EMBEDDED LINUX OPENCL INTRODUCTION REPORT

1 INTRODUCTION

This lab is an Introduction to OpenCL and GPU hardware acceleration. We learn the basics of OpenCL, how to set up the environment to send tasks to a GPU, and how to use the GPU to accelerate mathematical operations.

2 SETTING UP THE KERNEL ENVIRONMENT

OpenCL allows us to create programs meant to be run on a GPU. In the program we can define one or multiple kernels. A kernel is a subroutine that is meant to be run on multiple execution units in the GPU to exploit the SIMD (Single Instruction Multiple Data) architecture. Below is the kernel routine that we want to compile and send to the GPU.

```
__kernel void vector_add_kernel(    __global int* restrict inputA,
                                    __global int* restrict inputB,
                                    __global int* restrict output)
{
    /*
     * Set i to be the ID of the kernel instance.
     * If the global work size (set by clEnqueueNDRangeKernel) is n,
     * then n kernels will be run and i will be in the range [0, n - 1].
     */
    int i = get_global_id(0);
    /* Use i as an index into the three arrays. */
    output[i] = inputA[i] + inputB[i];
}

```

Figure 1- vector_add_kernel routine

This simple kernel takes two values in the A and B buffers, adds them, and stores the result in the C buffer. The variable "i" contains a unique identifier for the current unit of execution. We use this id to determine which item of the array this specific kernel needs to process.

To compile and send this kernel to the GPU we need a C++ program that sets up the OpenCL environment. Below is the program we wrote that shows the different steps to set up the environment.

```
int main(void) {
    /* VARIABLES DECLARATION */
    const string program_filename = "vector_add_opencl.cl";
    int i,j;
    const unsigned int VECTOR_SIZE=1024*1024;
    struct timeval start, end;
    double tdiff;
    //const int VECTOR_SIZE=256*256*256;
    void *va;
    void *vb;
    void *vc;
    int *A = new int[VECTOR_SIZE];
    int *B = new int[VECTOR_SIZE];
    int *C = new int[VECTOR_SIZE];

    for(i = 0; i < VECTOR_SIZE;i++){
        A[i] = i;
        B[i] = VECTOR_SIZE - i;
    }

    /* 1. SET UP OPENCL ENVIRONMENT: create context, command queue, program and kernel. */
    /* 1.a Create Context */
    /* Create an OpenCL context on a GPU on the first available platform. See createContext in common.h */
    cl_context context;
    createContext(&context);
    /* 1.b Create Command Queue */
    /* Create an OpenCL command queue for a given context. See create&commandQueue in common.h */
    cl_command_queue queue;
    cl_device_id device_id;
    createCommandQueue(context, &queue, &device_id);
    /* 1.c Create Program */
    /* Create an OpenCL program from a given file and compile it. See createProgram in common.h */
    cl_program program;
    createProgram(context, device_id, program_filename, &program);
    /* 1.d Create kernel */
    /* Create our OpenCL kernel for the kernel function. See clCreateKernel in OpenCL 1.2 Reference Pages */
    cl_int err;
    cl_kernel kernel;
    kernel = clCreateKernel(program, "vector_add_kernel", &err);
    if (err != CL_SUCCESS){
        cout << "Error : " << errorNumberToString(err) << endl;
        exit(0);
    }
    cout << "Kernel created without errors" << endl;
```

Figure 2- kernel environment setup

In this first part of the program we set up :

- The context : it is the platform on which we want to run the kernel. In our case it is an ARM Mali-T628 GPU
- The command queue : the commands that need to be sent to that specific GPU in order to run the kernel
- The program : the OpenCL program file that contains the code for our kernel. In our case "vector_add_opencl.cl"
- The kernel : the kernel in the program that needs to be compiled for the GPU. After this step we have a "cl_kernel" object that represents the compiled kernel code.

```
/* 2. SET UP MEMORY / DATA */
/* 2.a Create memory buffers */
/* Create 3 memory buffers for the input/output data. See clCreateBuffer in OpenCL 1.2 Reference Pages */
cl_mem cl_A, cl_B, cl_C;
cl_A = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int)*VECTOR_SIZE, NULL, &err);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
cl_B = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int)*VECTOR_SIZE, NULL, &err);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
cl_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int)*VECTOR_SIZE, NULL, &err);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}

cout << "Buffers created without errors" << endl;
/* 2.b Initialize the input data */
/* Map the input buffers to pointers. See clEnqueueMapBuffer in OpenCL 1.2 Reference Pages */
va = clEnqueueMapBuffer(queue, cl_A, CL_TRUE, CL_MAP_READ, 0, sizeof(int)*VECTOR_SIZE, 0, NULL,  NULL, &err);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
vb = clEnqueueMapBuffer(queue, cl_B, CL_TRUE, CL_MAP_READ, 0, sizeof(int)*VECTOR_SIZE, 0, NULL, NULL, &err);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}

cout << "Buffers mapped without errors" << endl;
/* Fill the input data */
```

Figure 3- Buffer creation and allocation

We now need to create and link the buffers in the GPU dedicated memory. Indeed, if the GPU need to checkout the data in the CPU memory for each operations, it would be very time consuming.

Here, using **clCreateBuffer** we allocate the memory in the GPU cache for each vectors (2 inputs and 1 output).

Then using the **clEnqueueMapBuffer**, we ensure we can access to the previously created buffer using the va and vb pointers. As we don't need to access to the output now, we don't map the output yet.

```
cout << "Buffers mapped without errors" << endl;
/* Fill the input data */
memcpy(va, A, sizeof(int)*VECTOR_SIZE);
memcpy(vb, B, sizeof(int)*VECTOR_SIZE);
cout << "A and B copied without errors" << endl;
/* Unmap the input data. See clEnqueueUnmapMemObject in OpenCL 1.2 Reference Pages */
err = clEnqueueUnmapMemObject(queue, cl_A, va, 0, NULL,  NULL);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
err = clEnqueueUnmapMemObject(queue, cl_B, vb, 0, NULL, NULL);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}

cout << "Buffers unmapped without errors" << endl;
```

Figure 4- data copy and unmapping

Now we only need to copy our A and B vector to the GPU buffer using **memcpy**, and unmap the buffer as now we won't need to access them.

```
/* 2.c Set the kernel arguments */
/* Pass the 3 memory buffers to the kernel as arguments. See clSetKernelArg in OpenCL 1.2 Reference Pages */
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &cl_A);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &cl_B);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &cl_C);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
cout << "Arguments set without errors" << endl;
```

Figure 5- kernel arguments

Now that the buffers are correctly assigned and initialized in the GPU memory, we can link them to the input parameters of our function. For example, using **clSetKernelArg(kernel, 0, sizeof(cl_mem), &cl_A**) sets the argument n°0 of **vector_add_kernel** as **cl_A**.

```
/* 3. EXECUTE THE KERNEL INSTANCES */
/* 3.a Define the global work size and enqueue the kernel. See clEnqueueNDRangeKernel in OpenCL 1.2 Reference Pages */
const size_t global_work_size = VECTOR_SIZE;
const size_t local_work_size = 1;

gettimeofday(&start, NULL);

err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_work_size, &local_work_size, 0, NULL, NULL);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
cout << "Kernel started without errors" << endl;
/* Each instance of our OpenCL kernel operates on a single element of each array so the number of instances needed is the number of
elements in the array. */

/* 3.b Wait for kernel execution completion */
/* See clFinish in OpenCL 1.2 Reference Pages */
err = clFinish(queue);
if (err != CL_SUCCESS){
    cout << "Error : " << errorNumberToString(err) << endl;
    exit(0);
}
cout << "Kernel execution finished without errors" << endl;
/* 4. AFTER EXECUTION */
```

Figure 6- screen error

Now we can start the kernel and wait for it to finish.

The **clEnqueueNDRRangeKernel** function will set the number of kernel instances and enqueue them. As we will run one instance for one vector value, we set the global_work_size at VECTOR_SIZE and the local_work_size at 1.

```
    /* 4.a Retrieve results: */
    /* Map the output buffer to a local pointer. See clEnqueueMapBuffer in OpenCL 1.2 Reference Pages */
     vc = clEnqueueMapBuffer(queue, cl_C, CL_TRUE, CL_MAP_WRITE, 0, VECTOR_SIZE, 0, NULL,  NULL,  &err);
    if (err != CL_SUCCESS){
        cout << "Error : " << errorNumberToString(err) << endl;
        exit(0);
    }
    /* Read the results using the mapped pointer */
    memcpy(C, vc, sizeof(int)*VECTOR_SIZE);

    for (i = VECTOR_SIZE-100; i < VECTOR_SIZE;i++){
        std::cout << A[i] << " + " << B[i] << " = " << C[i] << std::endl;
    }
    /* Unmap the output data. See clEnqueueUnmapMemObject in OpenCL 1.2 Reference Pages */
    err = clEnqueueUnmapMemObject(queue, cl_C, vc, 0, NULL, NULL);
    if (err != CL_SUCCESS){
        cout << "Error : " << errorNumberToString(err) << endl;
        exit(0);
    }
    /* 4.b Release OpenCL objects */
    /* See cleanUpOpenCL in common.h */
    cl_mem mem[] = {cl_A, cl_B, cl_C};
    cleanUpOpenCL( context, queue, program, kernel,mem, 3);

    tdiff = (double)(1000*(end.tv_sec-start.tv_sec))+((end.tv_usec-start.tv_usec)/1000);
    printf("ADDITION PROCESSING TIME: %f ms\n", tdiff);
    delete[] A;
    delete[] B;
    delete[] C;
    return 0;
}
```

Figure 7- end of program

The last step is to get the result from the GPU output buffer.

We map the buffer just as we did for the input, copy it to our local buffer, and unmap the buffer.

We can now clean our GPU using the **cleanUpOpenCL**, clean our CPU with the **delete**s, and close the application.

3 GPU VECTOR ADDITION VS. CPU ADDITION

Using the GPU vector addition, we observe at least a 6x improvement of execution time.

But the time measurements in the GPU application does not include the setup time, so depending of that it may not be interesting to use the GPU for small application.

```
1048574 + 2 = 1048576
1048575 + 1 = 1048576
ADDITION PROCESSING TIME: 23.000000 ms
odroid@odroid:~/LABS/TP_OPENCL$
```

Figure 8- CPU addition exec time

```
1048573 + 3 = 1048576
1048574 + 2 = 1048576
1048575 + 1 = 1048576
Profiling information:
Queued time:     0.182542ms
Wait time:       0.382071ms
Run time:        4.17025ms
odroid@odroid:~/LABS/TP_OPENCL$
```

Figure 9- GPU addition exec time

4 GPU ACCELERATION SCALING

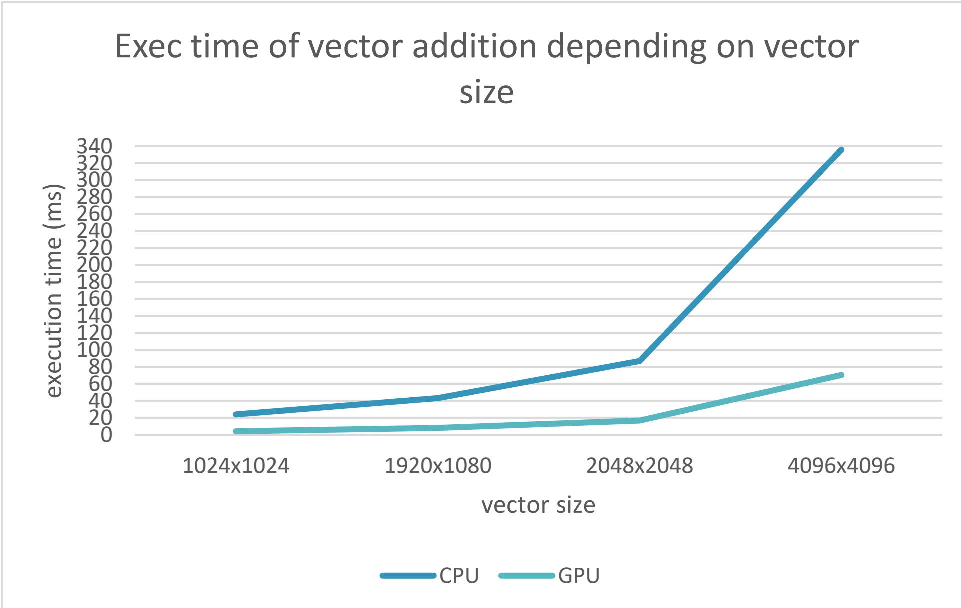

Figure 10- CPU vs GPU execution time

As shown on the graph, the GPU gives really good result compare to the CPU. As the vector size increase, the difference become more stark. But even the GPU follows the Amdahl law: as the vector size increase, the execution time increases too, and can't be reduced more than that.

The graph shows that if we need to do only one addition between two 2048x2048 pictures, the execution time may be too slow to be of use in a video stream (20ms/images = 30fps for one operation). To run some video games that needs more than 60fps and do more than just an addition between two images, we have to use better algorithms. For example, one that do an addition on only a part of the image.

5 CONCLUSION

We saw that using the GPU for small repeated operations can improve performances significantly. But as we don't have unlimited resources and one operation still need some time to be executed, we need to mind the actual resource consumption of our application and check if the execution time is still in tolerance range.