EXTRACTING DYNAMIC INFORMATION FROM VIDEO IN REAL TIME

An Undergraduate Research Scholars Thesis

by

JIALU ZHAO

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Tie Liu

May 2018

Major: Computer Engineering - CEEN

TABLE OF CONTENTS

ABSTRACT

Extract Dynamic Information from Video in Real Time

Jialu Zhao
Department of Electrical & Computer Engineering
Texas A&M University

Research Advisor: Dr. Tie Liu
Department of Electrical & Computer Engineering
Texas A&M University

Graphic processing units (GPU) play a major role in providing faster runtime with its parallel architecture. With hundreds of cores, GPUs help in accelerating the runtime of algorithms in various applications such as machine learning and image processing. Also CUDA parallel platform lets us to use CUDA enabled GPU for accelerating the runtime of this algorithm. In this work, we exploit the parallel computing capacity of GPU which will be programming in CUDA in video surveillance cameras. A fundamental task in video surveillance cameras is to capture sparse moving objects (foreground) in slowly changing background. A recent algorithm known as the *Prac-Re-Pro-CS* appears to be especially appealing from the performance viewpoint. However, the algorithm involves manipulations of large matrices and hence is very computationally intensive. The goal of this work is to achieve a real-time implementation of the PracReProCS algorithm by exploiting the parallel architecture of GPU.

ACKNOWLEDGEMENTS

I would like to thank my faculty advisor, Dr. Tie Liu, and my project partner, Lakshmi Venugopal, for their guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my mother and father for their encouragement.

CHAPTER I

INTRODUCTION

Prac-RePro-CS algorithm

A basic task for any "smart" video surveillance system is to extract dynamic information from the video feed in real or near real time. Different warning messages can then be generated based on further analysis of the extracted dynamic information. Many algorithms have been proposed in the literature, among which a recent algorithm proposed by Guo, Qiu, and Vaswani and known as the Prac-Re-Pro-CS appears to be especially appealing from the performance viewpoint [1].

In their algorithm, each video frame is modeled as the sum of a slowly changing background and a dynamic but sparse foreground. Because the background is barely changing and the foreground is relative dynamic. They assume that the background is constant. Through some frames' training, they obtained a training background and regarded this training background as the whole background. So for the next every frames, they will extract only dynamic sparse foreground from the video through ignoring the known dense background. This is the basic idea of the state-of-the-art sparse optimization techniques which is used to separate the dynamic foreground from the background. This algorithm is particularly suitable for surveillance systems, because the dynamic information is indeed sparse for most of the occasions.

Prac-RePro-CS algorithm has already been realized in Matlab and the results obtained by this algorithm are shown below in Fig. 1.

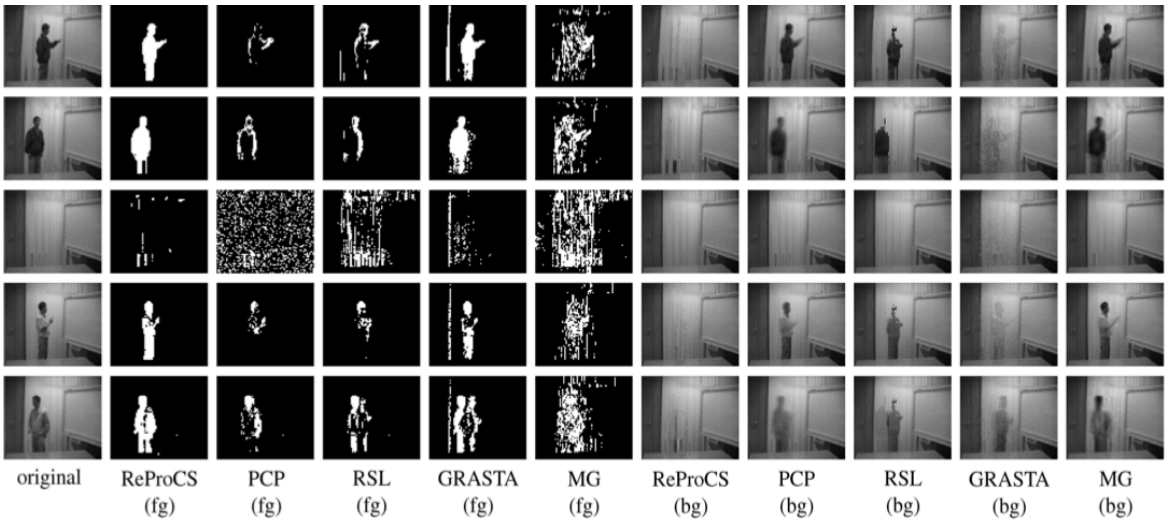

Fig. 1. Original video sequence and its foreground and background layer recovery results using ReProCS (ReProCS-pCA) and other algorithms.

From the results, we can see that compared to any other algorithm, the clarity of ReProCS algorithm is highest. That's the biggest reason that we choose to use this algorithm as our method to realize extracting dynamic information in real time. However, there are some disadvantages in this algorithm. Because they regard the background is always unchanging and use the previous training background as a known background. Therefore, some small changes such as rotating fans and moving trees will be regarded as a moving objects. For the second graph, there is supposed to be nothing, but we still can see some "dynamic foreground" which is caused by some small changes that we want to ignore and still treat as the static background. However, given the highest clarity of foreground, we choose to ignore this disadvantage and only consider how to realize it in real time.

Algorithm realization in C++

In this project, we aim at a real-time implementation of the PracReProCS algorithm using a graphics processing unit (GPU) based embedded system. There are two main challenges towards a real-time implementation. First of all, the algorithm deals with a huge amount of data

in both spatial and time dimensions (real-time video feeds from high-definition cameras). Secondly, the data processing algorithm involves manipulations of large matrices and hence is very computationally intensive. Our goal is to achieve a real-time implementation of the PracReProCS algorithm by exploiting the parallel architecture of GPU.

Because we will use CUDA C++ to realize this algorithm in real time, so the first step of our work is to convert the PracReProCS algorithm which is programmed in MATLAB to C++. My project partner Lakshmi Venugopal took charge of most part of this converting work and I gave some help on testing and coding. Finally, we have successfully converted it to C++ and run successfully on NVIDIA's Jetson TK1 development board with 192 CUDA cores. The camera can block most of the static background and only show the dynamic foreground.

The foreground is represented as a sparse signal, and the background is modeled as a dense but slowly changing signal. The Prac-RePro-CS algorithm detects the moving target even in the presence of noise in the background (such as a moving tree or a fan). The algorithm is run on NVIDIA's Jetson TK1 development board with 192 CUDA cores. A video surveillance IP camera (frame rate 15fps & resolution 1920*1080) is connected to the board to feed the video input. Without utilizing GPU, the Prac-ReProCS algorithm incurs 20 seconds delay per frame to produce the output.

The background frames (can be slowly changing due to noise) are given as training samples. Hence sufficiently large training set is required to ensure the correctness of the foreground detection. As memory is a constraint, we down sample the video frames to account for the memory used to store the training data. We then optimize the segments of the code that takes longer runtime using CUDA. The final expected result is a real-time implementation of the PracReProCS algorithm using GPU.

The results are shown by below in Fig. 2:

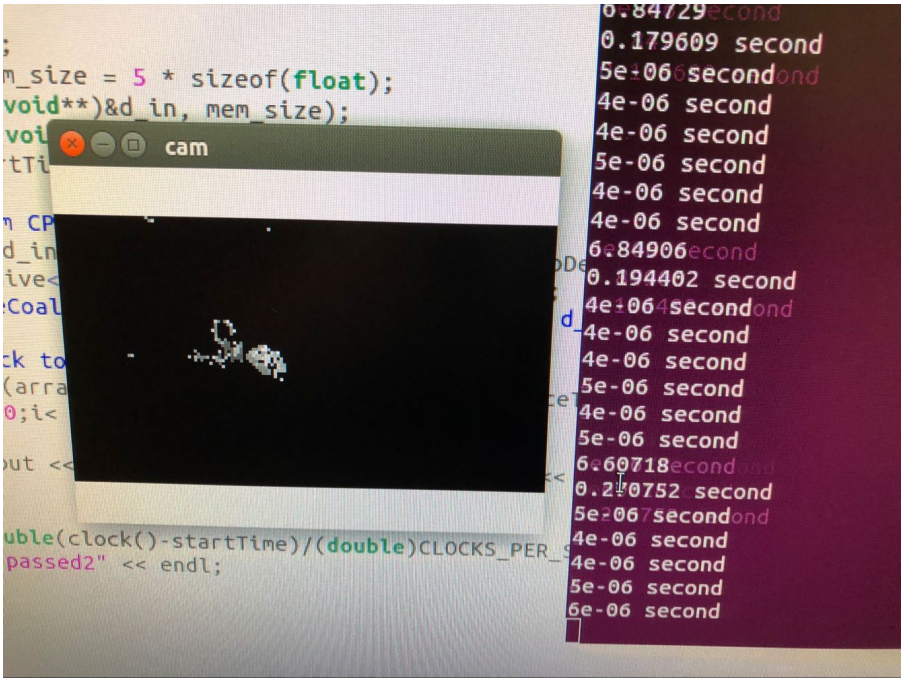

Fig. 2. Foreground and background layer recovery results using ReProCS (ReProCS-pCA) in C++.

From Figure 2 we can see that, the background has been totally blocked and only the foreground is been retained which is a person sitting in front of the table with her backpack. Also the quality of the video is relative good and most of the noises have been blocked.

Advantages of GPU parallel programming

In general, all the programs are running in CPU which has limited memory and therefore the speed pf program is really low. Otherwise, because of the limited memory, all the work in CPU has to be in sequential which means a lot of work which originally can be realized at the same time, now have to wait until other parts of work have been finished or in other words, there are some memory for them in CPU. This shortcoming is caused by relative fewer cores in CPU. But the appearance of GPU helps us to run some parts of work at the same time because that GPU has thousands of cores which can be seen in Fig. 3.

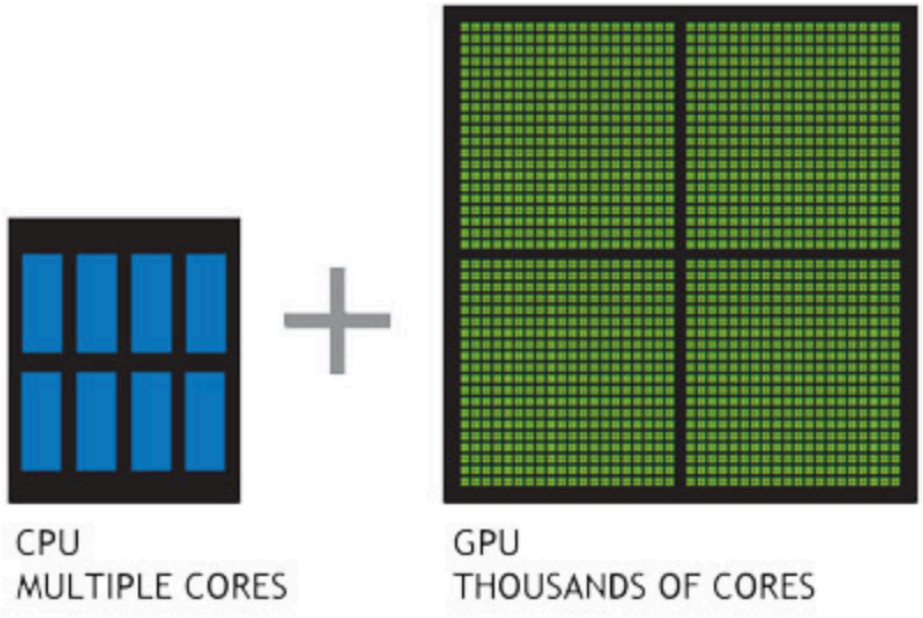

Fig. 3. CPU and GPU cores comparison.

In order to accelerate the running time of a lot of application, people always get help from graphics processing unit which is known as GPU. GPU has thousands of cores which can run different work at the same time on different threads. TABLE I shows some data which obtained by using different ways to realize our basic transpose function program:

TABLE I

transpose(camera_frame)	transpose(CPU)	transpose_naive(GPU)
0.000121s	0.000411s	0.000018s

From TABLE I we can find that after using GPU parallel architecture, the running time has been decreased a lot.

CHAPTER II

TRANSPOSE FUNCTION REALIZATION IN CUDA

Introduction to CUDA

In order to implement this algorithm in GPU, we are using a parallel computing platform called CUDA which is an application programming interface created by Nvidia. It is a really good tool for software engineers to use a CUDA-enabled graphics processing unit. The CUDA platform can work with a lot of programming languages such as C, C++, Fortran. This is easier for programmers to choose whatever the languages they like and then implement it in CUDA. Next, we want to talk about how CUDA works and how the data exchanged between CPU and GPU as shown in Figure 4.

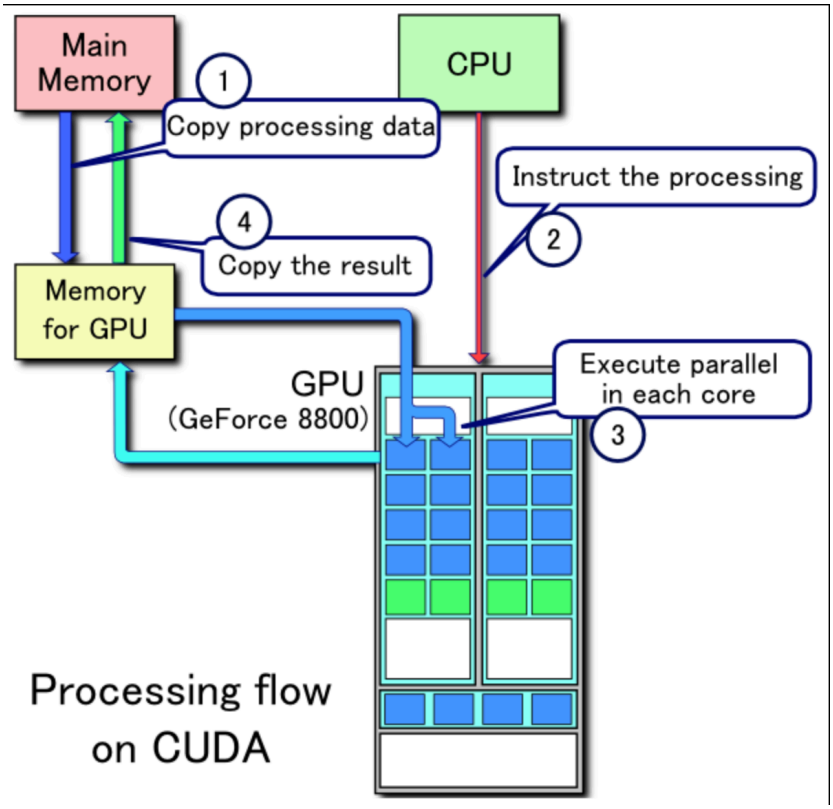

Fig. 4. Processing flow in CUDA

Because our whole program is still in CPU, all the data are still kept in CPU and we only use GPU to do part of our work at the same time in parallel. Therefore, the first step is to transfer our data in CPU to GPU. In other words, we want to copy data from our main memory in CPU to GPU memory. Secondly, we need to receive some instructions from CPU and pass it to GPU. After the GPU receive its instructions and then we could put different data and let the GPU run different processes in different threads in GPU in parallel. Finally, after we got the results in GPU, we need to transfer the results back to CPU. So we copy our results from the GPU memory back to CPU memory.

Transpose function in mathematics

In order to test the performance of GPU parallel architecture by using CUDA programming, we start from an easy but really important function in our algorithm: transpose function. So how the transpose function works in mathematics, as we can see from this Figure 5:

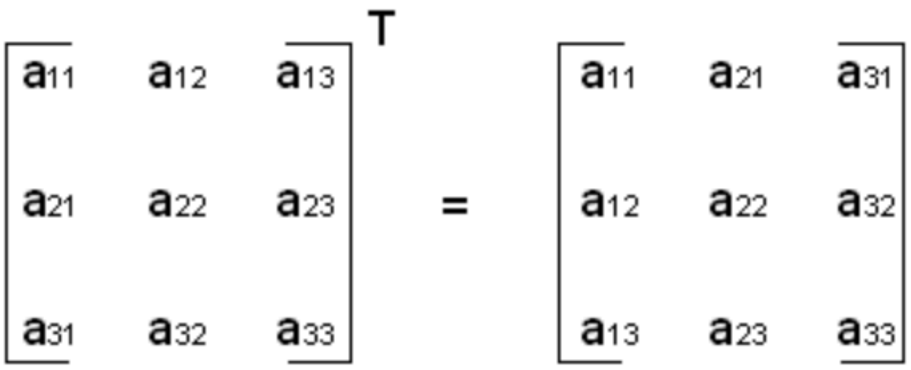

Fig. 5. Transpose function

We first started to run simple matrix copy kernels, because after we understand how the copy operation works and then we just need to reverse the index of x and y to get the transpose function. There are several steps to need to do if we want to realize this function in GPU by using CUDA. Firstly, we need to allocate enough memory in GPU. Secondly, we need to launch

different kernels to perform our operations. Thirdly, we need to transfer our data from the host which is known as CPU to the device which is known as the GPU. Finally, after we got the result, we need to transfer the data back to host and free the memory we are using in the GPU in order to make use of it next time. As discussed in An Efficient Matrix Transpose in CUDA C/C++ by Mark Harris, for both matrix copy function and transpose function, the relevant performance metric is effective bandwidth, calculated in GB/s. All the kernels we are using in these functions launch blocks of 32*8 threads and each thread block transpose a tile of size 32*32. Using the block with fewer threads will decrease the running time for the elements index. The copy and transpose example we are using in this paper are received from An Efficient Matrix Transpose in CUDA C/C++ by Mark Harris published in February 18, 2013. The kernels used in in those examples map threads to matrix elements by using Cartesian (x,y) mapping: threadIdx.x is horizontal and threadIdx.y is vertical.

Figure 6 is the example that how do we realize the copy function in CUDA:

```
__global__ void copy(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[(y+j)*width + x] = idata[(y+j)*width + x];
}
```

Fig. 6. Copy function in CUDA

TransposeNaive function

The TransposeNaive function as we can see from Figure 7 is very similar with the copy function, the only difference is that the indices for odata are flipped.

```
__global__ void transposeNaive(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}
```

Fig. 7. TransposeNaive function in CUDA

So we start from read from idata which is same with the copy function and all the data which reads from idata are coalesced as in the copy kernel. Figure 8 is the results of the copy and TransposeNaive kernels effective bandwidth which calculated in GB/s.

	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
transposeNaive	18.8	55.3

Fig. 8. Effective Bandwidth

The TransposeNaive kernel achieves only a small part of the effective bandwidth of the copy kernel. Because the kernel didn't do anything except for copying.

TransposeCoalesced function

The last function by just using copy without using the shared memory has really poor performance. The resolution for the poor performance is to use the shared memory to avoid the

large strides through global memory. Figure 9 is a graph to describe how we transfer data by using another shared memory.

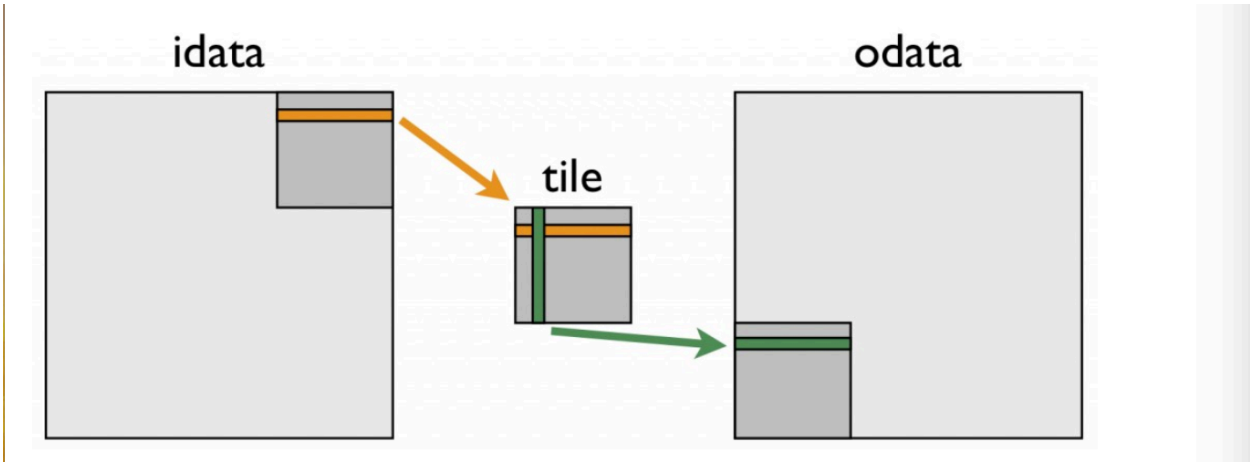

Fig. 9. Coalesced transpose via shared memory

And Figure 10 is how to code it in CUDA by using shared memory.

```
__global__ void transposeCoalesced(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Fig. 10. Coalesced transpose function via shared memory

In the first loop, a bunch of threads reads a lot of contiguous data from idata into rows of the standard memory tile. After recalculating the array indices, a column of the shared memory tile is written to contiguous addresses in odata.

Figure 11 is the comparison between different functions' effective bandwidth.

	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
copySharedMem	104.6	152.3
transposeNaive	18.8	55.3
transposeCoalesced	51.3	97.6

Fig. 11. Effective Bandwidth comparison

The transposeCoalesced results are a big progress over the transposeNaive function, but it is still really worse compared with the performance of the copy kernel.

Profiling and profiler

In computer programming, profiling is used to help analyze the dynamic program which measure the time complexity of a program. Profiling can be realized by some program source code or its binary executable form tool called profiler.

In the previous chapter, in order to show the advantages of GPU implementing over the CPU implementing, we just use CPU timer to find the time taken by a block of code to execute. However, since we want to get more accurate results for different functions, we could use profiler to find more accurate results.

Transpose function

We have already covered how to implement transpose function in two different ways in the previous chapter. Then we are using profiler to compare the time complexity of transpose function in GPU and CPU and also try to compare the time complexity between different transpose functions in GPU.

There is already a really convenient tool in Linux to help us to manage our program which is called GProf. GProf is a profiling program which collects and arranges statistic on our programs. According to Steve Wolfman, it will look into each of our functions and inserts some code at the beginning and the end of each one to collect timing information. After it collect all the time information, it will arrange the information properly and when we run this program as normal, it will create an execution file called “gomon.out” which has all the raw data which turns the gprof program into some profiling statistics. But the gprof is only working for the programs which are on CPU.

Fortunately, there is another tool called nvprof which is working for all programs on GPU programmed in CUDA. We use nvprof and gprof in the same way and we could also get some timing statistics for our programs.

Figure 12 shows that transpose function implemented in CPU and its timing statistics:

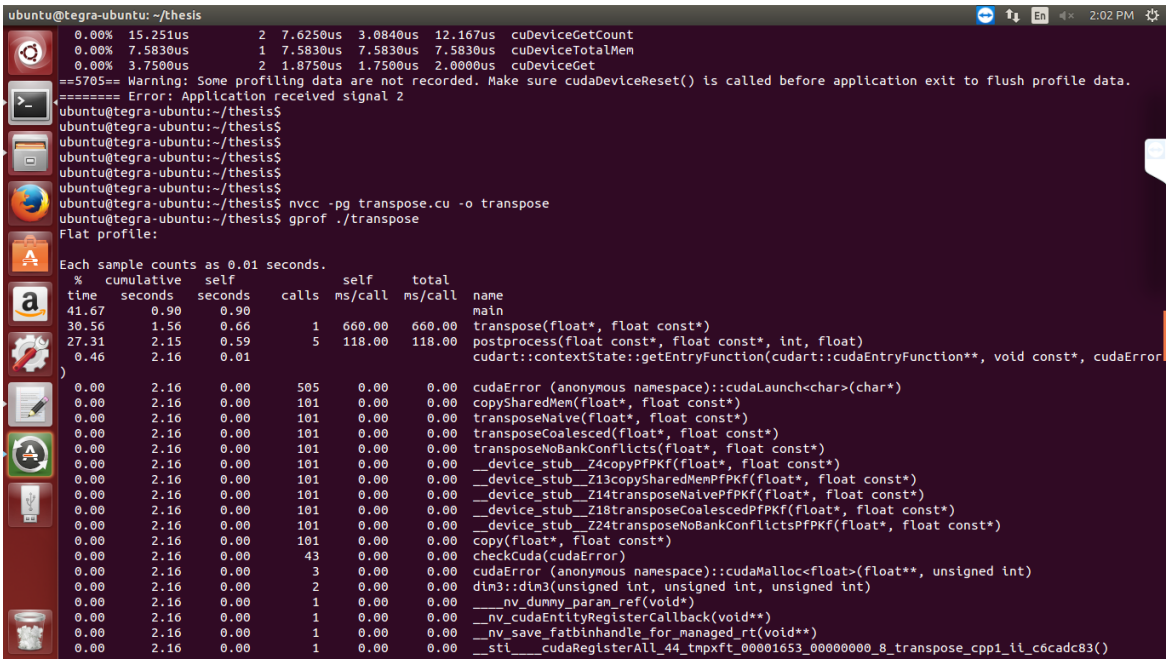

Fig. 12. Transpose function in CPU

We can find out that the total running time for transpose function in CPU is around 660 ms.

Next, we used nvprof to find out the timing statistics for different transpose functions in GPU. Figure 13 shows all the results:

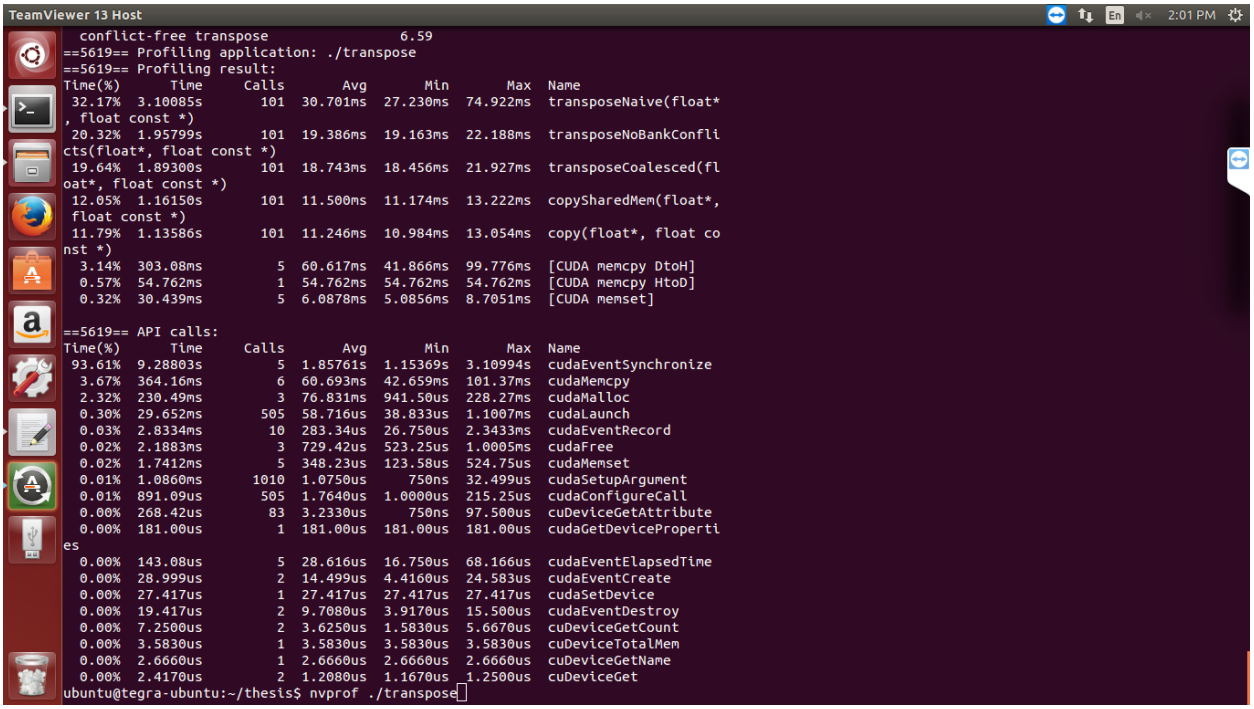

Fig. 13. Transpose function in GPU

By collecting all the statistics for transpose functions in CPU and GPU, we made TABLE II to compare the timing difference clearly.

TABLE II

transpose(CPU)	transposeNaive(GPU)	transposeCoalesced (GPU)
660.1ms	30.701ms	18.743ms

And from this table, we can find out that the running time has been reduced a lot in GPU compared with CPU, from 660 ms to 30ms. And among the transpose functions in GPU, we can find out that, if we are using the shared memory to do the transpose function, the time will also be reduced a little bit.

After going through all of our programs, we found that we used a lot of multiplications and additions, so if we could also implement the multiplication and addition function in GPU by using CUDA in parallel, the time complexity will be optimized a lot.

Multiply function

For the multiply function implemented in GPU, as what we did for the transpose function, we first created memory in GPU, then we copy the data from CPU to GPU, after that, we do all the multiplication function in GPU as normally in CPU, finally, we need to transfer the results back to CPU and also free the memory we used in GPU.

Figure 14 is a piece of code of multiplication function in CUDA:

```
__global__ void gpu_matrix_mult(int *a,int *b, int *c, int m, int n, int k)
{
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
int sum = 0;
if( col < k && row < m)
{
for(int i = 0; i < n; i++)
{
sum += a[row * n + i] * b[i * k + col];
}
c[row * k + col] = sum;
}
}
```

Fig. 14. Multiply function in GPU

After using pgrof to collect the statistics of multiply function in CPU and using nvprof to collect the statistics of multiply function in GPU, we got this raw data for the timing statistics. Figure 15 shows all the timing statistics.

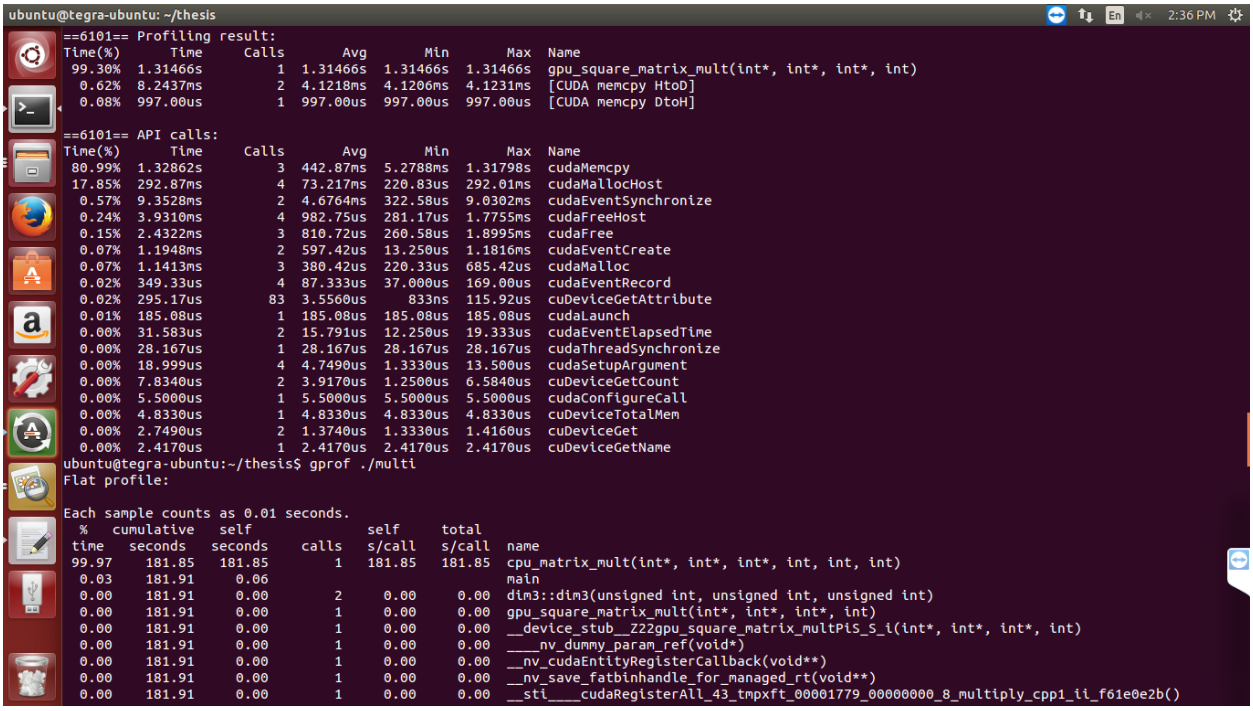

Fig. 15. Multiply function timing statistics

By recording the average time for GPU functions and the total time for CPU functions, we made TABLE III to compare between the same multiply function in CPU and GPU:

TABLE III

multiply(CPU)	multiply(GPU)
181.85s	1.31466s

From this table, we can find out that the running time for the multiply function has been reduced from 181.85 seconds to 1.31466 seconds which means our program has been optimized a lot.

Addition function

As doing the same procedure as what we did in transpose and multiply functions, we implement our addition function in GPU by CUDA. Figure 16 gives us the piece of code which implement addition function in CUDA:

```
__global__ void gpu_matrix_addition(int *a,int *b, int *c, int m, int n, int k)
{
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
int sum = 0;
if( col < k && row < m)
{
for(int i = 0; i < n; i++)
{
sum += a[row * n + i] + b[i * k + col];
}
c[row * k + col] = sum;
}
}
```

Fig. 16. Addition function in GPU

Same as before, we used gprof to compile the addition function in CPU and nvprof to compile the addition function in GPU. Figure 17 gives us the results of timing statistics:

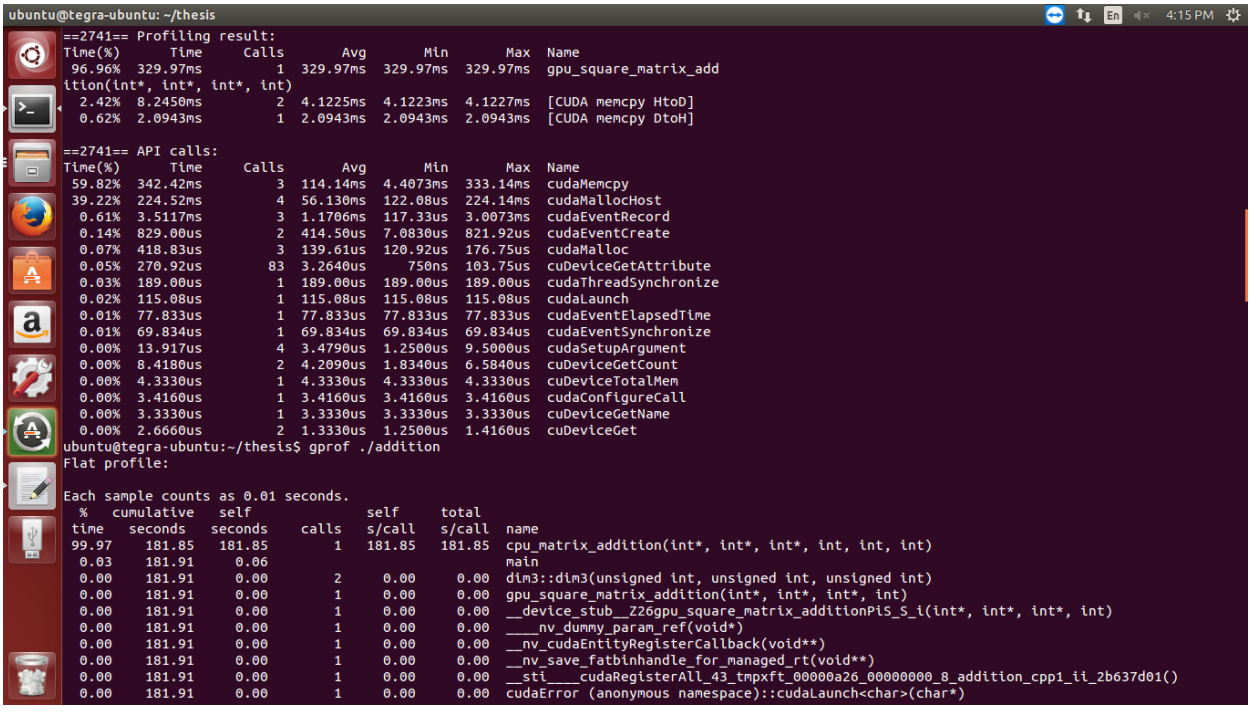

Fig. 17. Addition function timing statistics

We extracted the useful information from the raw data the profiler gave to us, we made TABLE IV:

TABLE IV

addition(CPU)	addition(GPU)
181.85s	114.14ms

From this table, we can find out that, after implementing the addition function in GPU, the running time for the same addition function has been reduced from 181.85 seconds to 114.14 ms.

CHAPTER III

L1 MINIMIZATION REALIZATION IN CUDA

The most important part in our algorithm is l1minimization and it helps to extract the dynamic information from the background. So if we want to improve our program in a big step, we should think about how to implement l1 minimization in GPU by using CUDA. At the beginning, we want to use the built in function in CUDA to implement our l1 minimization function, since CUDA is in its first period, so we have to implement it by ourselves. However, before we could implement it in GPU, we should understand l1 minimization well.

L1 minimization algorithm

First, we will introduce the l1 minimization from mathematics. L1 minimization has been really useful in signal processing and program optimizations. Specifically, given an unknown signal $x_0 \in R^n$, an underdetermined full-rank matrix $A \in R^{d \times n}$, and a $b = Ax$, the l1 minimization problem solves the following convex problem:

$$\min ||x||_1 \ subject\ to\ b = Ax$$

So our basic task is to solve this optimization problem.

ADMM Dual Prime method for l1 minimization

The algorithm that we used implement their algorithm in Matlab, and there is a package in Matlab to solve this l1 minimization problem. After we went through the package they used in the l1 minimization package, we find that they are using a method called ADMM Dual Prime method for l1 minimization. And we first will talk about how the ADMM Dual Prime method works.

Consider minimization problem:

$$\min_{x \in R^n} f_1(x_1) + f_2(x_2) \; subject \; to \; A_1X_1 + A_2X_2 = b$$

As usual, we augment the objective

$$\underset{x \in R^n}{\text{Min}} f_1(x_1) + f_2(x_2) + \frac{\rho}{2}||A_1X_1 + A_2X_2 - b||_2^2$$

$$subject \; to \; A_1X_1 + A_2X_2 = b$$

Write the augmented Lagrangian as

$$L_p(x_1, x_2, u) = f_1(x_1) + f_2(x_2) + u^T(A_1X_1 + A_2X_2 - b) + \frac{\rho}{2}||A_1X_1 + A_2X_2 - b||_2^2$$

ADMM method repeats these steps, for k = 1,2,3… until find the optimized solutions

$$x_1^{(k)} = \underset{x_1 \in R^{n_1}}{\operatorname{argmin}} L_\rho\left(x_1, x_2^{(k-1)}, u^{(k-1)}\right)$$

$$x_2^{(k)} = \underset{x_2 \in R^{n_2}}{\operatorname{argmin}} L_\rho\left(x_1^k, x_2, u^{(k-1)}\right)$$

$$u^{(k)} = u^{(k-1)} + \rho(A_1x_1^{(k)} + A_2x_2^{(k)} - b)$$

Matlab YALL1 package pseudo code

Our problem is to solve the l1 minimization problem as shown in the previous part. By extending the restriction, we got this form's l1 minimization:

L1 minimization

$$\min_x \|x\|_1 \;\; s.t. \; \|y_t - \varphi_t x\|_2 \leq \varepsilon$$

And YALL1 package is using ADMM method to solve this problem. Next we give the pseudocode we using in Matlab and we could follow this pseudocode to implement this algorithm in GPU.

Dimensions and initial values

Video frame dimension, number of rows * columns = p;

Dimension of y_t = p*1;

Mat_zeroes(r,c) : matrix of zeroes with r rows and c columns;

Tolerance: tol = 10^{-3} ;

Delta = $\|L_{t-1}\|_2$: Background frame at t-1;

Non-orth = True, for non-orthonormal rows;

3 main functions in the l1 solver package

1. Initial: sets initial values
2. Solve: l1 solver function
3. Check: function that has several stopping conditions for the iterations of l1 solver

Initial function

Input - y_t, φ_t

Output - x

randn(s1,0,1): fills array s1 with normally distributed random numbers of mean(0)and standard deviation(1)

s2 = $\varphi_t * (\varphi_t * s1)$

error = $\frac{\|s1 - s2\|_2}{\|s1\|_2}$

If error > 10^{-12}, non-orth = True

Else non-orth = False

If delta > 0 and $\|y_t\|_2 \leq delta$

{

x = mat_zeroes(p,1) ;

Return x;

}

$y_{max} = \|y_t\|_\infty$

if delta > 0 delta = $\frac{delta}{y_{max}}$

$y_1 = \frac{y_t}{y_{max}}$

x_1 = solve ($\varphi_t, y_1, non - orth$)

$x = x_1 * y_{max}$

return x

Solve function

Input - $\varphi_t, y_1, non - orth$

Output – x_1

mu = mean($|y_1|$)

$x_1 = \varphi_t * y_1$

if non-orth {

y = mat_zeroes (p,1);

φ'_{ty} = mat_zeroes (p,1);

}

$y_{dmu} = \frac{y_1}{mu}$

$d_{dmu} = \frac{delta}{mu}$

z = mat_zeroes (p,1)

for (iter = 0; iter < 9999; iter++){

$x_{dmu} = \frac{x_1}{mu}$

If non-orth = False {

y = $\varphi_t * (z - x_{dmu}) + y_{dmu}$

if delta > 0, y = 1 - $\frac{d_{dmu} * y}{\|y_t\|_2}$

}

Else {

r_y = $\varphi_t * (\varphi'_{ty} - z + x_{dmu}) - y_{dmu}$

$\varphi_{try} = \varphi'_t * r_y$

Stp = $\frac{r'_y * r_y}{(\varphi'_{try} * \varphi_{try}) + 2^{-52}}$

y = y – stp * r_y

}

$\varphi'_{ty} = \varphi'_t * y$

z = $\varphi'_{ty} + x_{dmu}$

z(i) = $\frac{z(i)}{\max(1,|z(i)|)}$

$r_d = \varphi'_{ty} - z$

$x_p = x_1$

$x_1 = x_1$ + (1.618 * mu) * r_d

Bool stop = check(x_1, x_p, y, r_d)

If stop, break

}

Return x1

Check function

Input - x_1, x_p, y, r_d

Output – True or False

Stop = False

if delta > 0, q = 0

else, q = 0.1

$x_{rel} = \frac{\|x_1 - x_p\|_2}{\|x_1\|_2}$

If $x_{rel} < tol * (1 - q)$ {stop = True; return stop}

If $x_{rel} \geq tol * (1 + q)$ return stop

If $\|r_d\|_2 \geq tol * \sqrt{p}$ return stop

Objp = $\sum_{i=1}^{p} x_1(i)$

Objd = $y_1' * y$

If $|\text{Objd} - \text{Objp}| \geq tol * |\text{Objp}|$ return stop

$r_p = \varphi_t * x_1$

If $\|r_p\|_2 \leq tol * \|y_1\|_2$ stop = True

Return stop

CHAPTER III

CONCLUSION

Present difficulties

Even though we have finished reviewing how the l1 minimization works in mathematics and also we understand how they are implemented in Matlab by using YALL package, we still can't decide which parts can be run in parallel and which parts have to wait until some other procedure finished. Therefore, the biggest difficulties for us right now is still trying to understand l1 minimization algorithm better and find out which parts can be run at the same time in parallel in GPU.

Future expectation

We will still keep working on this project and trying to find the best way to implement this algorithm in GPU by using CUDA.

REFERENCES

[1]H.Guo, C. Qiu and N. Vaswani, "An Online Algorithm for Separating Sparse and Low-Dimensional Signal Sequences From Their Sum " in IEEE transactions on signal processing, VOL. 62, NO. 16, AUGUST 15, 2014

[2] J. Sanders, E. Kandrot, "CUDA by Example, An introduction to general-purpose GPU programming", 2010

[3] J. Yang, Y. Zhang, "Alternating direction algorithms for l1-problems in compressive sensing" in CAAM Technical Report TR09-17, JUNE 6, 2010

[4] Y. Zhang, "User's guide for YALL1: Your algorithms for l1 optimization" in CAAM Technical Report TR09-17, MAY 13, 2009

[5] A. Yang, S. Sastry, A. Ganesh, Y. Ma, "Fast l1- minimization algorithms and an application in robust face recognition: a review" in 0 IEEE 17th International Conference on Image Processing, September 26-29, 2010

[6] B. Poczos, R. Tibshirani, "Dual methods and ADMM" in Convex Optimization 10-725/36-725

[7] Steve Wolfman, "What is gprof?"