Lab 2: CUDA Vector Add

Objective

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

Instructions

CPU Version

Implement a function that performs vector addition on the CPU. The vector addition function should add two vectors A and B to produce C, where $C_i = A_i + B_i$. Allow the user to provide the vector size as input from the command line. If your executable is named as `lab2` then the user should be able to run your program by typing the following on the command line:

```
./lab2 n,
```

where n is the size of the vector, e.g. 10000. Initialize the contents of the vectors A, B using the sine and cosine function as follows:

```
a[i] = sin(i)^2, b[i] = cos(i)^2.
```

Then add A and B component wise and save the results into the vector C. Sum up vector C and print the result divided by n (the size of the vector). The output should be approximately 1.

GPU Version

First, reset the contents of the host vector C to be all 0s. Print the sum of the entries of the host vector C to confirm that its elements have been reset to 0. Then convert your CPU vector addition function to a CUDA kernel. Do not remove your CPU function implementation. The same correctness check mentioned above can be used to check If your implementation is correct after transferring the data from the device vector C to the host vector C. Note that the CUDA kernel should just add A and B and save the results on the device vector C, i.e. do not sum the elements of the vector C in the CUDA kernel but rather on the host code after you have transferred the device vector C to host.

Reminder of some of the steps involved in the implementation of a CUDA kernel:

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

- Copy results from device to host
- Free device memory

Benchmarking

Time both your CPU and GPU vector add implementations with vectors of size 1000 (10^3), 10000 (10^4), 100000 (10^5), 1000000 (10^6), 10000000 (10^7), 100000000 (10^8), 1000000000 (10^9). Using your favorite plotting tool, plot the run times in milliseconds. Your plot should have vector size in the x-axis and run times for both CPU and GPU implementations on the y-axis. Also, record the speed ups obtained, computed as ratios of CPU / GPU execution times. Run your benchmarks multiple times and report average run times.

Time the host implementation using the clock_gettime() function, which has nanosecond precision. Use CLOCK_MONOTONIC_RAW for the parameter of type clockid_t, which represents the absolute elapsed wall-clock time since some arbitrary, fixed point in the past. Documentation about this function can be found here: https://man7.org/linux/man-pages/man3/clock_gettime.3.html. You can use the following example to time your host implementation.

```
// elapsed time in milliseconds
float cpu_time(timespec* start, timespec* end){
     return ((1e9*end->tv_sec + end->tv_nsec) - (1e9*start->tv_sec +
               start->tv_nsec))/1e6;
}

timespec ts, te;
clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
cpu_vec_add(…); // invoke the host function
clock_gettime(CLOCK_MONOTONIC_RAW, &te);

printf("\nCPU ellapsed time: %f\n", cpu_time(&ts, &te));
```

If you are using other programming languages (besides C) then consult with your instructor about the proper timing function to use.

Use the following code example to time the CUDA kernel implementation.

```
int main(){
   cudaEvent_t start, stop; //declare a start and stop event
   cudaEventCreate(&start); //create both events
   cudaEventCreate(&stop);
   …                        //copy data to the GPU and configure
   cudaEventRecord(start);  //insert the start event into the stream
```

```
   kernelCall<<<blocks, threads>>>(…);//run the kernel to be profiled
   cudaEventRecord(stop);  //insert the stop event into the stream
   …                       //add any other GPU activity to stream
   //all GPU events are in the stream, so stalls don't matter
   cudaEventSynchronize(stop); //wait for the stop event, if it isn't
done
   float milliseconds = 0;       //declare a variable to store runtime
   cudaEventElapsedTime(&milliseconds, start, stop); //get the elapsed
time
}
```

Error Checking

The following macro can be used to check for runtime errors.

```
//handle error macro
static void HandleError(cudaError_t err, const char *file,  int line ) {
        if (err != cudaSuccess) {
            printf("%s in %s at line %d\n", cudaGetErrorString(err),  file, line );
        }
}

#define HANDLE_ERROR(err) (HandleError( err, __FILE__, __LINE__ ))
```

You can then wrap each cuda API call with the above macro, for example:

```
            HANDLE_ERROR(cudaMalloc((void **)&d_a, n * sizeof(float)));
```

Questions

Answer the following questions:

1. How many floating operations are being performed in your vector add kernel? EXPLAIN.
2. How many global memory reads are being performed by your kernel? EXPLAIN.
3. How many global memory writes are being performed by your kernel? EXPLAIN.
4. Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.
5. Name three applications of vector addition.

Submission Instructions

You should upload a PDF of the answers to the above questions, plot of runtimes, the source code, and the binary executable to Canvas as part of a zipped folder named as your_msoeid-lab2.zip

Rubric

Code that crashes without producing output or code that doesn't compile are not considered to produce results and will be graded accordingly in the rubric.

CPU implementation **_____ / 30**

correct result (code compiles and executes)	_____ / 10
implementation / correct coding	_____ / 15
profiling/timing results	_____ / 5

GPU implementation **_____ / 50**

correct result (code compiles and executes)	_____ / 10
implementation / correct coding	_____ / 35
profiling / timing results	_____ / 5

Code Comments **_____ / 10**

Questions **_____ / 10**