Hasso Plattner Institut
Digital Engineering · Universität Potsdam

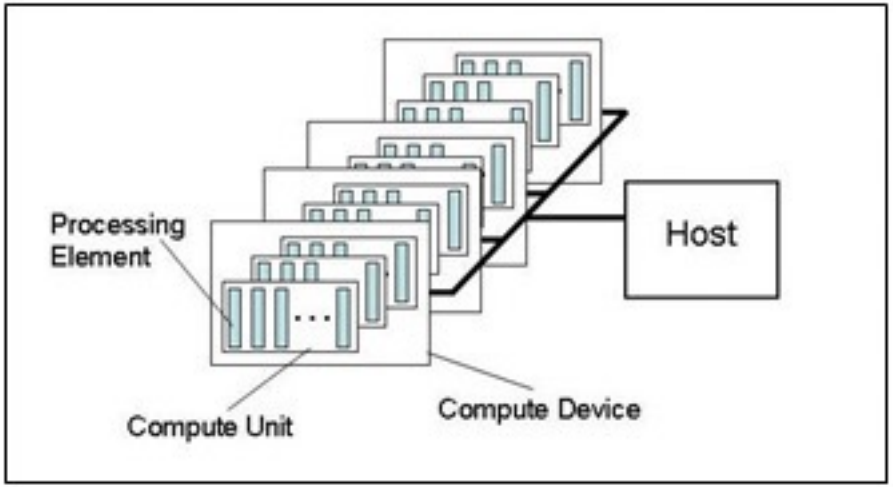
Parallel Programming and Heterogeneous Computing

Heterogeneous Computing with OpenCL

Sven Köhler, Lukas Wenzel, *Max Plauth*, and Andreas Polze
Operating Systems and Middleware Group

OpenCL: Platform Model

[OpenCL Specification]

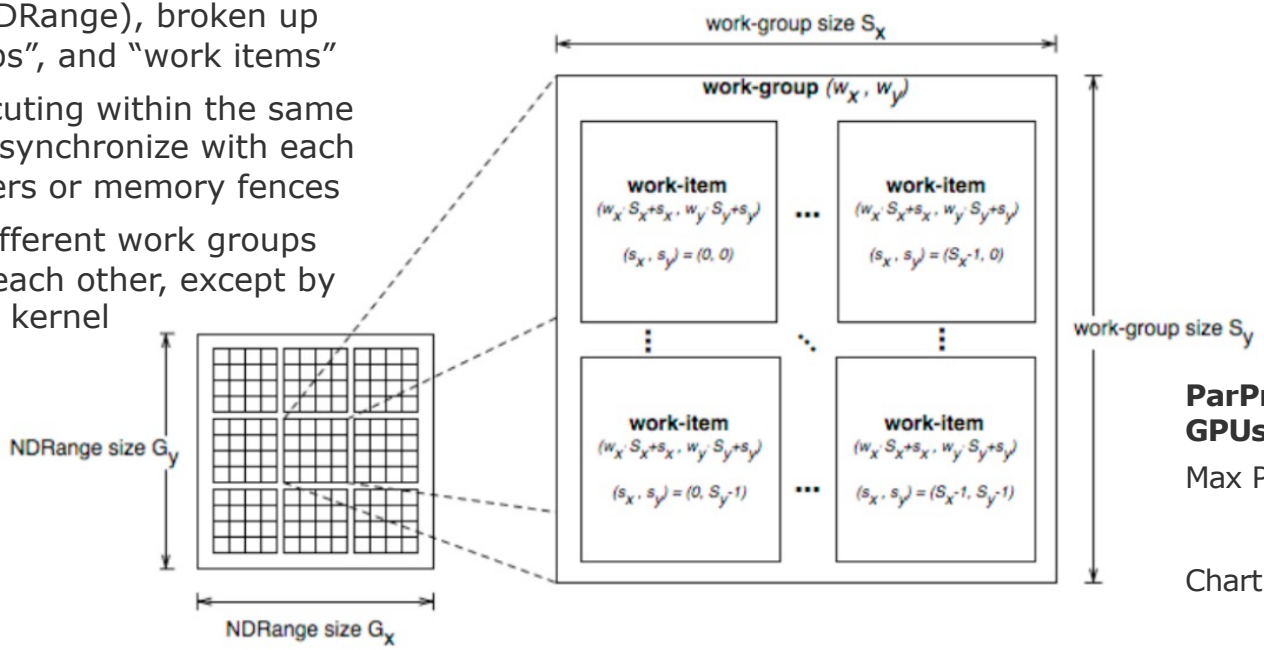


- OpenCL exposes CPUs, GPUs, FPGAs, and other Accelerators as *devices*
- Each *device* contains one or more *compute units*, i.e. SMs, CEs, ...
- Each *compute unit* contains one or more *processing elements*

OpenCL Execution Model

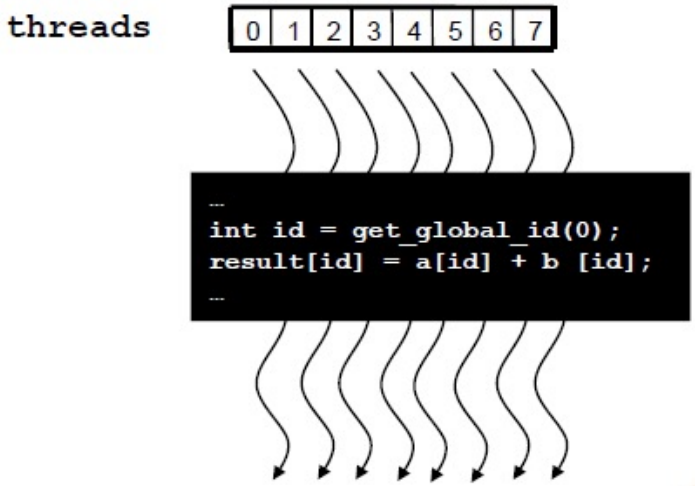
[OpenCL Specification]

- Parallel work is submitted to devices by launching kernels
- Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”
- Work items executing within the same work group can synchronize with each other with barriers or memory fences
- Work items in different work groups can’t sync with each other, except by launching a new kernel



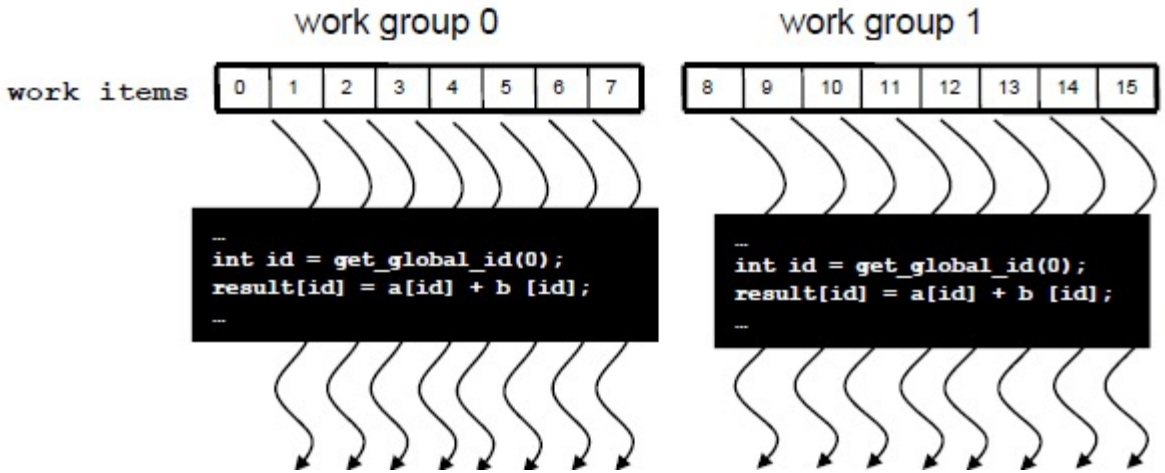
OpenCL Execution Model

- An OpenCL kernel is executed by an array of work items.
  - All work items run the same code (SPMD)
  - Each work item has an index that it uses to compute memory addresses and make control decisions



OpenCL Execution Model

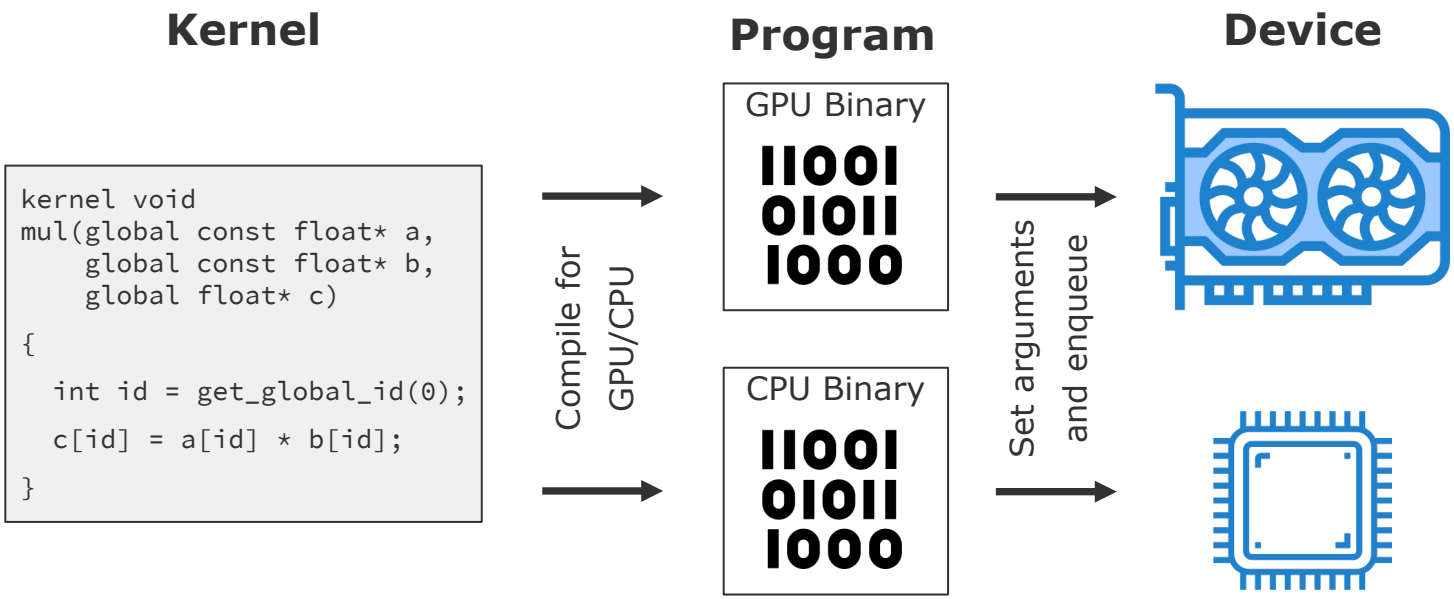
- Divide monolithic work item array into work groups
  - Work items within a work group cooperate via shared memory, atomic operations and barrier synchronization
  - Work items in different work groups cannot cooperate



OpenCL Execution Model: Kernels

- A subset of ISO C99 - without some C99 features
  - headers, function pointers, recursion, variable length arrays, and bit fields
- A superset of ISO C99 with additions for
  - Work-items and workgroups
  - Vector types (2,4,8,16): endian safe, aligned at vector length
  - Image types mapped to texture memory
  - Synchronization
  - Address space qualifiers
- Also includes a large set of built-in functions for image manipulation, work-item manipulation, specialized math routines, vectors, etc.

OpenCL Execution Model: Building and Executing Kernels



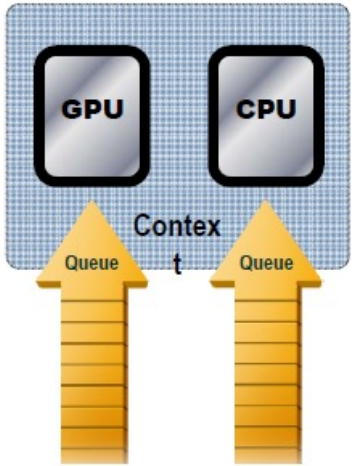
OpenCL codes must be prepared to deal with much greater hardware diversity (e.g. features are optional and may not be supported on all devices)

[Icons by freepik, srip, monkik, and Smashicons from www.flaticon.com]

OpenCL Execution Model

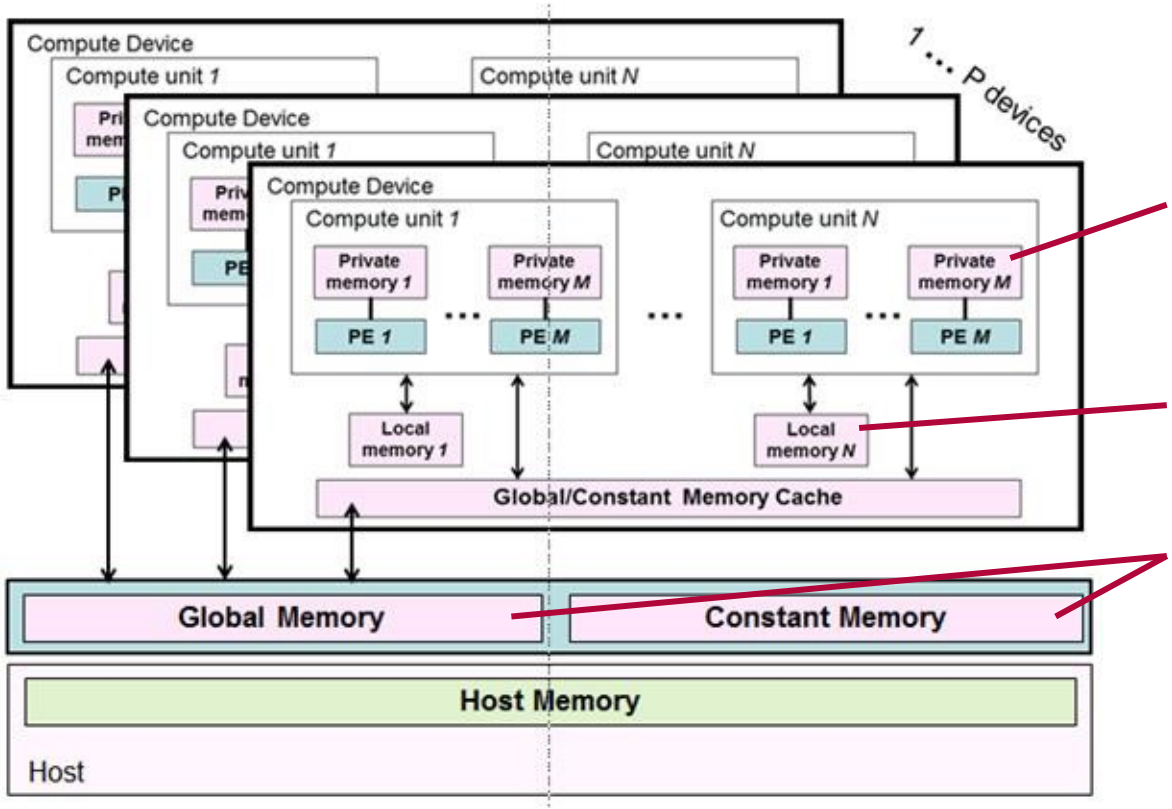
An OpenCL application runs on a host which submits work to the compute devices. Kernels are executed in contexts defined and manipulated by the host.

- **Work item**: the basic unit of work on an OpenCL device.
- **Kernel**: the code for a work item. Basically a C function
- **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
- **Context**: The environment within which work-items executes … includes devices and their memories and command queues.
- **Queue**: used to manage a device. (copy memory, start work item, …) In-order vs. out-of-order execution



OpenCL Memory Model

[OpenCL Specification]

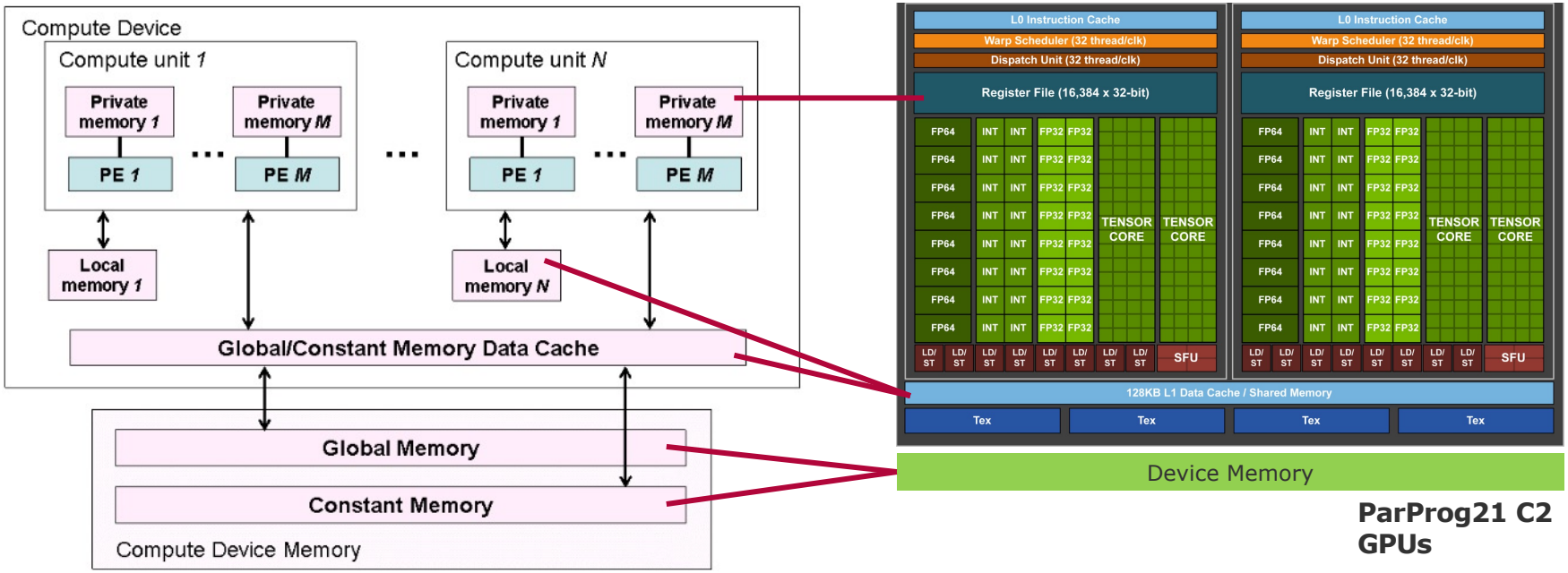


per work-item

shared within a workgroup

accessible by all workgroups

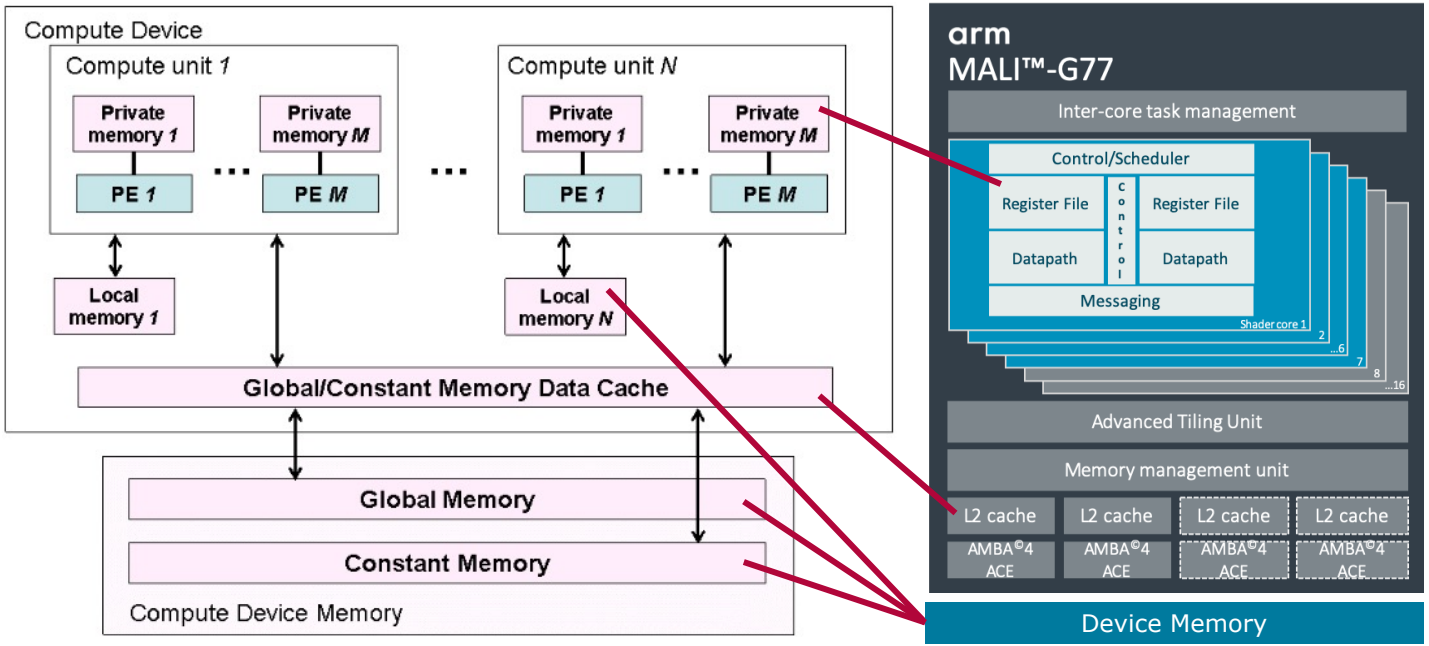
OpenCL: Mapping the Memory Model to Hardware



ParProg21 C2 GPUs

Max Plauth

OpenCL: Mapping the Memory Model to Hardware



Example: Vector Addition OpenCL Kernel

- Kernel body is instantiated once for each work item
  - Each instance is getting an unique index

```
__kernel void mul(__global const float* a,
                  __global const float* b,
                  __global       float* c)
{
  int id = get_global_id(0);
  c[id] = a[id] + b[id];
}
```

Code that actually executes on target devices

OpenCL Host Code Workflow (1/2)

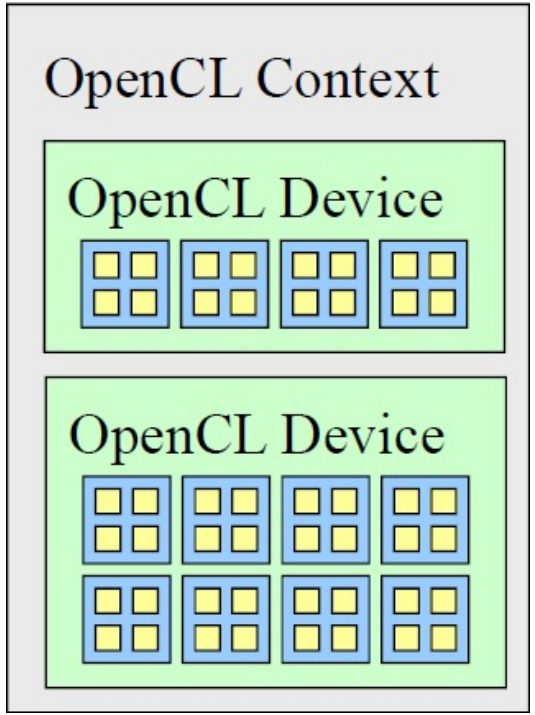
1. Query available *Platforms*
2. Create a *Context* for the desired *Platform*
   - The `CL_DEVICE_TYPE` flag is used to specify the desired device type
   - Only one *Platform* can be used per *Context*
3. Query available *Devices* in the *Context*
4. Create *Command Queues* for each *Device* that should be used
5. Build *Program* for the *Context*
   - A *Program* may contain multiple kernels and device functions
6. Create *Kernel* from *Program*

OpenCL Host Code Workflow (2/2)

7. Create *Buffers* on *Context* level
   - The choice of allocation flags is crucial
   - Read/rite properties: `CL_MEM_READ_ONLY`, `CL_MEM_READ_WRITE`
   - Memory location: `CL_MEM_USE_HOST_PTR`, `CL_MEM_ALLOC_HOST_PTR`
8. Enqueue *Write* or *Map* operations on *Buffers (Command Queue* level)
   - Asynchronous call
9. Set *Kernel* arguments
10. Enqueue *Kernel* (Command Queue level)
    - Asynchronous call
11. Enqueue *Read* or *Unmap* operation on *Buffers* (*Command Queue* level)
    - Asynchronous call
12. Wait for *Command Queue* to finish

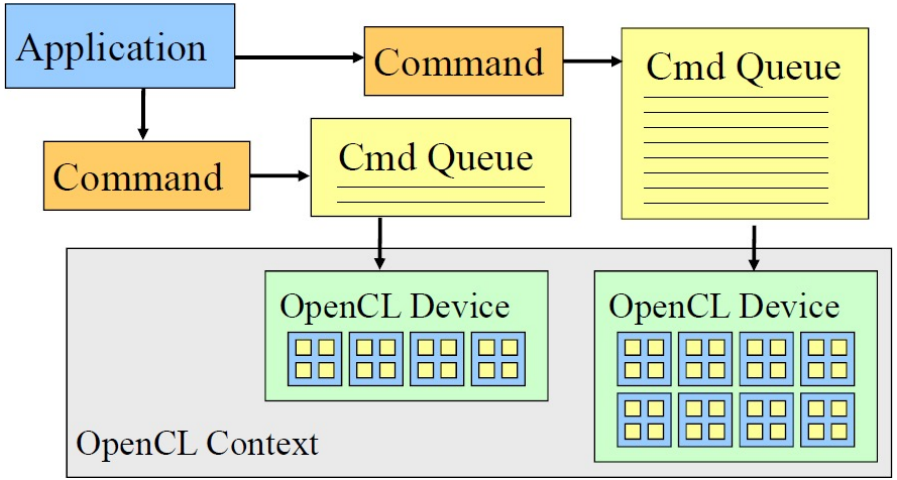
OpenCL Contexts and Devices

- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- clCreateBuffer() is the main data object allocation function
- error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory transfers are associated with a command queue (thus a specific device)



OpenCL Device Command Execution

- Command-queue - coordinates execution of kernels
  - Kernel execution commands
  - Memory commands: transfer or mapping of memory object data
  - Synchronization commands: constrains the order of commands



Example: Vector Addition OpenCL Host Code

```c
// Create context
cl_context ctx = clCreateContext(prop, 1, &device, NULL, NULL, &err);

// Create program
unsigned char* program_file = NULL;
size_t program_size = 0;
read_file(&program_file, &program_size, "vec_add.cl");

cl_program program =
    clCreateProgramWithSource(ctx, 1,
     (const char **)&program_file, &program_size, &err);

err = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

free(program_file);

// Allocate memory buffers (on the device)
cl_mem a = clCreateBuffer(ctx, CL_MEM_READ_ONLY, buf_size, NULL, &err);
cl_mem b = clCreateBuffer(ctx, CL_MEM_READ_ONLY, buf_size, NULL, &err);
cl_mem c = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY, buf_size, NULL, &err);

// Create command queue
cl_command_queue queue = clCreateCommandQueue(ctx, device, 0, NULL);

// Enqueue the write buffer commands
cl_event wb_events[2];

err = clEnqueueWriteBuffer(queue, a, CL_FALSE, 0, buf_size, data, 0,
                           NULL, &wb_events[0]);
err = clEnqueueWriteBuffer(queue, b, CL_FALSE, 0, buf_size, data, 0,
                           NULL, &wb_events[1]);
```

```c
// Enqueue the kernel execution command
cl_kernel kernel = clCreateKernel(program, "vec_add", &err);
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &c);
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &a);
err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &b);

const size_t global_offset = 0;
cl_event kernel_event;
err = clEnqueueNDRangeKernel(queue, kernel, 1, &global_offset,
                             &num_elems, NULL, 2, wb_events,
                             &kernel_event);

// Enqueue the read buffer command
err = clEnqueueReadBuffer(queue, c, CL_TRUE, 0, buf_size, data, 1,
                          &kernel_event, NULL);

// Wait until every commands are finished
err = clFinish(queue);
```

Error Handling

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

...

cl_int err;
cl::Context context =
      cl::Context(CL_DEVICE_TYPE_GPU, cprops, NULL, NULL, &err);
checkErr(err, "Error while creating cl::Context()");
```

Retrieve Build Errors

```
cl_int err;
ifstream cl_file("kernel.cl");

string cl_string(istreambuf_iterator<char>(cl_file), (istreambuf_iterator<char>()));
cl::Programm prog(context, cl_string.c_str());
err = prog.build();

if (err == CL_BUILD_PROGRAM_FAILURE) {
        cl::vector<cl::Device> devices;
        devices = context.getInfo<CL_CONTEXT_DEVICES>();
        checkErr(devices.size() > 0 ? CL_SUCCESS : -1, devices.size() > 0);
        std::string log = prog.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0], NULL);
        std::cerr << "Build Log: " << log << std::endl;
}
```