Gaussian Blur Convolution - Labwork 5

Le Minh Hoang - 2440051

Keywords:

Abstract. No abstract for the labwork 5 also, I guess.

1 Introduction

This report states the work completed in Labwork 5, which will attempt to implement a 7x7 Gaussian Blur filter using CUDA with both shared memory and non-shared memory approaches.

2 Implementation

2.1 Gaussian Kernel Construction

The Gaussian kernel is generated using the given in slide formula (I am too lazy to copy it here) where σ is the standard deviation, and μ_x, μ_y are the kernel center coordinates.

```
def create_gaussian_kernel(kernel_size=7, sigma=1.0):
    kernel = np.zeros((kernel_size, kernel_size), dtype=np.float32
       )
    center = kernel_size // 2
    for i in range(kernel_size):
        for j in range(kernel_size):
            x, y = i - center, j - center
            kernel[i, j] = np.exp(-(x**2 + y**2) / (2 * sigma**2))
    kernel /= kernel.sum()  # Normalize
    return kernel
```

2.2 GPU Implementation Without Shared Memory

The basic GPU implementation performs convolution by accessing the kernel directly from global memory:

```
@cuda.jit
def gaussian_blur_no_shared(src, dst, kernel, kernel_size):
    tx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    ty = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y

    if tx < dst.shape[1] and ty < dst.shape[0]:
        kernel_radius = kernel_size // 2
        for c in range(3):  # RGB channels
            pixel_value = 0.0
            for ky in range(kernel_size):
                for kx in range(kernel_size):
                    src_x = tx + kx - kernel_radius
                    src_y = ty + ky - kernel_radius
                    # Boundary check
```

Article version: October 10, 2025 17:25 CET

```
                    if 0 <= src_x < src.shape[1] and 0 <= src_y <
                       src.shape[0]:
                        pixel_value += src[src_y, src_x, c] *
                           kernel[ky, kx]
            dst[ty, tx, c] = pixel_value
```

- Map thread to unique pixel position (tx, ty)
- Validate the boundary
- Calculate offset for kernel centering (radius = 3 for 7x7)
- Loop through RGB channels
- Reset pixel_value to 0.0 for each channel
- Loop through all 49 kernel elements (7x7)
- Map kernel position to source image location
- Check for boundary
- Multiply pixel by kernel weight and accumulate
- Store the result into an array

2.3 GPU Implementation With Shared Memory

This implementation copies the kernel into shared memory for faster access

```
@cuda.jit
def gaussian_blur_shared(src, dst, kernel, kernel_size):
    tx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    ty = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y

    # Declare shared memory for kernel
    shared_kernel = cuda.shared.array(shape=(7, 7), dtype=numba.
       float32)

    # Load kernel into shared memory
    tid = cuda.threadIdx.y * cuda.blockDim.x + cuda.threadIdx.x
    if tid < kernel_size * kernel_size:
        ky = tid // kernel_size
        kx = tid % kernel_size
        shared_kernel[ky, kx] = kernel[ky, kx]

    # Synchronize threads
    cuda.syncthreads()

    # Perform convolution using shared kernel
    if tx < dst.shape[1] and ty < dst.shape[0]:
        kernel_radius = kernel_size // 2
        for c in range(3):
            pixel_value = 0.0
```

```
            for ky in range(kernel_size):
                for kx in range(kernel_size):
                    src_x = tx + kx - kernel_radius
                    src_y = ty + ky - kernel_radius
                    if 0 <= src_x < src.shape[1] and 0 <= src_y <
                       src.shape[0]:
                        pixel_value += src[src_y, src_x, c] *
                           shared_kernel[ky, kx]
            dst[ty, tx, c] = pixel_value
```

3 Results and Analysis

3.1 Difficulties with Shared Memory Implementation

All the run with shared memory was always slower, I tried to figure out but failed to get the solution. When I was about to accept the faith (https://stackoverflow.com/questions/10250325/cuda-shared-memory-not-faster-than-global) I thought that clearing the memory may do something. And it worked.

```
cuda.current_context().deallocations.clear()
```

3.2 Performance Comparison

After resolving the context issues, I achieved following result:

- Without shared memory: 0.0143 seconds
- With shared memory (after context clear): 0.0056 seconds
- Speedup: 2.57x

3.3 Visual Results

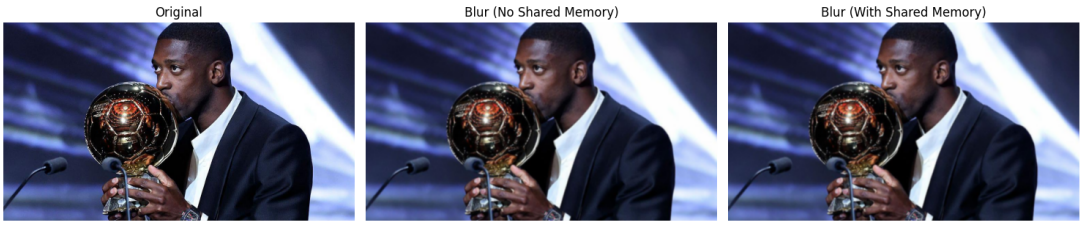

Figure 1: **Original Image and Gaussian Blur Result (7x7 kernel,** $\sigma = 1.0$**).**

3.4 Conclusion

From what have been implemented and the result that I have achieved, I can conclude that:

- Shared memory provides significant speedup (2.57x) for convolution operations
- CUDA context deallocation management is crucial for accurate performance measurements
- The kernel loading synchronization is critical - all threads must wait before computation
- The overhead of shared memory is worth it for repeated kernel access patterns like convolution (but maybe it will only have significant improvement for large kernel size).