High Performance Computer Architectures Practical Course - Exercise 8 -

Tutorium 1

David Jordan (6260776)
Florian Rüffer (7454628)
Michael Samjatin (7485765)

June 13, 2023

1_First

Part 1

In this part we have to compile and run the code for main1.cpp. The checkErr function is meant to check the return value of OpenCL function calls and returns an error message if one does occur.

```
   checkErr(cl_int err, const char *name)
{
        // Error codes are indeed CL_SUCCESS == 0
        if (err != CL_SUCCESS)
        {
                std::cerr << "ERROR: " << name
                          << " (" << err << ")" << std
                             ::endl;
                exit(EXIT_FAILURE);
        }
}
```

We proceed to vary the sizes of the LIST_SIZE variable. Upon running the program we see that the program provides all permutations of adding together two integers to get the specified LIST_SIZE. To summarize the functioning of the program: After the program creates respective OpenCL context, command queue and memory buffer for each vector, it copies A and B into their respective memory buffers. It creates a program from the kernel source and then builds the program. It then creates the OpenCL kernel and executes it on the LIST:

```
 size_t global_item_size = LIST_SIZE; // Process the
    entire lists
        size_t local_item_size = 1;          //
           Process in groups of 1

        ret = clEnqueueNDRangeKernel(command_queue,
           kernel, 1, NULL,
                                   &global_item_size
                                      , &
                                      local_item_size
                                      , 0, NULL,
                                      NULL);
        checkErr(ret, "clEnqueueNDRangeKernel");
```

It stores the memory buffer C on the device to the local variable C and displays the result of the addition on the console.

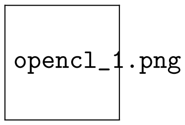

Figure 1: console output

Part 2

The following lines retrieve the build log information for the program compilation:

```
 char* build_log; // the pointer to a char
                  //array that will store the build log information
    size_t log_size;
    // First call to know the proper size
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);
    build_log = new char[log_size+1];
    // Second call to get the log
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, log_size, build_log, NULL);
    build_log[log_size] = '\0';
    cout << build_log << endl;
    delete[] build_log;
```

In order to have a comparison for the runtimes of the OpenCL Code, a normal scalar version of addition in C++ is implemented:

```
typedef float DataType;

void scalar_add( DataType A,  DataType B,  DataType C) {

      // Do the operation
    C = A + B;
}
```

This function is then later called together with a timing function. Upon executing the program with the simple addition, the scalar version turns out to be almost two orders of magnitude faster in the execution of this simple operation:

scalar_add.png

Figure 2: scalar addition is much faster

This can only be because of significant overhead in OpenCL compared to the standard C++ version.

Only when the complexity of the calculations is increased using expensive operations like sqrt and log() can the advantage of the OpenCL parallelisation come through. A sufficiently complicated calculation yields a threefold improvement of the Scalar time:

logs_andsqrt.png

Figure 3: for complex logs and sqrt times, parallel is much faster

This can only be because these more complicated operations offer the opportunity to be parallellized by OpenCL, that is parts of these operations, where there are no data dependencies are executed simultaneously.

Part 3

In this part we need to SIMDize the program.
We start off by creating the vector_add_kernel2.cl file, which contains code to utilize SIMD capabilities.

```
    __kernel void vector_add(__global float *A,
       __global float *B, __global float *C) {


    int i = get_global_id(0);


    float4 a = vload4(i, A);
    float4 b = vload4(i, B);


    vstore4( a + b, i, C );
}
```

File 1: vector_add_kernel2.cl

The first line defines a kernel function, which can be executed on an OpenCL device.
The function takes three parameters (A, B, C), which can be located in the global memory space. Furthermore line 4 gets the id is of the current work item and assigns the i-th group of A, B and C to it. In this case A, B and C are vectors with 4 elements each (line 7,8).
Finally line 11 stores the computed values into the $4 \cdot i$ element.

Subsequently we will add / load this kernel to our code in the main.cpp file.

```
#ifdef SIMD
    fp = fopen("vector_add_kernel2.cl", "r");
#else
    fp = fopen("vector_add_kernel.cl", "r");
#endif
    if (!fp) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
```

File 2: main.cpp

This part of the code add the new kernel to the main.cpp code base. If it is available it will be used, if not another kernel (w/o SIMD) will be used. If both of the cases above fail, we raise an error.

We also should not forget to add the preprocessing macro at the beginning of our code. This allows us to use SIMD in commands, such as #ifdef, to switch between scalar or simd execution during compile time.

```
    #define SIMD
```

File 3: main.cpp

Adjust our list size, to be divisible by 4.

```
    const int LIST_SIZE = 1024;
```

File 4: main.cpp

Last but not least, we need to compute the item size. We do this the same way, in which we have loaded the kernel.

```
    #ifdef SIMD
    size_t global_item_size = LIST_SIZE/4; // Process
       the entire lists
#else
    size_t global_item_size = LIST_SIZE;
#endif
    size_t local_item_size = 64; // Process in groups
       of 64
```

File 5: main.cpp

If we have defined SIMD, we will determine the number of work items we need. For our list size this would result in 256 work items. If we SIMD is not defined our item size will equal the list size. Lastly if both of the cases above fail to trigger, we will process in groups of 64.

Part 4

In this part sub devices are added.
Starting off by gathering device information:

```
1     cl_platform_id platform_id = NULL;
2     cl_device_id device_id = NULL;
3     cl_uint ret_num_devices;
4     cl_uint ret_num_platforms;
5     cl_int ret = clGetPlatformIDs(1, &platform_id, &
         ret_num_platforms);
6     checkErr(ret, "clGetPlatformIDs");
```

The first two lines initialize as NULL and will be used later on to refer the platform and device. The next two lines hold the number of platforms and devices. Line 5 denotes an error code (cl_int), which tells if the platform id was fetched successfully. Here only one platform is returned and written to platform_id, due to the first parameter. Also an error check is performed.

Here we perform the same task as described above, but for our device. CL_DEVICE_TYPE_CPU restricts us to devices of a certain type.

```
    ret = clGetDeviceIDs( platform_id,
       CL_DEVICE_TYPE_CPU, 1,
    &device_id, &ret_num_devices);
    checkErr(ret, "clGetDeviceIDs");
```

These two lines denote variables to hold the number and id of the created subdevices.

```
    cl_uint num_devices_ret;
    cl_device_id  out_devices[80];
```

This part denotes the properties based on which the device will be partitioned into subdevices. An equal partion means dividing it into as many parts as possible, while each part contains the same number of compute units (in this case 2). Subsequently the devices are created based on the defined properties. At the end again an error check is performed.

```
    const cl_device_partition_property props[] = {
       CL_DEVICE_PARTITION_EQUALLY, 2, 0};
    ret = clCreateSubDevices ( device_id, props, 80 ,
       out_devices , &num_devices_ret );
    checkErr(ret, "clCreateSubDevices");
```

The other partition methods, include partitioning by counts and affinity domain. When we partition by count, we can specify the exact number of compute units each subdevice should have. This allows a more find-grained control.

```
    const cl_device_partition_property props[] = {
        CL_DEVICE_PARTITION_BY_COUNTS, 1, 1,
        CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0};
```

Additionally there is the partitioning method by affinity domain. Those affinity domains are specific areas of the device (e.g. L1, L2 Cache). The subdevices will then be created and have an affinity to a certain affinity domain. The code below will result in an affinity (e.g. faster access) to the L1 Cache.

```
    const cl_device_partition_property props[] = {
        CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN,
        CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE, 0};
```

Part 5

In this exercise, we will focus on implementing a SIMD version. The SIMD (Single Instruction, Multiple Data) approach allows us to perform parallel computations on multiple data elements simultaneously, which can greatly enhance the performance of our program.

To begin, we need to load the corresponding kernel file. This file contains the code that will be executed on the GPU:

```
// Part of the source code of Solution/main5.cpp
fp = fopen("vector_add_kernel4.cl", "r");
```

Next, we need to build the kernel using the C++ option. Building the kernel involves compiling the GPU code so that it can be executed on the available OpenCL devices:

```
// Part of the source code of Solution/main5.cpp
ret = clBuildProgram(program, 1, &out_devices[0], "-x
    clc++", NULL, NULL);
```

Now, let's take a closer look at the kernel code. We define a function called `Add` that takes two `float4` vectors, `a` and `b`, and calculates their sum, storing it in the `sum` vector:

```
// Part of the source code of Solution/
    vector_add_kernel4.cl
void Add(float4 &a, float4 &b, float4 &sum) {
     sum = a + b;
}
```

The main kernel function, `vector_add`, performs the vector addition operation. It takes three global arrays, `A`, `B`, and `C`, as input parameters:

```
// Part of the source code of Solution/
   vector_add_kernel4.cl
__kernel void vector_add(__global float *A, __global
   float *B, __global float *C) {
    // Get the index of the current element
    int i = get_global_id(0);

    // Get the i-th group of 4
    float4 a = vload4(i, A);
    float4 b = vload4(i, B);
    float4 sum;
    Add(a, b, sum);

    // Store the sum of a and b to the 4*i-th element
    vstore4(sum, i, C);
}
```

In the `vector_add` function, we first obtain the index of the current element using the `get_global_id` function. This index will be used to access the corresponding elements in the input arrays.

Then, we load a group of four elements from arrays `A` and `B` using the `vload4` function, which loads the elements into `float4` vectors `a` and `b`. These vectors represent the data elements that will be processed in parallel.

Next, we create a `float4` vector called `sum`, and pass `a`, `b`, and `sum` to the `Add` function to perform the vector addition.

Finally, we store the sum of `a` and `b` into the output array `C` at the 4 times `i`-th element using the `vstore4` function.

By utilizing SIMD and parallel processing, we can achieve significant speedups in our vector addition computation.

Part 6

In this exercise, the objective is to perform vector addition on the GPU using OpenCL. The code provided includes two files: `main6.cpp` and `vector_add_kernel.cl` (or `vector_add_kernel2.cl` if SIMD is defined).

To run the code, compile `main6.cpp` with the necessary OpenCL and Vc libraries (if SIMD is enabled). Then execute the resulting binary. The program will measure the execution time of the parallel and sequential versions and compare their results for correctness.

The speed increase is consistent with preivous exercises, being multiple times faster, and even more this time.