CUDA Mirror Image by using OpenCV

Special Gift!!

```
-   Coding with CUDA+OpenCV (.cu format)
-   Coding with OpenCV (.cpp format)
```

What we need for This Project?

```
-   PulseSecure (VPN for Mahidol)
-   WinSCP/FileZilla (Recommend WinSCP for Windows)
-   PuTTy/SSH via Command Prompt
-   Code Editor
-   Image for Testing
-   Internet for connect to CUDA Lab at Mahidol University Mahidol Salaya
```

How To compile?

```
nvcc mirrornocv.cu -o mirrornocv `pkg-config --cflags --libs opencv` -std=c++11
# or
g++ mirrornocv.cpp -o mirrornocv `pkg-config --cflags --libs opencv` -std=c++11
```

เริ่มดูที่ Code ที่ใช้ OpenCV แบบแรก

```
#include <iostream>
#include "opencv2/opencv.hpp"
#include "opencv2/core.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/cudaarithm.hpp"

using namespace cv;

int main (int argc, char* argv[])
{
    try
    {
        cv::Mat src_host = cv::imread("image.png", cv::IMREAD_ANYCOLOR |
cv::IMREAD_ANYDEPTH);
        cv::flip(src_host,src_host,+1); // Mirror image by using OpenCV flip
method
        cv::cuda::GpuMat dst, src; // CUDA function by using GPU Matrix with dst
and source
```

```
        src.upload(src_host); // upload image to GPU
        cuda::cvtColor(src,dst, cv::COLOR_BGR2BGRA); // Convert to normal color
        cv::Mat result_host(dst); // use OpenCV matrix for copy image to host
        cv::imwrite("img.png",result_host); // write image instead using imshow
    }
    catch(const cv::Exception& ex)
    {
        std::cout << "Error: " << ex.what() << std::endl;
    }
    return 0;
}
```

หลักการอย่างง่ายคือเราใช้ OpenCV ตามปกติโหลดรูปเข้ามาในรูปแบบ Matrix ต่อมาคือเราใช้ฟังก์ชั่น Flip ในการพลิกรูปให้เป็นอีกด้านนึงนั่นเอง จากนั้นเราจะใช้ Cuda เพื่อทำ Matrix ใน GPU และอัพโหลดรูปไปที่การ์ดจอ CUDA จะทำการแปลงสีให้กลับมาเป็นเหมือนเดิม และจะอัพโหลดภาพมาที่เครื่องคอมหลังทำเสร็จแล้ว

เขียนแบบใช้ CUDA+OpenCV

```
#include <iostream>
#include <cuda_runtime.h>
#include "opencv2/opencv.hpp"
#include "opencv2/core.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/cudaarithm.hpp"
#include <vector>
#include <stdio.h>
#include <cuda_runtime.h>

using namespace std;
using namespace cv;

size_t numRows, numCols;

/* Mirror operations */

__global__ void mirror(const uchar4* const inputChannel, uchar4* outputChannel,
int numRows, int numCols, bool vertical)
{
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  if ( col >= numCols || row >= numRows )
  {
   return;
  }

  if(!vertical)
  {
    int thread_x = blockDim.x * blockIdx.x + threadIdx.x;
    int thread_y = blockDim.y * blockIdx.y + threadIdx.y;
```

```
    int thread_x_new = thread_x;
    int thread_y_new = numRows-thread_y;

    int myId = thread_y * numCols + thread_x;
    int myId_new = thread_y_new * numCols + thread_x_new;
    outputChannel[myId_new] = inputChannel[myId];

  }
  else
  {
        int thread_x = blockDim.x * blockIdx.x + threadIdx.x;
    int thread_y = blockDim.y * blockIdx.y + threadIdx.y;

    int thread_x_new = numCols-thread_x;
    int thread_y_new = thread_y;

    int myId = thread_y * numCols + thread_x;
    int myId_new = thread_y_new * numCols + thread_x_new;

        outputChannel[myId_new] = inputChannel[myId];  // linear data store in
global memory
  }
}

uchar4* mirror_ops(uchar4 *d_inputImageRGBA, size_t numRows, size_t numCols, bool
vertical)
{
        // Set reasonable block size (i.e., number of threads per block)
    const dim3 blockSize(4,4,1);
    // Calculate Grid SIze
    int a=numCols/blockSize.x, b=numRows/blockSize.y;
    const dim3 gridSize(a+1,b+1,1);

    const size_t numPixels = numRows * numCols;

    uchar4 *d_outputImageRGBA;
    cudaMalloc(&d_outputImageRGBA, sizeof(uchar4) * numPixels);

    // Call mirror kernel.
    mirror<<<gridSize, blockSize>>>(d_inputImageRGBA, d_outputImageRGBA, numRows,
numCols, vertical);

    cudaDeviceSynchronize();

    // Initialize memory on host for output uchar4*
    uchar4* h_out;
    h_out = (uchar4*)malloc(sizeof(uchar4) * numPixels);

    // copy output from device to host
    cudaMemcpy(h_out, d_outputImageRGBA, sizeof(uchar4) * numPixels,
cudaMemcpyDeviceToHost);

    // cleanup memory on device
    cudaFree(d_inputImageRGBA);
```

```
    cudaFree(d_outputImageRGBA);

    //return h_out
        return h_out;
}

void loadImageRGBA(string &filename, uchar4 **imagePtr,size_t *numRows, size_t
*numCols)
{
    // loading the image
    cv::Mat image = imread(filename.c_str(), cv::IMREAD_ANYCOLOR |
cv::IMREAD_ANYDEPTH);

    // forming a 4-channel(RGBA) image.
    cv::Mat imageRGBA;
    cvtColor(image, imageRGBA, cv::COLOR_BGR2BGRA);

    *imagePtr = new uchar4[image.rows * image.cols];
    unsigned char *cvPtr = imageRGBA.ptr<unsigned char>(0);
    for(size_t i = 0; i < image.rows * image.cols; ++i)
    {
        (*imagePtr)[i].x = cvPtr[4*i + 0];
        (*imagePtr)[i].y = cvPtr[4*i + 1];
        (*imagePtr)[i].z = cvPtr[4*i + 2];
        (*imagePtr)[i].w = cvPtr[4*i + 3];
    }
    *numRows = image.rows;
    *numCols = image.cols;
}

void saveImageRGBA(uchar4* image, string &output_filename,size_t numRows, size_t
numCols)
{
    // Forming the Mat object from uchar4 array.
    int sizes[2] = {numRows, numCols};
    Mat imageRGBA(2, sizes, CV_8UC4, (void *)image);
    // Converting back to BGR system
    Mat imageOutputBGR;
    cvtColor(imageRGBA, imageOutputBGR, cv::COLOR_BGR2BGRA);
    // Writing the image
    imwrite(output_filename.c_str(), imageOutputBGR);
}

uchar4* load_image_in_GPU(string filename)
{ // Load the image into main memory
  uchar4 *h_image, *d_in;
  loadImageRGBA(filename, &h_image, &numRows, &numCols);
  // Allocate memory to the GPU
  cudaMalloc((void **) &d_in, numRows * numCols * sizeof(uchar4));
  cudaMemcpy(d_in, h_image, numRows * numCols * sizeof(uchar4),
cudaMemcpyHostToDevice);
  // No need to keep this image in RAM now.
  free(h_image);
  return d_in;
```

```
}

int main(int argc, char **argv)
{
    string input_file = "image.png";
    string output_file = "output.png";

    uchar4 *d_in = load_image_in_GPU(input_file);
    uchar4 *h_out = NULL;

    h_out = mirror_ops(d_in, numRows, numCols, true);

    cudaFree(d_in);
    if(h_out != NULL)
    {
        saveImageRGBA(h_out, output_file, numRows, numCols);
    }

}
```

หลักการทำงาน

มาดูที่ int main ก่อน

```
string input_file = "image.png" // เราสร้าง path ขึ้นมาดึงรูปเรา
string output_file = "output.png"  // สร้าง output เป็นชื่อไฟล์

uchar4 *d_in = load_image_in_GPU(input_file);
// uchar4 คือชนิดข้อมูลของ cuda รูปแบบ struct เพื่อเอามาเรียกฟังก์ชั่น load_image_in_GPU โดย
เรียก image.png เข้าไปใช้
uchar4 *h_out = NULL;
// สร้าง output โดยที่ข้อมูลเป็น null
h_out = mirror_ops(d_in, numRows, numCols, true);
// ทำ output โดยเรียก mirror_ops(d_in, numRows, numCols, true);
//                              รูป,  ขนาดรูป,   ขนาดรูป,   พลิกรูป
cudaFree(d_in); // หลังจากเรียกทั้งสองฟังก์ชั่นแล้ว จะให้ cuda ปลดหน่วยความจำที่ถูกเรียกใช้ไว้เมื่อกี้
if(h_out != NULL)
{
    saveImageRGBA(h_out, output_file, numRows, numCols); // ทำรูปออกมาเป็นสีเหมือนเดิม
}
```

ต่อมาดูที่ load_image_in_GPU()

```
uchar4* load_image_in_GPU(string filename)
{ // Load the image into main memory
  uchar4 *h_image, *d_in; // ประกาศ pointer สำหรับ Cuda
  loadImageRGBA(filename, &h_image, &numRows, &numCols); // เรียกฟังก์ชั่น
loadImageRGBA
  // Allocate memory to the GPU
```

```
  cudaMalloc((void **) &d_in, numRows * numCols * sizeof(uchar4)); // จองพื้นที่หน่วย
ความจำในการ์ดจอโดยใช้ uchar4
  cudaMemcpy(d_in, h_image, numRows * numCols * sizeof(uchar4),
cudaMemcpyHostToDevice);
  // No need to keep this image in RAM now.
  free(h_image); // ปลดหน่วยความจำที่ RAM ใน CPU
  return d_in; // ส่งค่าไปที่ d_in
}
```

ต่อมาที่ loadImageRGBA()

```
void loadImageRGBA(string &filename, uchar4 **imagePtr,size_t *numRows, size_t
*numCols)
{
    // loading the image
    cv::Mat image = imread(filename.c_str(), cv::IMREAD_ANYCOLOR |
cv::IMREAD_ANYDEPTH);

    // forming a 4-channel(RGBA) image.
    cv::Mat imageRGBA;
    /**
     * R คือสีแดง
     * G คือสีเขียว
     * B คือสีน้ำเงิน
     * A คือคลื่นความถี่แอลฟ่า (alpha)
     * **/
    cvtColor(image, imageRGBA, cv::COLOR_BGR2BGRA); // ใช้ OpenCV แปลงสีจาก BGR เป็น
RGBA

    *imagePtr = new uchar4[image.rows * image.cols]; // ใช้ pointer เก็บขนาดรูปเป็น
พิกเซลทั้งหมด
    unsigned char *cvPtr = imageRGBA.ptr<unsigned char>(0);
    for(size_t i = 0; i < image.rows * image.cols; ++i)
    {
        (*imagePtr)[i].x = cvPtr[4*i + 0]; // พ้อยท์เตอร์สีแดง
        (*imagePtr)[i].y = cvPtr[4*i + 1]; // พ้อยท์เตอร์สีเขียว
        (*imagePtr)[i].z = cvPtr[4*i + 2]; // พ้อยท์เตอร์สีน้ำเงิน
        (*imagePtr)[i].w = cvPtr[4*i + 3]; // พ้อยท์เตอร์คลื่นความถี่แอลฟ่า
    }
    *numRows = image.rows; // ประกาศเพื่อเก็บขนาดแถว
    *numCols = image.cols; // ประกาศเพื่อเก็บขนาดคอลัมน์
}
```

```
uchar4* mirror_ops(uchar4 *d_inputImageRGBA, size_t numRows, size_t numCols, bool
vertical)
{
        // Set reasonable block size (i.e., number of threads per block)
    const dim3 blockSize(4,4,1);
    // Calculate Grid SIze
```

```
    int a=numCols/blockSize.x, b=numRows/blockSize.y; // ขนาดรูปที่จะเอาเข้ามาคำนวณ
    const dim3 gridSize(a+1,b+1,1);

    const size_t numPixels = numRows * numCols; // จัดการขนาดพิกเซล

    uchar4 *d_outputImageRGBA;
    cudaMalloc(&d_outputImageRGBA, sizeof(uchar4) * numPixels);

    // Call mirror kernel.
    mirror<<<gridSize, blockSize>>>(d_inputImageRGBA, d_outputImageRGBA, numRows,
numCols, vertical);

    cudaDeviceSynchronize(); // ใช้เพื่อกัน Race Condition

    // Initialize memory on host for output uchar4*
    uchar4* h_out;
    h_out = (uchar4*)malloc(sizeof(uchar4) * numPixels);

    // copy output from device to host
    cudaMemcpy(h_out, d_outputImageRGBA, sizeof(uchar4) * numPixels,
cudaMemcpyDeviceToHost);

    // cleanup memory on device
    cudaFree(d_inputImageRGBA);
    cudaFree(d_outputImageRGBA);

    //return h_out
        return h_out;
}
```

ต่อมาที่ kernel

```
__global__ void mirror(const uchar4* const inputChannel, uchar4* outputChannel,
int numRows, int numCols, bool vertical)
{
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

  if ( col >= numCols || row >= numRows )
  {
   return;
  }

  if(!vertical)
  {
    int thread_x = blockDim.x * blockIdx.x + threadIdx.x;
    int thread_y = blockDim.y * blockIdx.y + threadIdx.y;

    int thread_x_new = thread_x;
    int thread_y_new = numRows-thread_y;

    int myId = thread_y * numCols + thread_x;
```

```
    int myId_new = thread_y_new * numCols + thread_x_new;
    outputChannel[myId_new] = inputChannel[myId];

  }
  else
  {
        int thread_x = blockDim.x * blockIdx.x + threadIdx.x;
    int thread_y = blockDim.y * blockIdx.y + threadIdx.y;

    int thread_x_new = numCols-thread_x;
    int thread_y_new = thread_y;

    int myId = thread_y * numCols + thread_x;
    int myId_new = thread_y_new * numCols + thread_x_new;

        outputChannel[myId_new] = inputChannel[myId];  // linear data store in
global memory
  }
}
```

ข้างในนี้จะทำการเช็ค true/false **ว่าจะพลิกรูปแบบแนวตั้งหรือแนวนอน โดยใช้** threadIdx **แกน** X **และแกน** Y **ส่งข้อมูลออกไปที่** outputChannel[myId_new];

สุดท้ายการเซฟรูป

```
void saveImageRGBA(uchar4* image, string &output_filename,size_t numRows, size_t
numCols)
{
    // Forming the Mat object from uchar4 array.
    int sizes[2] = {numRows, numCols};
    Mat imageRGBA(2, sizes, CV_8UC4, (void *)image); // เราจะใช้ Matrix ในการทำรูป
กลับมาเป็นสีเดิม
    // Converting back to BGR system
    Mat imageOutputBGR;
    cvtColor(imageRGBA, imageOutputBGR, cv::COLOR_BGR2BGRA); // แปลงรูปจาก BGR เป็น
BGRA
    // Writing the image
    imwrite(output_filename.c_str(), imageOutputBGR); // เขียนรูปออกมา
}
```

Output

ก่อนพลิกรูป

หลังพลิกรูป

```
u6088130@cuda-machine:~/CPP_cuda$ nvprof ./mirrornocv
==21625== NVPROF is profiling process 21625, command: ./mirrornocv
==21625== Profiling application: ./mirrornocv
==21625== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   52.13%  6.3250ms         1  6.3250ms  6.3250ms  6.3250ms  [CUDA
memcpy HtoD]
                  47.27%  5.7346ms         1  5.7346ms  5.7346ms  5.7346ms  [CUDA
memcpy DtoH]
                   0.60%  72.386us         1  72.386us  72.386us  72.386us
mirror(uchar4 const *, uchar4*, int, int, bool)
      API calls:   96.12%  264.67ms         2  132.33ms  3.7291ms  260.94ms
cudaMalloc
                   2.21%  6.0797ms         2  3.0399ms  227.89us  5.8518ms
cudaMemcpy
                   0.93%  2.5728ms         1  2.5728ms  2.5728ms  2.5728ms
cudaDeviceSynchronize
                   0.54%  1.4737ms        97  15.192us     109ns  1.1243ms
cuDeviceGetAttribute
                   0.12%  323.64us         3  107.88us  1.0880us  165.42us
cudaFree
                   0.05%  133.04us         1  133.04us  133.04us  133.04us
```

```
cuDeviceTotalMem
                    0.02%  64.299us         1  64.299us  64.299us  64.299us
cuDeviceGetName
                    0.01%  24.834us         1  24.834us  24.834us  24.834us
cudaLaunchKernel
                    0.00%  1.7500us         3     583ns     143ns  1.1750us
cuDeviceGetCount
                    0.00%     619ns         2     309ns     161ns     458ns
cuDeviceGet
                    0.00%     237ns         1     237ns     237ns     237ns
```