```cpp
// header files
// standard headers
#include <stdio.h>
#include <math.h> // fabs()

// OpenCL headers
#include <CL/opencl.h>

#include "helper_timer.h"

// global variables
//const int iNumberOfArrayElements = 5;
const int iNumberOfArrayElements = 11444777;

cl_platform_id oclPlatformID;
cl_device_id oclDeviceID;

cl_context oclContext;
cl_command_queue oclCommandQueue;

cl_program oclProgram;
cl_kernel oclKernel;

float *hostInput1=NULL;
float *hostInput2=NULL;
float *hostOutput=NULL;
float* gold = NULL;

cl_mem deviceInput1=NULL;
cl_mem deviceInput2=NULL;
cl_mem deviceOutput=NULL;

float timeOnCPU = 0.0f;
float timeOnGPU = 0.0f;

// OpenCL kernel
const char *oclSourceCode =
"__kernel void vecAddGPU(__global float *in1, __global float *in2, __global float *out,int len)" \
"{" \
"int i=get_global_id(0);" \
"if(i < len)" \
"{" \
"out[i]=in1[i]+in2[i];" \
"}" \
"}";

// entry-point function
int main(void)
{
    // function declarations
    void fillFloatArrayWithRandomNumbers(float*, int);
    size_t roundGlobalSizeToNearestMultipleOfLocalSize(int, unsigned int);
    void vecAddCPU(const float*, const float*, float*, int);
    void cleanup(void);

    // variable declarations
    int size = iNumberOfArrayElements * sizeof(float);
    cl_int result;

    // code
    // host memory allocation
    hostInput1 = (float*)malloc(size);
    if (hostInput1 == NULL)
    {
        printf("Host Memory allocation is failed for hostInput1 array.\n");
        cleanup();
        exit(EXIT_FAILURE);
    }

    hostInput2 = (float*)malloc(size);
    if (hostInput2 == NULL)
    {
        printf("Host Memory allocation is failed for hostInput2 array.\n");
        cleanup();
        exit(EXIT_FAILURE);
    }

    hostOutput = (float*)malloc(size);
    if (hostOutput == NULL)
    {
        printf("Host Memory allocation is failed for hostOutput array.\n");
        cleanup();
        exit(EXIT_FAILURE);
    }

    gold = (float*)malloc(size);
    if (gold == NULL)
    {
        printf("Host Memory allocation is failed for gold array.\n");
        cleanup();
        exit(EXIT_FAILURE);
    }

    // filling values into host arrays
    fillFloatArrayWithRandomNumbers(hostInput1, iNumberOfArrayElements);
    fillFloatArrayWithRandomNumbers(hostInput2, iNumberOfArrayElements);

    // get OpenCL supporting platform's ID
    result = clGetPlatformIDs(1, &oclPlatformID, NULL);
    if (result != CL_SUCCESS)
    {
        printf("clGetPlatformIDs() Failed : %d\n", result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // get OpenCL supporting CPU device's ID
    result = clGetDeviceIDs(oclPlatformID, CL_DEVICE_TYPE_GPU, 1, &oclDeviceID, NULL);
    if (result != CL_SUCCESS)
    {
        printf("clGetDeviceIDs() Failed : %d\n", result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // create OpenCL compute context
    oclContext = clCreateContext(NULL, 1, &oclDeviceID, NULL, NULL, &result);
    if (result != CL_SUCCESS)
    {
        printf("clCreateContext() Failed : %d\n", result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // create command queue
    oclCommandQueue = clCreateCommandQueue(oclContext, oclDeviceID, 0, &result);
    if (result != CL_SUCCESS)
    {
        printf("clCreateCommandQueue() Failed : %d\n", result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // create OpenCL program from .cl
    oclProgram = clCreateProgramWithSource(oclContext, 1, (const char **)&oclSourceCode, NULL, &result);
    if (result != CL_SUCCESS)
    {
        printf("clCreateProgramWithSource() Failed : %d\n", result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // build OpenCL program
    result = clBuildProgram(oclProgram, 0, NULL, NULL, NULL, NULL);
    if (result != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];
        clGetProgramBuildInfo(oclProgram, oclDeviceID, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        printf("Program Build Log : %s\n", buffer);
        printf("clBuildProgram() Failed : %d\n", result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // create OpenCL kernel by passing kernel function name that we used in .cl file
    oclKernel = clCreateKernel(oclProgram, "vecAddGPU", &result);
    if (result != CL_SUCCESS)
    {
        printf("clCreateKernel() Failed : %d\n", result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // device memory allocation
    deviceInput1=clCreateBuffer
      (oclContext,CL_MEM_READ_ONLY,size,NULL,&result);
    if(result!=CL_SUCCESS)
    {
        printf("clCreateBuffer() Failed For 1st Input Array : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    deviceInput2=clCreateBuffer
      (oclContext,CL_MEM_READ_ONLY,size,NULL,&result);
    if(result!=CL_SUCCESS)
    {
        printf("clCreateBuffer() Failed For 2nd Input Array : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    deviceOutput=clCreateBuffer
      (oclContext,CL_MEM_WRITE_ONLY,size,NULL,&result);
    if(result!=CL_SUCCESS)
    {
        printf("clCreateBuffer() Failed For Output Array : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // set 0 based 0th argument i.e. deviceInput1
    result=clSetKernelArg(oclKernel,0,sizeof(cl_mem),(void *)&deviceInput1);
    if(result != CL_SUCCESS)
    {
        printf("clSetKernelArg() Failed For 1st Argument : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // set 0 based 1st argument i.e. deviceInput2
    result=clSetKernelArg(oclKernel,1,sizeof(cl_mem),(void *)&deviceInput2);
    if(result != CL_SUCCESS)
    {
        printf("clSetKernelArg() Failed For 2nd Argument : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // set 0 based 2nd argument i.e. deviceOutput
    result=clSetKernelArg(oclKernel,2,sizeof(cl_mem),(void *)&deviceOutput);
    if(result != CL_SUCCESS)
    {
        printf("clSetKernelArg() Failed For 3rd Argument : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // set 0 based 3rd argument i.e. len
    result=clSetKernelArg(oclKernel,3,sizeof(cl_int),(void *) &iNumberOfArrayElements);
    if(result != CL_SUCCESS)
    {
        printf("clSetKernelArg() Failed For 4th Argument : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // write abve 'input' device buffer to device memory
    result=clEnqueueWriteBuffer(oclCommandQueue,deviceInput1,CL_FALSE,0,size,hostInput1,0,NULL,NULL);
    if(result != CL_SUCCESS)
    {
        printf("clEnqueueWriteBuffer() Failed For 1st Input Device Buffer : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    result=clEnqueueWriteBuffer(oclCommandQueue,deviceInput2,CL_FALSE,0,size,hostInput2,0,NULL,NULL);
    if(result != CL_SUCCESS)
    {
        printf("clEnqueueWriteBuffer() Failed For 2nd Input Device Buffer : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // kernel configuration
  // size_t localWorkSize=5;
    size_t localWorkSize=256;
    size_t globalWorkSize;
    globalWorkSize=roundGlobalSizeToNearestMultipleOfLocalSize(localWorkSize, iNumberOfArrayElements);

    // start timer
    StopWatchInterface *timer = NULL;
    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);

    result=clEnqueueNDRangeKernel(oclCommandQueue,oclKernel,1,NULL,&globalWorkSize,&localWorkSize,0,NULL, NULL);
    if(result != CL_SUCCESS)
    {
        printf("clEnqueueNDRangeKernel() Failed : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // finish OpenCL command queue
    clFinish(oclCommandQueue);
```

```cpp

    // stop timer
    sdkStopTimer(&timer);
    timeOnGPU = sdkGetTimerValue(&timer);
    sdkDeleteTimer(&timer);

    // read back result from the device (i.e from deviceOutput) into cpu variable (i.e hostOutput)
    result=clEnqueueReadBuffer(oclCommandQueue,deviceOutput,CL_TRUE,0,size,hostOutput,0,NULL,NULL);
    if(result != CL_SUCCESS)
    {
        printf("clEnqueueReadBuffer() Failed : %d\n",result);
        cleanup();
        exit(EXIT_FAILURE);
    }

    // vector addition on host
    vecAddCPU(hostInput1, hostInput2, gold, iNumberOfArrayElements);

    // comparison
    const float epsilon = 0.000001f;
    int breakValue = -1;
    bool bAccuracy = true;
    for (int i = 0; i < iNumberOfArrayElements; i++)
    {
        float val1 = gold[i];
        float val2 = hostOutput[i];
        if (fabs(val1 - val2) > epsilon)
        {
            bAccuracy = false;
            breakValue = i;
            break;
        }
    }

    char str[128];
    if (bAccuracy == false)
        sprintf(str, "Comparison of CPU and GPU Vector Addition is not within accuracy of 0.000001 at array index %d", breakValue);
    else
        sprintf(str, "Comparison of CPU and GPU Vector Addition is within accuracy of 0.000001");

    // output
    printf("Array1 begins from 0th index %.6f to %dth index %.6f\n", hostInput1[0], iNumberOfArrayElements - 1, hostInput1[iNumberOfArrayElements - 1]);
    printf("Array2 begins from 0th index %.6f to %dth index %.6f\n", hostInput2[0], iNumberOfArrayElements - 1, hostInput2[iNumberOfArrayElements - 1]);
    printf("OpenCL Kernel Global Work Size = %lu and LOcal Work Size = %lu\n", globalWorkSize, localWorkSize);
    printf("Output Array begins from 0th index %.6f to %dth index %.6f\n", hostOutput[0], iNumberOfArrayElements - 1, hostOutput[iNumberOfArrayElements - 1]);
    printf("Time taken for Vector Addition on CPU = %.6f\n", timeOnCPU);
    printf("Time taken for Vector Addition on GPU = %.6f\n", timeOnGPU);
    printf("%s\n", str);

    // cleanup
    cleanup();

    return(0);
}

void fillFloatArrayWithRandomNumbers(float* arr, int len)
{
    // code
    const float fscale = 1.0f / (float)RAND_MAX;
    for (int i = 0; i < len; i++)
    {
        arr[i] = fscale * rand();
    }
}

void vecAddCPU(const float* arr1, const float* arr2, float *out, int len)
{
    // code
    StopWatchInterface* timer = NULL;
    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);

    for (int i = 0; i < len; i++)
    {
        out[i] = arr1[i] + arr2[i];
    }

    sdkStopTimer(&timer);
    timeOnCPU = sdkGetTimerValue(&timer);
    sdkDeleteTimer(&timer);
    timer = NULL;
}

size_t roundGlobalSizeToNearestMultipleOfLocalSize(int local_size, unsigned int global_size)
{
    // code
    unsigned int r = global_size % local_size;
    if(r == 0)
    {
        return(global_size);
    }
    else
    {
        return(global_size + local_size - r);
    }
}

void cleanup(void)
{
    // code
    if(deviceOutput)
    {
        clReleaseMemObject(deviceOutput);
        deviceOutput=NULL;
    }

    if(deviceInput2)
    {
        clReleaseMemObject(deviceInput2);
        deviceInput2=NULL;
    }

    if(deviceInput1)
    {
        clReleaseMemObject(deviceInput1);
        deviceInput1=NULL;
    }

    if(oclKernel)
    {
        clReleaseKernel(oclKernel);
        oclKernel=NULL;
    }

    if(oclProgram)
    {
        clReleaseProgram(oclProgram);
        oclProgram=NULL;
    }

    if(oclCommandQueue)
    {
        clReleaseCommandQueue(oclCommandQueue);
        oclCommandQueue=NULL;
    }

    if(oclContext)
    {
        clReleaseContext(oclContext);
        oclContext=NULL;
    }

    if(hostOutput)
    {
        free(hostOutput);
        hostOutput=NULL;
    }

    if(hostInput2)
    {
        free(hostInput2);
        hostInput2=NULL;
    }

    if(hostInput1)
    {
        free(hostInput1);
        hostInput1=NULL;
    }
}

```