4.1. OpenCL

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. It allows to write the universal code, which can be run both on the CPU and GPU giving software developers portable and efficient access to the power of the heterogeneous processing platforms. OpenCL supports a wide range of applications through a low-level, high-performance, portable abstraction. OpenCL consists of an API for coordinating parallel computation across heterogeneous processors and a cross-platform programming language with a well-specified computation environment. The OpenCL standard:

- supports both data- and task-based parallel programming models;
- utilises a subset of ISO C99 with extensions for parallelism;
- defines consistent numerical requirements based on IEEE 754;
- defines a configuration profile for handheld and embedded devices Efficiently interoperates with OpenGL, OpenGL ES and other graphics APIs.

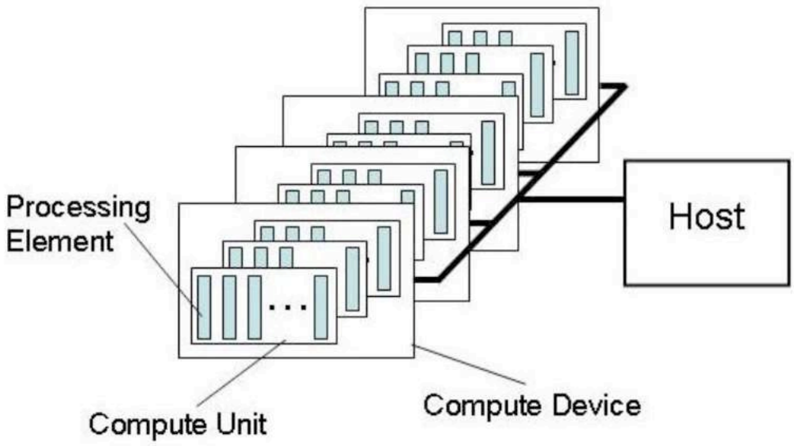

Fig. 1. Programming model of OpenCL.

OpenCL includes runtime API, which compiles kernels, Manage scheduling, compute, and memory resources and executes kernels.

The programming model of OpenCL (see Fig. 1) consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units (for example, one CPU of the server or one streaming multiprocessor of the GPU), which are further divided into one or more processing elements (cores of the CPU or streaming multiprocessor). The OpenCL application submits commands from the host to execute computations on the processing elements within a device. An OpenCL application runs on a host according to the models native to the host platform. The processing elements within a compute unit execute a single stream of instructions as SIMD units or each processing element maintains its own program counter.

The high abstraction level requires to specify the memory model. The memory in OpenCL is divided into several layers (Fig. 2). The host and device memory are separated. The largest memory available to the device is called a **global memory**. For CPU, for example, the global memory is RAM of the server, for GPU - RAM of the GPU. Usually, global memory is the slowest one. Some part of the global memory is considered as a **constant memory**. It is usually faster then global, because of caching. Each compute unit on the device has a **local memory**, which is typically on the processor die, and therefore has much higher bandwidth and lower latency than global memory. Local memory can be read and written by any work-item in a work-group, and thus allows for local communication between. Additionally, attached to each processing element is a **private memory**, which is typically not used directly by programmers, but is used to hold data for each work-item that does not fit in the processing element's registers. Usually the private memory physically is a part of the global, therefore it is also slow.

Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the kernels and manages their execution. Also it defines a queue of the tasks and runs corresponding kernels according to the queue. The data is divided into work groups (Fig. 3), which are assigned to the compute

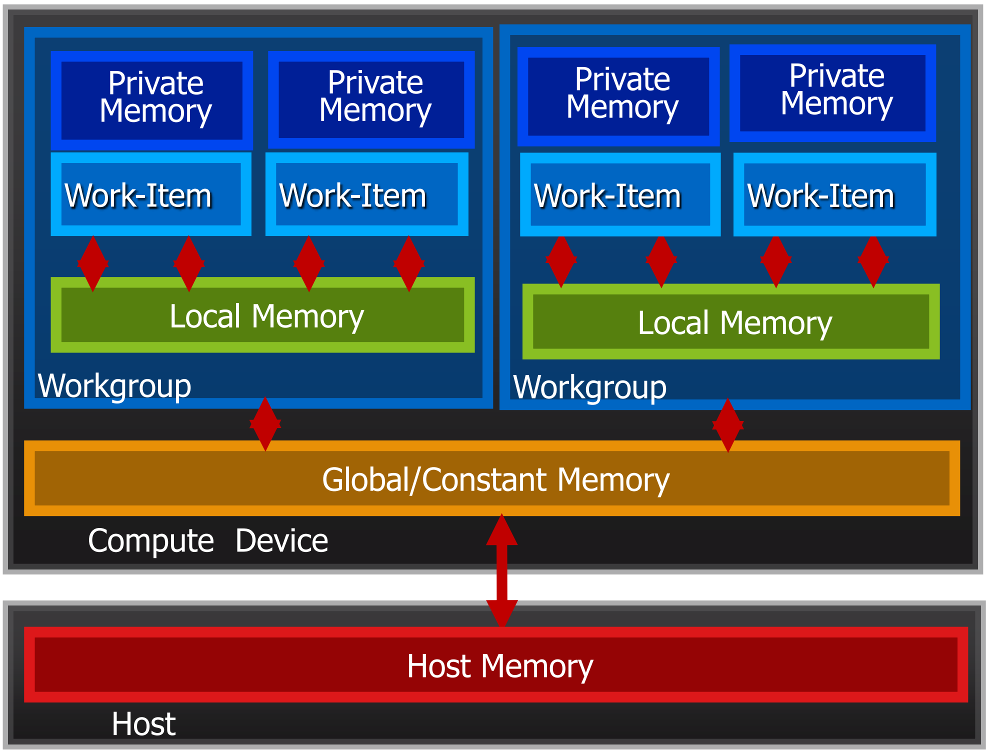

Fig. 2. The memory model of OpenCL.

unit. Each work group consist of work items. Work item is a basic unit of work, it runs the instance of the kernel on the individual processing element.

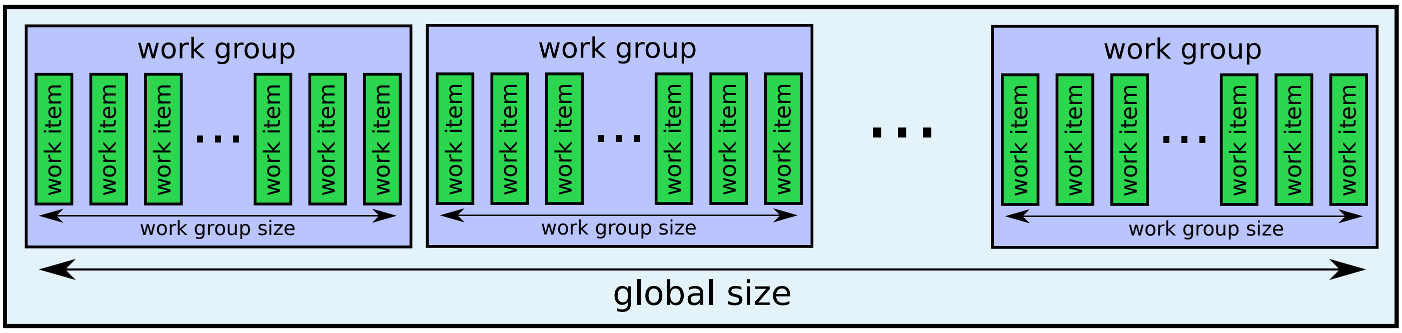

Fig. 3. Organisation of the data in OpenCL.

The structure of the host program:

- get a platform - the information about the whole;
- get a device - select the device for computations;
- set a context within which the program will work;
- create a command-queue;
- create a memory buffer;
- write the buffer (fill the buffer with the input data);
- create a program - an OpenCL object, the input for it - a *.cl file with the main kernel function;
- compile the program;
- create a kernel;
- set the kernel arguments;
- call the kernel;
- read the buffer;
- clean the memory.

Let us describe the functionality of OpenCL used for this. More detailed description can be found here: http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/.

1) cl_int clGetPlatformIDs(cl_uint num_entries,
cl_platform_id *platforms,
cl_uint *num_platforms)

Obtain the list of platforms available. num_entries - the maximum number of elements in platforms array, platforms - returned array of platform ids, num_platforms - returns number of available platforms. The function returns the error code.

2) cl_int clGetDeviceIDs (cl_platform_id platform ,
cl_device_type device_type ,
cl_uint num_entries ,
cl_device_id *devices ,
cl_uint *num_devices)

Obtain the list of devices available on a platform. platform - id of the platform, where we are looking for a device, device_type - type of a device (CL_DEVICE_TYPE_CPU to use CPU, CL_DEVICE_TYPE_GPU to use GPU or CL_DEVICE_TYPE_ALL to use both of them), num_entries - the maximum number of elements in devices array, devices - returned array of devices ids, num_devices - returns number of OpenCL devices available that match device_type. The function returns the error code.

3) cl_context clCreateContext(const cl_context_properties *properties,
cl_uint num_devices,
const cl_device_id *devices,
(void CL_CALLBACK *pfn_notify) (const char *errinfo,
const void *private_info, size_t cb,
void *user_data),
void *user_data,
cl_int *errcode_ret)

Creates an OpenCL context. An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context. properties - specifies a list of context property names and their corresponding values, num_devices - the number of devices specified in the devices argument; devices - array with device ids, which will be used within current context; errcode_ret - the error code returned by the function.

4) cl_command_queue clCreateCommandQueue(cl_context context,
cl_device_id device,
cl_command_queue_properties properties,
cl_int *errcode_ret)

Create a command-queue on a specific device. context - valid OpenCL context created before; device - id of a device from the context, for which the queue is created; properties - properties of the queue, if profiling should be enabled the value should be CL_QUEUE_PROFILING_ENABLE, if the execution mode should be out of order the value should be CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE; errcode_ret - the error code returned by the function.

5) cl_mem clCreateBuffer (cl_context context,
cl_mem_flags flags,
size_t size,
void *host_ptr,
cl_int *errcode_ret)

Creates a buffer object. context - valid OpenCL context created before; flags - specify allocation and usage information such as the memory arena that should be used to allocate the buffer object and how it will be used, value can be, for example, CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY; size - the size in bytes of the buffer memory object to be allocated; host_ptr - a pointer to the buffer data that may already be allocated by the application, the size of the buffer that host_ptr points to must be ≥ size bytes; errcode_ret - the error code returned by the function.

6) cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
cl_mem buffer,
cl_bool blocking_write,
size_t offset,
size_t size,
const void *ptr,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)

Enqueue commands to write to a buffer object from host memory. command_queue - refers to the command-queue in which the write command will be queued, command_queue and buffer must be created with the same OpenCL context; buffer - refers to a valid buffer object, offset - the offset in bytes in the buffer object to write to, size - the size in bytes of data being written; ptr - the pointer to buffer in host memory where data is to be written from; event_wait_list, num_events_in_wait_list - array of events together with its size to be waited before execution of the current function; event - returns an event object that identifies this particular write command and can be used to query or queue a wait for this particular command to complete. The function returns the error code.

7) cl_program clCreateProgramWithSource (cl_context context,
cl_uint count,
const char **strings,
const size_t *lengths,
cl_int *errcode_ret)

Creates a program object for a context, and loads the source code specified by the text strings in the strings array into the program object. context - valid OpenCL context created before; count - number of files (strings) to be compiled; strings - array of strings containing the source code; lengths - array with sizes of each string; errcode_ret - the error code returned by the function.

8) cl_int clBuildProgram (cl_program program,
cl_uint num_devices,
const cl_device_id *device_list,
const char *options,
void (CL_CALLBACK *pfn_notify)(cl_program program, void
*user_data),
void *user_data)

Builds (compiles and links) a program executable from the program source or binary. program - the program object; device_list - a pointer to a list of devices associated with the program; num_devices - the number of devices listed in device_list; options - compilation options, to build the program supporting c++-like functionality should be "-x clc++". The program returns the error code.

9) cl_int clGetProgramBuildInfo (cl_program program,
cl_device_id device,
cl_program_build_info param_name,
size_t param_value_size,
void *param_value,
size_t *param_value_size_ret)

Returns build information for each device in the program object. We will use it to print the build log. program - the program object being queried; device - specifies the device for which build information is being queried; param_name - specifies the information to query, in our case should be CL_PROGRAM_BUILD_LOG; param_value_size - the size in bytes of memory pointed to by param_value; param_value - a pointer to memory where the appropriate result being queried is returned; param_value_size_ret - returns the actual size of the log. The function returns the error code.

10) cl_kernel clCreateKernel (cl_program program,
const char *kernel_name,
cl_int *errcode_ret)

Creates a kernel object. program - a program object with a successfully built executable, kernel_name - a function name in the program declared with the __kernel qualifier; errcode_ret - the error code returned by the function.

11) cl_int clSetKernelArg (cl_kernel kernel,
cl_uint arg_index,
size_t arg_size,
const void *arg_value)

Used to set the argument value for a specific argument of a kernel. kernel - a valid kernel object; arg_index - the argument index, starts from 0; arg_size - specifies the size of the argument value; arg_value - a pointer to data that should be used as the argument value for argument specified by arg_index. The function returns the error code.

12) cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
cl_kernel kernel,
cl_uint work_dim,
const size_t *global_work_offset,
const size_t *global_work_size,
const size_t *local_work_size,

cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)

Enqueues a command to execute a kernel on a device. command_queue - a valid command-queue, the kernel will be queued for execution on the device associated with command_queue; kernel - a valid kernel object; work_dim - the number of dimensions used to specify the global work-items and work-items in the work-group; global_work_offset - can be used to specify an array of work_dim unsigned values that describe the offset used to calculate the global ID of a work-item; global_work_size - points to an array of work_dim unsigned values that describe the number of global work-items in work_dim dimensions that will execute the kernel function; local_work_size - points to an array of work_dim unsigned values that describe the number of work-items that make up a work-group (also referred to as the size of the work-group) that will execute the kernel specified by kernel; event_wait_list, num_events_in_wait_list - array of events together with its size to be waited before execution of the current function; event - returns an event object that identifies this particular write command and can be used to query or queue a wait for this particular command to complete.

13) cl_int clGetEventProfilingInfo (cl_event event,
cl_profiling_info param_name,
size_t param_value_size,
void *param_value,
size_t *param_value_size_ret)

Returns profiling information for the command associated with event if profiling is enabled. event - event to be profiled; param_name - specifies the profiling data to query (CL_PROFILING_COMMAND_QUEUED, CL_PROFILING_COMMAND_SUBMIT, CL_PROFILING_COMMAND_START, CL_PROFILING_COMMAND_END); param_value_size - specifies the size in bytes of memory pointed to by param_value; param_value - A pointer to memory where the appropriate result being queried is returned; param_value_size_ret - returns the actual size in bytes of data copied to param_value. The function returns the error code.

14) cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
cl_mem buffer,
cl_bool blocking_read,
size_t offset,
size_t size,
void *ptr,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)

Enqueue commands to read from a buffer object to the host memory. Parameters has the same description as for clEnqueueWriteBuffer().

15) cl_int clCreateSubDevices (cl_device_id in_device ,
const cl_device_partition_property *properties ,
cl_uint num_devices ,
cl_device_id *out_devices ,
cl_uint *num_devices_ret)

Creates an array of sub-devices that each reference a non-intersecting set of compute units within in_device. in_device - the device to be partitioned; properties - specifies how in_device should be partitioned described by a partition name and its corresponding value (CL_DEVICE_PARTITION_EQUALLY, CL_DEVICE_PARTITION_BY_COUNTS, CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN); num_devices - size of memory pointed to by out_devices specified as the number of cl_device_id entries; out_devices - the buffer where the OpenCL sub-devices will be returned; num_devices_ret - returns the number of sub-devices that device may be partitioned into according to the partitioning scheme specified in properties.

7_OpenCL/1_First: description

The first exercise is a simple program, which computes vector sum C=A+B. It consist of two parts: the host program (**main.cpp**) and the OpenCL kernel (**vector_add_kernel.cl**). The tasks for this exercise are:

Part 1:

- run and understand the code;
- check error codes, returned by each function (they should be equal to CL_SUCCESS==0);

- play: try to change size of the arrays (try 128, 64, 16, 1023), type (try float), etc.;
- solution is **main1.cpp** and **vector_add_kernel.cl**.

Part 2:
- display build log;
- measure the execution time
    1. increase the size of the array to 1000000, increase the **local_item_size**;
    2. because of the increased time comment the printing of the result on the screen;
    3. for comparison implement scalar version;
    4. try more complicated computations (log, sqrt);
- solution is **main2.cpp** and **vector_add_kernel.cl**.

Part 3:
- SIMDize;
- Solution is **main3.cpp** and **vector_add_kernel.cl** and **vector_add_kernel2.cl**.

Part 4:
- create sub devices;
- try CL_DEVICE_PARTITION_EQUALLY, CL_DEVICE_PARTITION_BY_COUNTS and CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN properties (more information you can find here: http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateSubDevices.html);
- solution is **main4.cpp** and **vector_add_kernel.cl** and **vector_add_kernel2.cl**.

Part 5:
- create a function into the kernel function for a sum calculation;
- we suggest to build the program in c++-like style: clBuildProgram(program, 1, &out_devices[0], "-x clc++", NULL, NULL);
- solution is **main5.cpp** and **vector_add_kernel.cl4**.

Part 6:
- run on GPU;
- try SIMD and scalar versions;
- try different sizes of working groups;
- solution is **main6.cpp** and **vector_add_kernel.cl** and **vector_add_kernel2.cl**.

7_OpenCL/1_First: solution

Part 1.
To check the error codes we suggest to create a function:

Part of the source code of Solution/main1.cpp

```cpp
inline void
checkErr(cl_int err, const char * name)
{
    if (err != CL_SUCCESS) {
        std::cerr << "ERROR: " << name
                  << " (" << err << ")" << std::endl;
        exit(EXIT_FAILURE);
    }
}
```

and call it after each OpenCL function, for example, after getting the platform:

Part of the source code of Solution/main1.cpp

```cpp
checkErr(ret, "clGetPlatformIDs");
```

To change the type throughout the whole code easily we introduced a type **DataType**:

Part of the source code of Solution/main1.cpp

```cpp
typedef float DataType;
```

It can be set to int or float. The code of the program should be modified respectively:

Part of the source code of Solution/main1.cpp

```
    DataType *A = (DataType*)malloc(sizeof(DataType)*LIST_SIZE);
    DataType *B = (DataType*)malloc(sizeof(DataType)*LIST_SIZE);
…
    cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            LIST_SIZE * sizeof(DataType), NULL, &ret);
    cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            LIST_SIZE * sizeof(DataType), NULL, &ret);
    cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            LIST_SIZE * sizeof(DataType), NULL, &ret);
…
    ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(DataType), A, 0, NULL, NULL);
    checkErr(ret, "clEnqueueWriteBuffer");
    ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(DataType), B, 0, NULL, NULL);
…
    DataType *C = (DataType*)malloc(sizeof(DataType)*LIST_SIZE);
    ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0,
            LIST_SIZE * sizeof(DataType), C, 0, NULL, NULL);
```

and the OpenCL kernel:

Part of the source code of Solution/main1.cpp

```
__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);

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

When changing the size of the arrays to 128 and 64 the program works fine. When changing it to 16 or 1023 - the OpenCL program will not be compiled, because the size of the array should be dividable on **local_item_size**. Setting **local_item_size**, for example, to one will solve the problem.

Part 2.

To extract the build log next lines should be added:

Part of the source code of Solution/main2.cpp

```
    // Shows the log
    char* build_log;
    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;
```

If we will underestimate the size, the log will be incomplete, if we will overestimate it - the log will be displayed together with a large empty region. Therefore the first time clGetProgramBuildInfo is called to calculate the size of the log. And the second time we get the log itself to the variable **build_log**.

To compare times of the OpenCL code and a normal c++ code the scalar function should be implemented:

Part of the source code of Solution/main2.cpp

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

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

and called together with the time measurement:

Part of the source code of Solution/main2.cpp

```
   TStopwatch timer;
   for(i = 0; i < LIST_SIZE; i++)
      scalar_add( A[i], B[i], C[i] );
   timer.Stop();
```

To improve the precision of the time measurement the size of the array should be increased:

Part of the source code of Solution/main2.cpp

```
   const int LIST_SIZE = 1000000;
```

To enable profiling of the OpenCL code the queue parameters should be modified:

Part of the source code of Solution/main2.cpp

```
     cl_command_queue command_queue = clCreateCommandQueue(context, device_id,
CL_QUEUE_PROFILING_ENABLE, &ret);
```

the event should be generated on the kernel execution and we should wait for this event:

Part of the source code of Solution/main2.cpp

```
   cl_event event;

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

   ret = clWaitForEvents(1 , &event);
```

And when the event is finished we can profile it and print the time:

Part of the source code of Solution/main2.cpp

```
   cl_ulong time_start, time_end;
   double total_time;
   clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
sizeof(time_start), &time_start, NULL);
   clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end),
&time_end, NULL);
   total_time = time_end - time_start;
   printf("Parallel time = %0.3f ms\n", (total_time / 1000000.0) );
```

To improve the performance and decrease overhead the **local_item_size** should be increased. But even after this the OpenCL code is mush slower. With more complicated calculations this will change.

Part 3.

To have the SIMD and scalar code in the same file we introduced a preprocessor macro:

Part of the source code of Solution/main3.cpp
```
#define SIMD // switch between vectorized and not vectorized versions
```

Here we again decrease the size of the array:

Part of the source code of Solution/main3.cpp
```
    const int LIST_SIZE = 1024;
```

Also we need to introduce a new kernel (**vector_add_kernel2.cl**):

Part of the source code of Solution/vector_add_kernel2.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);

    // store a+b to 4*i-th element
    vstore4( a + b, i, C );
}
```

And we need to load corresponding kernel:

Part of the source code of Solution/main3.cpp
```
#ifdef SIMD
    fp = fopen("vector_add_kernel2.cl", "r");
#else
    fp = fopen("vector_add_kernel.cl", "r");
#endif
```

and modify **global_item_size**:

Part of the source code of Solution/main3.cpp
```
#ifdef SIMD
    size_t global_item_size = LIST_SIZE/4; // Process the entire lists
#else
    size_t global_item_size = LIST_SIZE;
#endif
```

Part 4.

To create subdevices the code should be added:

Part of the source code of Solution/main4.cpp
```
    cl_uint num_devices_ret;
    cl_device_id  out_devices[80];
    /// CL_DEVICE_PARTITION_EQUALLY
```

```
Part of the source code of Solution/main4.cpp
    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");
    /// CL_DEVICE_PARTITION_BY_COUNTS
//      const cl_device_partition_property props[] = {CL_DEVICE_PARTITION_BY_COUNTS, 1, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0};
//      ret = clCreateSubDevices ( device_id, props, 80 , out_devices , &num_devices_ret );
//      checkErr(ret, "clCreateSubDevices");
    ///CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN
//      const cl_device_partition_property props[] = {CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE, 0};
//      ret = clCreateSubDevices ( device_id, props, 80 , out_devices , &num_devices_ret );
//      checkErr(ret, "clCreateSubDevices");
```

and we will use only the first device:

```
Part of the source code of Solution/main4.cpp
        cl_command_queue command_queue = clCreateCommandQueue(context, out_devices[0], CL_QUEUE_PROFILING_ENABLE, &ret);
    cl_program program = clCreateProgramWithSource(context, 1,
            (const char **)&source_str, (const size_t *)&source_size, &ret);
        clGetProgramBuildInfo(program, out_devices[0], CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);

    clGetProgramBuildInfo(program, out_devices[0], CL_PROGRAM_BUILD_LOG, log_size, build_log, NULL);
```

Part 5.

In this exercise we will work only with a SIMD version. We should load the corresponding file with a kernel:

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

should build the kernel with c++-option:

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

And the function should be added to the kernel:

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

__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);
```

Part of the source code of Solution/vector_add_kernel4.cl

```

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

    float4 sum;
    Add(a,b,sum);

    // store a+b to 4*i-th element
    vstore4( sum, i, C );
}
```