GPU Sobel Edge Detection:
Performance Analysis and Optimization

Thomas Giesbrecht
Avirup Bhattacharjee
Matthew Moran
CSE 4373/5373: General Purpose GPU Programming
Instructor: Dr. Alex J Dillhoff
April 23, 2024

Project Overview

Implementation of the Sobel edge detection algorithm

Four versions implemented:

- CPU Implementation
- GPU Naive Implementation
- GPU Optimized Implementation
- GPU Coarsened Implementation

Technologies used: **CUDA Toolkit, OpenCV**

CPU IMPLEMENTATION

- Standard convolution approach using Sobel kernels (Gx, Gy)
- Nested loops for pixel-by-pixel computation
- Gradient magnitude calculated and clamped
- Output image saved as cpu_output_image.jpg

```cpp
void applySobel(const cv::Mat& input, cv::Mat& output) {
    // Sobel kernels
    int Gx[3][3] = {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };

    int Gy[3][3] = {
        {-1, -2, -1},
        { 0,  0,  0},
        { 1,  2,  1}
    };

    // Initialize the output image
    output = cv::Mat::zeros(input.size(), CV_8UC1);

    // Apply the Sobel filter
    for (int y = 1; y < input.rows - 1; ++y) {
        for (int x = 1; x < input.cols - 1; ++x) {
            int sumX = 0;
            int sumY = 0;

            // Convolution with Sobel kernels
            for (int ky = -1; ky <= 1; ++ky) {
                for (int kx = -1; kx <= 1; ++kx) {
                    int pixel = input.at<uchar>(y + ky, x + kx);
                    sumX += pixel * Gx[ky + 1][kx + 1];
                    sumY += pixel * Gy[ky + 1][kx + 1];
                }
            }

            // Compute gradient magnitude
            int magnitude = std::sqrt(sumX * sumX + sumY * sumY);
            magnitude = std::min(255, magnitude); // Clamp to 255

            output.at<uchar>(y, x) = static_cast<uchar>(magnitude);
        }
    }
}
```

Naive GPU Implementation

- Basic CUDA kernel using direct global memory access
- Each thread computes gradient magnitude independently
- Coalesced global memory access
- Output image saved as gpu_output_image_naive.jpg

```
__global__ void sobelKernel(const unsigned char* input, unsigned char* output, int wi
    // Calculate the thread's position in the image
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Sobel kernels
    int Gx[3][3] = {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };

    int Gy[3][3] = {
        {-1, -2, -1},
        { 0,  0,  0},
        { 1,  2,  1}
    };

    // Ensure the thread is within bounds and not on the border
    if (x > 0 && x < width - 1 && y > 0 && y < height - 1) {
        int sumX = 0;
        int sumY = 0;

        // Apply the Sobel filter
        for (int ky = -1; ky <= 1; ++ky) {
            for (int kx = -1; kx <= 1; ++kx) {
                int pixel = input[(y + ky) * width + (x + kx)];
                sumX += pixel * Gx[ky + 1][kx + 1];
                sumY += pixel * Gy[ky + 1][kx + 1];
            }
        }

        // Compute gradient magnitude
        int magnitude = sqrtf(sumX * sumX + sumY * sumY);
        magnitude = min(255, magnitude); // Clamp to 255

        // Write the result to the output image
        output[y * width + x] = static_cast<unsigned char>(magnitude);
    }
}
```

Optimized GPU Implementation

- Utilizes shared memory to minimize global memory accesses
- 18x18 shared memory tile for 16x16 pixel blocks, loading additional border pixels
- Thread synchronization (__syncthreads()) for data consistency
- Output image saved as gpu_output_image_optimized.jpg

```
__global__ void sobelKernelOptimized(const unsigned char* input, unsigned char* outp
    // Shared memory for the tile
    __shared__ unsigned char sharedMem[18][18]; // Block size (16x16) + 2 for border

    // Calculate global thread coordinates
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate shared memory coordinates
    int sharedX = threadIdx.x + 1;
    int sharedY = threadIdx.y + 1;

    // Load data into shared memory
    if (x < width && y < height) {
        sharedMem[sharedY][sharedX] = input[y * width + x];

        // Load border pixels into shared memory
        if (threadIdx.x == 0 && x > 0) {
            sharedMem[sharedY][0] = input[y * width + (x - 1)];
        }
        if (threadIdx.x == blockDim.x - 1 && x < width - 1) {
            sharedMem[sharedY][sharedX + 1] = input[y * width + (x + 1)];
        }
        if (threadIdx.y == 0 && y > 0) {
            sharedMem[0][sharedX] = input[(y - 1) * width + x];
        }
        if (threadIdx.y == blockDim.y - 1 && y < height - 1) {
            sharedMem[sharedY + 1][sharedX] = input[(y + 1) * width + x];
        }

        // Load corner pixels
        if (threadIdx.x == 0 && threadIdx.y == 0 && x > 0 && y > 0) {
            sharedMem[0][0] = input[(y - 1) * width + (x - 1)];
        }
        if (threadIdx.x == blockDim.x - 1 && threadIdx.y == 0 && x < width - 1 && y
            sharedMem[0][sharedX + 1] = input[(y - 1) * width + (x + 1)];
        }
        if (threadIdx.x == 0 && threadIdx.y == blockDim.y - 1 && x > 0 && y < height
            sharedMem[sharedY + 1][0] = input[(y + 1) * width + (x - 1)];
        }
        if (threadIdx.x == blockDim.x - 1 && threadIdx.y == blockDim.y - 1 && x < wi
            sharedMem[sharedY + 1][sharedX + 1] = input[(y + 1) * width + ))];
        }
```

Performance Findings

- Optimized GPU implementation unexpectedly slower than naive GPU
- Main reasons:
  - Compute-bound workload
  - Shared memory overhead
  - Synchronization overhead (__syncthreads() latency)
  - Modern GPU efficient global memory handling

Recommendations

- **Reduce shared memory overhead:**
  - Minimize unnecessary data in shared memory
  - Adjust shared memory tile sizes
- **Decrease synchronization cost:**
  - Implement warp-level synchronization (__shfl_sync())
- **Hybrid approach:**
  - Naive kernel for compute-bound tasks
  - Optimized kernel for memory-bound tasks

Visual Outputs Comparison

Fig: input_image.jpg

Fig: cpu_output_image.jpg

Visual Outputs Comparison(contd.)

Fig: gpu_output_image_naive.jpg

Fig: gpu_output_image_optimized.jpg

Thread Coarsening

After updating for Thread coarsening (4x coarsening factor) the Optimized almost caught up to the naive (rather than twice as slow, is about 5% slower). However, the output is slightly different

Final Summary and Takeaways

- **Key Insight:** Optimization must align with workload nature — compute-bound cases favor simpler designs
- **Surprising Result:** Naive GPU kernel outperformed optimized version due to lower overhead
- **Improvement Noted:** Thread coarsening helped optimized kernel almost match naive performance
- **Conclusion:** Always profile and analyze before optimizing — simple designs can outperform complex ones on modern GPUs
- **Future Work:** Test across GPU architectures, refine warp-level coordination, and explore dynamic kernel strategies

Thank You!