

# Module 8 Lab

## Stencil

*GPU Teaching Kit – Accelerated Computing*

### OBJECTIVE

The purpose of this lab is to perform shared-memory tiling by implementing a 7-point stencil.

### INSTRUCTIONS

- Edit the code to implement a 7-point stencil.
- Edit the code to launch the kernel you implemented. The function should launch 2D CUDA grid and blocks.
- Answer the questions found in the questions tab.

### ALGORITHM

You will be implementing a 7-point stencil without having to deal with boundary conditions. The result is clamped so the range is between the values of 0 and 255.

```
for i from 1 to height-1:  # notice the ranges exclude the boundary
    for j from 1 to width-1: # this is done for simplification
        for k from 1 to depth-1: # the output is set to 0 along the boundary
            res = in(i, j, k + 1) + in(i, j, k - 1) + in(i, j + 1, k) +
                  in(i, j - 1, k) + in(i + 1, j, k) + in(i - 1, j, k) -
                  6 * in(i, j, k)
            out(i, j, k) = Clamp(res, 0, 255)
        end
    end
end
```

With Clamp defined as

```
def Clamp(val, start, end):
    return Max(Min(val, end), start)
end
```

And `in(i, j, k)` and `out(i, j, k)` are helper functions defined as

```
#define value(array, i, j, k) array[(( i ) * width + (j)) * depth + (k)]
#define in(i, j, k)    value(input_array, i, j, k)
#define out(i, j, k)   value(output_array, i, j, k)
```

## QUESTIONS

(1) How many global memory reads does your program make?

ANSWER: **This code makes  $\text{BLOCK\_SIZE} * \text{BLOCK\_SIZE} * \text{ceil}(\text{size\_x} / \text{TILE\_SIZE}) * \text{ceil}(\text{size\_y} / \text{tile\_size})$  reads.**

(2) How many shared memory reads does your program make?

ANSWER: **Each thread reads 7 values, and there are  $\text{BLOCK\_SIZE} * \text{BLOCK\_SIZE} * \text{ceil}(\text{size\_x} / \text{TILE\_SIZE}) * \text{ceil}(\text{size\_y} / \text{TILE\_SIZE})$  threads.**

(3) This stencil would need a 3x3 convolution kernel, where the center entry is -6 and the adjacent entries are 1. Corner entries would be 0.

ANSWER: **If you already had a working convolution code, how could you use it to implement this stencil?**

(4) Does your stencil make more, equal, or fewer memory accesses than the equivalent 3x3 convolution code would?

ANSWER: **This code makes fewer accesses - the equivalent convolution code would load the values where the kernel is 0 on the corners.**

## CODE TEMPLATE

The following code is suggested as a starting point for students. The code handles the import and export as well as the checking of the solution. Students are expected to insert their code in the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
1  #include <wb.h>
2
3  #define wbCheck(stmt)                                     \
4      do {                                                 \
5          cudaError_t err = stmt;                           \
6          if (err != cudaSuccess) {                         \
7              wbLog(ERROR, "Failed to run stmt ", #stmt);   \
8              wbLog(ERROR, "Got CUDA error ... ", cudaGetErrorString(err)); \
9              return -1;                                     \
10         }                                                 \
11     } while (0)
12
13  __global__ void stencil(float *output, float *input, int width, int height,
```

```

14         int depth) {
15     ///@@ INSERT CODE HERE
16 }
17
18 static void launch_stencil(float *deviceOutputData, float *deviceInputData,
19                             int width, int height, int depth) {
20     ///@@ INSERT CODE HERE
21 }
22
23 int main(int argc, char *argv[]) {
24     wbArg_t arg;
25     int width;
26     int height;
27     int depth;
28     char *inputFile;
29     wbImage_t input;
30     wbImage_t output;
31     float *hostInputData;
32     float *hostOutputData;
33     float *deviceInputData;
34     float *deviceOutputData;
35
36     arg = wbArg_read(argc, argv);
37
38     inputFile = wbArg_getInputFile(arg, 0);
39
40     input = wbImport(inputFile);
41
42     width  = wbImage_getWidth(input);
43     height = wbImage_getHeight(input);
44     depth  = wbImage_getChannels(input);
45
46     output = wbImage_new(width, height, depth);
47
48     hostInputData  = wbImage_getData(input);
49     hostOutputData = wbImage_getData(output);
50
51     wbTime_start(GPU, "Doing GPU memory allocation");
52     cudaMalloc((void **)&deviceInputData,
53                width * height * depth * sizeof(float));
54     cudaMalloc((void **)&deviceOutputData,
55                width * height * depth * sizeof(float));
56     wbTime_stop(GPU, "Doing GPU memory allocation");
57
58     wbTime_start(Copy, "Copying data to the GPU");
59     cudaMemcpy(deviceInputData, hostInputData,
60                width * height * depth * sizeof(float),
61                cudaMemcpyHostToDevice);
62     wbTime_stop(Copy, "Copying data to the GPU");
63
64     wbTime_start(Compute, "Doing the computation on the GPU");
65     launch_stencil(deviceOutputData, deviceInputData, width, height, depth);
66     wbTime_stop(Compute, "Doing the computation on the GPU");

```

```

67
68     wbTime_start(Copy, "Copying data from the GPU");
69     cudaMemcpy(hostOutputData, deviceOutputData,
70               width * height * depth * sizeof(float),
71               cudaMemcpyDeviceToHost);
72     wbTime_stop(Copy, "Copying data from the GPU");
73
74     wbSolution(arg, output);
75
76     cudaFree(deviceInputData);
77     cudaFree(deviceOutputData);
78
79     wbImage_delete(output);
80     wbImage_delete(input);
81
82     return 0;
83 }

```

## CODE SOLUTION

The following is a possible implementation of the lab. This solution is intended for use only by the teaching staff and should not be distributed to students.

```

1  #include <wb.h>
2
3  #define wbCheck(stmt)                                     \
4      do {                                                 \
5          cudaError_t err = stmt;                           \
6          if (err != cudaSuccess) {                         \
7              wbLog(ERROR, "Failed to run stmt ", #stmt);   \
8              wbLog(ERROR, "Got CUDA error ... ", cudaGetErrorString(err)); \
9              return -1;                                     \
10         }                                                 \
11     } while (0)
12
13     #define dx 32
14     #define dy 32
15     #define TILE_SIZE 32
16
17     __host__ __device__ float Clamp(float val, float start, float end) {
18         return max(min(val, end), start);
19     }
20
21     void stencil_cpu(float *_out, float *_in, int width, int height,
22                     int depth) {
23
24         #define out(i, j, k) _out[((i)*width + (j)) * depth + (k)]
25         #define in(i, j, k) _in[((i)*width + (j)) * depth + (k)]
26
27         float res;
28         for (int i = 1; i < height - 1; ++i) {

```

```

29     for (int j = 1; j < width - 1; ++j) {
30         for (int k = 1; k < depth - 1; ++k) {
31             res = in(i, j, k + 1) + in(i, j, k - 1) + in(i, j + 1, k) +
32                 in(i, j - 1, k) + in(i + 1, j, k) + in(i - 1, j, k) -
33                 6 * in(i, j, k);
34             out(i, j, k) = Clamp(res, 0, 255);
35         }
36     }
37 }
38 }
39
40 __global__ void stencil(float *_out, float *_in, int width, int height,
41                        int depth) {
42
43     int k = blockIdx.z * TILE_SIZE + threadIdx.z;
44     int j = blockIdx.x * TILE_SIZE + threadIdx.x;
45
46     __shared__ float ds_A[TILE_SIZE][TILE_SIZE];
47     float bottom = in(0, j, k);
48     float current = in(1, j, k);
49
50     float top = in(2, j, k);
51
52     ds_A[threadIdx.z][threadIdx.x] = current;
53
54     __syncthreads();
55
56     for (int i = 1; i < height - 1; i++) {
57         float temp = 0;
58
59         if (k < depth - 1 && k > 0 && j < width - 1 && j > 0) {
60
61             temp = bottom + top;
62             if (threadIdx.z > 0) {
63                 temp += ds_A[threadIdx.z - 1][threadIdx.x];
64             } else {
65                 temp += in(i, j, k - 1);
66             }
67             if (threadIdx.z < TILE_SIZE - 1) {
68                 temp += ds_A[threadIdx.z + 1][threadIdx.x];
69             } else {
70                 temp += in(i, j, k + 1);
71             }
72
73             if (threadIdx.x > 0) {
74                 temp += ds_A[threadIdx.z][threadIdx.x - 1];
75             } else {
76                 temp += in(i, j - 1, k);
77             }
78             if (threadIdx.x < TILE_SIZE - 1) {
79                 temp += ds_A[threadIdx.z][threadIdx.x + 1];
80             } else {
81                 temp += in(i, j + 1, k);

```

```

82     }
83
84     temp -= 6 * current;
85
86     out(i, j, k) = Clamp(temp, 0, 255);
87 }
88
89 bottom = current;
90
91 __syncthreads();
92 ds_A[threadIdx.z][threadIdx.x] = top;
93 __syncthreads();
94 current = top;
95
96 top = in(i + 2, j, k);
97 }
98 }
99
100 static void launch_stencil(float *deviceOutputData, float *deviceInputData,
101                             int width, int height, int depth) {
102     /// INSERT CODE HERE
103
104     const unsigned int zBlocks = (depth - 1) / TILE_SIZE + 1;
105     const unsigned int xBlocks = (width - 1) / TILE_SIZE + 1;
106
107     dim3 GridD(xBlocks, 1, zBlocks);
108     dim3 BlockD(TILE_SIZE, 1, TILE_SIZE);
109
110     stencil<<<GridD, BlockD>>>(deviceOutputData, deviceInputData, width,
111                                 height, depth);
112 }
113
114 int main(int argc, char *argv[]) {
115     wbArg_t arg;
116     int width;
117     int height;
118     int depth;
119     char *inputFile;
120     wbImage_t input;
121     wbImage_t output;
122     float *hostInputData;
123     float *hostOutputData;
124     float *deviceInputData;
125     float *deviceOutputData;
126
127     arg = wbArg_read(argc, argv);
128
129     inputFile = wbArg_getInputFile(arg, 0);
130
131     input = wbImport(inputFile);
132
133     width = wbImage_getWidth(input);
134     height = wbImage_getHeight(input);

```

```

135     depth = wbImage_getChannels(input);
136
137     output = wbImage_new(width, height, depth);
138
139     hostInputData = wbImage_getData(input);
140     hostOutputData = wbImage_getData(output);
141
142     wbTime_start(GPU, "Doing GPU memory allocation");
143     cudaMalloc((void **)&deviceInputData,
144               width * height * depth * sizeof(float));
145     cudaMalloc((void **)&deviceOutputData,
146               width * height * depth * sizeof(float));
147     wbTime_stop(GPU, "Doing GPU memory allocation");
148
149     wbTime_start(Copy, "Copying data to the GPU");
150     cudaMemcpy(deviceInputData, hostInputData,
151               width * height * depth * sizeof(float),
152               cudaMemcpyHostToDevice);
153     wbTime_stop(Copy, "Copying data to the GPU");
154
155     wbTime_start(Compute, "Doing the computation on the GPU");
156     launch_stencil(deviceOutputData, deviceInputData, width, height, depth);
157     wbTime_stop(Compute, "Doing the computation on the GPU");
158
159     wbTime_start(Copy, "Copying data from the GPU");
160     cudaMemcpy(hostOutputData, deviceOutputData,
161               width * height * depth * sizeof(float),
162               cudaMemcpyDeviceToHost);
163     wbTime_stop(Copy, "Copying data from the GPU");
164
165     wbSolution(arg, output);
166
167     cudaFree(deviceInputData);
168     cudaFree(deviceOutputData);
169
170     wbImage_delete(output);
171     wbImage_delete(input);
172
173     return 0;
174 }

```