EE-147 Lab 4
CUDA Streams

Cody Simons
861177050

May 30, 2018

1 Introduction

In this lab we revisit our vector addition code from lab one. The outcome should be the same, but this time we are modifying our code to make use of streaming. This allows us to begin calculations before all of the data is written to the GPU, which helps to offset the cost of moving data to and from the GPU.

2 Results

Comparing the streaming and the non-streaming version of vector add, we can see that the time taken to do the computation is a lot lower. This is because the kernel doesn't have to wait for the memory copy to complete before it begins its computation. There is some overhead to create the streams, so the streaming version does take longer to run. A comparison of the computation time and memcpy time is shown in fig 1. The time line generated by the Nvidia visual profiler is also shown in fig 2 and in fig 3. We can see that in the non-streaming version, the computation and memory copies are overlapping and take up much less space.

3 Questions

1. What is the speed up between the non-Stream and Stream version of Vector Add? Where do the improvement comes from?
   The speed up comes from the stream essentially pipelining the vector add kernel. It allows the vector addition to start while the data is still streaming to the GPU. Then before the kernel finishes running it can move the result back to host.

2. How can data transfers be further optimized?
   You could initialize the host variable in pinned memory so that the you don't have to copy the variable into pinned memory before you start the transfer.

3. Do ordering of various CUDA API calls on the host side matter when implementing streams? Why or why not?

non-stream vector add	116.21 ms
memcpy HtoD	1.74 ms
memcpy DtoH	1.92 ms
kernel	0.103 ms
stream vector add	181.69 ms
memcpy HtoD	1.51 ms
memcpy DtoH	0.65 ms
kernel	0.08 ms

Figure 1: Comparison of streaming and non-streaming memory

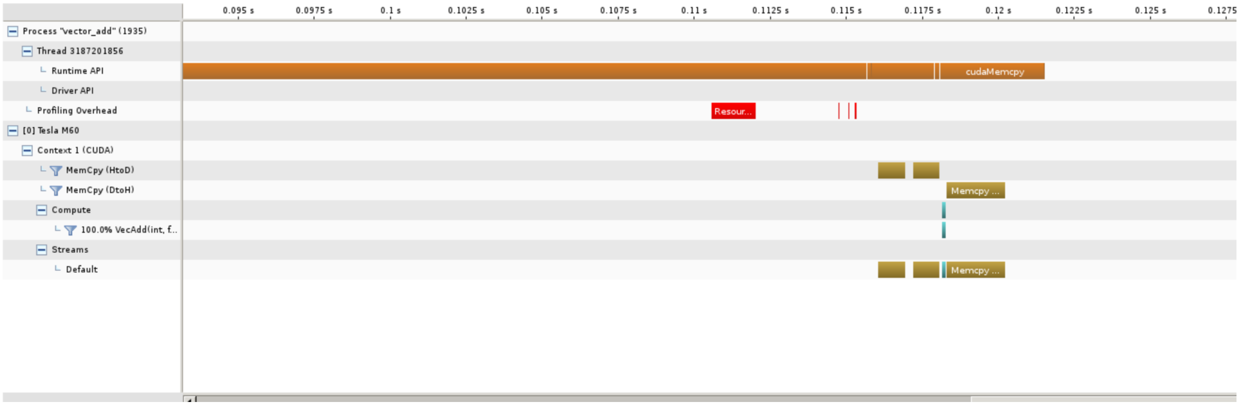

Figure 2: Non-streaming Vector Add

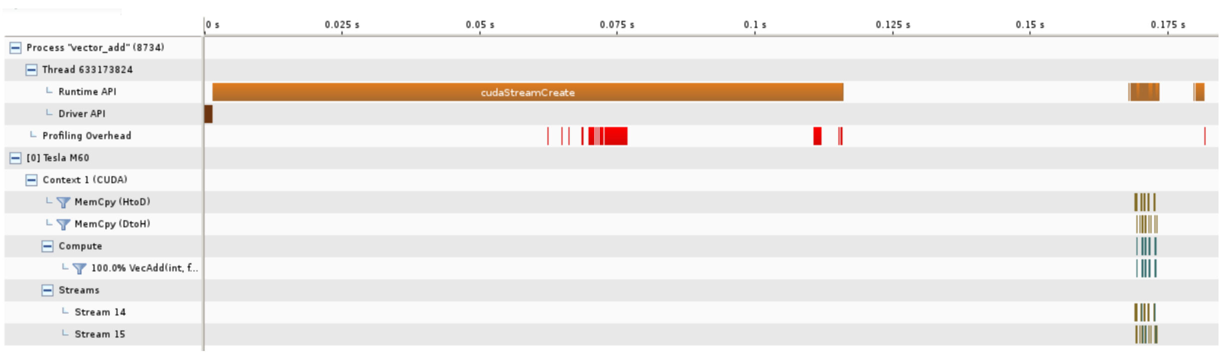

Figure 3: Streaming Vector Add

Yes, generally you need to start streaming the data before you call the kernel. It would also be better to start streaming the data for both stream before you start the kernels, so that you maximize how much is happening in parallel.

4 Code

In this lab the kernel code is reused from lab 1 with no modification, so it is not shown. The main file is modified to create two streams and split the computation between the two streams. The code is shown below.

4.1 main.cu

```
/******************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
 ******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "kernel.cu"
#include "support.cu"

#define SEG_SIZE 100000
#define BLOCK_SIZE 256

int main (int argc, char *argv[])
{
    unsigned VecSize;
```

```
    if (argc == 1) {
        VecSize = 1000000;
    } else if (argc == 2) {
        VecSize = atoi(argv[1]);
    } else {
        printf("\nOh no!\nUsage: ./vecAdd <Size>");
        exit(0);
    }

    //set standard seed
    srand(217); //Defualt value 217, DO NOT TOUCH

    Timer timer;
    cudaError_t cuda_ret;

    // Initialize host variables ----------------------------------------------

    printf("\nSetting up the problem...\n"); fflush(stdout);
    startTime(&timer);

    cudaStream_t stream0, stream1;
    printf("Creating stream0...\n"); fflush(stdout);
    cuda_ret = cudaStreamCreate(&stream0);
    if(cuda_ret != cudaSuccess) {
        printf("Failed to create stream 0, exiting"); fflush(stdout);
        return 0;
    }
    printf("Stream 0 created!\nCreating stream1...\n"); fflush(stdout);
    cuda_ret = cudaStreamCreate(&stream1);
    if(cuda_ret != cudaSuccess) {
        printf("Failed to create stream 0, exiting"); fflush(stdout);
        return 0;
    }
    printf("Stream 1 created!\n"); fflush(stdout);

    float *A_d0, *B_d0, *C_d0; // device memory for stream 0
    float *A_d1, *B_d1, *C_d1; // device memory for stream 1

    float *A_h, *B_h, *C_h;
    size_t d0_sz, d1_sz;

    dim3 dim_grid, dim_block;

    d0_sz = VecSize/2;
    d1_sz = (VecSize-1)/2 + 1;

    A_h = (float*) malloc(sizeof(float)*VecSize);
    for (unsigned int i=0; i < VecSize; i++) { A_h[i] = (rand()%100)/100.00; }

    B_h = (float*) malloc(sizeof(float)*VecSize);
    for (unsigned int i=0; i < VecSize; i++) { B_h[i] = (rand()%100)/100.00; }

    C_h = (float*) malloc(sizeof(float)*VecSize);

    stopTime(&timer); printf("%f s\n", elapsedTime(timer));
    printf("    size Of vector: %u x %u\n  ", VecSize,1);

    // Allocate device variables ----------------------------------------------

    printf("Allocating device variables..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    cudaMalloc((void **) &A_d0, sizeof(float)*d0_sz);
    cudaMalloc((void **) &B_d0, sizeof(float)*d0_sz);
    cudaMalloc((void **) &C_d0, sizeof(float)*d0_sz);

    cudaMalloc((void **) &A_d1, sizeof(float)*d1_sz);
    cudaMalloc((void **) &B_d1, sizeof(float)*d1_sz);
    cudaMalloc((void **) &C_d1, sizeof(float)*d1_sz);

```

```
    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Run streams here ------------------------------------------------------

    printf("Running streaming operations..."); fflush(stdout);
    startTime(&timer);

    dim3 DimGrid((SEG_SIZE-1)/BLOCK_SIZE+1,1,1);
    dim3 DimBlock(BLOCK_SIZE,1,1);

    for(int i=0;i<VecSize;i+=SEG_SIZE*2) {
        if(i+SEG_SIZE*2<VecSize) {
            cudaMemcpyAsync(A_d0, A_h+i, SEG_SIZE*sizeof(float),
                            cudaMemcpyHostToDevice, stream0);
            cudaMemcpyAsync(B_d0, B_h+i, SEG_SIZE*sizeof(float),
                            cudaMemcpyHostToDevice,stream0);
            cudaMemcpyAsync(A_d1, A_h+i+SEG_SIZE, SEG_SIZE*sizeof(float),
                            cudaMemcpyHostToDevice,stream1);
            cudaMemcpyAsync(B_d1, B_h+i+SEG_SIZE, SEG_SIZE*sizeof(float),
                            cudaMemcpyHostToDevice,stream1);
        } else {
            cudaMemcpyAsync(A_d0, A_h+i, (VecSize-i)/2*sizeof(float),
                            cudaMemcpyHostToDevice, stream0);
            cudaMemcpyAsync(B_d0, B_h+i, (VecSize-i)/2*sizeof(float),
                            cudaMemcpyHostToDevice,stream0);
            cudaMemcpyAsync(A_d1, A_h+i+(VecSize-i)/2, ((VecSize-i-1)/2+1)*sizeof(float),
                            cudaMemcpyHostToDevice,stream1);
            cudaMemcpyAsync(B_d1, B_h+i+(VecSize-i)/2, ((VecSize-i-1)/2+1)*sizeof(float),
                            cudaMemcpyHostToDevice,stream1);
        }

        VecAdd<<<DimGrid, DimBlock, 0, stream0>>>(d0_sz, A_d0, B_d0, C_d0);
        VecAdd<<<DimGrid, DimBlock, 0, stream1>>>(d1_sz, A_d1, B_d1, C_d1);

        if(i+SEG_SIZE*2<VecSize) {
            cudaMemcpyAsync(C_h+i, C_d0, SEG_SIZE*sizeof(float),
                            cudaMemcpyDeviceToHost,stream0);
            cudaMemcpyAsync(C_h+i+SEG_SIZE, C_d1, SEG_SIZE*sizeof(float),
                            cudaMemcpyDeviceToHost,stream1);
        } else {
            cudaMemcpyAsync(C_h+i, C_d0, (VecSize-i)/2*sizeof(float),
                            cudaMemcpyDeviceToHost,stream0);
            cudaMemcpyAsync(C_h+i+(VecSize-i)/2, C_d1, ((VecSize-i-1)/2+1)*sizeof(float),
                            cudaMemcpyDeviceToHost,stream1);
        }
    }

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Verify correctness ----------------------------------------------------

    printf("Verifying results..."); fflush(stdout);

    verify(A_h, B_h, C_h, VecSize);


    // Free memory -----------------------------------------------------------

    free(A_h);
    free(B_h);
    free(C_h);

    //INSERT CODE HERE
    cudaFree(A_d0);
    cudaFree(B_d0);
    cudaFree(C_d0);

    cudaFree(A_d1);
    cudaFree(B_d1);
```

```
    cudaFree(C_d1);

    return 0;

}
```

5 Program Output

The output is shown below, the verification output is suppressed, because it would print literally a million elements.

```
bender /home/eemaj/csimons/EE-147/Lab 4 $ ./vector_add

Setting up the problem...
Creating stream0...
Stream 0 created!
Creating stream1...
Stream 1 created!
0.498974 s
    size Of vector: 1000000 x 1
  Allocating device variables...0.000967 s
Running streaming operations...0.005558 s
Verifying results...
TEST PASSED
```