Project on Local Contrast Enhancement using NVIDIA Cuda

Mentor: **Dr. Sushanta Mukhopadhayay**

1. Rajesh kumar sinha 2013JE0333
2. Ashish kumar 2013JE0606
3. Parichaya walia 2013JE0336
4. Harshit 2013JE0455
5. Ajit singh 2013JE0567
6. Maheswara Reddy Chennuru 2013JE0889

Introduction –

CUDA® is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).
With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA. Here are a few examples:

Identify hidden plaque in arteries: Heart attacks are the leading cause of death worldwide. Harvard Engineering, Harvard Medical School and Brigham & Women's Hospital have teamed up to use GPUs to simulate blood flow and identify hidden arterial plaque without invasive imaging techniques or exploratory surgery.

Analyze air traffic flow: The National Airspace System manages the nationwide coordination of air traffic flow. Computer models help identify new ways to alleviate congestion and keep airplane traffic moving efficiently. Using the computational power of GPUs, a team at NASA obtained a large performance gain, reducing analysis time from ten minutes to three seconds.

Visualize molecules: A molecular simulation called NAMD (nanoscale molecular dynamics) gets a large performance boost with GPUs. The speed-up is a result of the parallel architecture of GPUs, which enables NAMD developers to port compute-intensive portions of the application to the GPU using the CUDA Toolkit.
-

This entire project is based on Image processing using CUDA libraries. Here, we have implemented three codes using the cuda libraries.

- 1. Burn combine
- 2. Image Close reconstruction
- 3. Image Open reconstruction

In all the above programs, we have created threads for simultaneous execution of the same procedure, so as to yield an improved running time. It has been implemented using CUDA libraries. CUDA enables us to use the concept of multiprocessing in the field of image processing too.

The Header file:

imageio.h

This header file allows us to take the input of an image file and to give the output of an image file. It takes the input as a .pgm and .ppm file.

```
#include<stdio.h>
#include<stdlib.h>
#define FALSE 0
#define TRUE 1
#define uc unsigned char

int i,j;
struct imagestruct{
   int *image;
   int numberOfColumns, numberOfRows, numberOfBands, highVal, totalPixels, header;
};

int readImage(FILE *fpIn,struct imagestruct *img){
        char * string;
        int doneReading = FALSE;
        char c;
        if(fpIn == NULL){
           printf("Such a file does not exist...");
        }
        string = (char *) calloc(256,1);
        while(!doneReading && (c = (char) fgetc(fpIn)) != '\0')
                switch(c){
                        case 'P':
                                c = (char) fgetc(fpIn);
                                switch(c){
                                        case '2':
```

```
                        img->header = 2;
                        img->numberOfBands = 1;
                        //pgm plain
                break;
                case '5':
                        img->header = 5;
                        img->numberOfBands = 1;
                        //pgm Normal
                break;
                case '3':
                        img->header = 3;
                        img->numberOfBands = 3;
                        //ppm plain
                break;
                case '6':
                        img->header = 6;
                        img->numberOfBands = 3;
                        //ppm Normal
                break;
        }
        c = (char) fgetc(fpIn);
        if(c != 0x0A){
                ungetc(c, fpIn);
        }
        else{
                ungetc(c, fpIn);
                fgets(string, 256, fpIn);
        }
break;
case '#':
fgets(string, 256, fpIn);
printf("File you entered is %s\n",string);
```

```
                        break;
                        case '1':
                        case '2':
                        case '3':
                        case '4':
                        case '5':
                        case '6':
                        case '7':
                        case '8':
                        case '9':
                                ungetc(c, fpIn);
                                fscanf(fpIn,"%d %d %d", &(img->numberOfColumns), &(img->numberOfRows), &(img->highVal));
                                doneReading = TRUE;
                                fgetc(fpIn);
                        break;
                }
        img->totalPixels = img->numberOfRows*img->numberOfColumns*img->numberOfBands;
        img->image = (int *) malloc (img->totalPixels);
        fread(img->image,1,img->totalPixels,fpIn);
        printf("Reading the image was sucessfull...\n");
        return 0;
}
int writeImage(FILE *fpOut,struct imagestruct *img){
        if(fpOut == NULL){
           printf("Error couldn't write the image ...\n");
        }
        fprintf(fpOut, "P%d\n%d %d\n%d\n",img->header,img->numberOfColumns, img->numberOfRows, img->highVal );
        fwrite(img->image,1,img->totalPixels,fpOut);
        printf("Wrote the image into ...\n");
        return 0;
}
```

Burn Combine

Cuda_functions_burn_combine.h

This header file contains all the Cuda functions that have been used in the main program. All the functions are related to combining the images and modifying a particular image in the desired way.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);
__global__ void work(int *c, const int *a, const int *b,int sel)
{
    int i = threadIdx.x;
      switch(sel)
    {
        case 1:
            c[i] = a[i] + b[i];
            break;
        case 2:
            c[i] = a[i] - b[i];
            break;
        case 3:
            c[i] = a[i] * b[i];
            break;
        case 4:
```

```
            c[i] = (a[i]+1) / (1+b[i]);
            break;
        case 5:
            c[i] = a[i] - b[i] ;
            if(c[i]<0)c[i]=-c[i];
            break;
        case 6:
            if(a[i]==b[i])c[i]=0;
            else c[i]=255;
            break;
        case 7:
            if(a[i]==b[i])c[i]=255;
            else c[i]=0;
        case 8:
            c[i]=max(a[i],b[i]);
            break;
        case 9:
            c[i]=min(a[i],b[i]);
            break;
        case 10:
            c[i] = (a[i] + b[i])/2;
            break;
 }
    //printf("%d %d %d",a[i],b[i],c[i]);
}
// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size,int sel)
{
    int *dev_a = 0;
    int *dev_b = 0;
```

```
int *dev_c = 0;
cudaError_t cudaStatus;
// Choose which GPU to run on, change this on a multi-GPU system.
cudaStatus = cudaSetDevice(0);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");
    goto Error;
}
// Allocate GPU buffers for three vectors (two input, one output)    .
cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Copy input vectors from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
```

```
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    // Launch a kernel on the GPU with one thread for each element.
    work<<<1, size>>>(dev_c, dev_a, dev_b, sel);

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
        goto Error;
    }

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n",
cudaStatus);
        goto Error;
    }

    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

```
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    return cudaStatus;
}
```

Kernel.cu

The main function which implements several operations on the image.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include "imageio.h"
#include "cuda_functions_burn_combine.h"

int main()
{

            FILE *fpIn1, *fpIn2, *fpOut;
            fpIn1 = fopen("input2.pgm", "rb");
            fpIn2 = fopen("input2.pgm", "rb");
            fpOut = fopen("avg.pgm", "wb");
```

```
        struct imagestruct img1, img2;
        readImage(fpIn1, &img1);
        readImage(fpIn2, &img2);
        printf("%d  %d", img1.totalPixels, img2.totalPixels);

        cudaError_t cudaStatus = addWithCuda(img1.image, img1.image, img2.image, 1000,1);
        writeImage(fpOut, &img1);
}
```

Here is the output of adding two images.

Close reconstruction

Cuda_close_recons.h

This header file contains all the CUDA functions related to close reconstruction of an image using a suitable structuring element.

```
__global__ void cuda_check(int *a, int *b)
{
   int i = threadIdx.x;
   if(a[i]!=b[i])
   {
      int t=a[i];
      a[i]=b[i];
      b[i]=t;
   }
   //printf("%d %d %d",a[i],b[i],c[i]);
```

```
}

__global__ void cuda_U_Limit(int *a, int *b)
{
    int i = threadIdx.x;
    if(a[i]<b[i])
    b[i]=a[i];
    //printf("%d %d %d",a[i],b[i],c[i]);
}

__global__ void cuda_shift(int *ff, int *bb,int row,int col,int xorg,int yorg)
{
    int i = threadIdx.x;
    int ii,jj;
    ii = i%col;
            jj = i / col;
    int temp=ii*col + jj;
    int temp2=(ii+xorg)*col+jj+yorg;
    bb[temp] = ff[temp2];
}
```

Kernel.cu

The main function does the close reconstruction of an image.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
```

```
#include "imageio.h"
#include "cuda_close_recons.h"

FILE  *imgfilein, *imgfileout;
int   xsize, ysize, sxsize, sysize, xorg, yorg, ngr, count, itern;
int   *f, *c, *d, **s;

void determineIndexInImage(int index, int row, int col, int* x, int* y)
{
        *x = index%col;
        *y = index / col;
        return;
}

int max(int **t)
{
        int ii, jj, x;

        x = t[0][0];
        for (ii = 0; ii <= sxsize - 1; ii++)
                for (jj = 0; jj <= sysize - 1; jj++)
                        if (x < t[ii][jj]) x = t[ii][jj];

        return(x);
}

int determineIndexInArray(int x, int y, int col)
{
        return y*col + x;
}
```

```
int **alloc_mem(int row_size, int col_size)
{

        int i, j;
        int **p;

        p = (int **)malloc(row_size*sizeof(int *));
        for (i = 0; i<row_size; i++) p[i] = (int *)malloc(col_size*sizeof(int));

        return(p);
}/* end alloc_mem() */

int *alloc_mem_1D(int row_size, int col_size)
{
        int *p;
        p = (int *)malloc(row_size*col_size*sizeof(int));
        return(p);
}



int min(int** t, int ix, int fx, int iy, int fy)
{
        int minm, i, j;

        minm = t[ix][iy];
        for (i = ix; i <= fx - 1; i++)
                for (j = iy; j <= fy - 1; j++)
                        if (t[i][j]<minm) minm = t[i][j];
```

```
	return(minm);
}

int dilation(int* ff, int stxsize, int stysize, int** ss, int* bb)//cuda
{
	int ii, jj, k, l, temp1, temp2, minm, **aa;
	int new_xsize, new_ysize;



	aa = alloc_mem(stxsize, stysize);

	for (ii = 0; ii < xsize*ysize; ii++)
		bb[ii] = 0;



	new_xsize = xsize + stxsize - xorg;
	new_ysize = ysize + stysize - yorg;
	for (ii = -xorg; ii <= new_xsize - 1; ii++)
		for (jj = -yorg; jj <= new_ysize - 1; jj++)
		{
			int tempo = determineIndexInArray(ii + xorg, jj + yorg, xsize);
			for (k = 0; k <= stxsize - 1; k++)
				for (l = 0; l <= stysize - 1; l++)
				{
					temp1 = k - xorg;
					temp2 = l - yorg;
					if ((ii - temp1) >= 0 && (jj - temp2) >= 0 && (ii - temp1)<xsize
&& (jj - temp2)<ysize)
						aa[k][l] = ff[determineIndexInArray(ii - temp1, jj -
temp2, xsize)] + ss[k][l];
					else   aa[k][l] = -999;
```

```
                        }
                    minm = min(aa, 0, stxsize, 0, stysize);
                    if (minm <0)bb[tempo] = 0;
                    else if (minm > ngr - 1) bb[tempo] = ngr - 1;
                    else bb[tempo] = minm;
                }
        return(0);
}

int erosion_3x3(int* ff, int* bb)//cuda
{
        int ii, jj, k, l, temp1, temp2, minm, **aa;
        int new_xsize, new_ysize;
        int bxsize = 3, bysize = 3;

        aa = alloc_mem(bxsize, bysize);

        for (ii = 0; ii < xsize*ysize; ii++)
                bb[ii] = 0;

        for (ii = 0; ii <= xsize - 1; ii++)
                for (jj = 0; jj <= ysize - 1; jj++)
                {
                    int tempo = determineIndexInArray(ii, jj, xsize);
                    for (k = 0; k <= 2; k++)
                        for (l = 0; l <= 2; l++)
                        {
                            temp1 = k - 1;
                            temp2 = l - 1;
                            if ((ii - temp1) >= 0 && (jj - temp2) >= 0 && (ii - temp1)<xsize
&& (jj - temp2)<ysize)
```

```
                                                aa[k][l] = ff[determineIndexInArray(ii - temp1, jj -
temp2, xsize)];
                                        else   aa[k][l] = -999;
                                }
                        minm = min(aa, 0, 3, 0, 3);
                        if (minm <0)bb[tempo] = 0;
                        else if (minm > ngr - 1) bb[tempo] = ngr - 1;
                        else bb[tempo] = minm;
                }

        return(0);
}

int shift(int* ff, int* bb)//cuda
{
        int ii, jj;

        for (ii = 0; ii < xsize*ysize; ii++)
                bb[ii] = 0;

        int *dev_a, *dev_b;
        int size = xsize*ysize;
        cudaMalloc((void**)&dev_a, xsize * ysize * sizeof(int));
        cudaMalloc((void**)&dev_b, xsize * ysize * sizeof(int));
        cudaMemcpy(dev_a, ff, size* sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, bb, size * sizeof(int), cudaMemcpyHostToDevice);

        cuda_shift << <1, size >> > (dev_a, dev_b, ysize, xsize, xorg, yorg);
        cudaMemcpy(ff, dev_a, size * sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(bb, dev_b, size * sizeof(int), cudaMemcpyDeviceToHost);
        cudaFree(dev_a);
```

```
	cudaFree(dev_b);

	return(0);
}


int erosion(int* ff, int stxsize, int stysize, int** ss, int* bb)//cuda
{
	int ii, jj, k, l, temp1, temp2, **aa;
	int new_xsize, new_ysize;

	aa = alloc_mem(stxsize, stysize);

	for (ii = 0; ii < xsize*ysize; ii++)
		bb[ii] = 0;


	new_xsize = xsize - stxsize + xorg;
	new_ysize = ysize - stysize + yorg;

	for (ii = xorg; ii <= new_xsize - 1; ii++)
		for (jj = yorg; jj <= new_ysize - 1; jj++)
		{
			int tempo = determineIndexInArray(ii, jj, xsize);
			for (k = 0; k <= stxsize - 1; k++)
				for (l = 0; l <= stysize - 1; l++)
				{
					temp1 = k - xorg;
					temp2 = l - yorg;
					aa[k][l] = ff[determineIndexInArray(ii + temp1, jj + temp2,
xsize)] - ss[k][l];
```

```
                                        }


                                bb[tempo] = min(aa, 0, stxsize, 0, stysize);
                                if (bb[tempo] < 0) bb[tempo] = 0;
                                if (bb[tempo] > ngr - 1) bb[tempo] = ngr - 1;
                        }
        return(0);
}

int gl_close(int *ff, int stxsize, int stysize, int **ss, int *bc)
{
        int *bb, *cc;

        bb = alloc_mem_1D(xsize + stxsize, ysize + stysize);
        cc = alloc_mem_1D(xsize + stxsize, ysize + stysize);

        dilation(ff, stxsize, stysize, ss, bb);
        erosion(bb, stxsize, stysize, ss, cc);
        shift(cc, bc);

        return(0);
}
int struc_element()
{
        int ii, jj;
        /*
        printf("enter no. of rows of the structuring element :");
        scanf("%d", &sxsize);
        printf("enter no. of columns of the structuring element :");
```

```
	scanf("%d", &sysize);
	*/
	s = alloc_mem(sxsize, sysize);
	 xorg = sxsize / 2;                                    //mid pt
	yorg = sysize / 2;

	for (ii = 0; ii <= sxsize - 1; ii++)
		for (jj = 0; jj <= sysize - 1; jj++)
			s[ii][jj] = 0;

	return(0);
}


int maxmin()
{
	int amax, amin, temp;
	int   i, j;
	amax = f[0];
	amin = f[0];
	for (i = 0; i <= xsize*ysize; i++)
	{
		if (amax < f[i])amax = f[i];
		if (amin > f[i])amin = f[i];
	}

	ngr = 1;		temp = amax - 1;
	do
	{
		temp = temp / 2;
```

```
                ngr = ngr * 2;
        } while (temp > 0);
 }
int L_limit(int *bb, int *ff)
{
        int ii, jj;
        for (ii = 0; ii<xsize*ysize; ii++)
                if (ff[ii] > bb[ii]) bb[ii] = ff[ii];
}


int check(int *bb, int* ff, int* count)//cuda
{
        int ii, jj, temp;

        int *dev_a, *dev_b;
        int size = xsize*ysize;
        cudaMalloc((void**)&dev_a, xsize * ysize * sizeof(int));
        cudaMalloc((void**)&dev_b, xsize * ysize * sizeof(int));
        cudaMemcpy(dev_a, bb, size* sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, ff, size * sizeof(int), cudaMemcpyHostToDevice);

        cuda_check << <1, size >> > (dev_a, dev_b);
        cudaMemcpy(bb, dev_a, size * sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(ff, dev_b, size * sizeof(int), cudaMemcpyDeviceToHost);
        cudaFree(dev_a);
        cudaFree(dev_b);
        return(0);
}
```

```
int main()
{
	/* check argument errors */
	sxsize = 3;		sysize = 3;
	imgfilein = fopen("input.pgm", "rb");
	imgfileout = fopen("output.pgm", "wb");
	if (imgfilein == NULL)
	{
		printf("Not found input image file. Check directory. \n");
		exit(0);
	}

	if (imgfileout == NULL)
	{
		printf("Error in opening output image file. \n");
		exit(0);
	}

	imagestruct img;
	readImage(imgfilein, &img);//all the parameter of img is set now
	xsize = img.numberOfColumns;
	ysize = img.numberOfRows;

	f = alloc_mem_1D(xsize, ysize);
	 for (int i = 0; i<xsize*ysize; i++)
		f[i] = img.image[i];

	c = alloc_mem_1D(xsize, ysize);
	d = alloc_mem_1D(xsize, ysize);
```

```
        maxmin();
        struc_element();

        gl_close(f, sxsize, sysize, s, c);

        itern = 1;
        do
        {
                erosion_3x3(c, d);
                L_limit(d, f);
                check(c, d, &count);
                /* printf("\n iteration = %d   diff. count = %d \n", itern++, count);  */
        } while (count != 0);

        for (int i = 0; i<xsize*ysize; i++)
                img.image[i] = c[i];
        writeImage(imgfileout, &img);
}
```

Here is the output of close reconstruction of an image.

Open reconstruction

Cuda_open_recons.h

The header file contains all the cuda functions realted to open reconstruction of image.

```
__global__ void cuda_check(int *a, int *b)
{
   int i = threadIdx.x;
```

```
    if(a[i]!=b[i])
    {
        int t=a[i];
        a[i]=b[i];
        b[i]=t;
    }
    //printf("%d %d %d",a[i],b[i],c[i]);
}

__global__ void cuda_U_Limit(int *a, int *b)
{
    int i = threadIdx.x;
    if(a[i]<b[i])
    b[i]=a[i];
 }

__global__ void cuda_shift(int *ff, int *bb,int row,int col,int xorg,int yorg)
{
    int i = threadIdx.x;
    int ii,jj;
    ii = i%col;
            jj = i / col;
    int temp=ii*col + jj;
    int temp2=(ii+xorg)*col+jj+yorg;
    bb[temp] = ff[temp2];
}
```

Kernel.cu

The main function does the open reconstruction of the image.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include "imageio.h"
#include "cuda_open_recons.h"

FILE  *imgfilein, *imgfileout;
int   xsize, ysize, sxsize, sysize, xorg, yorg, ngr, count, itern;
int   *f, *c, *d, **s;

void determineIndexInImage(int index, int row, int col, int* x, int* y)
{
        *x = index%col;
        *y = index / col;
        return;
}

int determineIndexInArray(int x, int y, int col)
{
        return y*col + x;
}
int min(int** t, int ix, int fx, int iy, int fy)
{
        int minm, i, j;
        minm = t[ix][iy];
        for (i = ix; i <= fx - 1; i++)
                for (j = iy; j <= fy - 1; j++)
                        if (t[i][j]<minm) minm = t[i][j];
```

```
	return(minm);
}

int **alloc_mem(int row_size, int col_size)
{

	int i, j;
	int **p;

	p = (int **)malloc(row_size*sizeof(int *));
	for (i = 0; i<row_size; i++) p[i] = (int *)malloc(col_size*sizeof(int));

	return(p);
}

int *alloc_mem_1D(int row_size, int col_size)
{
	int *p;
	p = (int *)malloc(row_size*col_size*sizeof(int));
	return(p);
}

int max(int **t, int stxsize, int stysize)
{
	int ii, jj, x;
	x = t[0][0];
	for (ii = 0; ii <= stxsize - 1; ii++)
		for (jj = 0; jj <= stysize - 1; jj++)
			if (x < t[ii][jj]) x = t[ii][jj];
	return(x);
```

```
}

int erosion(int* ff, int stxsize, int stysize, int** ss, int* bb)//cuda
{
	int ii, jj, k, l, temp1, temp2, **aa;
	int new_xsize, new_ysize;

	aa = alloc_mem(stxsize, stysize);

	for (ii = 0; ii < xsize*ysize; ii++)
		bb[ii] = 0;



	new_xsize = xsize - stxsize + xorg;
	new_ysize = ysize - stysize + yorg;

	for (ii = xorg; ii <= new_xsize - 1; ii++)
		for (jj = yorg; jj <= new_ysize - 1; jj++)
		{
			int tempo = determineIndexInArray(ii, jj, xsize);
			for (k = 0; k <= stxsize - 1; k++)
				for (l = 0; l <= stysize - 1; l++)
				{
					temp1 = k - xorg;
					temp2 = l - yorg;
					aa[k][l] = ff[determineIndexInArray(ii + temp1, jj + temp2,
xsize)] - ss[k][l];
				}
			bb[tempo] = min(aa, 0, stxsize, 0, stysize);
			if (bb[tempo] < 0) bb[tempo] = 0;
		if (bb[tempo] > ngr - 1) bb[tempo] = ngr - 1;
```

```
                }
        return(0);
}
int dilation(int* ff, int stxsize, int stysize, int** ss, int* bb)//cuda
{
        int ii, jj, k, l, temp1, temp2, maxm, **aa;
        int new_xsize, new_ysize;



        aa = alloc_mem(stxsize, stysize);


        for (ii = 0; ii < xsize*ysize; ii++)
                bb[ii] = 0;
        new_xsize = xsize + stxsize - xorg;
        new_ysize = ysize + stysize - yorg;


        for (ii = -xorg; ii <= new_xsize - 1; ii++)
                for (jj = -yorg; jj <= new_ysize - 1; jj++)
                {
                        int tempo = determineIndexInArray(ii + xorg, jj + yorg, xsize);
                        for (k = 0; k <= stxsize - 1; k++)
                                for (l = 0; l <= stysize - 1; l++)
                                {
                                        temp1 = k - xorg;
                                        temp2 = l - yorg;
                                        if ((ii - temp1) >= 0 && (jj - temp2) >= 0 && (ii - temp1)<xsize
&& (jj - temp2)<ysize)
                                                aa[k][l] = ff[determineIndexInArray(ii - temp1, jj -
temp2, xsize)] + ss[k][l];
                                        else   aa[k][l] = -999;
                                }
```

```
                        maxm = max(aa, stxsize, stysize);
                        if (maxm <0)bb[tempo] = 0;
                        else if (maxm > ngr - 1) bb[tempo] = ngr - 1;
                        else bb[tempo] = maxm;
                }
        return(0);
}

int dilation_3x3(int* ff, int* bb)//cuda
{
        int ii, jj, k, l, temp1, temp2, maxm, **aa;
        int new_xsize, new_ysize;
        int bxsize = 3, bysize = 3;

        aa = alloc_mem(bxsize, bysize);

        for (ii = 0; ii < xsize*ysize; ii++)
                bb[ii] = 0;

        for (ii = 0; ii <= xsize - 1; ii++)
                for (jj = 0; jj <= ysize - 1; jj++)
                {
                        int tempo = determineIndexInArray(ii, jj, xsize);
                        for (k = 0; k <= 2; k++)
                                for (l = 0; l <= 2; l++)
                                {
                                        temp1 = k - 1;
                                        temp2 = l - 1;
                                        if ((ii - temp1) >= 0 && (jj - temp2) >= 0 && (ii - temp1)<xsize
&& (jj - temp2)<ysize)
```

```
                                                aa[k][l] = ff[determineIndexInArray(ii - temp1, jj -
temp2, xsize)];
                                        else   aa[k][l] = -999;
                                }
                        maxm = max(aa, 3, 3);
                        if (maxm <0)bb[tempo] = 0;
                        else if (maxm > ngr - 1) bb[tempo] = ngr - 1;
                        else bb[tempo] = maxm;
                }

        return(0);
}

int shift(int* ff, int* bb)//cuda
{
        int ii, jj;

        for (ii = 0; ii < xsize*ysize; ii++)
                bb[ii] = 0;

        int *dev_a, *dev_b;
        int size = xsize*ysize;
        cudaMalloc((void**)&dev_a, xsize * ysize * sizeof(int));
        cudaMalloc((void**)&dev_b, xsize * ysize * sizeof(int));
        cudaMemcpy(dev_a, ff, size* sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, bb, size * sizeof(int), cudaMemcpyHostToDevice);

        cuda_shift << <1, size >> > (dev_a, dev_b,ysize,xsize,xorg,yorg);
        cudaMemcpy(ff, dev_a, size * sizeof(int), cudaMemcpyDeviceToHost);
        cudaMemcpy(bb, dev_b, size * sizeof(int), cudaMemcpyDeviceToHost);
        cudaFree(dev_a);
```

```
	cudaFree(dev_b);

	return(0);
}

int gl_open(int *ff, int stxsize, int stysize, int** ss, int* bc)
{
	int *bb, *cc;

	bb = alloc_mem_1D(xsize, ysize);
	cc = alloc_mem_1D(xsize + stxsize, ysize + stysize);

	erosion(ff, stxsize, stysize, ss, bb);
	dilation(bb, stxsize, stysize, ss, cc);
	shift(cc, bc);

	return(0);
}
int struc_element()
{
	int ii, jj;
	/*
	printf("enter no. of rows of the structuring element :");
	scanf("%d", &sxsize);
	printf("enter no. of columns of the structuring element :");
	scanf("%d", &sysize);
	*/
	s = alloc_mem(sxsize, sysize);

	/*
```

```
        printf("enter origin -xorg,yorg  of the structuring element :");
        scanf("%d %d", &xorg,&yorg);
        */
        xorg = sxsize / 2;                                      //mid pt
        yorg = sysize / 2;
        for (ii = 0; ii <= sxsize - 1; ii++)
                for (jj = 0; jj <= sysize - 1; jj++)
                        s[ii][jj] = 0;
        return(0);
}
int maxmin()
{
        int amax, amin, temp;
        int   i, j;
        //amax = -32768;            //amin = 32767;
        amax = f[0];
        amin = f[0];
        for (i = 0; i <= xsize*ysize; i++)
        {
                if (amax < f[i])amax = f[i];
                if (amin > f[i])amin = f[i];
        }

        ngr = 1;
        temp = amax - 1;
        do
        {
                temp = temp / 2;
                ngr = ngr * 2;
        } while (temp > 0);
```

```
	return (0);
}

int U_limit(int *bb, int* ff)//cuda
{
	int ii;
	for (ii = 0; ii<xsize*ysize; ii++)
		if (ff[ii] < bb[ii]) bb[ii] = ff[ii];
	return(0);
	return 0;
}
int check(int *bb, int* ff, int* count)//cuda
{
	int ii, jj, temp;
	int *dev_a, *dev_b;
	int size = xsize*ysize;
	cudaMalloc((void**)&dev_a, xsize * ysize * sizeof(int));
	cudaMalloc((void**)&dev_b, xsize * ysize * sizeof(int));
	cudaMemcpy(dev_a, bb, size* sizeof(int), cudaMemcpyHostToDevice);
	cudaMemcpy(dev_b, ff, size * sizeof(int), cudaMemcpyHostToDevice);

	cuda_check << <1, size >> > (dev_a, dev_b);
	cudaMemcpy(bb, dev_a, size * sizeof(int), cudaMemcpyDeviceToHost);
	cudaMemcpy(ff, dev_b, size * sizeof(int), cudaMemcpyDeviceToHost);
	cudaFree(dev_a);
	cudaFree(dev_b);
	return(0);
}
```

```
int main()
{
	sxsize = 3;		sysize = 3;
	imgfilein = fopen("input.pgm", "rb");
	imgfileout = fopen("output.pgm", "wb");
	if (imgfilein == NULL)
	{
		printf("Not found input image file. Check directory. \n");
	}

	if (imgfileout == NULL)
	{
		printf("Error in opening output image file. \n");
		exit(0);
	}

	imagestruct img;
	readImage(imgfilein, &img);//all the parameter of img is set now
	xsize = img.numberOfColumns;
	ysize = img.numberOfRows;

	f = alloc_mem_1D(xsize, ysize);
	for (int i = 0; i<xsize*ysize; i++)
		f[i] = img.image[i];

	c = alloc_mem_1D(xsize, ysize);
	d = alloc_mem_1D(xsize, ysize);

	maxmin();
	struc_element();
```

```
        gl_open(f, sxsize, sysize, s, c);
        itern = 1;
        do
        {
                dilation_3x3(c, d);
                U_limit(d, f);
                check(c, d, &count);
        } while (count != 0);

        for (int i = 0; i<xsize*ysize; i++)
        {
                img.image[i] = c[i];
        }
        writeImage(imgfileout, &img);
}
```

Here is the output of the open reconstruction of an image.

IMAGE RESULTS

Input Image

Input Image

Output Image

Output Image