Tam Phi (thp29)
Hanh Do Phung (hdd29)
ECEC 622

Assignment 4 Report - CUDA Blur Filter

Introduction:

In this assignment, a 2D grayscale image blur filter is parallelize using CUDA, offloading SIMD computations to a GPU in a heterogeneous system. The filter is an element-wise averaging operation that is done on a matrix of the image's pixel values. One element's new value after the operation is 1/9 of the sum of itself and 8 other elements adjacent to it.

Design and Implementation:

Overview of image structure

The structure of an image is as follows:

```
/* Define the structure for our image or 2D array. */
typedef struct image_s {
    int size; /* Size of one side of the image. */
    float *element; /* Individual pixels in the image. */
} image_t;
```

The integer ***size*** denotes the image height that is a user-input value. The float ***element*** denotes an array with size* size number of elements of random floating-point numbers. ***Element*** represents the number of pixels within an image.

Parallelization in the kernel

There are 2 steps in implementing the parallel version of a blur filter:

1) Exec() function:

-A function that takes in ***image_t in*** and ***image_t out*** that are input image, and result store in output image, respectively. It then allocates memory, copies image to device, and creates thread blocks

Pseudo code{

/*Allocate memory on device */

d_IN = Allocate for input image

d_OUT = Allocate for output image

/* Copy image to device */

Copy image from input (d_IN) to device

Copy image from output (d_OUT) to device

/* Create thread blocks of size TILE_SIZE x TILE_SIZE 8/

/* Each grid of size [d_IN.size / threads.x] x [d_OUT.size/ threads.y] */

```
dim3 threads(TILE_SIZE, TILE_SIZE);
dim3 grid(d_IN.size/threads.x, d_OUT.size/threads.y);
```

2)Blur filter in GPU kernel:

-A function that finds the blur value of an interior pixel. Speedups utilize thread block implementation of serial version.

-Each thread block contains the following information:

- Location of thread block within the grid (blockIdx)
- Location of each thread within the thread block (threadIdx)

-Then the row and column can be identified using the following formula:

Pseudo code {

/* blockDim.x is dimension of row within the thread block */

/* blockDim.y is dimension of col within the thread block */

Col = blockDim.x * blockIdx.x + threadIdx.x;

Row= blockDim.y * blockIdx.y + threadIdx.y;

```
/* Obtain thread location within the block */
int threadX = threadIdx.x;
int threadY=threadIdx.y;
/* Obtain block index within the grid */
int blockX = blockIdx.x;
int blockY = blockIdx.y;

/*TILE_SIZE = blockDim.x or blockDim.y */
int col = TILE_SIZE*blockX + threadX;
int row = TILE_SIZE*blockY + threadY;
```

The calculation of blur value is similar to the serial one and the results are store in the ***elements of image_t out.***

```
/* Apply blur filter to current pixel */
float blur_value = 0.0;
int 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 += 1;
        }
    }
}
out[row *size + col] = blur_value/num_neighbors;

return;
```

Result and Discussion:

The time measured only accounted for the computation of new pixel values, not including any data moving on the bus between CPU and GPU. All tests were passed with various image sizes and thread block sizes. The speed up is significantly better than that applications parallelized with pthreads and openMP, with execution time on GPU is faster than gold computation on CPU by up to 24890 times (observed when a 8192x8192 image is passed in, and block dimension of 32 is used).

Detail timing data on the execution time is shown in the Table 1, while the magnitude of speed up is illustrated in Figure 1 and Figure 2.

pixel/edge	block_dim	gold_time	gpu_time	test_failed	speedup
512	8	0.009886	5.5E-05	0	179.745453
512	16	0.007886	5.5E-05	0	143.381821
512	32	0.009841	5.1E-05	0	192.960785
1024	8	0.033425	5E-05	0	668.499512
1024	16	0.032999	6.1E-05	0	540.967224
1024	32	0.033513	8.3E-05	0	403.771057
4096	8	0.455709	6.6E-05	0	6904.681641
4096	16	0.493828	7.5E-05	0	6584.373047
4096	32	0.495384	7E-05	0	7076.914062
8192	8	1.721446	8.4E-05	0	20493.40625
8192	16	1.692522	6.8E-05	0	24890.029297
8192	32	1.657041	7.5E-05	0	22093.880859

Table 1: Timing results of blur filter execution time

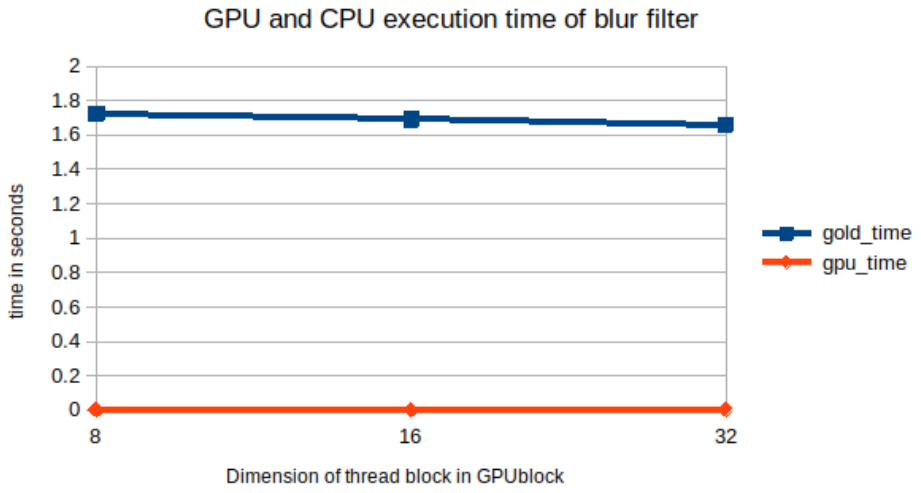

Figure 1. GPU and CPU execution time in seconds.

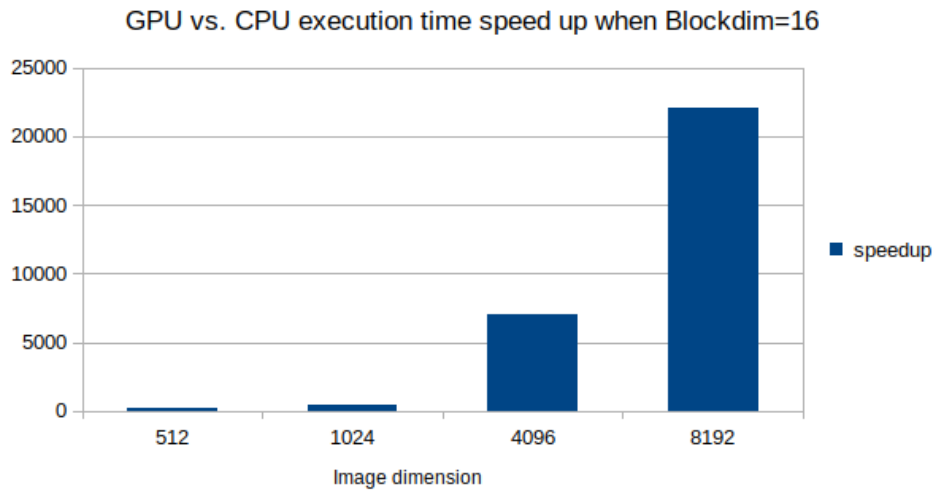

Figure 2. GPU execution time speed up relative to that of CPU (when BLOCKDIM=16)

From the above figures, 2 trends can be observed:

1) The speed up of the parallelized version, when executed on same size images, does not vary significantly with the user defined GPU thread block dimension.

2) However, with the same thread block dimension, the larger the image is, or in other words, the more computation operations there are, the more magnificent the speed up is.

Note that the small decrease in gold execution time in figure 1 between runs might be caused by the CPU cache warming up after runs, decreasing the number of miss in cache.

Conclusion:

Using the GPU with CUDA APIs, the blur filter was parallelized successfully with significant speed up when compared to the serial version on CPU. However, as the implementation only made use of the GPU's global memory, not each SM's shared memory, there is still room for improvement.