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(arry, i, j, k) arry[(( 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** $BLOCK\_SIZE * BLOCK\_SIZE * ceil(size\_x/TILE\_SIZE) * ceil(size\_y/tile\_size)$ **reads.**

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

ANSWER: **Each thread reads 7 values, and there are** $BLOCK\_SIZE * BLOCK\_SIZE * ceil(size\_x/TILE\_SIZE) * ceil(size\_y/TILE\_SIZE)$ **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 is the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
#include <wb.h>

#define wbCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
      wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));      \
      return -1;                                                          \
    }                                                                     \
  } while (0)

__global__ void stencil(float *output, float *input, int width, int height,
                        int depth) {
  //@@ INSERT CODE HERE
}

static void launch_stencil(float *deviceOutputData, float *deviceInputData,
                           int width, int height, int depth) {
  //@@ INSERT CODE HERE
}

int main(int argc, char *argv[]) {
  wbArg_t arg;
  int width;
  int height;
  int depth;
  char *inputFile;
  wbImage_t input;
  wbImage_t output;
  float *hostInputData;
  float *hostOutputData;
  float *deviceInputData;
  float *deviceOutputData;

  arg = wbArg_read(argc, argv);

  inputFile = wbArg_getInputFile(arg, 0);

  input = wbImport(inputFile);

  width  = wbImage_getWidth(input);
  height = wbImage_getHeight(input);
  depth  = wbImage_getChannels(input);

  output = wbImage_new(width, height, depth);

  hostInputData  = wbImage_getData(input);
  hostOutputData = wbImage_getData(output);

  wbTime_start(GPU, "Doing GPU memory allocation");
  cudaMalloc((void **)&deviceInputData,
             width * height * depth * sizeof(float));
  cudaMalloc((void **)&deviceOutputData,
             width * height * depth * sizeof(float));
  wbTime_stop(GPU, "Doing GPU memory allocation");

  wbTime_start(Copy, "Copying data to the GPU");
  cudaMemcpy(deviceInputData, hostInputData,
             width * height * depth * sizeof(float),
             cudaMemcpyHostToDevice);
  wbTime_stop(Copy, "Copying data to the GPU");

  wbTime_start(Compute, "Doing the computation on the GPU");
  launch_stencil(deviceOutputData, deviceInputData, width, height, depth);
  wbTime_stop(Compute, "Doing the computation on the GPU");

  wbTime_start(Copy, "Copying data from the GPU");
  cudaMemcpy(hostOutputData, deviceOutputData,
             width * height * depth * sizeof(float),
             cudaMemcpyDeviceToHost);
  wbTime_stop(Copy, "Copying data from the GPU");

  wbSolution(arg, output);

  cudaFree(deviceInputData);
  cudaFree(deviceOutputData);

  wbImage_delete(output);
  wbImage_delete(input);

  return 0;
}
```

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.

```
#include <wb.h>

#define wbCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
      wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));      \
      return -1;                                                          \
    }                                                                     \
  } while (0)

#define dx 32
#define dy 32
#define TILE_SIZE 32

__host__ __device__ float Clamp(float val, float start, float end) {
  return max(min(val, end), start);
}

void stencil_cpu(float *_out, float *_in, int width, int height,
                 int depth) {

#define out(i, j, k) _out[((i)*width + (j)) * depth + (k)]
#define in(i, j, k) _in[((i)*width + (j)) * depth + (k)]

  float res;
  for (int i = 1; i < height - 1; ++i) {
    for (int j = 1; j < width - 1; ++j) {
      for (int k = 1; k < depth - 1; ++k) {
        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);
      }
    }
  }
}

__global__ void stencil(float *_out, float *_in, int width, int height,
                        int depth) {

  int k = blockIdx.z * TILE_SIZE + threadIdx.z;
  int j = blockIdx.x * TILE_SIZE + threadIdx.x;

  __shared__ float ds_A[TILE_SIZE][TILE_SIZE];
  float bottom  = in(0, j, k);
  float current = in(1, j, k);

  float top = in(2, j, k);

  ds_A[threadIdx.z][threadIdx.x] = current;

  __syncthreads();

  for (int i = 1; i < height - 1; i++) {
    float temp = 0;

    if (k < depth - 1 && k > 0 && j < width - 1 && j > 0) {

      temp = bottom + top;
      if (threadIdx.z > 0) {
        temp += ds_A[threadIdx.z - 1][threadIdx.x];
      } else {
        temp += in(i, j, k - 1);
      }
      if (threadIdx.z < TILE_SIZE - 1) {
        temp += ds_A[threadIdx.z + 1][threadIdx.x];
      } else {
        temp += in(i, j, k + 1);
      }

      if (threadIdx.x > 0) {
        temp += ds_A[threadIdx.z][threadIdx.x - 1];
      } else {
        temp += in(i, j - 1, k);
      }
      if (threadIdx.x < TILE_SIZE - 1) {
        temp += ds_A[threadIdx.z][threadIdx.x + 1];
      } else {
        temp += in(i, j + 1, k);
      }

      temp -= 6 * current;

      out(i, j, k) = Clamp(temp, 0, 255);
    }

    bottom = current;

    __syncthreads();
    ds_A[threadIdx.z][threadIdx.x] = top;
    __syncthreads();
    current = top;

    top = in(i + 2, j, k);
  }
}

static void launch_stencil(float *deviceOutputData, float *deviceInputData,
                           int width, int height, int depth) {
  //@@ INSERT CODE HERE

  const unsigned int zBlocks = (depth - 1) / TILE_SIZE + 1;
  const unsigned int xBlocks = (width - 1) / TILE_SIZE + 1;

  dim3 GridD(xBlocks, 1, zBlocks);
  dim3 BlockD(TILE_SIZE, 1, TILE_SIZE);

  stencil<<<GridD, BlockD>>>(deviceOutputData, deviceInputData, width,
                             height, depth);
}

int main(int argc, char *argv[]) {
  wbArg_t arg;
  int width;
  int height;
  int depth;
  char *inputFile;
  wbImage_t input;
  wbImage_t output;
  float *hostInputData;
  float *hostOutputData;
  float *deviceInputData;
  float *deviceOutputData;

  arg = wbArg_read(argc, argv);

  inputFile = wbArg_getInputFile(arg, 0);

  input = wbImport(inputFile);

  width  = wbImage_getWidth(input);
  height = wbImage_getHeight(input);
  depth  = wbImage_getChannels(input);

  output = wbImage_new(width, height, depth);

  hostInputData  = wbImage_getData(input);
  hostOutputData = wbImage_getData(output);

  wbTime_start(GPU, "Doing GPU memory allocation");
  cudaMalloc((void **)&deviceInputData,
             width * height * depth * sizeof(float));
  cudaMalloc((void **)&deviceOutputData,
             width * height * depth * sizeof(float));
  wbTime_stop(GPU, "Doing GPU memory allocation");

  wbTime_start(Copy, "Copying data to the GPU");
  cudaMemcpy(deviceInputData, hostInputData,
             width * height * depth * sizeof(float),
             cudaMemcpyHostToDevice);
  wbTime_stop(Copy, "Copying data to the GPU");

  wbTime_start(Compute, "Doing the computation on the GPU");
  launch_stencil(deviceOutputData, deviceInputData, width, height, depth);
  wbTime_stop(Compute, "Doing the computation on the GPU");

  wbTime_start(Copy, "Copying data from the GPU");
  cudaMemcpy(hostOutputData, deviceOutputData,
             width * height * depth * sizeof(float),
             cudaMemcpyDeviceToHost);
  wbTime_stop(Copy, "Copying data from the GPU");

  wbSolution(arg, output);

  cudaFree(deviceInputData);
  cudaFree(deviceOutputData);

  wbImage_delete(output);
  wbImage_delete(input);

  return 0;
}
```