Sobel Edge Detection and Image Sharpening in CUDA using Libtorch

Reya Sadhu

Agenda:

- Environment: Libtorch and CUDA
- Convolution in Images
- Sobel Edge Detection
    - Global Memory
    - Naïve Shared Memory
    - Optimized Shared Memory
    - Reorganized Shared memory
- Sharpening Filter
    - Global Memory
    - Shared Memory
- Results

Environment

- I am using libtorch and OpenCV in cuda.
- OpenCV reads the images to process and libtorch transforms it into a tensor.
- The processing happens on tensor.
- Output is converted to images using OpenCV.
- This can be used to create any custom function to run with Pytorch using Pybind!

Convolution

- Convolution is a general-purpose filter effect for images.
- It's a matrix applied to an image and a mathematical operation comprised of integers.
- It works by determining the value of a central pixel by adding the weighted values of all its neighbors.
- The output is a new modified filtered image

99	99	99	99	99
103	101	99	101	98
96	99	100	97	101
99	101	96	99	103
99	97	98	99	106

Pixel Values

1	0	4	2	125	67
8	2	5	4	34	12
20	13	25	15	240	2
76	8	6	6	100	76
34	66	134	223	201	3
255	123	89	55	32	2

Kernel 3 x 3 Pixels

1	2	1
2	4	2
1	2	1

2 x 1 = 2
2 x 5 = 10
1 x 4 = 4
2 x 13 = 26
25 x 4 = 100
15 x 2 = 30
8 x 1 = 8
6 x 2 = 12
6 x 1 = 6

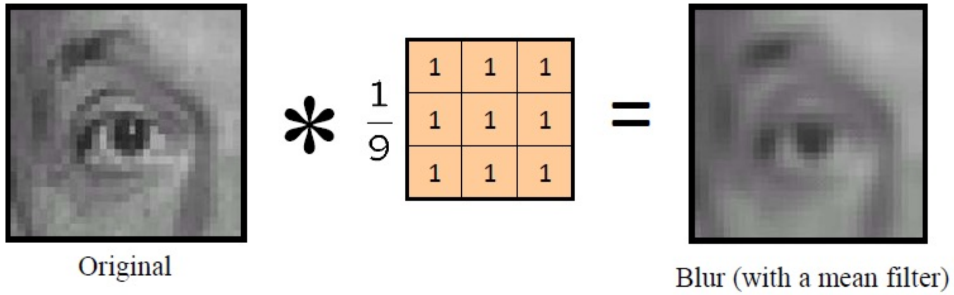
2 + 10 + 4 + 26 + 100 + 30 + 8 + 12 + 6 = 198

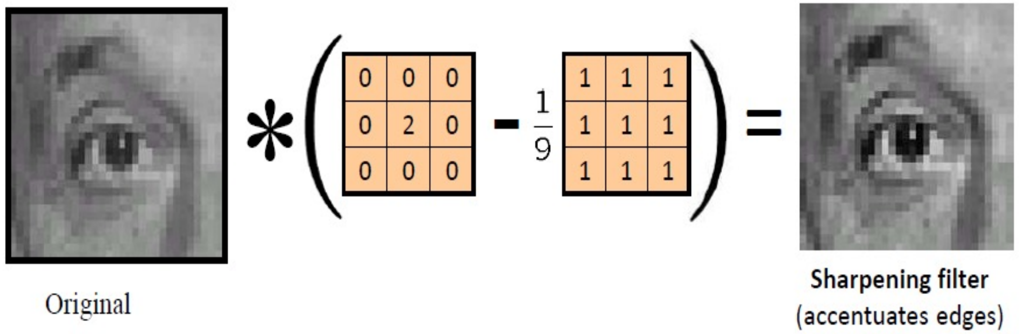
Convoluted Image

		198			

Convolution

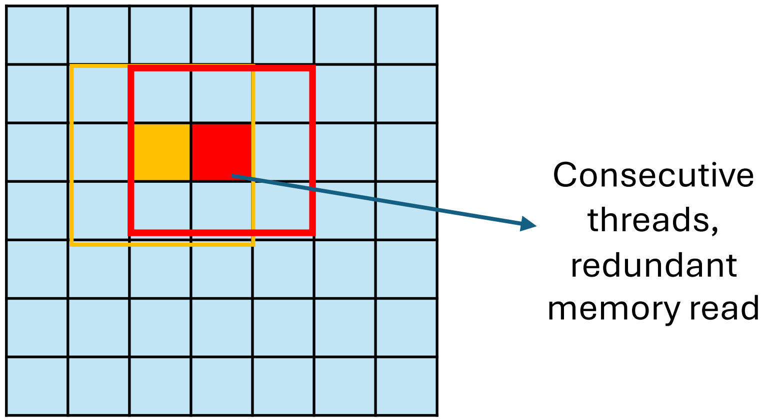
Changing the values and sizes of kernel, we can achieve different tasks like blurring, embossing, sharpening, edge detection





Global Memory

- Use global memory to send data to device and each thread accesses this to compute convolution kernel.
- Our convolution kernel size is radius 1
- Each thread processes 9 read from memory.
- Neighboring thread read same elements. Not efficient.



- Global memory is on-board, high latency memory.

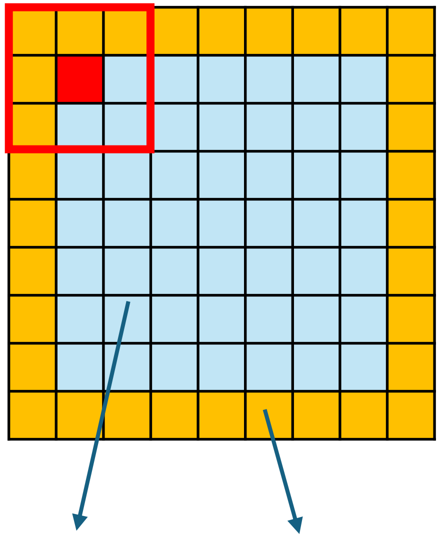
```
__global__ void global_sobelEdgeDetection(float* srcImage, float* dstImage)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float Ky[3][3] = { -1, 0, 1, -2, 0, 2, -1, 0, 1 };
    float Kx[3][3] = { 1, 2, 1, 0, 0, 0, -1, -2, -1 };


    // only threads inside image will write results
    if ((x >= 1) && (x < (W_input - 1)) && (y >= 1) && (y < (H_input - 1)))
    {

        float Gx = 0;
        float Gy = 0;

        for (int ky = -1; ky <= 1; ky++) {
            for (int kx = -1; kx <= 1; kx++) {

                float fl = srcImage[(y + ky) * W_input + (x + kx)];
                Gx += fl * Kx[ky + 1][kx + 1];
                Gy += fl * Ky[ky + 1][kx + 1];
            }
        }



        dstImage[y * W_input + x] = sqrt(Gx * Gx + Gy * Gy);
    }
}
```

Naïve Shared Memory

Shared memory is on-chip, low latency and band width

- **A thread block first stores all global data in the shared memory.**
- **Each thread reads one global memory while storing.**
- **Coalesced Memory Read**
- **No redundant Memory Read: Optimization**

While computing, the pixels at the edge of the shared memory array will depend on pixels not in shared memory.



Shared Memory same as Block Size, no space for pixels on apron

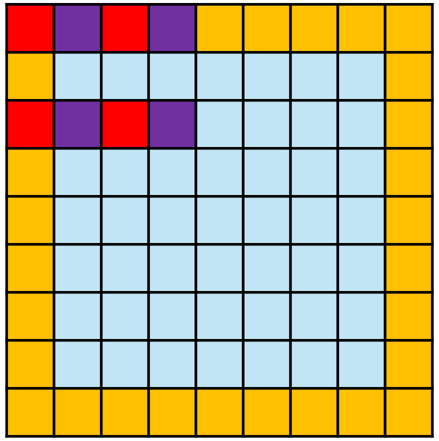
Wait till all threads finish storing

Warp Divergence

data is read from global memory for edge pixels.

```
__global__ void naive_shared_sobelEdgeDetection(float* d_Data, float* d_Result)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;


    float Ky[3][3] = { -1, 0, 1, -2, 0, 2, -1, 0, 1 };
    float Kx[3][3] = { 1, 2, 1, 0, 0, 0, -1, -2, -1 };

    __shared__ float shm[BLOCK_SIZE][BLOCK_SIZE];
    shm[threadIdx.y][threadIdx.x] = d_Data[row * W_input + col];

    __syncthreads();
    float Gx = 0.0, Gy = 0.0;
    for (int i = 0; i < MASK_WIDTH; i++) {
        for (int j = 0; j < MASK_WIDTH; j++) {
            int current_x_global = col - RADIUS + j;
            int current_y_global = row - RADIUS + i;
            int current_x = threadIdx.x - RADIUS + j;
            int current_y = threadIdx.y - RADIUS + i;
            if (current_x_global>=0 && current_x_global<W_input && current_y_global >= 0 && current_y_global < H_input) {
                if (current_x >= 0 && current_x < BLOCK_SIZE && current_y >= 0 && current_y < BLOCK_SIZE) {
                    Gx += shm[current_y][current_x] * Kx[i][j];
                    Gy += shm[current_y][current_x] * Ky[i][j];
                }
                else {
                    Gx += d_Data[current_y_global * W_input + current_x_global] * Kx[i][j];
                    Gy += d_Data[current_y_global * W_input + current_x_global] * Ky[i][j];
                }
            }
        }
    }

    d_Result[row * W_input + col] = sqrt(Gx*Gx+Gy*Gy);
```

Shared Memory: Multiple Pixels per thread



- If one thread is used for each pixel loaded into shared memory, then the threads loading the apron pixels will be idle during the filter computation.
- As the radius of the filter increases, the percentage of idle threads increases. This wastes much of the available parallelism.

Shared Memory size increased to accommodate the apron

Each thread read and store four pixels.

Coalesced Memory Read

No Warp Divergence

```
// Original image-based coordinates
const int x0 = threadIdx.x + blockIdx.x * blockDim.x;
const int y0 = threadIdx.y + blockIdx.y * blockDim.y;
// global mem address of this thread
const int gLoc = x0 + W_input * y0;
__shared__ float data[BLOCK_SIZE + 2 * RADIUS][BLOCK_SIZE + 2 * RADIUS];
// Case 1: upper left
x = x0 - RADIUS;
y = y0 - RADIUS;
if (x < 0 || y < 0)
    data[threadIdx.y][threadIdx.x] = 0;
else
    data[threadIdx.y][threadIdx.x] = d_Data[x + W_input * y];

// Case 2: upper right
x = x0 + RADIUS;
y = y0 - RADIUS;
if (x > W_input - 1 || y < 0)
    data[threadIdx.y][threadIdx.x+2*RADIUS] = 0;
else
    data[threadIdx.y][threadIdx.x+2*RADIUS] = d_Data[x+ W_input *y];

// Case 3: lower left
x = x0 - RADIUS;
y = y0 + RADIUS;
if (x < 0 || y > H_input - 1)
    data[threadIdx.y + 2 * RADIUS][threadIdx.x] = 0;
else
    data[threadIdx.y + 2 * RADIUS][threadIdx.x] = d_Data[x + W_input * y];

// Case 4: lower right
x = x0 + RADIUS;
y = y0 + RADIUS;
if (x > W_input - 1 || y > H_input - 1)
    data[threadIdx.y +2 * RADIUS][threadIdx.x + 2 * RADIUS] = 0;
else
    data[threadIdx.y + 2 * RADIUS][threadIdx.x +2 * RADIUS] = d_Data[x + W_input * y];

__syncthreads();

float Gx = 0,Gy=0;
for (int i = 0; i <MASK_WIDTH; i++) {
    for (int j = 0; j <MASK_WIDTH; j++) {
        Gx += data[threadIdx.y + i][threadIdx.x + j] *Kx[i][j] ;
        Gy += data[threadIdx.y + i][threadIdx.x + j] *Ky[i][j] ;
    }
}
d_Result[gLoc] = sqrt(Gx*Gx+Gy*Gy);
```

Reorganized shared memory – 1D

- I used 2D array of shared memory to make indexing a bit simpler. Inside computation loop, there is possibility of bank conflict for warp since each thread access first column major memory at the same time.
- Rearrange this shared memory to 1D array so that all threads access data consequently and optimize memory bus.

```
__global__ void reorganized_shared_sobelEdgeDetection(float* d_Data, float* d_Result) {

    const int shm_width = BLOCK_SIZE + 2 * RADIUS;
    __shared__ float data[shm_width*shm_width];
```

```
// Case 1: upper left
x = x0 - RADIUS;
y = y0 - RADIUS;
if (x < 0 || y < 0)
    data[threadIdx.y*shm_width+threadIdx.x] = 0;
else
    data[threadIdx.y * shm_width + threadIdx.x] = d_Data[x + W_input * y];

// Case 2: upper right
x = x0 + RADIUS;
y = y0 - RADIUS;
if (x > W_input - 1 || y < 0)
    data[threadIdx.y * shm_width + threadIdx.x + 2 * RADIUS] = 0;
else
    data[threadIdx.y * shm_width + threadIdx.x + 2 * RADIUS] = d_Data[x + W_input * y];

// Case 3: lower left
x = x0 - RADIUS;
y = y0 + RADIUS;
if (x < 0 || y > H_input - 1)
    data[(threadIdx.y + 2 * RADIUS)* shm_width + threadIdx.x] = 0;
else
    data[(threadIdx.y + 2 * RADIUS)* shm_width + threadIdx.x] = d_Data[x + W_input * y];

// Case 4: lower right
x = x0 + RADIUS;
y = y0 + RADIUS;
if (x > W_input - 1 || y > H_input - 1)
    data[(threadIdx.y + 2 * RADIUS)* shm_width + threadIdx.x + 2 * RADIUS] = 0;
else
    data[(threadIdx.y + 2 * RADIUS) * shm_width + threadIdx.x + 2 * RADIUS] = d_Data[x + W_input * y];

__syncthreads();

// Convolution
float Gx = 0, Gy = 0;

for (int i = 0; i < MASK_WIDTH; i++) {
    for (int j = 0; j < MASK_WIDTH; j++) {
        Gx += data[(threadIdx.y + i)* shm_width + threadIdx.x + j] * Kx[i*MASK_WIDTH+j];
        Gy += data[(threadIdx.y + i)* shm_width + threadIdx.x + j] * Ky[i*MASK_WIDTH+j];
    }
}
d_Result[gLoc] = sqrt(Gx * Gx + Gy * Gy);
```

Results: Sobel Edge Detection

*

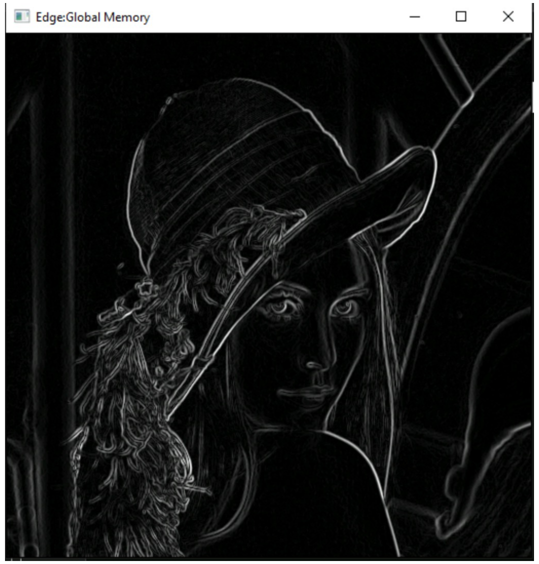
Vertical

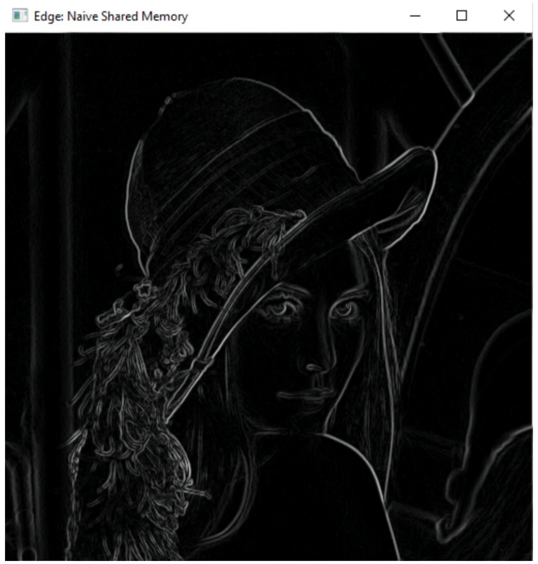
-1	0	1
-2	0	2
-1	0	1

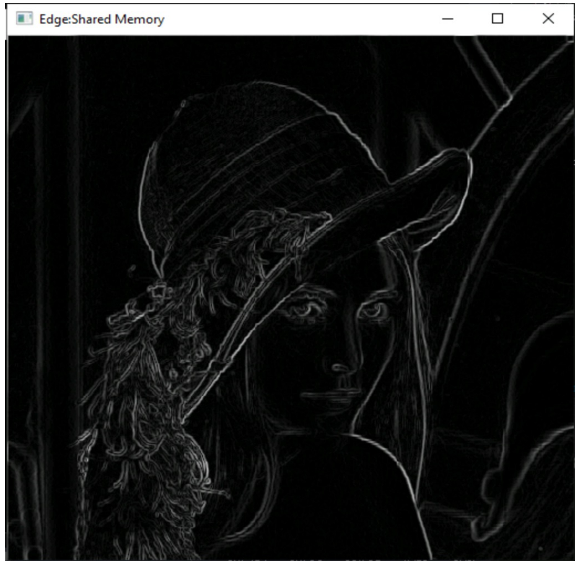
Horizontal

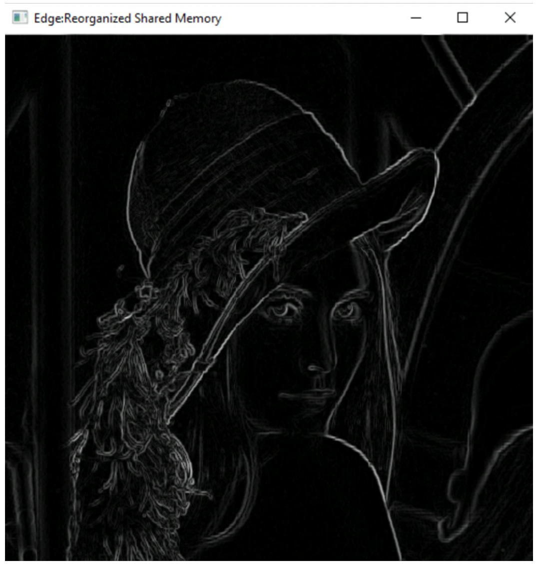
-1	0	1
-2	0	2
-1	0	1

=







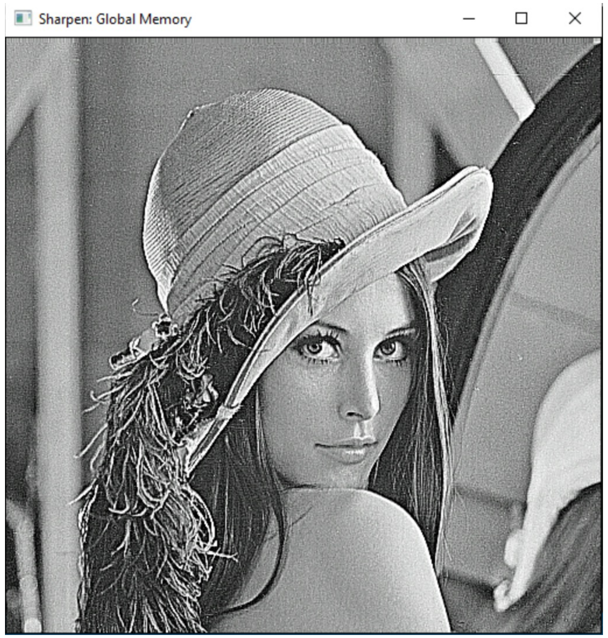


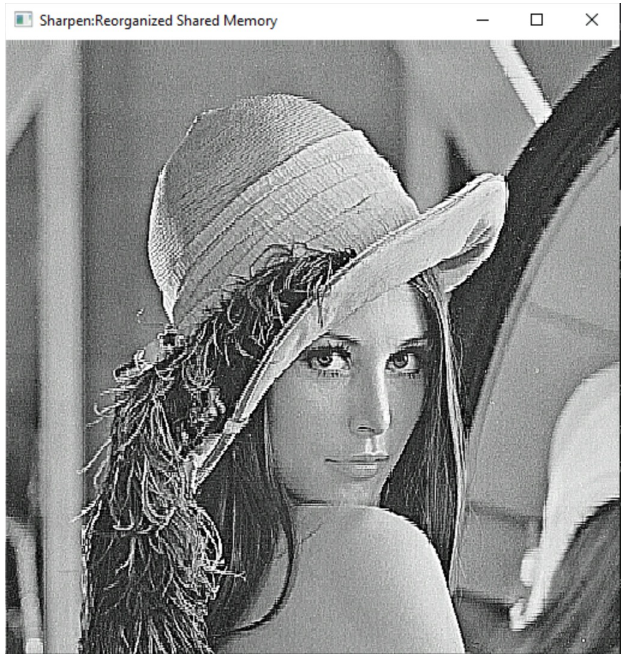
Results: Sharpening Filter

*

-1	-1	-1
-1	9	-1
-1	-1	-1

=





Processing Time

BLOCK_SIZE=32

```
Check image dimensions
pixel dim x: 512
pixel dim x: 512

Processing time Edge:Global Memory (ms): 0.099328

Processing time Edge: Naive Shared Memory (ms): 0.095232

Processing time Edge:Shared Memory (ms): 0.065536

Processing time Edge:Reorganized Shared Memory (ms): 0.054272

Processing time Sharpen: Global Memory (ms): 0.086016

Processing time Sharpen:Reorganized Shared Memory (ms): 0.048128
```

Global Memory > Naïve Shared Memory >Shared Memory(Multiple pixel) > Reorganized shared memory

4% ⬆ 31%⬆ 17%⬆

45% ⬆

Sobel Filter

44% ⬆

Sharpening Filter

Processing Time

BLOCK_SIZE=16

```
Check image dimensions
pixel dim x: 512
pixel dim x: 512

Processing time Edge:Global Memory (ms): 0.073728

Processing time Edge: Naive Shared Memory (ms): 0.060416

Processing time Edge:Shared Memory (ms): 0.053248

Processing time Edge:Reorganized Shared Memory (ms): 0.047296

Processing time Sharpen: Global Memory (ms): 0.071680

Processing time Sharpen:Reorganized Shared Memory (ms): 0.052224

D:\projects\Project\build\src\test\Release\test.exe (process 33412) exited with code 0.
Press any key to close this window . . .
```

Smaller block size giving better results!

Larger block sizes typically consume more shared memory per block. As the kernel relies heavily on shared memory and the available shared memory per block is limited, using a smaller block size might allow more blocks to run concurrently on the GPU.

Global Memory > Naïve Shared Memory >Shared Memory(Multiple pixel) > Reorganized shared memory

18% ↑ 12% ↑ 11% ↑

36% ↑

Sobel Filter

27% ↑

Sharpening Filter

Reproducing Results:

- Add Extern/opencv/build to environmental variable **OpenCV_DIR**
- After building solution in Release Mode, Add opencv_world480.dll from \Extern\opencv\build\x64\vc16\bin to \build\src\test\Release.
- Uncomment wait_key(0) to visualize the results.