DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

Drexel University

Electrical and Computer Engineering Dept.

ECEC-413

Blur Filter with CUDA

Chris Kasper
Prof. Naga Kandasamy
DATE: 05/6/19

GPU Implementation (using CUDA)

To implement a box blur filter suitable for execution on the GPU, the CUDA programming language was utilized. Doing so, the CUDA code is broken into two parts essentially: initialization/free and the kernel. In initialization/free, the data structures are allocated onto the "device" (in this case the GPU), and then freed once the kernel function is finished. The kernel function is the logic being ran on the device to compute the box blur filter.

The initialization/free code is as follows:

```
void
compute_on_device (const image_t in, image_t out)
{
    struct timeval start, stop;

    /* Allocate device memory */
    image_t d_in = allocate_image_on_device(in);
    image_t d_out = allocate_image_on_device(out);

    /* Copy image to device memory */
    copy_image_to_device(d_in, in);

    /* Set up execution grid */
    dim3 thread_block(THREAD_BLOCK_SIZE, 1, 1);
    dim3 grid(NUM_THREAD_BLOCKS,1);

    /* Launch kernel */
    gettimeofday(&start, NULL);
    blur_filter_kernel<<<grid, thread_block>>>(d_in.element, d_out.element, d_in.size);
    gettimeofday(&stop, NULL);
    printf ("Execution time for GPU = %fs. \n", (float)(stop.tv_sec - start.tv_sec +\
                (stop.tv_usec - start.tv_usec)/(float)1000000));

    /* Check if kernel execution generated an error */
    cudaError_t err = cudaGetLastError();
    if (cudaSuccess != err) {
        fprintf (stderr, "Kernel execution failed: %s\n", cudaGetErrorString (err));
        exit (EXIT_FAILURE);
    }
```

```
    // Copy out back over
    copy_image_from_device(out, d_out);

    /* Clean up memory on GPU */
    cudaFree(d_in.element);
    cudaFree(d_out.element);

    return;
}
```

The first thing we do in this code is allocate both the input and output images onto the device. The function *allocate_image_on_device()* is a function I created to make the code cleaner here, but it utilizes *CudaMalloc()* to get the space allocated. Next, the input image is copied over to a pointer that represents a pointer to the input image on the device, using *copy_image_to_device()*, which is another user created function that uses *cudaMemcpy()*. With the data structures initialized, the execution grid for the kernel can be set up. For this assignment, we have a grid of 240 threads blocks, each containing 128 threads, for a total of 30,720 threads. This grid and thread block parameters are inputs to the kernel function, as well as the input and out images on the device. The execution time is then computed for the kernel, and we check for any errors that may have occurred. If there are no errors, the output image on the device is copied to back to the host version, and the device's input and output images are freed with *cudaFree()*.

Most of the kernel function's code is the same as the serial version. However, there are several small tweaks that will make sure the threads are being properly executed on the device. Here is the kernel code:

```
__global__ void
blur_filter_kernel (const float *in, float *out, int size)
{
    int pix, i, j;
    int row, col;
    int curr_row, curr_col;
    float blur_value;
    int num_neighbors;

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (pix = idx; pix < size * size; pix+=stride) {
        row = pix/size;
        col = pix % size;

        blur_value = 0.0;
        num_neighbors = 0;

        for (i = -BLUR_SIZE; i < (BLUR_SIZE + 1); i++) {
            for (j = -BLUR_SIZE; j < (BLUR_SIZE + 1); j++) {
                curr_row = row + i;
                curr_col = col + j;

                if ((curr_row > -1) && (curr_row < size) &&\
                        (curr_col > -1) && (curr_col < size)) {
                            blur_value += in[curr_row * size + curr_col];
                            num_neighbors++;
                        }
            }
        }

        out[pix] = blur_value / num_neighbors;
    }

    return;
}
```

Highlighting the changes, the first one is finding the index of the thread that's running the kernel function and getting the stride length for a thread in the case the number of threads is less than the number of elements in the image. The loop over the pixels for a particular thread now iterates based on the stride length. The rest of the code is the same as the serial, and the pixel value for the output image is stored.

Speedup

Table 1: Speedup with Blur Filter on xunil-05

Image Size	CPU	GPU
2048 x 2048	129 ms	64 µs
4096 x 4096	456 ms	85 µs
8192 x 8192	1.66 s	77 µs

Note: xunil-05 has a Nvidia 1080 GTX GPU