Ehi Simon - 14352307 ECEC 413-001
Alisha Augustin - 14389451 05/30/2023

CUDA: Blur Filter

In this project, we implement a parallel box blur filter suitable for execution on a Graphics Processing Unit (GPU) using CUDA. CUDA is a parallel computing platform and programming model developed for GPU acceleration. The implementation consists of several key components, including memory allocation, data transfer between the host (CPU) and the device (GPU), kernel function invocation, and result validation.

First, memory is allocated for the input and output images (*in, out_gold, and out_gpu*) using the malloc function. The input image is populated with random values between -0.5 and 0.5. The blur filter is then computed on the CPU using the *compute_gold* function. Next, the code proceeds to compute the blur filter on the GPU using the *compute_on_device* function. This function follows a specific pattern outlined for GPU computation. First, device memory is allocated for the input and output images using *cudaMalloc*. The input image is then copied from the host to the device using *cudaMemcpy*. The block and grid dimensions for the GPU kernel are defined based on the input image size. The GPU kernel function *blur_filter_kernel* is launched with the specified block and grid dimensions to perform the actual blur filter calculation in parallel. The output image is then copied back from the device to the host. Finally, the device memory is freed using *cudaFree*. And the *check_results* function is used to validate the results.

```
void compute_on_device(const image_t in, image_t out)
{
    /* Allocate device memory for the input and output images */
    float *d_in, *d_out;
    size_t image_size = sizeof(float) * in.size * in.size;
    cudaMalloc((void **)&d_in, image_size);
    cudaMalloc((void **)&d_out, image_size);

    /* Copy the input image from host to device */
    cudaMemcpy(d_in, in.element, image_size, cudaMemcpyHostToDevice);

    dim3 blockDim(32, 32);
    dim3 gridDim((in.size + blockDim.x - 1) / blockDim.x, (in.size + blockDim.y - 1) / blockDim.y);
    blur_filter_kernel<<<gridDim, blockDim>>>(d_in, d_out, in.size);


    cudaMemcpy(out.element, d_out, image_size, cudaMemcpyDeviceToHost);

    /* Free device memory */
    cudaFree(d_in);
    cudaFree(d_out);
}
```

Figure 1: Implementation of *compute_on_device*

```cpp
__global__ void blur_filter_kernel (const float *in, float *out, int size)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int index = row * size + col;

    /* Check if the thread is within the image bounds */
    if (row < size && col < size) {
        /* Perform the box blur on the current pixel */
        int num_neighbors = 0;
        float sum = 0.0f;

        /* Iterate over the neighbors of the current pixel */
        for (int i = -BLUR_SIZE; i <= BLUR_SIZE; i++) {
            for (int j = -BLUR_SIZE; j <= BLUR_SIZE; j++) {
                int neighbor_row = row + i;
                int neighbor_col = col + j;

                /* Check if the neighbor is within the image bounds */
                if (neighbor_row >= 0 && neighbor_row < size && neighbor_col >= 0 && neighbor_col < size) {
                    num_neighbors++;
                    sum += in[neighbor_row * size + neighbor_col];
                }
            }
        }

        /* Calculate the average value for the current pixel */
        out[index] = sum / num_neighbors;
    }
}

#endif /* _BLUR_FILTER_KERNEL_H_ */
```

Figure 2: Implementation of *blur_filter_kernel*

The kernel function *blur_filter_kernel* is the core element responsible for calculating the blur filter on each pixel in parallel. The function takes three parameters: *in* (input image), *out* (output image), and *size* (size of the image). Within the function, the row and column indices of the current thread are calculated based on the block and thread indices used to determine the pixel position being processed. To ensure that each thread performs the computation only on valid pixels, the code includes a conditional statement that checks if the thread is within the image bounds. If the condition is met, the computation proceeds; otherwise, the thread does nothing. Inside the conditional statement, the code initializes variables to track the number of neighboring pixels (*num_neighbors*) and the sum of their values (*sum*).

Using nested loops, the code iterates over the rows and columns from *-BLUR_SIZE to BLUR_SIZE* so all neighboring pixels are considered. For each neighbor, the code calculates the row and column indices and checks if the neighbor is within the image bounds. After this, the code calculates the average value for the current pixel by dividing the sum of neighbor values by the number of neighbors. Finally, the calculated average value is assigned to the corresponding output pixel.

Performance (Execution Time) Comparison

Image Size	Serial	Parallel
512x512	0.010401s	0.243658s
4096x4096	0.459274s	0.275878s
8192x8192	1.657029s	0.456612s

Speedup Comparison

Image Size	Speedup
512x512	0.0427
4096x4096	1.665
8192x8192	3.629