Activity 1

1. Host

The host refers to the device that runs the OpenCL application, typically the Central Processing Unit (CPU) of a computer. It is responsible for dispatching computational tasks to other devices and managing memory.

2. Device

A device refers to the hardware component capable of performing computational tasks, such as a Graphics Processing Unit (GPU), Digital Signal Processor (DSP), or other accelerators. The host assigns computational tasks to the device for parallel processing.

3. Compute Unit

A compute unit is a parallel processing unit inside a device. Each compute unit consists of multiple processing elements that are capable of executing tasks in parallel.

4. Processing Elements

Processing elements are the basic execution units within a compute unit, responsible for executing instructions. Each processing element typically corresponds to a processing core.

5. Context

Context refers to the execution space within the OpenCL environment, containing devices, memory objects, program objects, and other resources needed to execute OpenCL programs. The context manages the interaction between the host and the devices.

6. Command Queue

A command queue is the mechanism used to send tasks to the device. The host uses the command queue to pass computational tasks to the device for execution. These tasks can include kernel execution, memory operations, etc.

7. Host program

The host program is the application running on the host device (usually the CPU). It is responsible for setting up the OpenCL context, creating the command queue, compiling kernels, and managing data exchanges with the devices.

8. Program kernel

A program kernel refers to the parallelized code in an OpenCL program that executes on the device. It defines the function to perform a specific task and is typically loaded and executed on the device by the host program.

Activity 2

1.

```
sit315-001@sit315-001-VirtualBox:~/opencl_project$ g++ vector_ops.cpp -lOpenCL -o vector_ops
sit315-001@sit315-001-VirtualBox:~/opencl_project$ ./vector_ops 8
83 86 77 15 93 35 86 92
---------------------------
GPU not found
6889 7396 5929 225 8649 1225 7396 8464
---------------------------
sit315-001@sit315-001-VirtualBox:~/opencl_project$
```

2.

```c
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

#define PRINT 1

int SZ = 8;
int *v;

// OpenCL device to execute the kernel on (e.g., CPU or GPU)
cl_mem bufV;

// OpenCL memory object for storing the vector data on the device
cl_device_id device_id;
// OpenCL memory object for storing the vector data on the device
cl_context context;
// OpenCL memory object for storing the vector data on the device
cl_program program;
// OpenCL memory object for storing the vector data on the device
cl_kernel kernel;
// OpenCL memory object for storing the vector data on the device
cl_command_queue queue;
cl_event event = NULL;

int err;

// Creates and returns an OpenCL device for kernel execution
cl_device_id create_device();
// Creates and returns an OpenCL device for kernel execution
void setup_openCL_device_context_queue_kernel(char *filename, char *kernelname);
// Creates and returns an OpenCL device for kernel execution
cl_program build_program(cl_context ctx, cl_device_id dev, const char *filename);
// Creates and returns an OpenCL device for kernel execution
void setup_kernel_memory();
// Creates and returns an OpenCL device for kernel execution
void copy_kernel_args();
// Creates and returns an OpenCL device for kernel execution
void free_memory();

void init(int *A, int size);
void print(int *A, int size);

int main(int argc, char **argv)
{
  if (argc > 1)
    SZ = atoi(argv[1]);

  init(v, SZ);

  // OpenCL memory object for storing the vector data on the device
  size_t global[1] = {(size_t)SZ};

  //initial vector
  printf("v, SZ);

  setup_openCL_device_context_queue_kernel((char *)"./vector_ops.cl", (char *)"square_magnitude");

  setup_kernel_memory();
  copy_kernel_args();

  // clEnqueueNDRangeKernel schedules the execution of a kernel on the OpenCL device.
  // It takes the command queue, kernel object, number of dimensions, global and local work sizes, and synchronization events as arguments.
  clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global, NULL, 0, NULL, &event);
  clWaitForEvents(1, &event);

  // clEnqueueReadBuffer reads the results from the device memory (bufV) back to the host (v).
  // Arguments: command queue, buffer object, blocking mode, offset, size of data, pointer to the host memory, event wait list, and event.
  clEnqueueReadBuffer(queue, bufV, CL_TRUE, 0, SZ * sizeof(int), &v[0], 0, NULL, NULL);

  //result vector
  print(v, SZ);

  //frees memory for device, kernel, queue, etc.
  //you will need to modify this to free your own buffers
  free_memory();
}

void init(int *A, int size)
{
  A = (int *)malloc(sizeof(int) * size);

  for (long i = 0; i < size; i++)
  {
    A[i] = rand() % 100; // any number less than 100
  }
}
void print(int *A, int size)
{
  if (PRINT == 0)
  {
    return;
  }

  if (PRINT == 1 && size > 15)
  {
    for (long i = 0; i < 5; i++)
    {                           //rows
      printf("%d ", A[i]); // print the cell value
    }
    printf(" .... ");
    for (long i = size - 5; i < size; i++)
    {                           //rows
      printf("%d ", A[i]); // print the cell value
    }
  }
  else
  {
    for (long i = 0; i < size; i++)
    {                           //rows
      printf("%d ", A[i]); // print the cell value
    }
  }
  printf("\n----------------------------\n");
}

void free_memory()
{
  //free the buffers
  clReleaseMemObject(bufV);

  //free opencl objects
  clReleaseKernel(kernel);
  clReleaseCommandQueue(queue);
  clReleaseProgram(program);
  clReleaseContext(context);

  free(v);
}


void copy_kernel_args()
{
  // clSetKernelArg sets the arguments for the kernel.
  // Arguments: kernel object, argument index, size of the argument, and pointer to the argument value.
  clSetKernelArg(kernel, 0, sizeof(int), (void *)&SZ);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&bufV);

  if (err < 0)
  {
    perror("Couldn't create a kernel argument");
    printf("error = %d", err);
    exit(1);
  }
}

void setup_kernel_memory()
{
  // clCreateBuffer creates a memory buffer object on the device.
  // Arguments: context, memory flags, size of buffer, host pointer (optional), and error code.
  // The second parameter, cl_mem_flags, defines the usage of the buffer (e.g., read-write or read-only).
  bufV = clCreateBuffer(context, CL_MEM_READ_WRITE, SZ * sizeof(int), NULL, NULL);

  // Copy matrices to the GPU
  clEnqueueWriteBuffer(queue, bufV, CL_TRUE, 0, SZ * sizeof(int), &v[0], 0, NULL, NULL);
}

void setup_openCL_device_context_queue_kernel(char *filename, char *kernelname)
{
  device_id = create_device();
  cl_int err;

  // clCreateContext creates an OpenCL context, which manages devices and resources.
  // Arguments: properties, number of devices, list of devices, error callback, user data, and error code.
  context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
  if (err < 0)
  {

    perror("Couldn't create a context");
    exit(1);
  }

  program = build_program(context, device_id, filename);

  // clCreateCommandQueueWithProperties creates a command queue for the device.
  // Arguments: context, device, queue properties, and error code.
  queue = clCreateCommandQueueWithProperties(context, device_id, 0, &err);
  if (err < 0)
  {
    perror("Couldn't create a command queue");
    exit(1);
  };


  kernel = clCreateKernel(program, kernelname, &err);
  if (err < 0)
  {
    perror("Couldn't create a kernel");
    printf("error =%d", err);
    exit(1);
  };
}

cl_program build_program(cl_context ctx, cl_device_id dev, const char *filename)
{

  cl_program program;
  FILE *program_handle;
  char *program_buffer, *program_log;
  size_t program_size, log_size;

  /* Read program file and place content into buffer */
  program_handle = fopen(filename, "r");
  if (program_handle == NULL)
  {
    perror("Couldn't find the program file");
    exit(1);
  }
  fseek(program_handle, 0, SEEK_END);
  program_size = ftell(program_handle);
  rewind(program_handle);
  program_buffer = (char *)malloc(program_size + 1);
  program_buffer[program_size] = '\0';
  fread(program_buffer, sizeof(char), program_size, program_handle);
  fclose(program_handle);

  // clCreateProgramWithSource creates an OpenCL program object from source code. It takes the context, number of strings, an array of source code strings, and their lengths as arguments.
  program = clCreateProgramWithSource(ctx, 1,
                                      (const char **)&program_buffer, &program_size, &err);
  if (err < 0)
  {
    perror("Couldn't create the program");
    exit(1);
  }
  free(program_buffer);

  /* Build program

  The fourth parameter accepts options that configure the compilation.
  These are similar to the flags used by gcc. For example, you can
  define a macro with the option -DMACRO=VALUE and turn off optimization
  with -cl-opt-disable.
  */
  err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  if (err < 0)
  {

    /* Find size of log and print to std output */
    clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG,
                          0, NULL, &log_size);
    program_log = (char *)malloc(log_size + 1);
    program_log[log_size] = '\0';
    clGetProgramBuildInfo(program, dev, CL_PROGRAM_BUILD_LOG,
                          log_size + 1, program_log, NULL);
    printf("%s\n", program_log);
    free(program_log);
    exit(1);
  }

  return program;
}

cl_device_id create_device() {

  cl_platform_id platform;
  cl_device_id dev;
  int err;

  /* Identify a platform */
  err = clGetPlatformIDs(1, &platform, NULL);
  if(err < 0) {
    perror("Couldn't identify a platform");
    exit(1);
  }

  // Access a device
  // GPU
  err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &dev, NULL);
  if(err == CL_DEVICE_NOT_FOUND) {
    // CPU
    printf("GPU not found\n");
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &dev, NULL);
  }
  if(err < 0) {
    perror("Couldn't access any devices");
    exit(1);
  }

  return dev;
}
```

```
// This is the OpenCL kernel that performs the operation of squaring each element of the vector.
// It will be executed on the OpenCL device (e.g., GPU) in parallel.
__kernel void square_magnitude(const int size, __global int* v) {

    // Each thread gets a unique index representing its position in the global space
    const int globalIndex = get_global_id(0);

    // Uncomment the line below to print the index for debugging purposes
    // printf("Kernel process index :(%d)\n ", globalIndex);

    // Perform the square operation for the element at this thread's global index
    v[globalIndex] = v[globalIndex] * v[globalIndex];
}

```

3.

```
sit315-001@sit315-001-VirtualBox:~/opencl_project$ g++ vector_ops.cpp -lOpenCL -o vector_ops
sit315-001@sit315-001-VirtualBox:~/opencl_project$ ./vector_ops
Vector A: 83 86 77 15 93 35 86 92
Vector B: 49 21 62 27 90 59 63 26
OpenCL result: 132 107 139 42 183 94 149 118
OpenCL execution time: 1.01305 s
Multithreaded CPU result: 132 107 139 42 183 94 149 118
Multithreaded CPU execution time: 0.000398865 s
sit315-001@sit315-001-VirtualBox:~/opencl_project$
```

In this task, I compared the performance of OpenCL parallel vector addition with the multi-threaded CPU version. The multi-threaded CPU version runs directly on the CPU without the overhead of data transfer, performing well with a reasonable number of threads. In contrast, the OpenCL version excels at large-scale parallel computation, especially when GPU support is available, where it demonstrates superior performance. However, in a virtual machine environment without a GPU, OpenCL falls back to CPU execution, resulting in similar performance to the multi-threaded version while introducing additional memory copy overhead.