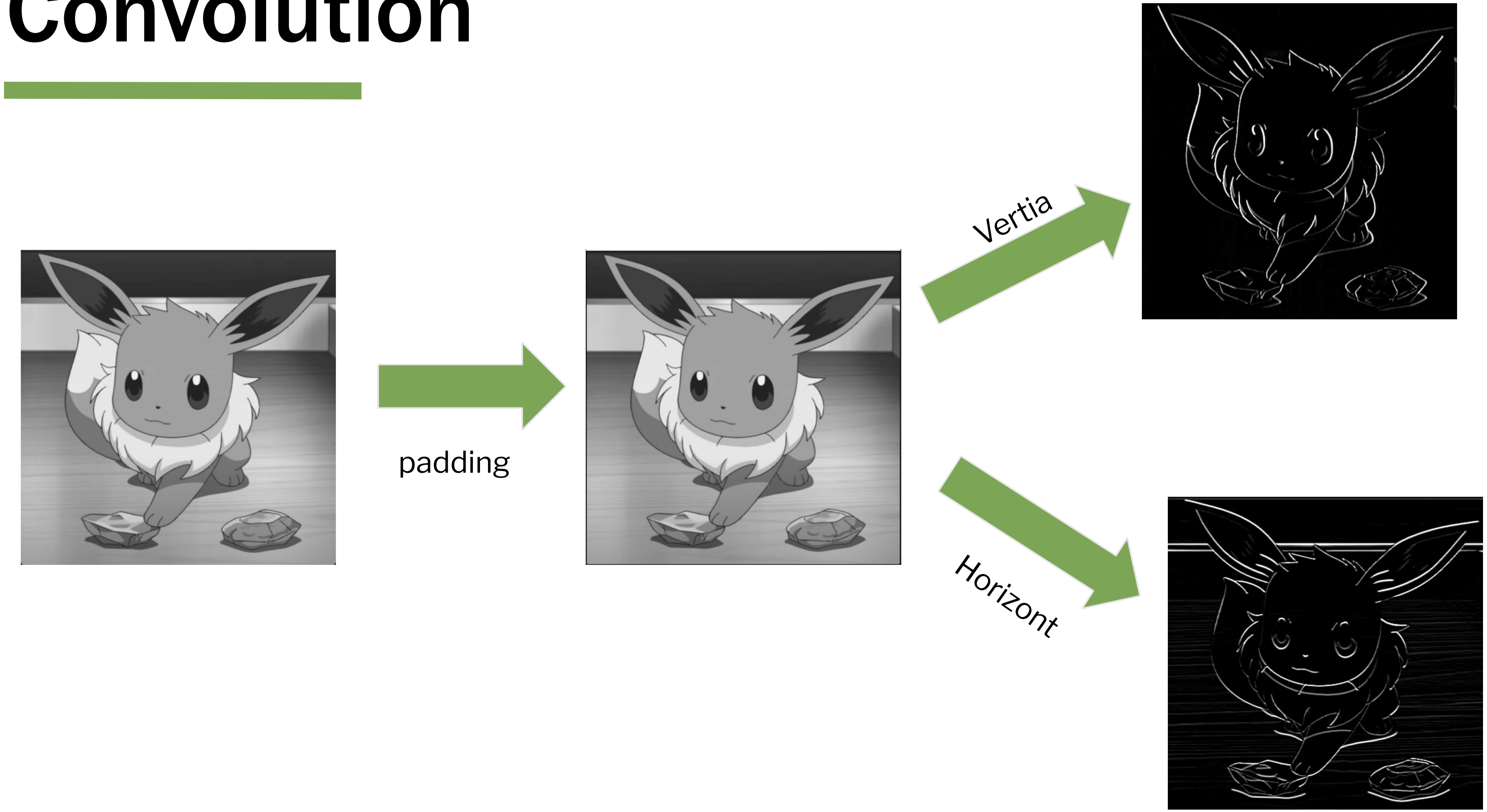
Gamma = 2

Gamma = 0.5

Convolution



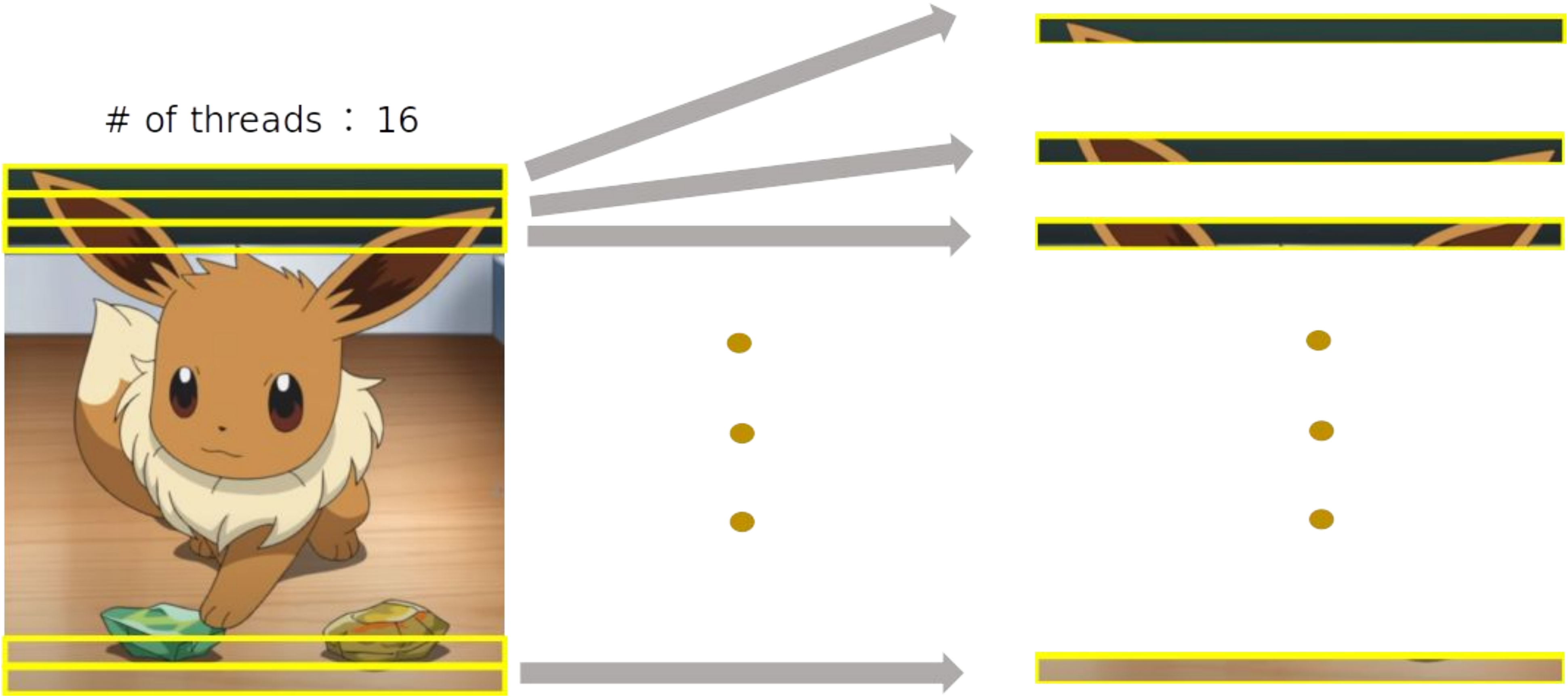
Convolution



Method

Pthreads

Separate the image into 16(threads number) parts by its height

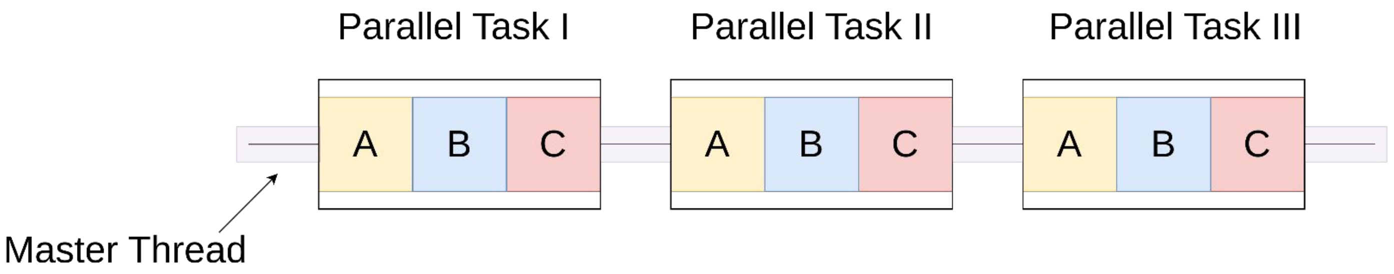


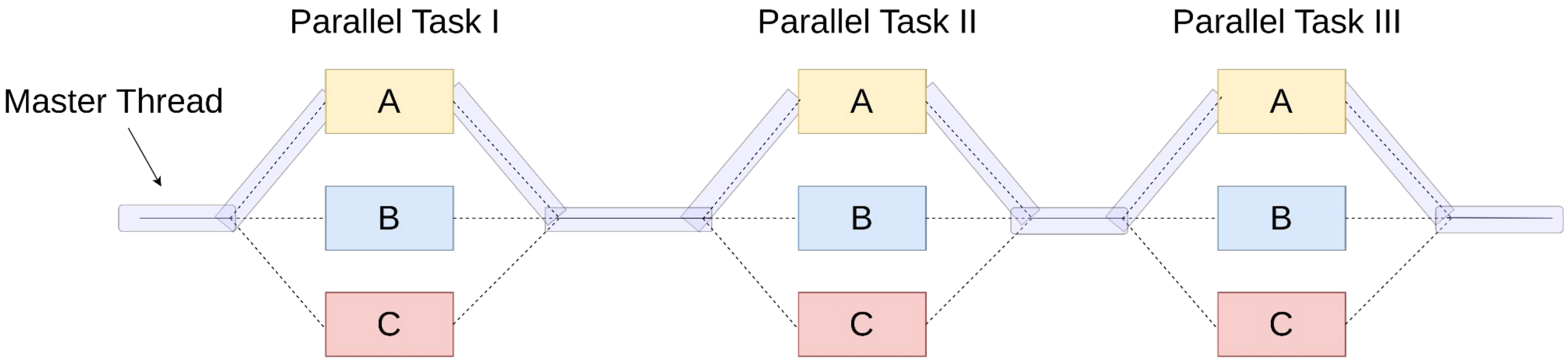
Pthread

```
Arg* arg = new Arg[MAXTHREADSCOUNT]();

int part = img.rows / MAXTHREADSCOUNT;
for (int i = 0; i < MAXTHREADSCOUNT; i++)
{
    arg[i].thread_id = i;
    arg[i].start = part * i;
    arg[i].end = part * (i + 1);
    arg[i].inImg1 = &after_padding_img;
    arg[i].kernel = &kernel;
    arg[i].outImg = &result_image;
}
arg[MAXTHREADSCOUNT - 1].end = img.rows;

for (int i = 1; i < MAXTHREADSCOUNT; i++)
{
    pthread_create(&threads[i], NULL, Partial_Convolution, (void*)&arg[i]);
}
//Master thread
Partial_Convolution((void*)arg);
```

OpenMp

Let the sum is shared, every partial sum will parallel calculate, and the final main thread will collect all the answers.

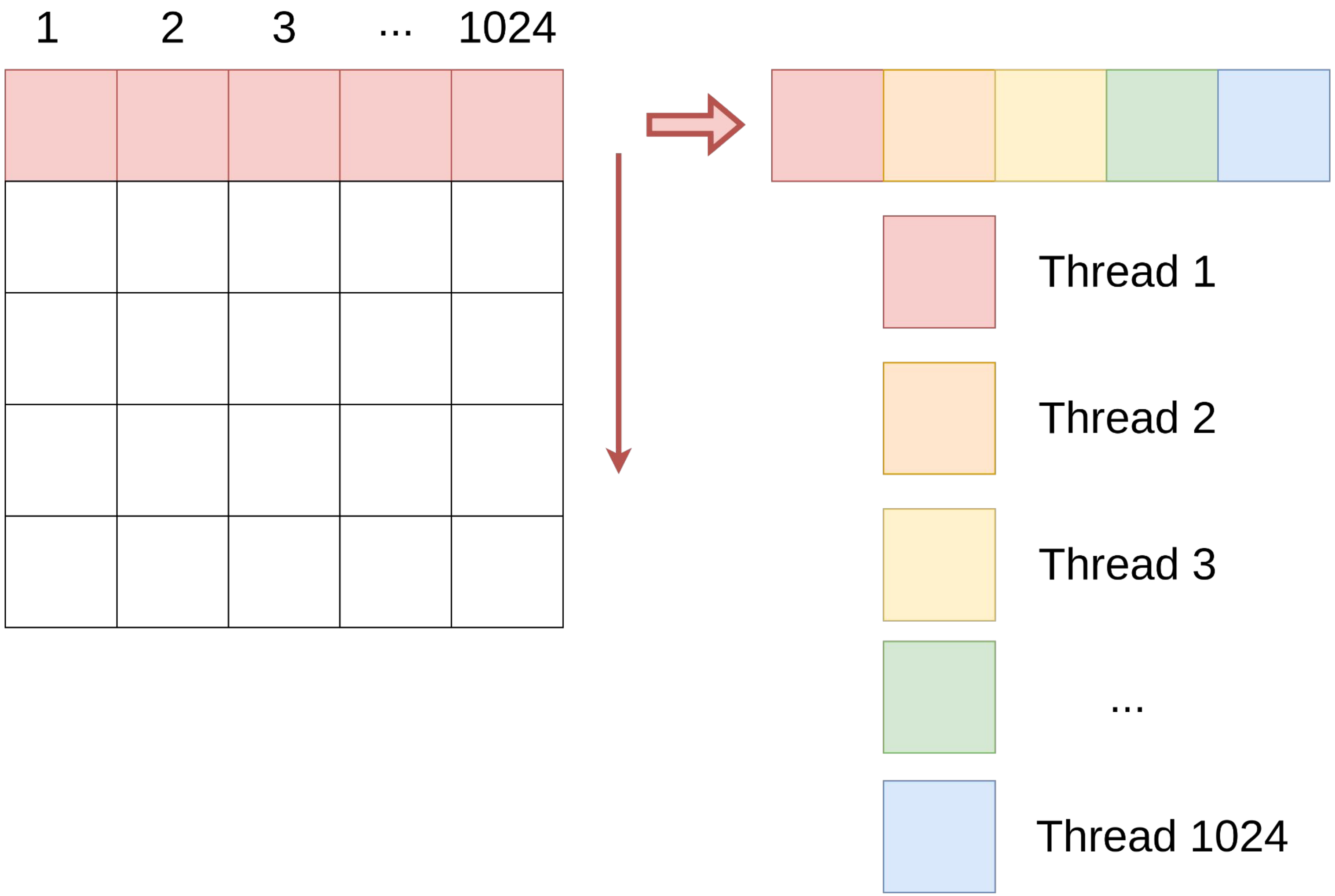




OpenMp

```
#pragma omp parallel for reduction(+:sum)
for (int c = 0; c < channels; c++)
    for (int i = 0; i < result_image.rows; i++)
        for (int j = 0; j < result_image.cols; j++)
        {

            sum = 0;
            for (int curr_row = 0; curr_row < kernel.rows; curr_row++)
                for (int curr_col = 0; curr_col < kernel.cols; curr_col++)
                {
                    sum += (kernel.at<float>(curr_row, curr_col) * after_padding_img.at<Vec3b>(curr_row + i, curr_col + j)[c]);
                }
            result_image.at<Vec3b>(i, j)[c] = (int)sum;
        }
```

CUDA



CUDA

```
cudaMalloc((void**)&device_inImg, sizeof(uchar) * imgPaddedSize);
cudaMalloc((void**)&device_inKernel, sizeof(float) * kernelSize);
cudaMalloc((void**)&device_outImg, sizeof(uchar) * imgSize);

cudaMemcpy(device_inImg, host_inImg, sizeof(uchar) * imgPaddedSize, cudaMemcpyHostToDevice);
cudaMemcpy(device_inKernel, host_inKernel, sizeof(float) * kernelSize, cudaMemcpyHostToDevice);

int BlockNum = min(after_padding_img.rows / ThreadNum, MaxBlockNum);
BlockNum = (BlockNum == 0)? 1 : BlockNum;
Partial_Convolution_Cuda << <BlockNum, ThreadNum >> > (device_outImg, device_inImg, device_inKernel, bios * 2, inImg.rows, inImg.cols, inImg.channels(),

cudaMemcpy(host_outImg, device_outImg, sizeof(uchar) * imgSize, cudaMemcpyDeviceToHost);

cudaFree(device_inImg);
cudaFree(device_inKernel);
cudaFree(device_outImg);
```

SIMD

Store 8 elements into a vector, and calculate the 8 output pixels once again.

(row,col)	(row,col+1)	(row,col+2)	…	(row,col+7)	Input
			×		
(row,col)	(row,col)	(row,col)	…	(row,col)	Kernel
			=		
sum	sum	sum	…	sum	Temp sum

SIMD

```
__m256 sum = _mm256_setzero_ps(); // set sum to zero
for (int curr_row = 0; curr_row < kernel.rows; curr_row++)
{
    for (int curr_col = 0; curr_col < kernel.cols; curr_col++)
    {
        // load eight pixels
        __m256 img_pixels = _mm256_set_ps(
            after_padding_img.at<Vec3b>(curr_row + i, curr_col + j + 7)[c],
            after_padding_img.at<Vec3b>(curr_row + i, curr_col + j + 6)[c],
            after_padding_img.at<Vec3b>(curr_row + i, curr_col + j + 5)[c],
            after_padding_img.at<Vec3b>(curr_row + i, curr_col + j + 4)[c],
            after_padding_img.at<Vec3b>(curr_row + i, curr_col + j + 3)[c],
            after_padding_img.at<Vec3b>(curr_row + i, curr_col + j + 2)[c],
            after_padding_img.at<Vec3b>(curr_row + i, curr_col + j + 1)[c],
            after_padding_img.at<Vec3b>(curr_row + i, curr_col + j)[c]
        );
        __m256 kern_val = _mm256_set1_ps(kernel.at<float>(curr_row, curr_col)); // set kern_val
        sum = _mm256_add_ps(sum, _mm256_mul_ps(img_pixels, kern_val)); // sum += img_pixels *
    }
}
```

Assumption

Assumption

Based on our experience and the GPU speed,
We think that the speed will be:

cv::filter2D > CUDA > OpenMP> Pthread> SIMD > Serial

Experiment

Platform

- ➢ CPU：AMD Ryzen 7 5800X (8 cores 16 threads)
- ➢ GPU：NVIDIA GeForce RTX 3080
- ➢ OS：Window11
- ➢ OpenCV：4.6.0

Parameter Introduce

FHD

2M pixels

2K

5M pixels

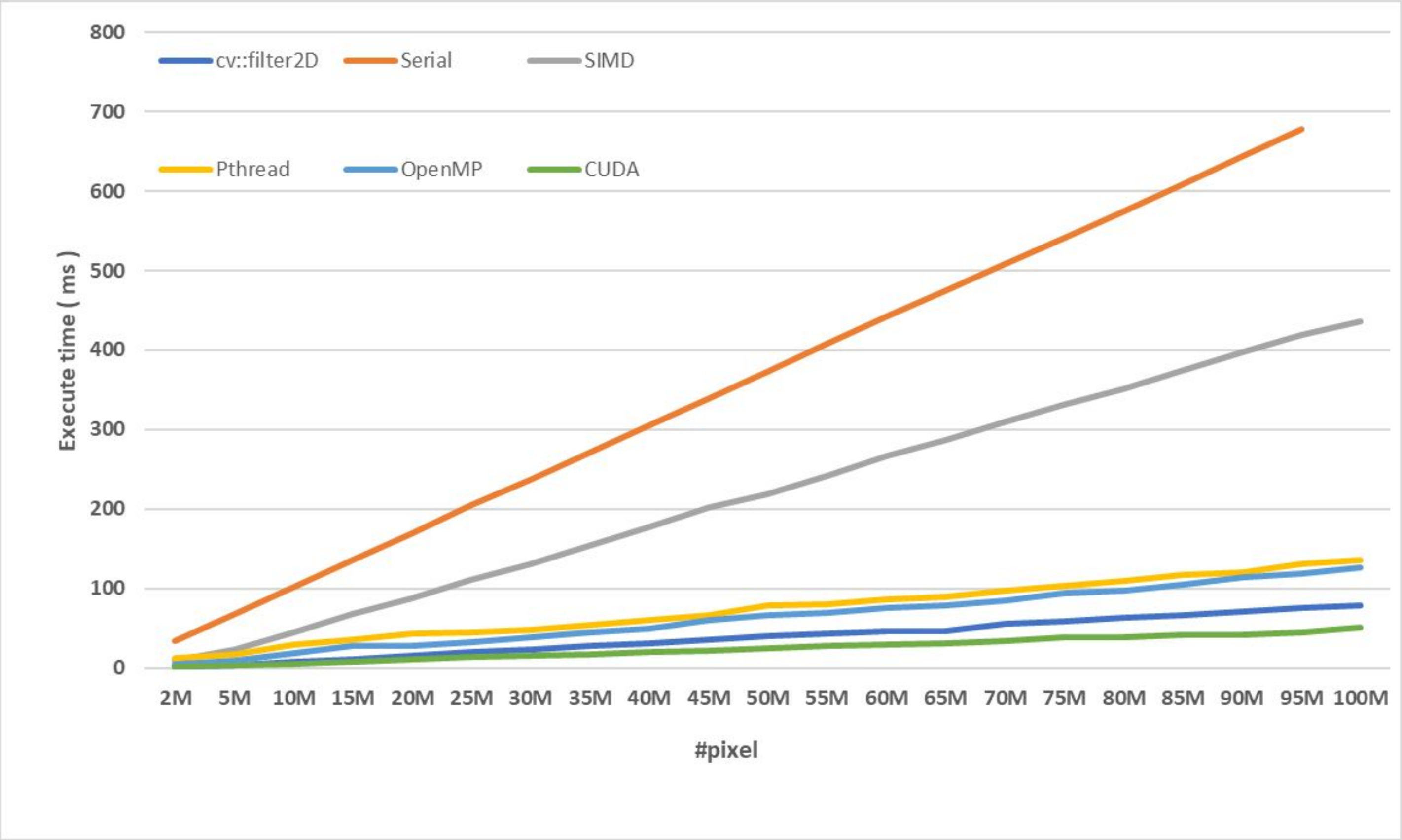
4K

10M pixels

8K

33M pixels

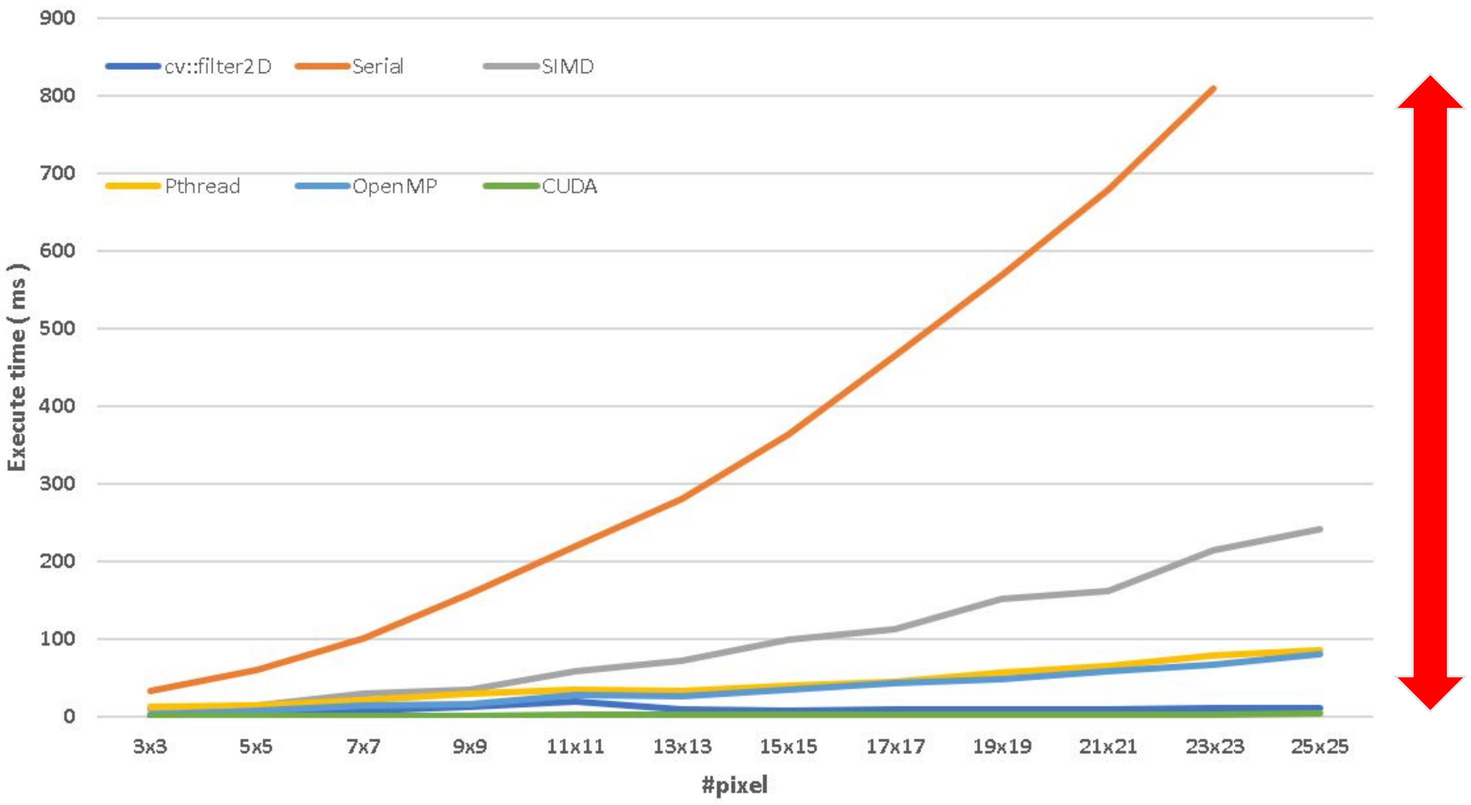
Experimental - Filter size:3x3



CUDA faster than serial

13.33x

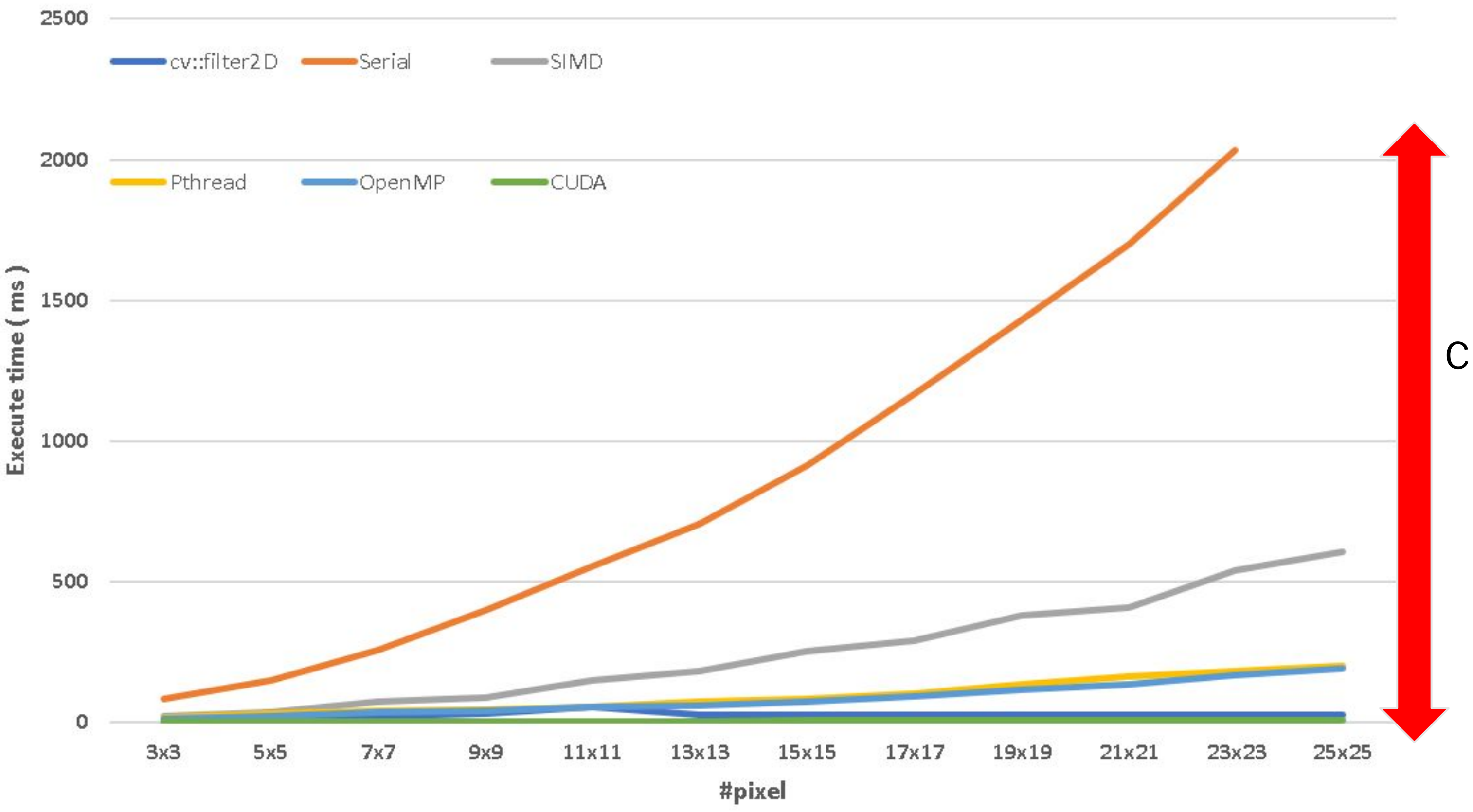
Experimental - Pixel:2M



CUDA faster than serial

255x

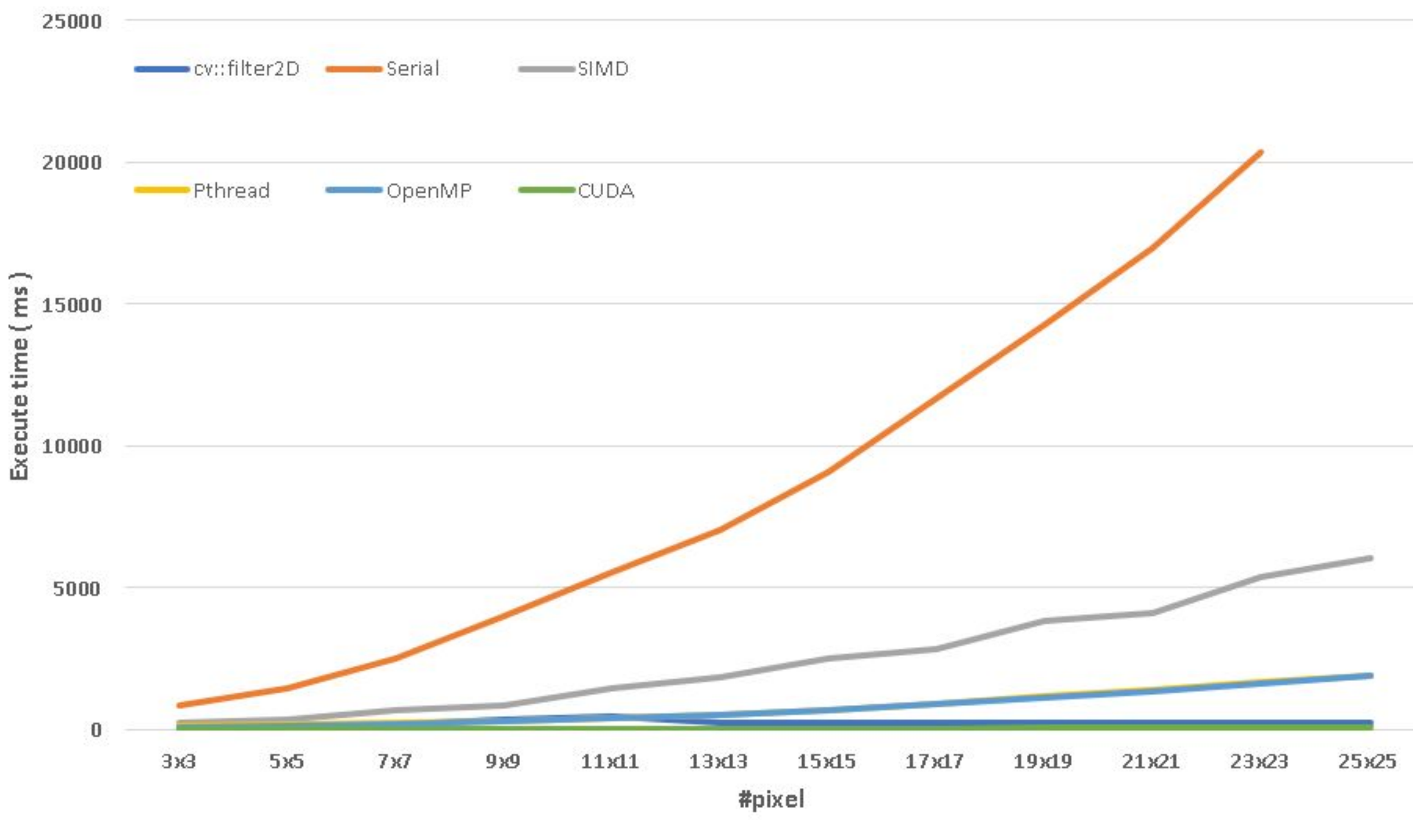
Experimental - Pixel:20M



CUDA faster than serial

294.97x

Experimental - Pixel:50M



CUDA faster than serial

349.92x

Experiment Result

- CUDA > cv::filter2D
  - We think this is beacuse our cv::filter2D is based on CPU version , so GPU wins cv:filter2D

- SIMD performance is bad
  - It is because we only use 8/per iter, while others use 16, it should do 16 too.

Conclusion

Conclusion

- Using parallel methods to complete image processing can greatly improve performance, among which CUDA has the best performance than others.

- Not only the number of pixels will affect the performance, but filter size also affects the performance.

Conclusion

- Using parallel methods to complete image processing can greatly improve performance, among which CUDA has the best performance than others.

- Not only the number of pixels will affect the performance, but filter size also affects the performance.

Thanks for listening !!

Q & A

Appendix-3x3

Image Size	cv::filter2D	Serial	SIMD	Pthread	OpenMP	CUDA
2M	1.6	14.04	9	12.7	4.3	1.5
5M	3.98	34.47	22.9	17.3	9.5	3.3
10M	7.89	67.72	45.4	30.1	19.5	5.4
15M	11.5	102.53	67.5	35.6	28.1	7.9
20M	15.56	135.95	88.1	43.4	28.3	11
25M	19.70	169.85	111.3	44.4	33.2	13.6
30M	23.70	204.81	131.6	48	38.6	15.5
35M	27.86	237.61	155	54	44.4	17.4
40M	31.29	271.51	176.9	59.7	49.6	20.3
45M	35.07	305.19	202.3	66.5	60.9	22.1
50M	39.66	339.63	219.6	78.3	66.3	25.1
55M	43.11	373.65	242.3	79.8	69.6	28.7
60M	47.16	407.77	266.3	86.1	75.1	28.9
65M	46.29	441.86	286.9	89.6	78.7	31.5
70M	55.25	474.63	309.8	96.9	85.3	34.4
75M	59.20	509.15	330.7	103.4	93.6	38.1
80M	62.97	540.92	352	109.7	96.8	39.1
85M	66.94	574.38	374.2	116.9	105.5	41.4
90M	70.87	608.24	398	121	114.4	42.2
95M	74.98	643.71	418.5	131.4	118.5	45.4
100M	78.87	677.5	436.8	136	125.8	50.8

Appendix-5M

Kernel Size	cv::filter2D	Serial	SIMD	Pthread	OpenMP	CUDA
3x3	1.6	14.5	9.2	12.4	3.4	1.5
5x5	4.2	33	13.9	14.6	8.3	1.5
7x7	8.2	60.2	29.1	20.6	13.8	1.6
9x9	13	100.6	35.1	29.3	16.2	1.7
11x11	19.6	158.6	58.4	34.1	28.2	1.8
13x13	9.7	218.9	72.8	32.4	27.1	1.9
15x15	8.5	280	99.7	39.5	34	2.5
17x17	8.7	364.1	113.5	45.6	43.3	2.2
19x19	8.8	465	151.3	56.9	49.2	2.4
21x21	9	568.5	162.2	64.5	58.6	2.6
23x23	11.2	678.6	215.2	79	67	3
25x25	10.9	810	241.4	85.3	80	3.6

Appendix-20M

Kernel Size	cv::filter2D	Serial	SIMD	Pthread	OpenMP	CUDA
3x3	4.2	35.4	22.7	16.6	9.9	3.2
5x5	10.6	83.6	34.9	29.6	19.4	3.2
7x7	20.4	147.1	71.5	39.2	33.6	3.4
9x9	33	256.6	86.4	45.5	38.3	3.6
11x11	55.4	400.5	146.8	54	53.6	3.9
13x13	25.2	553.5	183.4	73.2	56.8	4.2
15x15	27	704.7	249.9	83.6	73.6	4.9
17x17	25.4	913.3	287.8	103	92.3	4.9
19x19	25.6	1168.3	381.2	132.2	115.6	5.9
21x21	25.4	1429.4	409.6	162.1	136.3	5.9
23x23	25.2	1699.1	538.3	180.8	165.2	6.4
25x25	25.8	2035.3	604.5	202	190.7	6.9

Appendix-50M

Kernel Size	cv::filter2D	Serial	SIMD	Pthread	OpenMP	CUDA
3x3	43	359	233	97	82.8	26.3
5x5	102	816	331	170	135.9	26.2
7x7	201	1470	699	209	203.6	27.2
9x9	325	2525	850	284	284.1	28.9
11x11	482	3963	1454	407	397	31.2
13x13	247	5525	1821	531	529.4	33.8
15x15	248	7023	2491	696	695	37.2
17x17	249	9086	2831	912	880	39.1
19x19	247	11667	3836	1192	1131	42.4
21x21	248	14259	4090	1375	1359.4	47.1
23x23	250	16997	5381	1650	1606.7	49.2
25x25	252	20359	6011	1900	1884.7	58.3