Module 3 Lab

CUDA Vector Add

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

The purpose of this lab is to introduce the student to the CUDA API by implementing vector addition. The student will implement vector addition by writing the GPU kernel code as well as the associated host code.

PREREQUISITES

Before starting this lab, make sure that:

- You have completed all of Module 2 in the teaching kit
- You have completed the "Device Query" lab

INSTRUCTIONS

Edit the code in the code tab to perform the following:

- Allocate device memory
- Copy host memory to device
- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernel
- Copy results from device to host
- Free device memory
- Write the CUDA kernel

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

LOCAL SETUP INSTRUCTIONS

The most recent version of source code for this lab along with the build-scripts can be found on the Bitbucket repository. A description on how to

use the CMake tool in along with how to build the labs for local development found in the README document in the root of the repository.

The executable generated as a result of compiling the lab can be run using the following command:

```
./VectorAdd_Template -e <expected.raw> -i <intput1.raw>,<input2.raw> \
  -o <output.raw> -t vector
```

where `<expected.raw>` is the expected output, `<input0.raw>,<input1.raw>` is the input dataset, and `<output.raw>` is an optional path to store the results. The datasets can be generated using the dataset generator built as part of the compilation process.

QUESTIONS

(1) How many floating operations are being performed in your vector add kernel? EXPLAIN.

ANSWER: **N - one for each pair of input vector elements.**

(2) How many global memory reads are being performed by your kernel? EXPLAIN.

ANSWER: **2N - one for each input vector element.**

(3) How many global memory writes are being performed by your kernel? EXPLAIN.

ANSWER: **N - one for each output vector element.**

(4) Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

ANSWER: **Split into multiple kernels to overlap data transfer and kernel execution.**

(5) Name three applications of vector addition.

ANSWER: **Big Data analysis, statistics, FFT.**

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.

```
1  #include <wb.h>
2
3  __global__ void vecAdd(float *in1, float *in2, float *out, int len) {
4    //@@ Insert code to implement vector addition here
```

```
}

int main(int argc, char **argv) {
  wbArg_t args;
  int inputLength;
  float *hostInput1;
  float *hostInput2;
  float *hostOutput;
  float *deviceInput1;
  float *deviceInput2;
  float *deviceOutput;

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostInput1 =
      (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
  hostInput2 =
      (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
  hostOutput = (float *)malloc(inputLength * sizeof(float));
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The input length is ", inputLength);

  wbTime_start(GPU, "Allocating GPU memory.");
  //@@ Allocate GPU memory here

  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  //@@ Copy memory to the GPU here

  wbTime_stop(GPU, "Copying input memory to the GPU.");

  //@@ Initialize the grid and block dimensions here

  wbTime_start(Compute, "Performing CUDA computation");
  //@@ Launch the GPU Kernel here

  cudaDeviceSynchronize();
  wbTime_stop(Compute, "Performing CUDA computation");

  wbTime_start(Copy, "Copying output memory to the CPU");
  //@@ Copy the GPU memory back to the CPU here

  wbTime_stop(Copy, "Copying output memory to the CPU");

  wbTime_start(GPU, "Freeing GPU Memory");
  //@@ Free the GPU memory here

  wbTime_stop(GPU, "Freeing GPU Memory");

  wbSolution(args, hostOutput, inputLength);
```

```
58
59    free(hostInput1);
60    free(hostInput2);
61    free(hostOutput);
62
63    return 0;
64  }
```

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   __global__ void vecAdd(float *in1, float *in2, float *out, int len) {
4     //@@ Insert code to implement vector addition here
5     int index = threadIdx.x + blockIdx.x * blockDim.x;
6     if (index < len) {
7       out[index] = in1[index] + in2[index];
8     }
9   }
10
11  int main(int argc, char **argv) {
12    wbArg_t args;
13    int inputLength;
14    float *hostInput1;
15    float *hostInput2;
16    float *hostOutput;
17    float *deviceInput1;
18    float *deviceInput2;
19    float *deviceOutput;
20
21    args = wbArg_read(argc, argv);
22
23    wbTime_start(Generic, "Importing data and creating memory on host");
24    hostInput1 =
25        (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
26    hostInput2 =
27        (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
28    hostOutput = (float *)malloc(inputLength * sizeof(float));
29    wbTime_stop(Generic, "Importing data and creating memory on host");
30
31    wbLog(TRACE, "The input length is ", inputLength);
32
33    wbTime_start(GPU, "Allocating GPU memory.");
34    //@@ Allocate GPU memory here
35    cudaMalloc((void **)&deviceInput1, inputLength * sizeof(float));
36    cudaMalloc((void **)&deviceInput2, inputLength * sizeof(float));
37    cudaMalloc((void **)&deviceOutput, inputLength * sizeof(float));
38    wbTime_stop(GPU, "Allocating GPU memory.");
```

```
39
40    wbTime_start(GPU, "Copying input memory to the GPU.");
41    //@@ Copy memory to the GPU here
42    cudaMemcpy(deviceInput1, hostInput1, inputLength * sizeof(float),
43               cudaMemcpyHostToDevice);
44    cudaMemcpy(deviceInput2, hostInput2, inputLength * sizeof(float),
45               cudaMemcpyHostToDevice);
46    wbTime_stop(GPU, "Copying input memory to the GPU.");
47
48    //@@ Initialize the grid and block dimensions here
49    dim3 blockDim(32);
50    dim3 gridDim(ceil(((float)inputLength) / ((float)blockDim.x)));
51
52    wbLog(TRACE, "Block dimension is ", blockDim.x);
53    wbLog(TRACE, "Grid dimension is ", gridDim.x);
54
55    wbTime_start(Compute, "Performing CUDA computation");
56    //@@ Launch the GPU Kernel here
57    vecAdd<<<gridDim, blockDim>>>(deviceInput1, deviceInput2, deviceOutput,
58                                  inputLength);
59    cudaDeviceSynchronize();
60    wbTime_stop(Compute, "Performing CUDA computation");
61
62    wbTime_start(Copy, "Copying output memory to the CPU");
63    //@@ Copy the GPU memory back to the CPU here
64    cudaMemcpy(hostOutput, deviceOutput, inputLength * sizeof(float),
65               cudaMemcpyDeviceToHost);
66    wbTime_stop(Copy, "Copying output memory to the CPU");
67
68    wbTime_start(GPU, "Freeing GPU Memory");
69    //@@ Free the GPU memory here
70    cudaFree(deviceInput1);
71    cudaFree(deviceInput2);
72    cudaFree(deviceOutput);
73    wbTime_stop(GPU, "Freeing GPU Memory");
74
75    wbSolution(args, hostOutput, inputLength);
76
77    free(hostInput1);
78    free(hostInput2);
79    free(hostOutput);
80
81    return 0;
82  }
```