# CUDA Total Variation Denoise

**Final Report**

赵申剑 5110309748
陈蕾宇 5110309754
徐文康 5110309788

## 1. Problem Description

Our project is based on the paper: *A Highly Scalable Parallel Algorithm for Isotropic Total Variation Models*, which introduced the total variation (TV) models to remove noise in a given image based on Alternating Direction Method of Multipliers (ADMM) algorithm, termed FAD. The motivation for this model came from the fact that a noisy signal generally implies high total variation.

Given an image $Y \in \mathrm{R}^{m\times n}$, the discrete version of the Rudin-Osher-Fatemi (ROF) model is:

$$\min_X \frac{1}{2}\|X-Y\|_F^2 + \lambda\|X\|_{TV}, \qquad (1)$$

where $\|\cdot\|_F$ and $\|\cdot\|_{TV}$ are the Frobenius norm and TV norm respectively, $\lambda$ is a positive parameter, and $X \in \mathrm{R}^{m\times n}$ is the image to be recovered. We consider the isotropic TV norm:

$$\|X\|_{TV} = \sum_{i=1}^{m}\sum_{j=1}^{n}\left\|D_{i,j}X\right\|_2, \qquad (2)$$

where $D_{i,j}X = ((D_1X)_{i,j}, (D_2X)_{i,j})^T$ is the discretized gradient at pixel (i, j). For simplicity, we use the forward difference to define $D_{i,j}X$, i.e.,

$$(D_1X)_{i,j} = \begin{cases} x_{i+1,j} - x_{i,j} & if\ i < m \\ 0 & if\ i = m' \end{cases} \qquad (3)$$

$$(D_2X)_{i,j} = \begin{cases} x_{i,j+1} - x_{i,j} & if\ j < n \\ 0 & if\ j = n' \end{cases} \qquad (4)$$

It is worthwhile to mention that there are fast algorithms for the “anisotropic” TV model, where $\|X\|_{TV}$ is defined by $\sum_{i=1}^{m}\sum_{j=1}^{n}\left\|D_{i,j}X\right\|_1$. However, those algorithms, which are usually based on graph cuts or maximum flow, are not applicable to isotropic TV models..

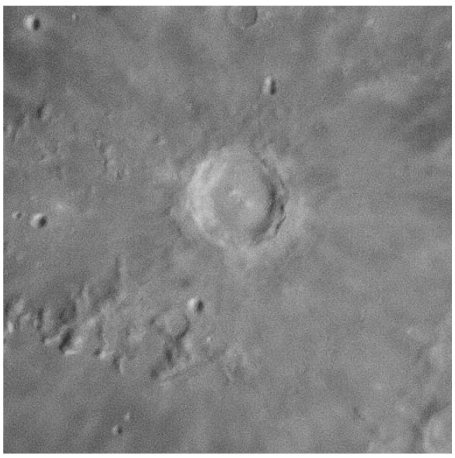
We solved the TV models by ADMM based algorithm. FAD is based on a novel decomposition strategy of the problem domain. As a result, the TV model in (1) can be decoupled into a set of small and independent subproblems. Each subproblem involves at most three variables and admits a closed-form solution. Thus, all of the subproblems can be solved efficiently in parallel. The author use MPI and Matlab to solve this problem in parallel. We implement our algorithm via CUDA which gain 17X speedup compare to the MPI approach.

## 2. Problem Application

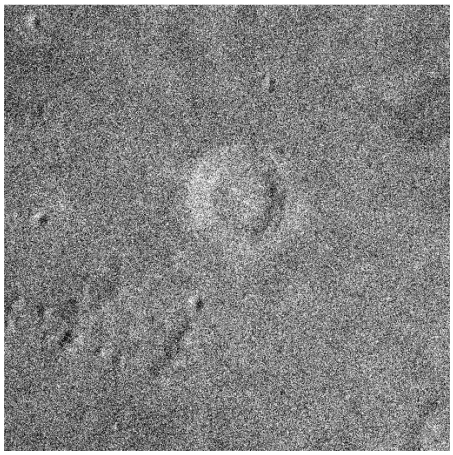
We apply FAD to the problems of image denoising. For the image denoising problem, the original image is a high-resolution photograph. The noisy image is obtained by adding

Gaussian noise $\mathrm{N}(0, \lambda^2)$. In the paper, the regularization parameter $\lambda$ is set to be 0.2. To give a better view, this paper shows the results over a sub-image with lower pixels, as showed below:

Original image



Subimage



Noisy subimage

Denoised result

However, we test with $\lambda = 0.05$ which is appealing to get better result.

## 3. Method

We apply ADMM to the image denoising problem. ADMM makes use of the decomposability of dual ascent and meanwhile has the good convergence properties of the multipliers' method. Given a problem:

$$\min_{X,z}\{f(x) + g(z): x - z = 0\}, \qquad (5)$$

We assume that both f(x) and g(z) are convex. The augmented Lagrangian is given by:

$$L_\gamma(x, z; \theta) = f(x) + g(z) + \frac{\gamma}{2}\|x - (z - \theta)\|^2, \qquad (6)$$

ADMM attempts to solve problem (5) by iteratively minimizing $L_\gamma(x, z; \theta)$ over x and y, respectively, and updating $\theta$ accordingly.

By the decomposition strategy, the TV model can be decoupled into a set of small and

independent subproblems, all of which can be solved independently and efficiently in parallel. One of the advantages of ADMM is that each node updates independently without exchanging information, which is suitable for GPU computing. In this paper, the author divides the image into three parts. In this schema, we can update each pixel independently. The details are illustrated in the paper.

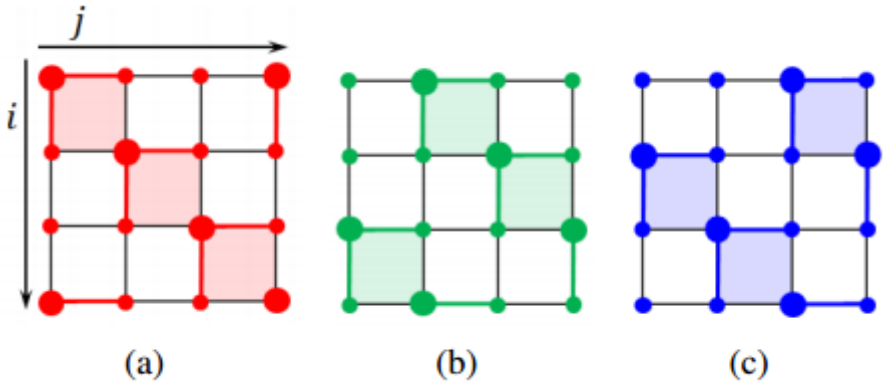
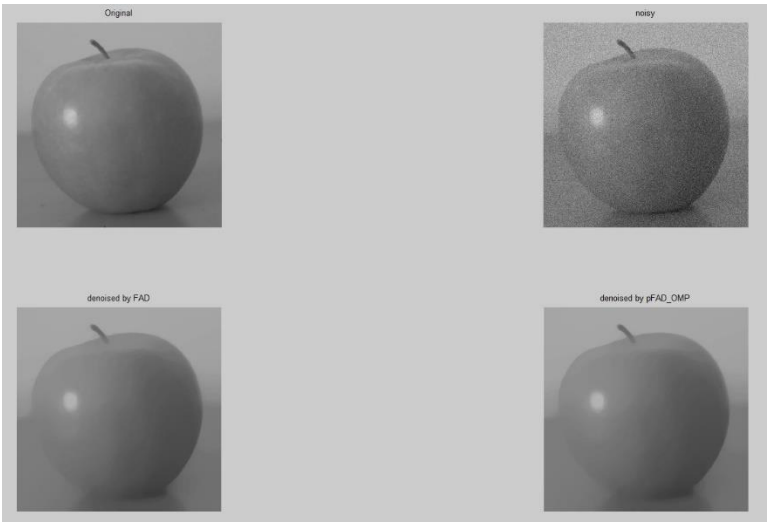



*Figure 1.* Illustration of the decomposition.

## 4. Implementation using OpenCV

Since we need to use CUDA, it is easier to implement it using OpenCV instead of Matlab. We use OpenCV to manipulate the images and test the built-in denoising algorithm. The results of the paper are shown in following figure.



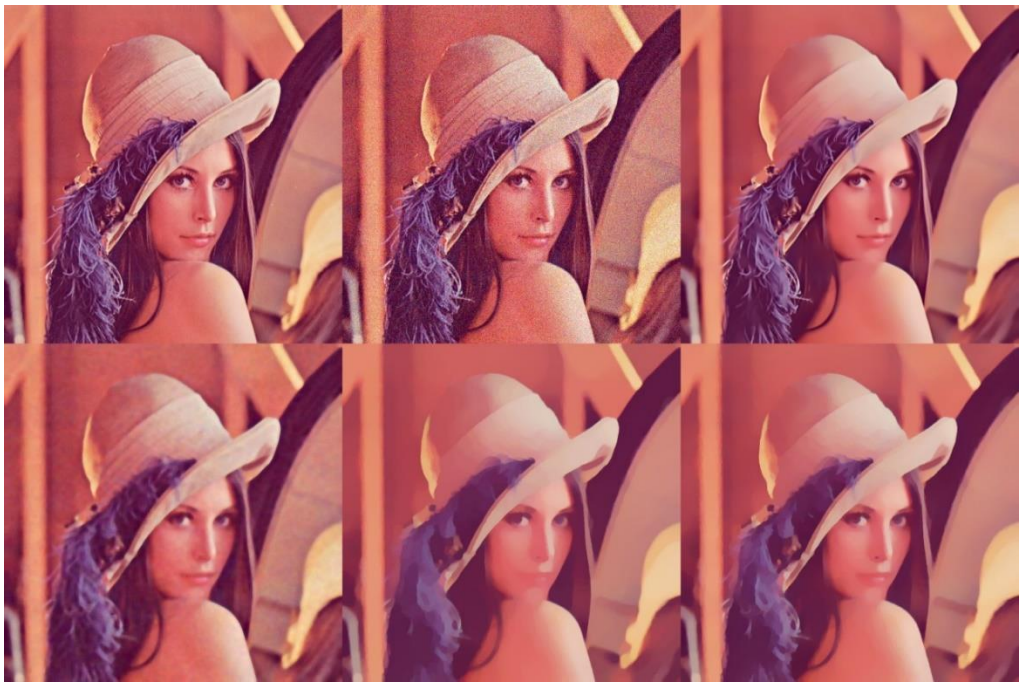

## 5. Division of Labor

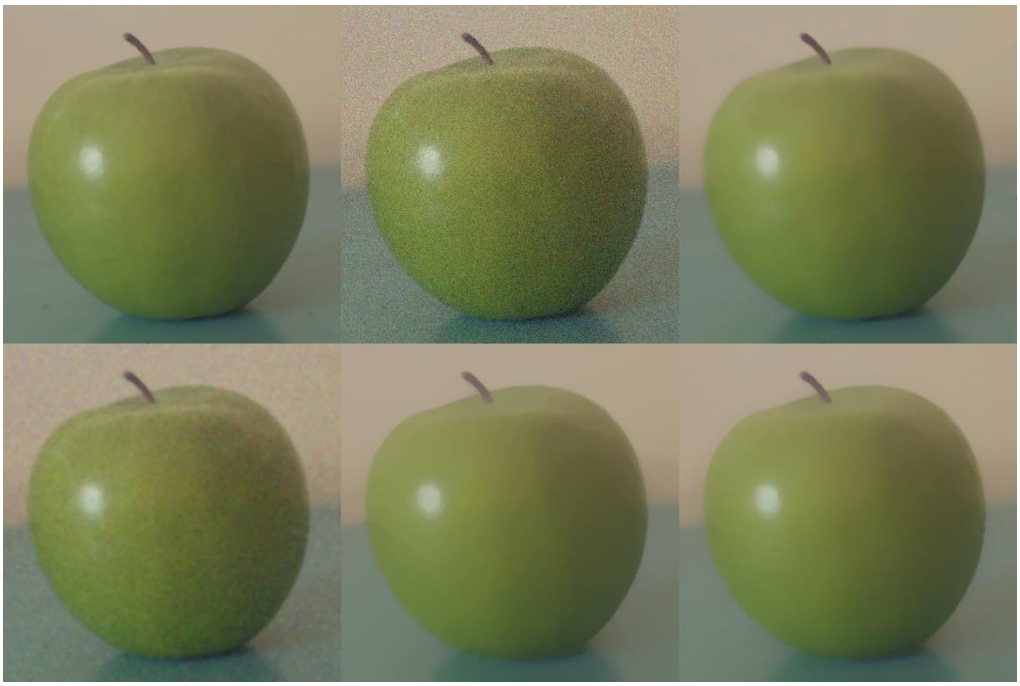
We all read the paper, and discussed the implementation. The major part of the FAD algorithm is updating. The division is shown in the following table.

| | | |
|---|---|---|
| 赵申剑 | **Interior update, Tests** | **34%** |
| 陈蕾宇 | **Boundary update, Tests** | **33%** |
| 徐文康 | **Stop condition, Tests** | **32%** |

## 6. Results

We test median filter, built-in and FAD algorithms to denoise image and compare the results they obtained. From the original image, we obtain the noisy image by adding Gaussian noise. And then we apply these algorithms to the same noisy image and gain the results showing below: (from left to right, top to bottom, the subimage is original image, noisy image, built-in algorithm result, median filter result, OpenMP result, CUDA result).





From these two images, we can draw conclusion that our implementation is correct at least. But the result may not be very good because of the algorithm. However the built-in algorithm needs 2s, but our CUDA implementation only 1s, which is more efficient.

## 7. Optimization

We use several optimization techniques to gain speedup. We compare result using the main bottleneck (inside_update) time. And test with a 512*512 image.

i. double to float: we change the main data type from double to float. However, the double type is need in some place, for example netwon iteration.
   Gain:
   1.26s → 1.01s

ii. using double buffer: we use double buffer to avoid memory transferring.
   Gain:
   1.01s → 933ms

iii. using for newton and device copy: We use while condition test to get optimal newton root at first which will cause divergence, so we use fixed 15 iterations which is proved well. In addition, we use device copy instead of double buffer, which is more efficient.
   Gain:
   933ms → 742ms

And use suitable register, asynchronous.

## 8. Discussion

Comparing to author's result, we can see that our implementation is correct. So we only need to compare efficiency. We test with a 512*512 image, and the running time is recorded as below.

Single Channel:

Matlab Serial: 11.3907 s

Matlab Parallel: 5.1889 s

Three Channel:

Serial: 34.454 s

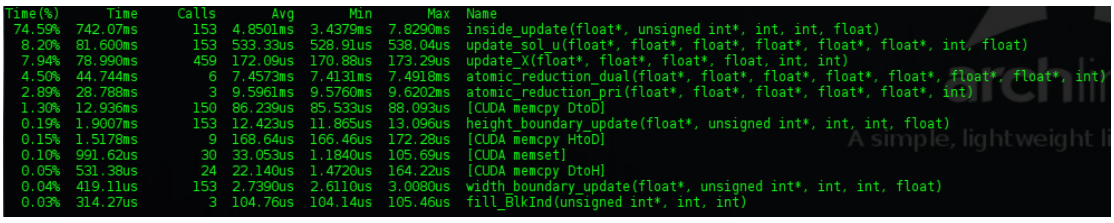
OpenMP: 17.363 s (i5 four cores)

NAÏVE CUDA: 1.758 s (GT540M)

OPT CUDA: 1.184 s (GT540M)

We have three channel, so our results of CPU version that using OpenCV and OpenMP are the same as those of the author's implementation using matlab. From the results above, we can see that we achieved 34X speedup using CUDA comparing to serial CPU version, and 17X speedup comparing to OpenMP, with the same denoising effect.

## 9. Limitation



| Time(%) | Time | Calls | Avg | Min | Max | Name |
|---|---|---|---|---|---|---|
| 74.59% | 742.07ms | 153 | 4.8501ms | 3.4379ms | 7.8290ms | inside_update(float*, unsigned int*, int, int, float) |
| 8.20% | 81.600ms | 153 | 533.33us | 528.91us | 538.04us | update_sol_u(float*, float*, float*, float*, float*, float*, int, float) |
| 7.94% | 78.990ms | 459 | 172.09us | 170.88us | 173.29us | update_X(float*, float*, float*, float, int, int) |
| 4.50% | 44.744ms | 6 | 7.4573ms | 7.4131ms | 7.4918ms | atomic_reduction_dual(float*, float*, float*, float*, float*, float*, int) |
| 2.89% | 28.788ms | 3 | 9.5961ms | 9.5760ms | 9.6202ms | atomic_reduction_pri(float*, float*, float*, float*, float*, int) |
| 1.30% | 12.936ms | 150 | 86.239us | 85.533us | 88.093us | [CUDA memcpy DtoD] |
| 0.19% | 1.9007ms | 153 | 12.423us | 11.865us | 13.096us | height_boundary_update(float*, unsigned int*, int, int, float) |
| 0.15% | 1.5178ms | 9 | 168.64us | 166.46us | 172.28us | [CUDA memcpy HtoD] |
| 0.10% | 991.62us | 30 | 33.053us | 1.1840us | 105.69us | [CUDA memset] |
| 0.05% | 531.38us | 24 | 22.140us | 1.4720us | 164.22us | [CUDA memcpy DtoH] |
| 0.04% | 419.11us | 153 | 2.7390us | 2.6110us | 3.0080us | width_boundary_update(float*, unsigned int*, int, int, float) |
| 0.03% | 314.27us | 3 | 104.76us | 104.14us | 105.46us | fill_BlkInd(unsigned int*, int, int) |

According to the image above, the proportion of running time of inside_update is 74%, which has the largest share. So this suggests that the bottleneck is clearly the inside_update. That is, the efficiency of inside_update algorithm had a huge influence on that of total process. After testing and analyzing the data, we obtained the reason was that there are lots of pixels and the newton iteration costs a lot of computation resource. We have no idea to reduce the time by these algorithm. We need new algorithm to eliminate the step to find newton root. So we will dig into the algorithm and take a further research on new optimization in other papers if we are capable enough.

Another problem is the efficiency of memory access pattern. Since we only need 1 pixel in every 3 pixel, memory accessing is inefficient enough. The algorithm fails to take good advantage of global memory accessing pattern in CUDA. And 2/3 of the memory bandwidth is wasted.

## 10. Summary

In this project, we select the subject relevant to parallel programming by ourselves. We mainly focus on the problem of image denoising. To have a better understanding of the principle, we read the paper *A Highly Scalable Parallel Algorithm for Isotropic Total Variation Models*, researched the algorithms which relevant to machine learning, and then implemented the algorithms on parallel programming.

The author of this paper provided source code in MATLAB. Having researching and comprehending the algorithm in this paper, we have implemented this algorithm in C++ using the interface of OpenCV. After successfully running our program in parallel on CPU, we implemented the algorithm in CUDA. And we try lots of ways of optimization to gain more speedup. Then we compared these solutions by their efficiencies. The process of researching for the project make us master the skill of programming via CUDA.