

Úvod do OpenCL

Michal Kula

Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 2, 612 66 Brno
www.fit.vutbr.cz/~ikula



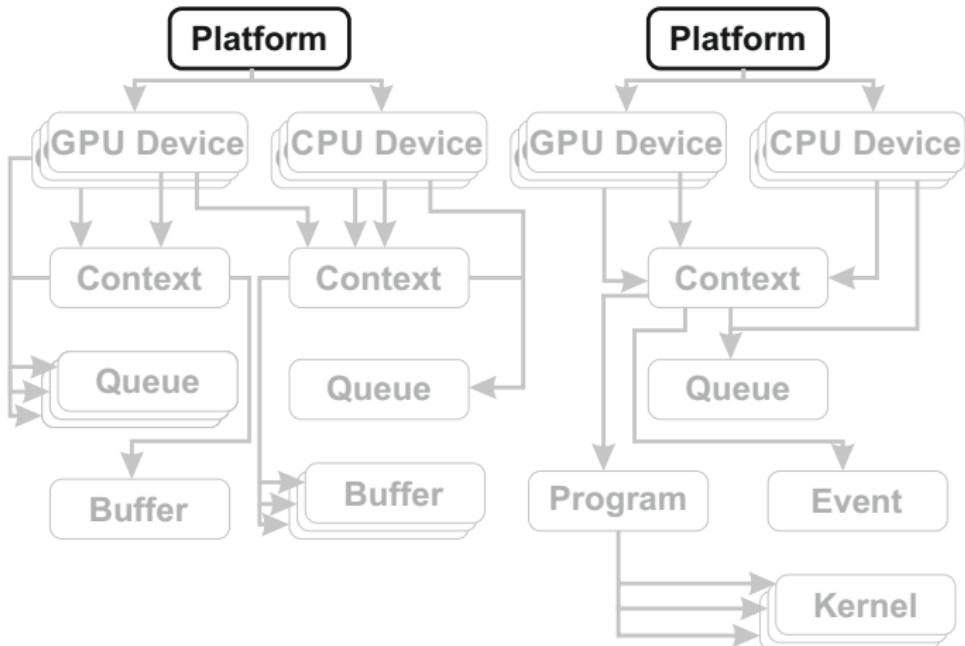
FACULTY
OF INFORMATION
TECHNOLOGY

Knihovna OpenCL

- Linux, Windows, MacOS X, Android ...
- možnost rozšíření pomocí extensions
- aktuální verze 3.0 - návrat k funkcionalitě verze 1.2 (nejvíce podporované) - funkcionality verzí 2.0 pomocí extensions
- návod na zprovoznění: Zdroj

Podpora HW

- AMD CPU - CPU s podporou SSE 2.0 (Athlon 64, Pentium M) a novější
- AMD GPU - Terascale 1. generace řada R700 (řada HD 4000) a novější
- Intel CPU - CPU s podporou SSE 4.1 (Core 2 Penryn) a novější
- Intel GPU - architektura Ivy bridge (3. gen. core procesorů) a novější
- Nvidia GPU - architektura Tesla (řada 8000) a novější



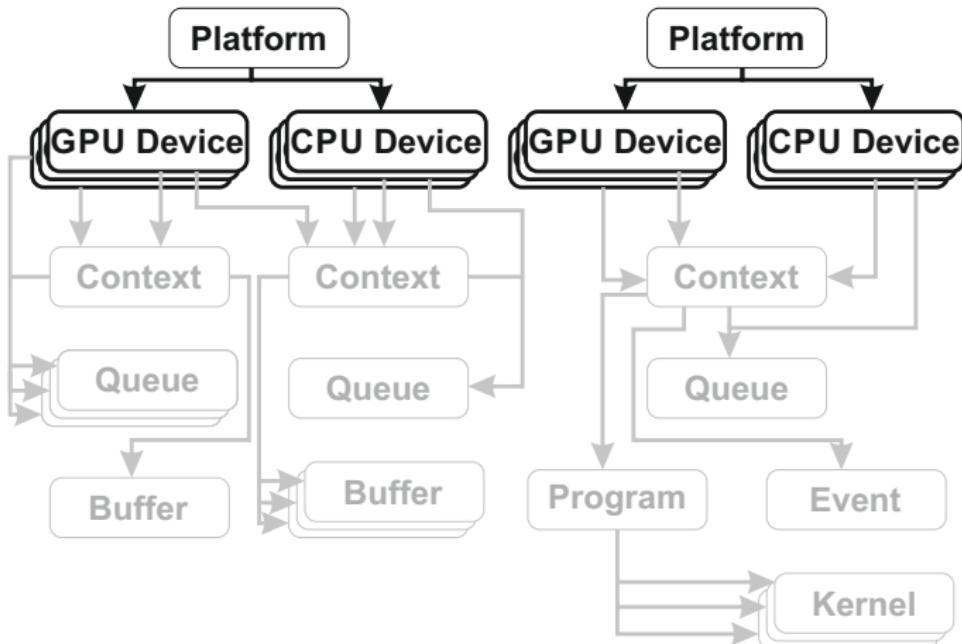
- AMD, Intel, Nvidia ...
- vázané na driver
- komunikace jen v rámci platformy

```
// ziskani platforem
cl_int clGetPlatformIDs(cl_uint num_entries, cl_platform_id *platforms,
                       cl_uint *num_platforms);

// ziskani platforem C++
static cl_int cl::Platform::get(VECTOR_CLASS<Platform> * platforms)

// info o platforeme
cl_int clGetPlatformInfo(cl_platform_id platform,
                        cl_platform_info param_name, size_t param_value_size,
                        void *param_value, size_t *param_value_size_ret)

// info o platforeme C++
template <cl_int name> typename
    detail::param_traits<detail::cl_platform_info, name>::param_type
cl::Platform::getInfo(void)
```



- výběr požadovaného typu zařízení:
CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU, ...
- možná rozšíření: double, atomické instrukce, textury ...
- v novějších verzích se dostávají některá rozšíření do specifikace - povinnost podpory ze strany zařízení

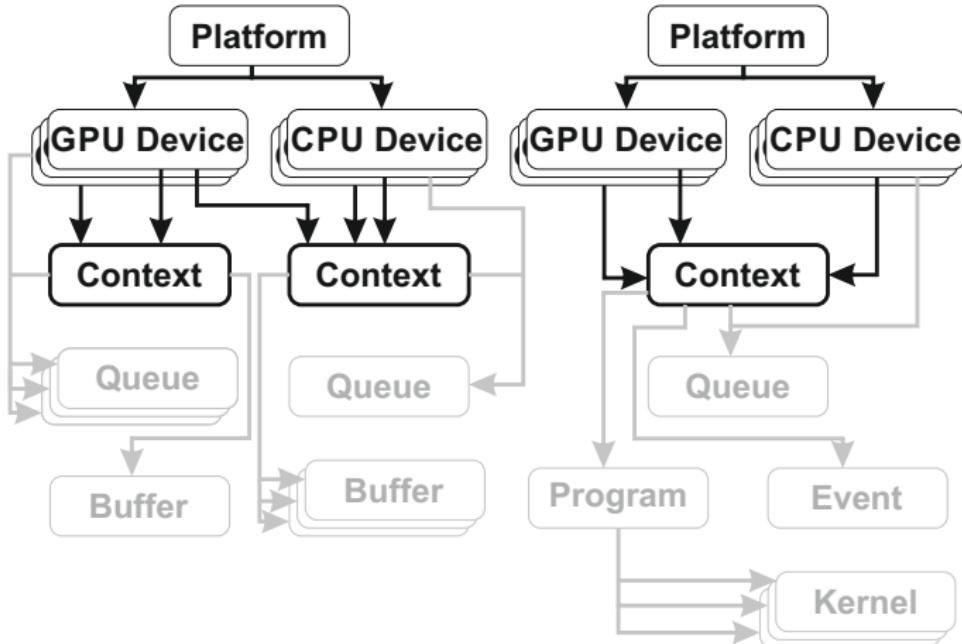
```
// povoleni double rozsireni
#pragma OPENCL EXTENSION cl_khr_fp64

// ziskani zarizeni
cl_int clGetDeviceIDs(cl_platform_id platform ,
                      cl_device_type device_type, cl_uint num_entries ,
                      cl_device_id *devices, cl_uint *num_devices )

// ziskani zarizeni C++
cl_int cl::Platform::getDevices(cl_device_type type,
                                VECTOR_CLASS<Device> *devices)

// info o zarizeni
cl_int clGetDeviceInfo(cl_device_id device,
                      cl_device_info param_name, size_t param_value_size,
                      void *param_value, size_t *param_value_size_ret)

// info o zarizeni C++
template <cl_int name> typename
    detail::param_traits<detail::cl_device_info, name>::param_type
cl::Device::getInfo(void)
```



- pouze ze zařízení v rámci platformy
- zprostředkovává komunikaci pomocí eventů
- lze filtrovat pomocí `cl_context_properties`

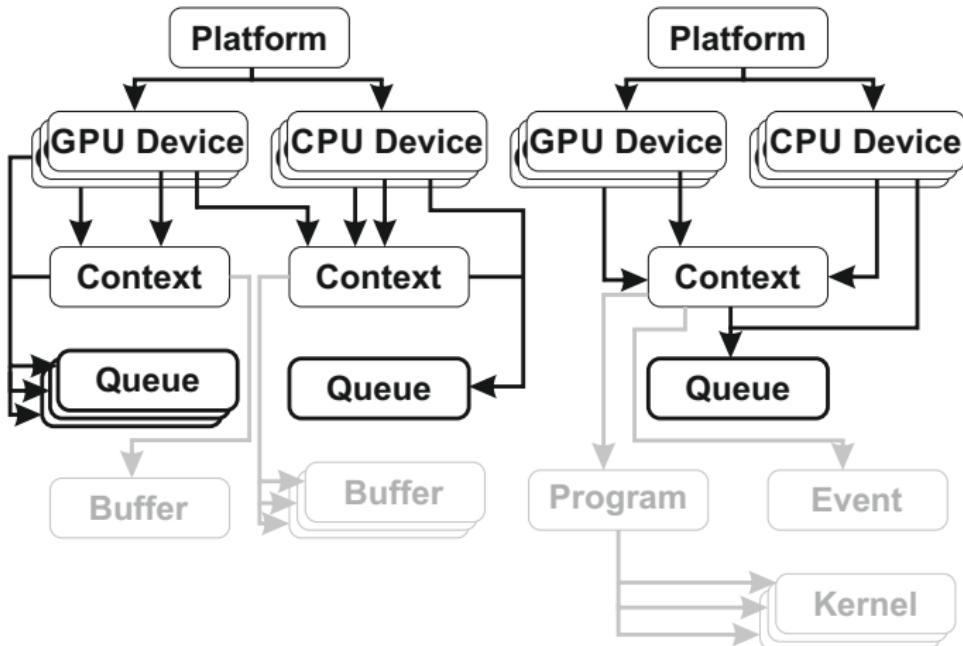
```
// vytvoreni kontextu
cl_context clCreateContext(
    const cl_context_properties *prop,
    cl_uint num_devices,
    const cl_device_id *devices,
    (void CL_CALLBACK *pfn_notify) (
        const char *errinfo,
        const void *private_info,
        size_t cb,
        void *user_data),
    void *user_data,
    cl_int *errcode_ret)

// vytvoreni kontextu C++
cl::Context::Context(VECTOR_CLASS<Device>& devices,
    cl_context_properties * properties = NULL,
    void (CL_CALLBACK * pfn_notify) (
        const char * errorinfo,
        const void * private_info_size,
        ::size_t cb,
        void * user_data) = NULL,
    void * user_data = NULL,
    cl_int * err = NULL)
```

- lze vytvořit i s implicitním zařízením daného typu

```
// vytvoreni kontextu daneho typem zarizeni
cl_context clCreateContextFromType(
    const cl_context_properties *prop,
    (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)

// vytvoreni kontextu daneho typem zarizeni C++
cl::Context::Context(cl_device_type type,
    cl_context_properties * properties = NULL,
    void (CL_CALLBACK * pfn_notify) (
        const char * errorinfo,
        const void * private_info_size,
        ::size_t cb,
        void * user_data) = NULL,
    void * user_data = NULL,
    cl_int * err = NULL)
```



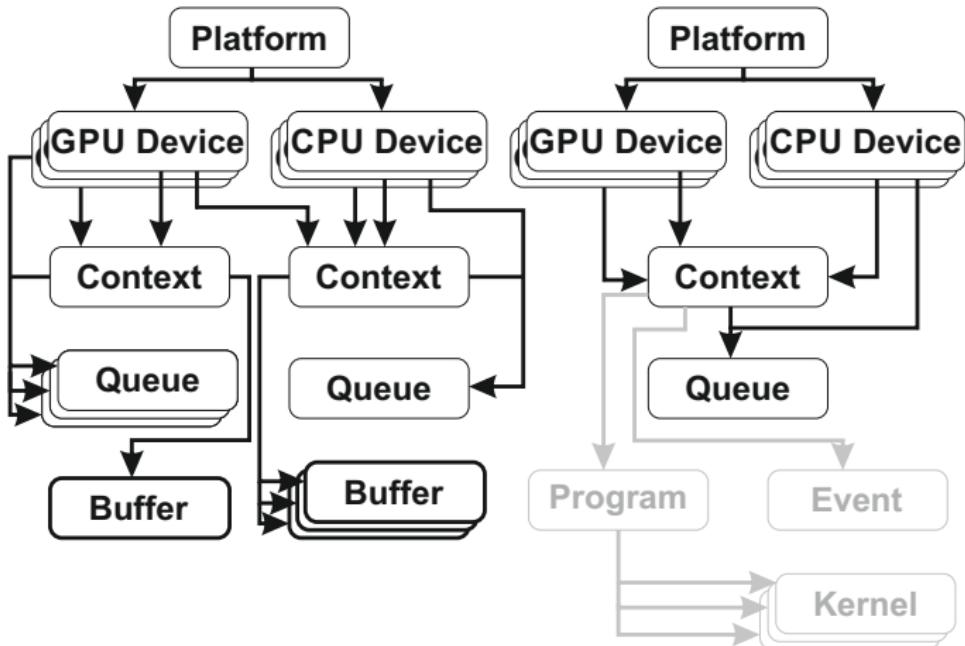
- in-order, out-of-order - `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`
- `enqueue*()` funkce pro zařazení příkazu do fronty
- více front na zařízení
- aktivace profilování - `CL_QUEUE_PROFILING_ENABLE`

```
// vytvoreni fronty
cl_command_queue clCreateCommandQueue(
    cl_context context, cl_device_id device,
    cl_command_queue_properties properties,
    cl_int *errcode_ret)

// vytvoreni fronty C++
cl::CommandQueue::CommandQueue(
    const Context& context, const Device& device,
    cl_command_queue_properties properties = 0,
    cl_int * err = NULL)

// format prikazu pouzivajicich frontu
clEnqueue*(cl_command_queue queue, ....,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)

// format prikazu pouzivajicich frontu C++
cl::CommandQueue::enqueue*(, ....,
    const VECTOR_CLASS<Event> * events = NULL,
    Event * event = NULL)
```



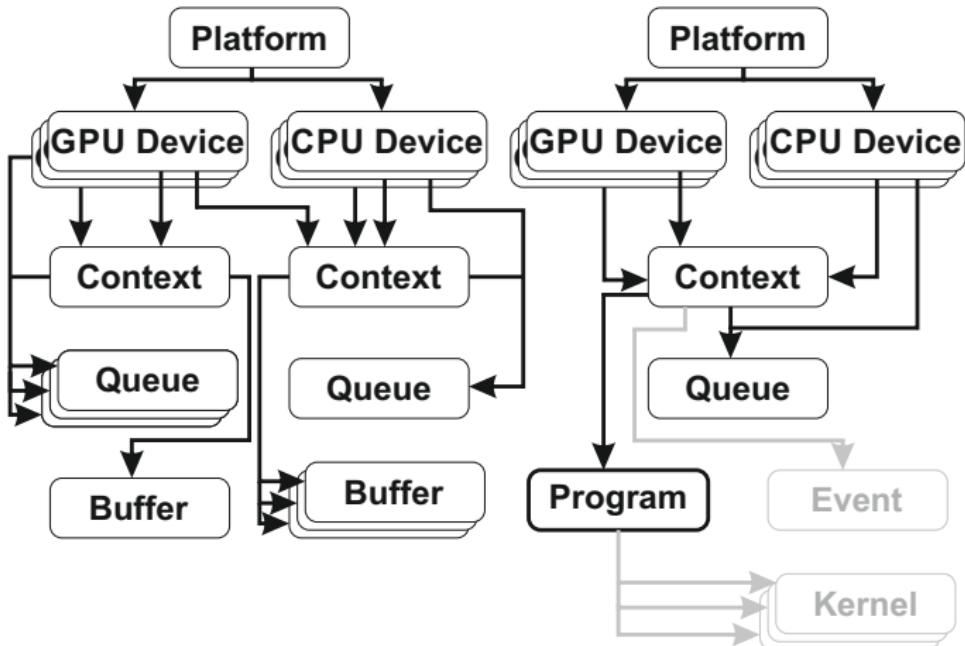
- Image, Buffer, Rectangle
- přímý přístup do paměti
- vytváření nad kontextem
- manipulace s buffery se zařazuje do fronty
- možnost synchronního vykonání manipulace

```
// vytvoreni bufferu
cl_mem clCreateBuffer(cl_context context, cl_mem_flags flags,
                      size_t size, void *host_ptr, cl_int *errcode_ret)

// vytvoreni bufferu C++
cl::Buffer::Buffer(
    const Context& context, cl_mem_flags flags, ::size_t size,
    void * host_ptr = NULL, cl_int * err = NULL)

// kopirovani ze zarizeni
cl_int clEnqueueReadBuffer(cl_command_queue command_queue,
                           cl_mem buffer, cl_bool blocking_read, size_t offset,
                           size_t cb, void *ptr, cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list, cl_event *event)

// kopirovani ze zarizeni C++
cl_int cl::CommandQueue::enqueueReadBuffer(const Buffer& buffer,
                                           cl_bool blocking_read, ::size_t offset,
                                           ::size_t size, const void * ptr,
                                           const VECTOR_CLASS<Event> * events = NULL,
                                           Event * event = NULL)
```



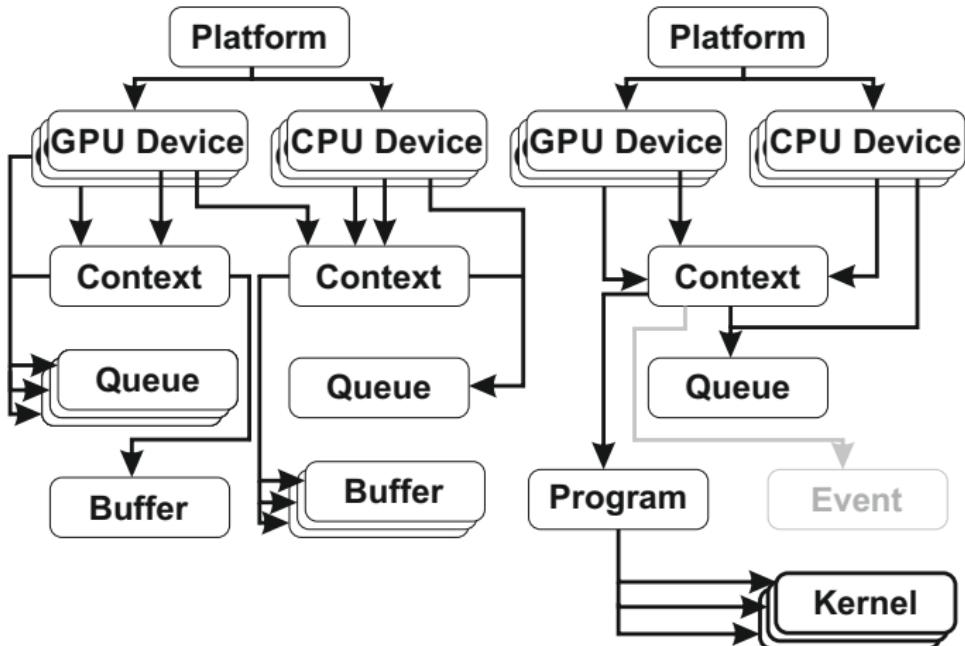
- načítání ze zdrojového nebo nativního kódu
- zdrojový kód je nutné zkompilovat

```
// vytvoreni programu ze zdrojoveho kodu
cl_program clCreateProgramWithSource(cl_context context,
                                     cl_uint count, const char **strings,
                                     const size_t *lengths, cl_int *errcode_ret)

// vytvoreni programu ze zdrojoveho kodu C++
cl::Program::Program(const Context& context,
                     const Sources& sources,
                     cl_int * err = NULL)

// komplikace programu
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)

// komplikace programu C++
cl_int cl::Program::build(const VECTOR_CLASS<Device> devices,
                         const char * options = NULL,
                         (CL_CALLBACK * pfn_notify)
                         (cl_program, void * user_data) = NULL, void * data = NULL)
```



- vytvářeny z programu
- funkce volatelné z runtimu
- ve zdrojovém kódu označeny jako `__kernel`

```
// vytvoreni kernelu z programu
cl_kernel clCreateKernel(cl_program program,
                        const char *kernel_name,
                        cl_int *errcode_ret)

// vytvoreni kernelu z programu C++
cl::Kernel::Kernel(const Program& program,
                    const char * name, cl_int * err = NULL)

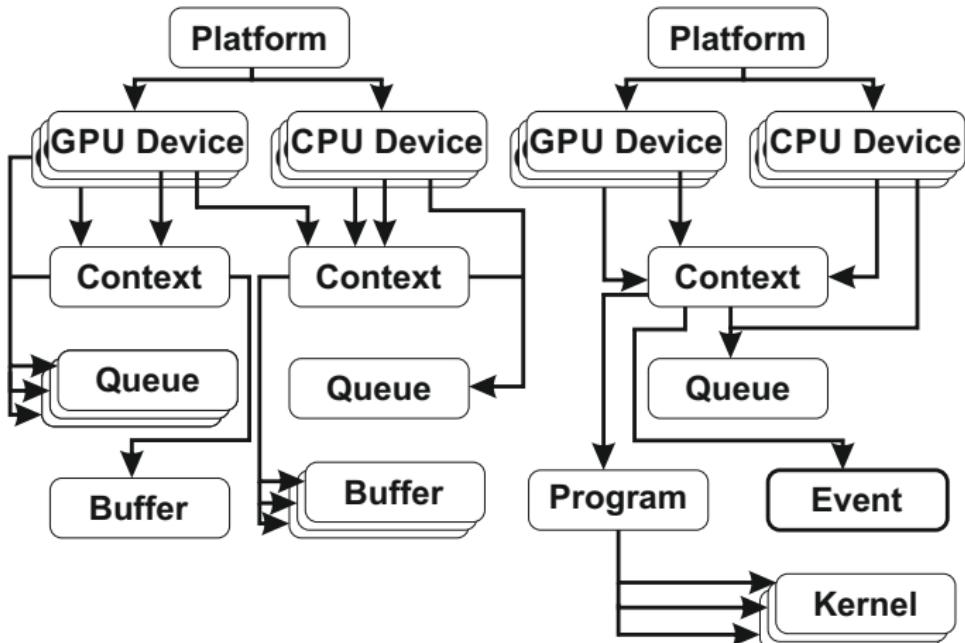
// nastaveni parametru kernelu
cl_int clSetKernelArg(cl_kernel kernel,
                      cl_uint arg_index,
                      size_t arg_size,
                      const void *arg_value)

// nastaveni parametru kernelu C++
template <typename T> cl_int cl::Kernel::setArg(cl_uint index, T value)
```

- počet vláken v jednotlivých dimenzích musí být zarovnaný na velikost skupiny v těchto dimenzích - je nutné ošetřit přesahující hodnoty
- id v kernelu `get_global_id(dim)`
- id ve skupině `get_local_id(dim)`
- id skupiny `get_group_id(dim)`

```
// zalozeni kernelu
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)

// zalozeni kernelu C++
cl_int cl::CommandQueue::enqueueNDRangeKernel(
    const Kernel& kernel,
    const NDRANGE& offset,
    const NDRANGE& global,
    const NDRANGE& local,
    const VECTOR_CLASS<Event> * events = NULL,
    Event * event = NULL)
```



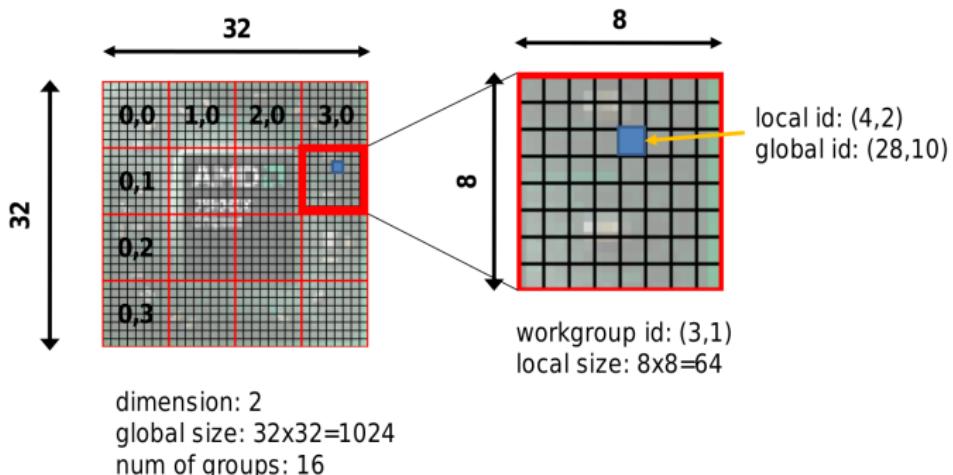
- profilování
- asynchronní čekání na výpočty
- enqueue příkazy cekají na dokončení operací v seznamu eventů

```
// vytvoreni eventu
cl_event clCreateUserEvent(cl_context context,
                           cl_int *errcode_ret)

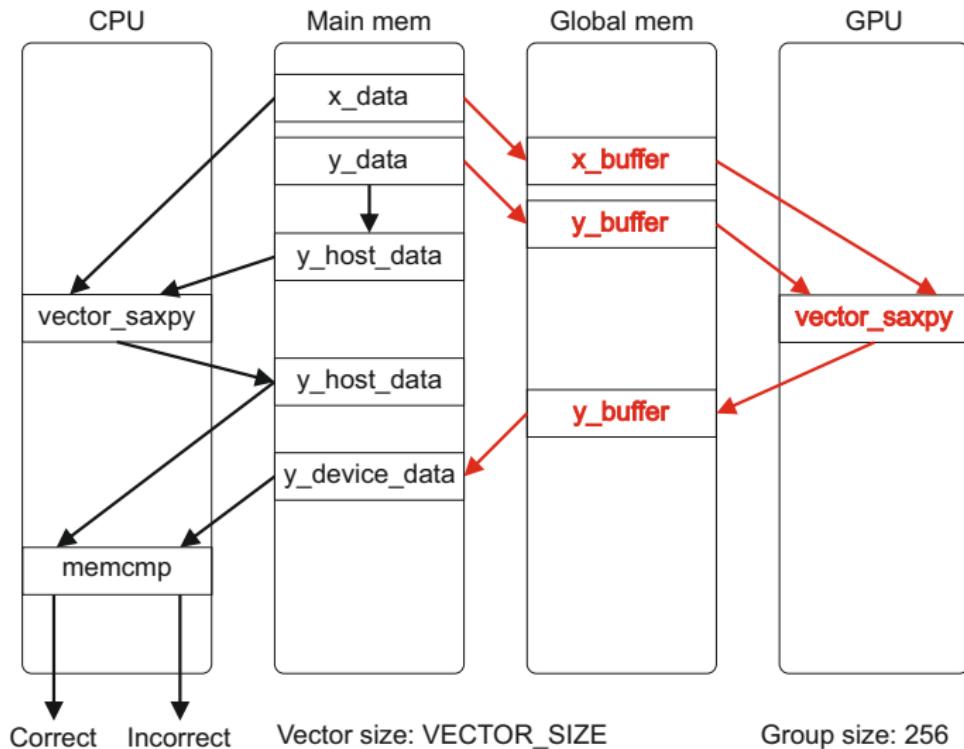
// vytvoreni eventu C++
cl::UserEvent::UserEvent(Context& context, cl_int *err = NULL)

// pouziti eventu
clEnqueue*(cl_command_queue queue,....,
           cl_uint pocet_eventu_v_seznamu,
           const cl_event *seznam_eventu,
           cl_event *produkowany_event)

// pouziti eventu C++
cl::CommandQueue::enqueue*(,....,
                           const VECTOR_CLASS<Event> * events = NULL,
                           Event * event = NULL)
```



```
// globalni index vlakna v dimezi dimindx
size_t get_global_id(int dimindx);
// index vlakna ve skupine v dimezi dimindx
size_t get_local_id(int dimindx);
// globalni pocet vlaken v dimenzi dimindx
size_t get_global_size(int dimindx);
// pocet vlaken ve skupine v dimenzi dimindx
size_t get_local_size(int dimindx);
// indexace v 1D poli pomocí mrizky 2D vlaken
size_t id = get_global_id(0) + get_global_id(1) * get_global_size(0);
```



Vaším úkolem je

- doplnit kód v runtimu - 2 body
- doplnit kernel - 1 bod

Postup:

- vybrat GPU zařízení
- vytvořit kontext a frontu nad zařízením
- vytvořit buffery `x_buffer` a `y_buffer` na zařízení
- nastavit parametry kernelu
- naplnit gpu buffery z bufferů `x_data` a `y_data`
- pustit kernel `saxpy` nad vektorem dat s velikostí skupiny 256
- nakopírovat výsledná data do `y_device_data`
- doplnit kernel `saxpy`

Odkazy:

- khronos - OpenCL 1.2 - Reference pages
- khronos - OpenCL 1.2 - Specification
- khronos - OpenCL 1.2 C++ binding - Specification

Děkuji za pozornost!