Parallelized Matrix Filtering using OpenMP

Tianjiao Ding, Jiahao Li, Xin Qin @ SIST, Shanghaitech University

I. Introduction

Matrix filtering has been widely used in many applications, especially in the fields of image processing and machine learning. The time complexity of sequential execution grows quadraticly with the width of the target matrix and kernel. The contention of computational resources could be relieved by using parallel computing techniques, and OpenMP is one of them. Not until recently did OpenCV add built-in support for 3rd party multi-threading. This project identifies convolution as the major matrix filtering method, but other methods could be optimized using similar techniques. This paper reviews the theory and application of image filtering in Section II, discusses two ways to parallelize the algorithm of convolution in Section III, evaluates one of them under different conditions in Section IV.

II. Theory and Application of matrix filtering

A. Brief review of matrix filtering

1) Convolution: During convolution, two matrixes are involved. One matrix is treat by another matrix, which is called kernel. For each entry in the matrix to be treated, we replace the entry with the weighted sum of its neighbors. How many neighbors should be considered is determined by the size of kernel matrix. The weight of each neighbor entry considered, is determined by the value of corresponding entry in kernel matrix. Equation (1) demonstrate the procedure of convolution.

$$(f * g)[m,n] = \sum_{k,l} f[m-k, n-l]g[k,l] \tag{1}$$

2) Median filtering: It is done by looking at the m*m neighbours, and replacing the entry with the median.

B. Applications in computer vision

Pictures in computer vision are stored as matrixes. Matrix filtering can help remove noise, Sharpening the image, blur the image, and so on. For example, for Salt-and-papper noise, median filter can have a very good effect. Figure one showed an experiment done in CV [1].

C. Applications in machine learning

a) Convolutional Neural Network(CNN): One advantage of CNN is that, instead of complicated preprocessing of the picture, the neural networks can directly input the origin picture. This advantage is achieved by convolution operation.

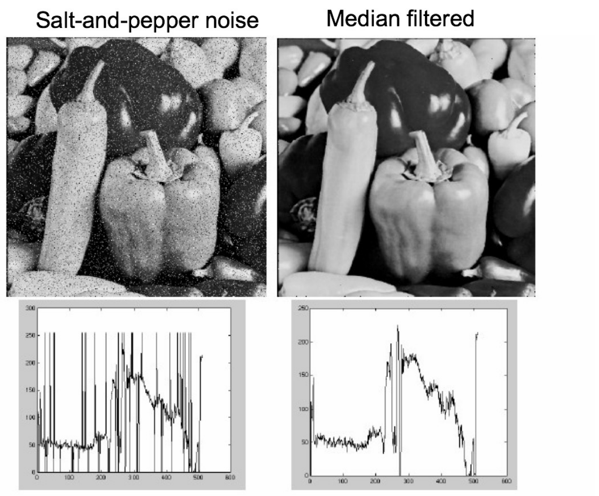


Fig. 1. Example of median filtering

D. Applications in Real-time application

a) Mobile App: Many image processing apps on mobile, particularly selfie apps, requires matrix filtering as a key functionality. To achieve different effects of selfie, the real-time procedure is actually linear combinations of many filtering operations. An interesting example is shown in Figure 2: By adding the original image with a gaussian-blurred image, we can produce a hybrid image[2]. The formula is shown in Equation (2). When observing far away from the image, we see a different expression of the person compared to seeing the image at a close distance. Since modern cameras on mobile phones usually produces high-definition image like 2K or even 4K, many real-time apps have low frame rate, typically 10fps - 25fps.

$$f + \alpha(f - f * g) \tag{2}$$

b) Real-time AR: Video See-through HMD (Head-mounted displays) is one popular implementation of Real-time AR, which involves video processing. Treatment for each frame of the video can involve several matrix filtering operations.

III. Choice of algorithm

This project identifies convolution as the major matrix filtering method, but other methods could be optimized using similar techniques. We are going to discusses two parallelizable algorithms on convolution in this Section. For the simplicity of discussion, suppose we have a square matrix

Fig. 2. Example of hybrid image

M_{m*m} that serves as the signal or target for the convolution, and a square matrix K_{k*k} that serves as the kernel. We denote the outcome matrix as matrix X_{m*m}. We ignore the padding stuff that happens at the boundary of matrices.

A. Propagation-based method

For each entry (i, j) in the signal matrix M, it affects the outcome matrix X by k*k neighbour entries that surrounds (i, j). The kernel could be viewed as the weight of how much the entry affect the surrounding entries. Based on this fact, we propagate the entry (i, j) in the signal matrix M by multiplying the weighting factors and add them respectively to k*k neighbours in outcome matrix. We present the pseudo code to illustrate the algorithm.

```
for each entry [m][n] {
  for (int i=-k/2; i<= k/2; i++){
    for (int j=-k/2; j<= k/2; j++){
      X[m-i][n-j] += M[m][n]*K[i][j]
    }
  }
}
```

This algorithm cann't be parallelized over different entries naively using OpenMP because there are certain dependencies among adjacent entries, and different processors writing to the same adjacent entry would cause race conditions. Using mutex locks isn't a good choice either, since there would be a lock for each entry, and the number of entries could be large.

However, we show that it's feasible to paralleize this algorithm using MPI, although it's beyond the discussion of this paper. Suppose we have a p*p mesh of processors that could be mapped to entries of matrix. We partition the matrix in blocks so that each entry in one block corresponds to one processor. Each processor reads the entry in signal matrix M, multiply it by the weights in kernel K, and send the (k*k-1) terms respectively to nearby processors. Each processor get (k*k-1) terms, sum them up along with one term from the processor itself, and write it to the entry in outcome matrix that corresponds to this processor. Such design raises no contention, but requires carefulness of programmers.

B. Gathering-based method

For each entry (i, j) in the outcome matrix M, it is affected by k*k neighbour entries in the signal matrix. The kernel, again, could be viewed as the weight of how much the surrounding entry affect this entry. Based on this fact, we gather the terms by multiplying the weighting factors respectively with k*k neighbours in signal matrix M and add them to entry (i, j) in the outcome matrix X. We present the pseudo code to illustrate the algorithm.

```
for each entry [m][n] {
  float sum = 0.0;
  for (int i=-k/2; i<= k/2; i++){
    for (int j=-k/2; j<= k/2; j++){
      sum += M[m-i][n-j]*K[i][j]
    }
  }
  X[m,n] = sum;
}
```

This algorithm, unlike the above one, could be parallelized using OpenMP. Because each processor reads from different entries but only writes to the entry that it has exclusive access to, there won't be data races.

C. Comparison between two methods

Beside different parallizing techniques, we show that the two algorithm present different spatial locality towards reading from the signal matrix and writing to the outcome matrix.

a) Reading from the signal matrix: Propagation-based method has a better behaviour when reading from the signal matrix. At each iteration, one processor only reads one entry from the signal matrix. However, the Gathering-based method, unfortunately, has to read all the k*k neighbours in the signal matrix.

b) Writing to the outcome matrix: Gathering-based method has a better behaviour when writing to the outcome matrix. At each iteration, one processor only writes one entry to the outcome matrix. However, the Propagation-based method, unfortunately, has to send the terms to all the k*k adjacent processors.

IV. EVALUATION

This section presents the evaluation results for matrix convolution using OpenMP, for the simplicity of discussion, we choose the kernel matrix to be common Gaussian matrix, but any matrix works fine.

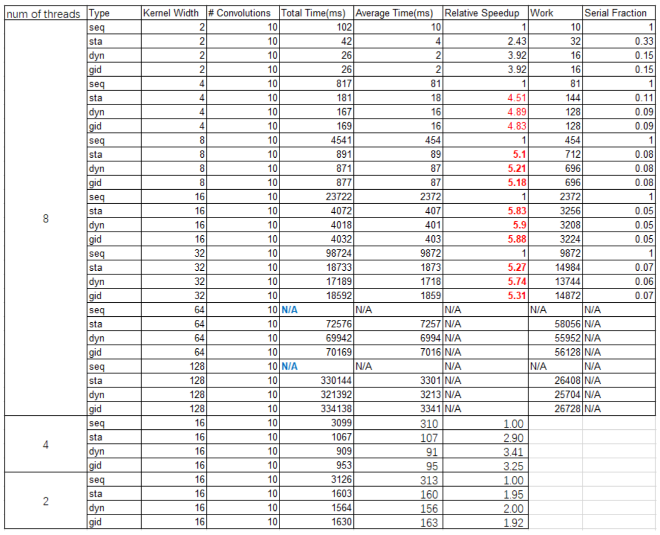
A. Basic Performance Evaluation

Figure 2 shows the benchmark of the Gaussian Blur program using OpenMP. Higher speedup is better. Our test is based on a 4 core 8 thread cpu which is intel i7 6700. Due to the speed for the Gaussian Bluer with small kernel with is too fast, we do the same convolution for 10 times and calculate the total time.

OpenMP have 4 (runtime is just a selector) kinds of loop scheduling. For the reason that schedule(auto) is delegated to the compiler, we decided to test our Gaussian Blur program with schedule(static), schedule(dynamic) and schedule(guided). In OpenMP, static schedule divides the loop into chunks as equally as possible. Dynamic schedule uses Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, Dynamic schedule retrieves the next block of loop iterations from the top of the work queue. Finally, guided schedule is similar to dynamic scheduling while the chunk size starts off large and decreases to better handle load imbalance between iterations.

B. Data Dependency

As mentioned above, there are two algorithms to implement parallel Gaussian Blur program. The algorithm using OpenMP calculates the new color for the new image independently from one corner to the corner opposite. In this algorithm, we calculate the Gaussian kernel before processing the image. During the parallel part, every thread only needs the data from the original image and data from the Gaussian kernel, so data in each thread is independent. Therefore, there is no need to add any lock in the code.



num of threads	Type	Kernel Width	# Convolutions	Total Time(ms)	Average Time(ms)	Relative Speedup	Work	Serial Fraction
8	seq	2	10	102	10	1	10	1
	sta	2	10	42	4	2.43	32	0.33
	dyn	2	10	26	2	3.92	16	0.15
	gid	2	10	26	2	3.92	16	0.15
	seq	4	10	817	81	1	81	1
	sta	4	10	181	18	4.51	144	0.11
	dyn	4	10	167	16	4.89	128	0.09
	gid	4	10	169	16	4.83	128	0.09
	seq	8	10	4541	454	1	454	1
	sta	8	10	891	89	**5.1**	712	0.08
	dyn	8	10	871	87	**5.21**	696	0.08
	gid	8	10	877	87	**5.18**	696	0.08
	seq	16	10	23722	2372	1	2372	1
	sta	16	10	4072	407	**5.83**	3256	0.05
	dyn	16	10	4018	401	**5.9**	3208	0.05
	gid	16	10	4032	403	**5.88**	3224	0.05
	seq	32	10	98724	9872	1	9872	1
	sta	32	10	18733	1873	**5.27**	14984	0.07
	dyn	32	10	17189	1718	**5.74**	13744	0.06
	gid	32	10	18592	1859	**5.31**	14872	0.07
	seq	64	10	**N/A**	N/A	N/A	N/A	N/A
	sta	64	10	72576	7257	N/A	58056	N/A
	dyn	64	10	69942	6994	N/A	55952	N/A
	gid	64	10	70169	7016	N/A	56128	N/A
	seq	128	10	**N/A**	N/A	N/A	N/A	N/A
	sta	128	10	330144	3301	N/A	26408	N/A
	dyn	128	10	321392	3213	N/A	25704	N/A
	gid	128	10	334138	3341	N/A	26728	N/A
4	seq	16	10	3099	310	1.00		
	sta	16	10	1067	107	2.90		
	dyn	16	10	909	91	3.41		
	gid	16	10	953	95	3.25		
2	seq	16	10	3126	313	1.00		
	sta	16	10	1603	160	1.95		
	dyn	16	10	1564	156	2.00		
	gid	16	10	1630	163	1.92		

Fig. 3. Evaluation of performance

C. Effect of different dataset size

The size of the dataset in our program is determined by the width of the gaussian kernel. Figure 2 shows the speed up of OpenMP (static,dynamic,guided) with gaussian kernel width varying from 2 to 32 using 8 threads. As we can see in the figure, the speedup keeps rising until the kernel width is larger than 16, and dynamic scheduling always achieves the greatest speed up. For those tests with kernel width smaller than 16, we think that the execution time of parallel part does not account for a large proportion in total execution time. For tests with kernel width larger than 16, the difference between the amount of calculation of middle pixels and edges pixels becomes obvious. As a result, dynamic scheduling still achieves good speedup. What's more, the total work of tests with gaussian kernel width larger than 16 may be a little bit 'heavy' to the operating system, so the system may not give 100% resource to our program to ensure that other running processor works well. That may lead to the slow down of the speedup.

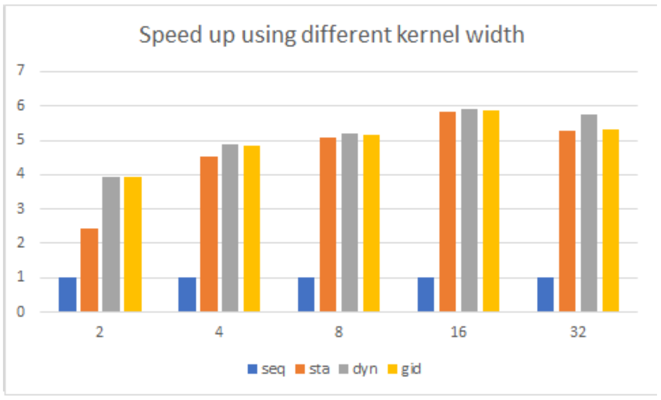


Fig. 4. Speedup using different kernel sizes

D. Effect of different number of threads

For the reason that tests with kernel width 16 achieves the greatest speed up, we tested it with 2,4,8 threads. For 8 threads, the maximum speed up is 5.9. For 4 threads, the maximum speedup is 3.41 which is much closer to the number of threads used. This phenomenon many depend on the physical structure of our cpu (4 cores 8 threads). As we have learnt from the class, multithreading is a technic which aims for making full use of the cores' computation resources. It means that the ability of a single thread using multithreading is not as good as that without multithreading. Let's move on. The maximum speedup of 2 threads is almost 2. For a system using 4 cores , it is easy for it to give a program full resources of 2 threads, this might be the reason why our tests with 2 threads achieve 'super' speedup.

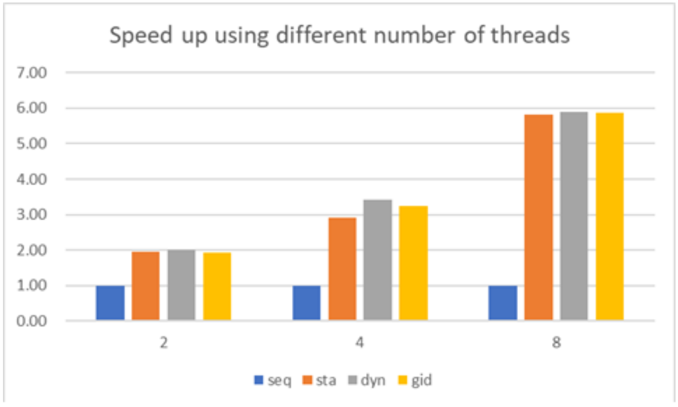


Fig. 5. Speedup using different number of threads

APPENDIX

The source code for this project is available at https://github.com/RunningCool/openmp_convolution.

ACKNOWLEDGMENT

We thanks the teaching staff and students of CS121 for the support and patience.

References

[1] Example shown in computer vision course, 2016
[2] A. Oliva, A. Torralba, P.G. Schyns, Hybrid Images, SIGGRAPH 2006