Module 14 Lab

Vector Addition Using CUDA Streams

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

The purpose of this lab is to get you familiar with using the CUDA streaming API by re-implementing a the vector addition lab to use CUDA streams.

PREREQUISITES

Before starting this lab, make sure that:

- You have completed the vector addition lab
- You have completed all Module 14 lecture videos

INSTRUCTION

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

- You MUST use at least 4 CUDA streams in your program, but you may adjust it to be larger for largest datasets.
- Allocate device memory
- Interleave the host memory copy to device to hide
- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernel
- Copy results from device to host asynchronously

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_Stream_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) Compared to the non-streamed vector addition, what performance gain do you get?

ANSWER: **Since the problem is bandwidth bound, you may hide some of the memory copy overhead.**

(2) How did you hide the copy / kernel latency?

ANSWER: **You can operate on the vector in chunks, assigning each chunk to a different sream.**

(3) Can you hide two memory copy latencies at the same time?

ANSWER: **Yes, although the PCI-E bus is shared between invocations, the PCI-E bus has different channels for reads and writes, so reads and writes and be overlaped.**

(4) Describe how you'd use streaming to optimize the image convolution MP.

ANSWER: **One can either operate each channel on a different stream, but because the image channels are interleaved that would not be efficient. A better solution would be to subdivide the image and operate each on a different scheme. The halos need to be either recomputed or shared between stream invocations.**

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
```

```
__global__ void vecAdd(float *in1, float *in2, float *out, int len) {
  //@@ 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");

  wbSolution(args, hostOutput, inputLength);

  free(hostInput1);
  free(hostInput2);
  free(hostOutput);

  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)

__global__ void vecAdd(float *in1, float *in2, float *out, int len) {
```

```
14    //@@ Insert code to implement vector addition here
15    int i = threadIdx.x + blockDim.x * blockIdx.x;
16    if (i < len)
17      out[i] = in1[i] + in2[i];
18  }
19
20  int main(int argc, char **argv) {
21    wbArg_t args;
22    int inputLength;
23    float *hostInput1;
24    float *hostInput2;
25    float *hostOutput;
26
27    args = wbArg_read(argc, argv);
28
29    wbTime_start(Generic, "Importing data and creating memory on host");
30    hostInput1 =
31        (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
32    hostInput2 =
33        (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
34    hostOutput = (float *)malloc(inputLength * sizeof(float));
35    wbTime_stop(Generic, "Importing data and creating memory on host");
36
37    cudaStream_t stream[4];
38    float *d_A[4], *d_B[4], *d_C[4];
39    int i, k, Seglen = 1024;
40    int Gridlen = (Seglen - 1) / 256 + 1;
41
42    for (i = 0; i < 4; i++) {
43      cudaStreamCreate(&stream[i]);
44      wbCheck(cudaMalloc((void **)&d_A[i], Seglen * sizeof(float)));
45      wbCheck(cudaMalloc((void **)&d_B[i], Seglen * sizeof(float)));
46      wbCheck(cudaMalloc((void **)&d_C[i], Seglen * sizeof(float)));
47    }
48
49    for (i = 0; i < inputLength; i += Seglen * 4) {
50      for (k = 0; k < 4; k++) {
51        cudaMemcpyAsync(d_A[k], hostInput1 + i + k * Seglen,
52                        Seglen * sizeof(float), cudaMemcpyHostToDevice,
53                        stream[k]);
54        cudaMemcpyAsync(d_B[k], hostInput2 + i + k * Seglen,
55                        Seglen * sizeof(float), cudaMemcpyHostToDevice,
56                        stream[k]);
57        vecAdd<<<Gridlen, 256, 0, stream[k]>>>(d_A[k], d_B[k], d_C[k],
58                                               Seglen);
59      }
60      cudaStreamSynchronize(stream[0]);
61      cudaStreamSynchronize(stream[1]);
62      cudaStreamSynchronize(stream[2]);
63      cudaStreamSynchronize(stream[3]);
64      for (k = 0; k < 4; k++) {
65        cudaMemcpyAsync(hostOutput + i + k * Seglen, d_C[k],
66                        Seglen * sizeof(float), cudaMemcpyDeviceToHost,
```

```
67                        stream[k]);
68       }
69     }
70     cudaDeviceSynchronize();
71
72     wbSolution(args, hostOutput, inputLength);
73
74     free(hostInput1);
75     free(hostInput2);
76     free(hostOutput);
77
78     for (k = 0; k < 4; k++) {
79       cudaFree(d_A[k]);
80       cudaFree(d_B[k]);
81       cudaFree(d_C[k]);
82     }
83
84     return 0;
85   }
```