UNIVERSITÉ DU LUXEMBOURG

High Performance Computing & Big Data Services

hpc.uni.lu

hpc@uni.lu

@ULHPC

LUXEMBOURG LET'S MAKE IT HAPPEN

Running and Writing OpenCL kernel for FPGA

Emmanuel Kieffer

EUMaster4HPC

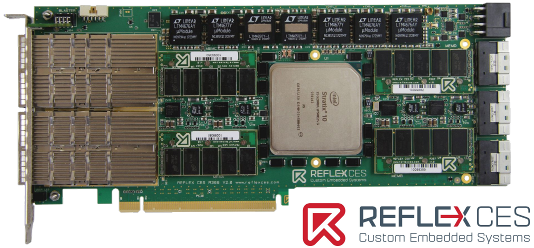
UNIVERSITÉ DU LUXEMBOURG

Intel® FPGA SDK for OpenCL™ Software Technology

UNIVERSITÉ DU LUXEMBOURG

Preferred vendors

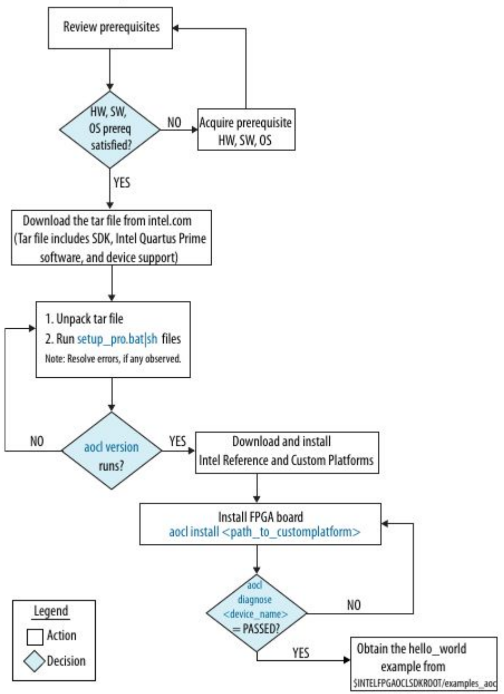
- Vendors
  - BittWare
  - Gidel
  - Terasic
  - Reflexces
  - Etc ...
- Intel preferred board for FPGA
- Board for High-Performance computing
- BSP provided by vendor and compatible with the SDK
- Provide software layer to interact with host code including drivers



REFLEX CES
Custom Embedded Systems

Setting Up the OpenCL Design Environment

- Intel® Quartus® Prime Design Software
- Intel® FPGA SDK for OpenCL™ Software Technology
- BSP installation files provided by vendor
  - <Quartus_installation_directory>/hld/board
  - FlexLM license installed by the administrator
  - Env variable `$LM_LICENCSE`
- Environment variables:
  - **`$QUARTUS_ROOTDIR`**: Quartus installation directory
  - **`$QSYS_ROOTDIR`**: Qsys is a software tool provided by Altera to build SOPC system on FPGA
  - **`$INTELFPGAOCLSDKROOT`**: path to the SDK's installation directory
  - **`$ALTERAOCLSDKROOT`** == **$INTELFPGAOCLSDKROOT`**
  - **`$AOCL_BOARD_PACKAGE_ROOT`**: BSP path
  - **`$OCL_ICD_VENDORS`** == **`$ACL_BOARD_VENDOR_PATH`**: same as **`$AOCL_BOARD_PACKAGE_ROOT`**



(source: Intel)

Example of Environment

```
export LM_LICENSE_FILE=1800@192.168.1.80:/usr/.../license/lic1512.dat
export QUARTUS_ROOTDIR=/usr/.../20.4/quartus
export QSYS_ROOTDIR="$QUARTUS_ROOTDIR"/sopc_builder/bin
export INTELFPGAOCLSDKROOT=/usr/.../20.4/hld
export AOCL_BOARD_PACKAGE_ROOT="$INTELFPGAOCLSDKROOT"/board/bittware/520nmx
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:"$QUARTUS_ROOTDIR"/linux64:"$AOCL_BOARD_PACKAGE_ROOT"/linux64/lib:"$ALTERA
OCLSDKROOT"/host/linux64/lib
export
PATH=$PATH:"$QUARTUS_ROOTDIR"/bin:"$ALTERAOCLSDKROOT"/linux64/bin:"$ALTERAOCLSDKROOT"/bin:/usr/.../16.1/qsys/bin
```

Quartus Prime

- Intel® Quartus® Prime Design Software
- Intel® FPGAs, SoCs, CPLD
- Design entry and synthesis to optimization, verification, and simulation
- Build the hardware configuration (VHDL, verilog, …)
- <u>Without:</u> You cannot develop OpenCL code for FPGA

Supported Devices	Pro Edition	Standard Edition	Lite Edition (Free)
Intel® Agilex™	✓	-	-
Intel® Stratix® 10	✓	-	-
Intel® Arria® 10	✓	✓	-
Intel® Cyclone® 10 GX	✓	-	-
Intel® Cyclone® 10 LP	-	✓	✓

HPC FPGA cards only

(source: Intel)

Directory structure of the SDK

Folder	Description
…/bin	Main binaries and utilities
…/board	Design files related to specific supported boards
…/ip	Ip cores required for kernel compilation
…/host	Files used by compilation flow for users program
…/host/include	OpenCL header files + interface files for compilation
…/host/windows64/lib .../host/linux64/lib …/host/arm32/lib	OpenCL host runtime libraries

(source: Intel)

AOCL tool

- The `aocl` command is the main tool to compile kernels and manage your FPGA workflows
- It comes with some utilities functions:

Host compilation utilities	
aocl compile-config	Displays compiler flags for the host program
aocl link-config	Show links options needed by the host program
aocl makefile	Provide makefile fragments for your application
Board management command	
aocl install	Install a board driver onto your system
aocl diagnose	Run the board vendor's test program
aocl flash	Program the on-board flash through JTAG

AOCL tool

- The `aocl` command is the main tool to compile kernels and manage your FPGA workflows
- It comes with some utilities functions:

aocl list-devices	Lists all installed devices
aocl program	Configure a new FPGA image onto the board
aocl initialize	Configure a default FPGA image onto the board
Kernel compilation report	
aocl profile	Displays kernel execution profiler data
aocl env	Show the compilation environment of a binary (aocx)
Help for commands	
aocl help <subcommand>	Show help for a particular subcommand

Diagnostic / Listing boards

- You can detect whether the environment variables of the FPGA board are set:
    - `aoc -list-boards`
    - (Below) Two cards Intel Stratix 10MX 16GB HBM (High Bandwith Memory)

```
[u100057@mel3009 ~]$ aoc -list-boards

Board list:
  p520_hpc_m210h_g3x16 (default)
     Board Package: /apps/USE/easybuild/staging/2022.1/software/520nmx/20.4
     Memories:      HBM0, HBM1, HBM2, HBM3, HBM4, HBM5, HBM6, HBM7, HBM8, HBM9, HBM10, HB
M11, HBM12, HBM13, HBM14, HBM15, HBM16, HBM17, HBM18, HBM19, HBM20, HBM21, HBM22, HBM23,
HBM24, HBM25, HBM26, HBM27, HBM28, HBM29, HBM30, HBM31

  p520_max_m210h_g3x16
     Board Package: /apps/USE/easybuild/staging/2022.1/software/520nmx/20.4
     Memories:      HBM0, HBM1, HBM2, HBM3, HBM4, HBM5, HBM6, HBM7, HBM8, HBM9, HBM10, HB
M11, HBM12, HBM13, HBM14, HBM15, HBM16, HBM17, HBM18, HBM19, HBM20, HBM21, HBM22, HBM23,
HBM24, HBM25, HBM26, HBM27, HBM28, HBM29, HBM30, HBM31
     Channels:      kernel_input_ch0, kernel_output_ch0, kernel_input_ch1, kernel_output_
ch1, kernel_input_ch2, kernel_output_ch2, kernel_input_ch3, kernel_output_ch3
```

UNIVERSITÉ DU LUXEMBOURG

Diagnostic / Listing boards

- You can diagnose each of these cards using:
  - `aocl diagnose <device-names>`
  - OR `aocl diagnose all`
- The command perform:
  - ICD diagnostics
  - BSP Diagnostics
  - Global Memory checks
  - Bandwidth checks

```
As a reference:
PCIe Gen1 peak speed: 250MB/s/lane
PCIe Gen2 peak speed: 500MB/s/lane
PCIe Gen3 peak speed: 985MB/s/lane

Writing 262144 KBs with block size (in bytes) below:

Block_Size Avg     Max     Min     End-End (MB/s)
  524288 5194.50 5299.05 4588.95 4999.68
 1048576 6485.19 6634.84 4628.66 6325.43
 2097152 7255.27 7482.35 5390.30 7153.00
 4194304 7748.90 7993.86 4967.19 7689.52
 8388608 8196.49 8251.45 7866.94 8163.93
16777216 8354.29 8393.30 8109.59 8336.92
33554432 8436.10 8462.38 8311.30 8426.86
67108864 8471.51 8493.00 8416.22 8467.03
134217728 8496.22 8510.09 8482.39 8494.25
268435456 8464.67 8464.67 8464.67 8464.67

Reading 262144 KBs with block size (in bytes) below:

Block_Size Avg     Max     Min     End-End (MB/s)
  524288 5030.72 5175.60 4622.94 4845.66
 1048576 6319.06 6440.88 6055.18 6169.15
 2097152 6993.57 7207.20 5310.98 6902.53
 4194304 7566.28 7633.49 7453.88 7513.31
 8388608 7831.68 7879.42 7774.49 7802.62
16777216 7963.20 7985.26 7939.50 7949.14
33554432 8036.71 8057.63 8012.66 8029.03
67108864 8073.08 8082.68 8062.21 8070.20
134217728 8098.88 8104.39 8093.38 8097.88
268435456 8111.71 8111.71 8111.71 8111.71

Write top speed = 8510.09 MB/s
Read top speed = 8111.71 MB/s
Throughput = 8310.90 MB/s
```

Compiling OpenCL programs

UNIVERSITÉ DU LUXEMBOURG

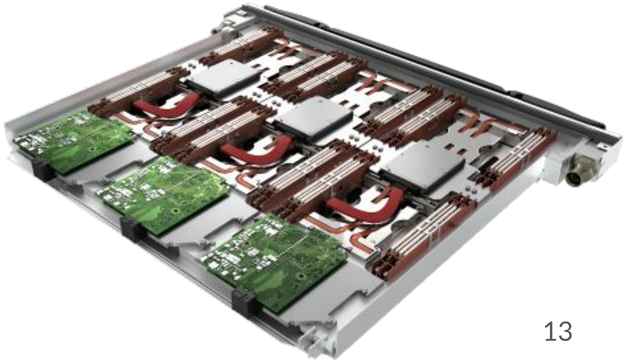
Compiling the host code

- Include CL/opencl.h or CL/cl.hpp
- Install and use a C/C++ compiler
    - MS Visual Studio
    - GNU GCC
    - Intel
- Use aocl compile-config for the OpenCL included
- Use aocl link-config to link to the Intel FPGA lib

```
#include <CL/opencl.h>
#define AOCL_ALIGNMENT 64
#define N (1024*1024)

int main()
{ …
return 0;}
```

C compiler

FPGA libs



Compiling kernels

- `aoc -list-boards`
  - List available boards within the current package

```
[u100057@mel3009 first_code]$ aoc -list-boards
Board list:
  p520_hpc_m210h_g3x16 (default)
     Board Package: /apps/USE/easybuild/staging/2022.1/software/520nmx/20.4
     Memories:      HBM0, HBM1, HBM2, HBM3, HBM4, HBM5, HBM6, HBM7, HBM8, HBM9, HBM10, HBM11, HBM12, HBM1
3, HBM14, HBM15, HBM16, HBM17, HBM18, HBM19, HBM20, HBM21, HBM22, HBM23, HBM24, HBM25, HBM26, HBM27, HBM2
8, HBM29, HBM30, HBM31

  p520_max_m210h_g3x16
     Board Package: /apps/USE/easybuild/staging/2022.1/software/520nmx/20.4
     Memories:      HBM0, HBM1, HBM2, HBM3, HBM4, HBM5, HBM6, HBM7, HBM8, HBM9, HBM10, HBM11, HBM12, HBM1
3, HBM14, HBM15, HBM16, HBM17, HBM18, HBM19, HBM20, HBM21, HBM22, HBM23, HBM24, HBM25, HBM26, HBM27, HBM2
8, HBM29, HBM30, HBM31
     Channels:      kernel_input_ch0, kernel_output_ch0, kernel_input_ch1, kernel_output_ch1, kernel_inpu
t_ch2, kernel_output_ch2, kernel_input_ch3, kernel_output_ch3
```

- `aoc -board=<board> <kernel file>`
  - Compile the kernel for a specific board
  - Generate the kernel hardware system
  - Call Intel Quartus Prime software to create the aocx file

```
[u100057@mel3009 first_code]$ aoc -board=p520_hpc_m210h_g3x16 first_kernel.cl
```

```
#define N (1024*1024)

__kernel void first_kernel ( __global const int * restrict k_din,
                             __global int * restrict k_dout )
{

      for(unsigned int i=0; i<N; i++)
        k_dout[i] = k_din[i] + 40;


}
```

AOC

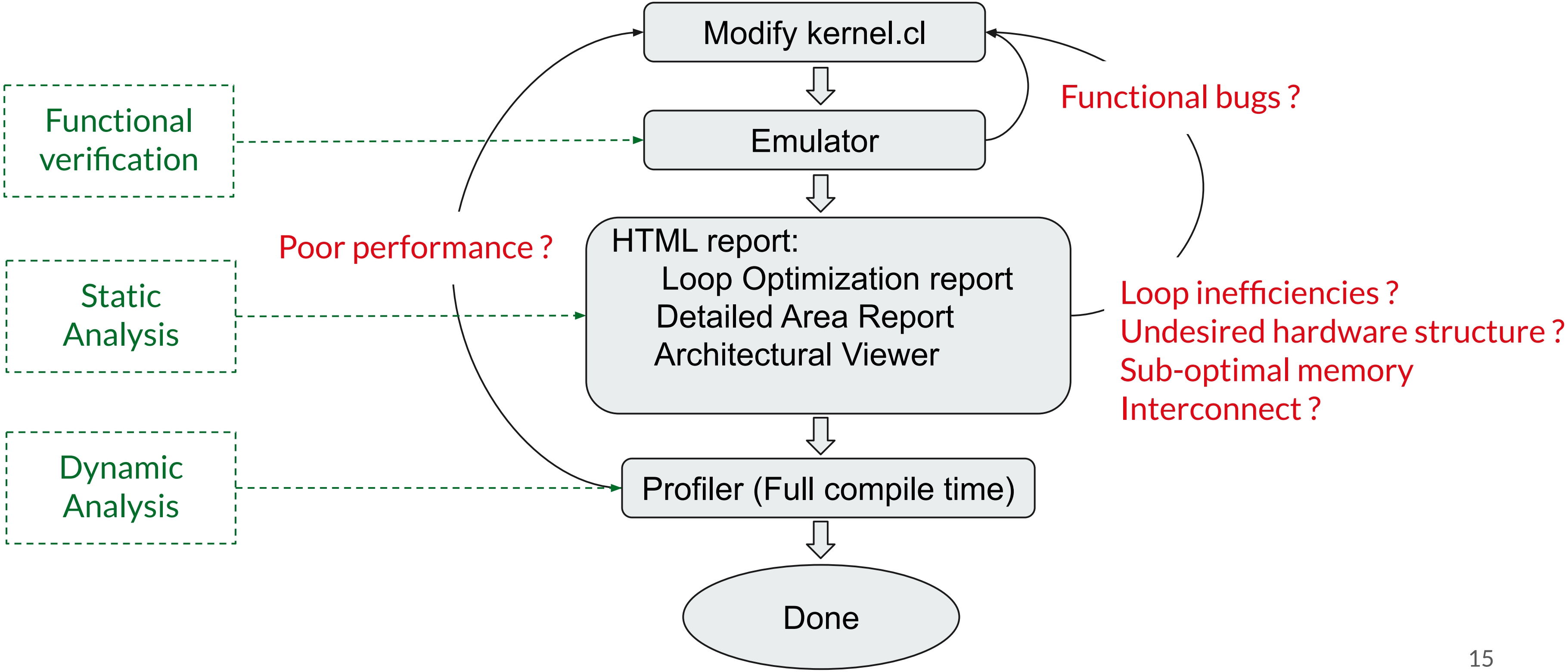
⇩

kernel.aocx

⇩

(source:Intel)

Compilation workflow

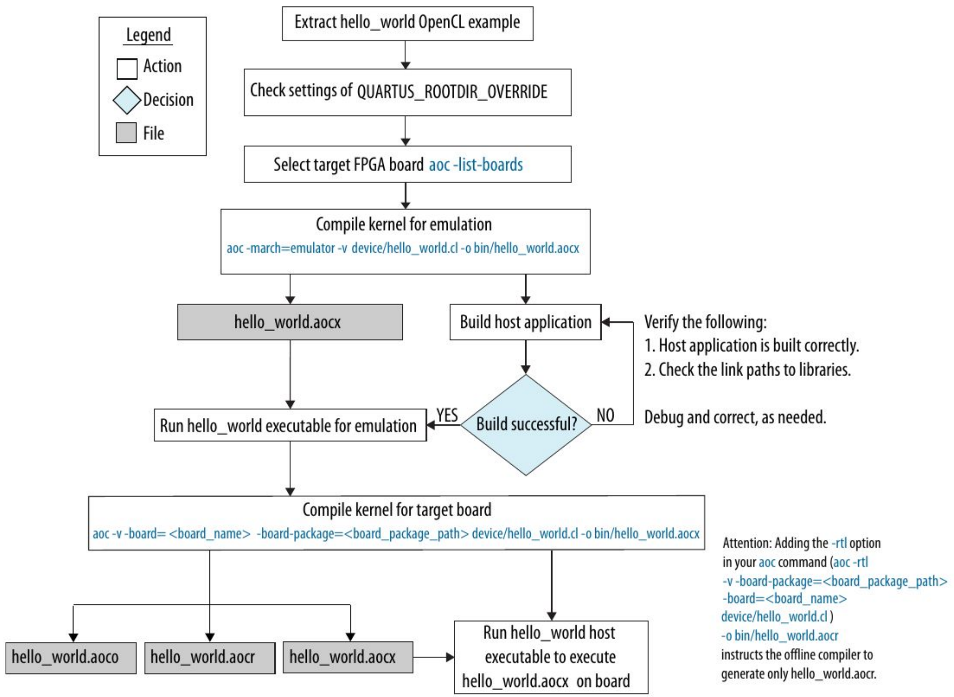


UNIVERSITÉ DU LUXEMBOURG

AOC command (most common)

Options	Description
-report	Print area estimates to screen
-c	Creates .aoco object fickle and bootstrap an Intel Quartus Prime design project
-o <file>	Use to specify a non-default name for the output file
-I <directory>	Adds <directory> to header search path
-L <directory>	Adds <directory> to library search path
-l <library.aoclib>	Specify OpenCL library file
-D <name>	Defines a macro called <name>
-march=emulator	Create kernels that can be executed and debugged on the host without the board
-profile	Enable profile support when generating aocx file

Compilation workflow



(source: Intel)

Writing OpenCL programs

On the host

Platforms IDs

```
cl_int clGetPlatformIDs(cl_unit num_entries,
                        cl_platform_id *platforms,
                        clu_uint *numplatforms)
```

- Obtain the list of available platforms
- Parameters:
  - `cl_unit num_entries` : size of platforms
  - `cl_platform_id *platforms` : list of platform IDs
  - `clu_uint *numplatforms` : total number of platforms available

- Return :
  - `cl_int` : error code

Device IDs

```
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 available devices supported by the platform
- Parameters:
  - `cl_platform_id platform` : platform to look in
  - `cl_device_type device_type` : Device types
    - CPU, GPU, FPGA, Default, All
  - `cl_uint num_entries` : size of devices
  - `cl_device_id *devices` : list of devices IDs
  - `cl_uint *num_devices` : total number of available devices

- Return :
  - `cl_int` : error code

Device IDs

```
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 available devices supported by the platform
- Parameters:
  - `cl_platform_id platform` : platform to look in
  - `cl_device_type device_type` : Device types
    - CPU, GPU, FPGA, Default, All
  - `cl_uint num_entries` : size of devices
  - `cl_device_id *devices` : list of devices IDs
  - `cl_uint *num_devices` : total number of available devices
- Return :
  - `cl_int` : error code

Create the context

```
cl_context clCreateContext(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);
```

- Create and return a context
- `void CL_CALLBACK *pfn_notify` : callback to handle errors in the context

Host code setup workflow

- Call `clGetPlatformIDs` to get available platforms
- Allocate space to hold platform information
- Call `clGetPlatformIDs` to create a list of platforms
- Call `clGetDeviceIDs` to get the number of available devices in the selected platforms
- Allocate space to hold device information
- Call `clGetDeviceIDs` to create a list of devices
- Call `clCreateContext` to create a context managing kernel execution

Platform example code

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/opencl.h>
...
// Get openCL first platform
cl_platform_id fpga_paltform = NULL;
clGetPlatformIDs(1, &fpga_paltform, NULL);

// Get openCL first device
cl_device_id fpga_device = NULL;
clGetDeviceIDs(fpga_paltform, CL_DEVICE_TYPE_ACCELERATOR, 1, &fpga_device, NULL);

// Create a context.
cl_context context = clCreateContext(NULL, 1, &fpga_device, NULL, NULL, NULL);
```

The command queue

- The host request action by the device through a command queue
- Each queue is linked to single device but devices may have several queues
- Host submits commands for example:
  - Write to device
  - Execute kernel
  - Read from device
- Operations in the queue will execute in-order for Intel FPGA

Create a command queue

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

- Parameters:
  - `cl_context *context`: previously generated context
  - `cl_device_id device`: device associated with the context
  - `cl_command_queue_properties properties`: queue properties (ex: profiling)
  - `cl_int *errcode_ret` : error code
- Returns:
  - `Cl_command_queue` : the generated command queue

Memory transfer management

- Host and device have their own physical memory
- Data must be move from the host to the device before kernel execution
- The host is in charge to retrieve the result of the computation after kernel execution
- OpenCL provide functions to allocate, transfer and free memory
    - These functions generate memory objects sent through the command queues

UNIVERSITÉ DU LUXEMBOURG

Memory objects

- Host and device have their own physical memory
- Data must be move from the host to the device before kernel execution
- The host is in charge to retrieve the result of the computation after kernel execution
- OpenCL provide functions to allocate, transfer and free memory
  - These functions generate memory objects sent through the command queues
  - Memory objects ⇔ Data encapsulation
- Two types of memory objects defined by OpenCL specification
  - Buffers (Onde dimensional collection of elements)
  - Images (single or array)

Create a buffer (one dimension collection)

```
cl_mem clCreateBuffer(cl_context *context,
                      cl_mem_flags flags,
                      size_t size,
                      cl_int *errcode_ret);
```

- Can be scalar (int, char, float), vector data types, or structures
- In the host, dereferencing memory object is not allowed -> transformed to pointer inside the kernel
- Parameters:
  - `cl_context *context`: a valid context
  - `cl_mem_flags flags`: special flags
  - `size_t size`: size in bytes
  - `void *host_ptr`: size in bytes
  - `cl_int *errcode_ret` : error code
- Returns:
  - `cl_mem` : memory object

Data transfers with the command queue (write)

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

Valid command queue

Destination buffer

Blocking if set to CL_TRUE

Offset in bytes in the buffer

Size in bytes of data to be written

Source host pointer

Events for synchronization

Data transfers with the command queue (read)

```
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);
```

Valid command queue

Destination buffer

Blocking if set to CL_TRUE

Offset in bytes in the buffer

Size in bytes of data to be written

Source host pointer

Events for synchronization

Buffer example code

```
// Host side data
int *host_din, *host_dout;
// Align mem in order to use DMA -- POSIX
posix_memalign((void **)(&host_din), AOCL_ALIGNMENT , sizeof(int)*N);
posix_memalign((void **)(&host_dout), AOCL_ALIGNMENT, sizeof(int)*N);
for(int i=0; i<N; i++){
   host_din[i] = i;
   host_dout[i] = 0;
}
// Create memory Object
cl_mem dev_din = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int)*N, NULL, NULL);
cl_mem dev_dout = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int)*N, NULL, NULL);
// FPGA side data
clEnqueueWriteBuffer(queue, dev_din, CL_TRUE, 0, sizeof(int)*N, host_din, 0,NULL, NULL);
// Execute kernel
…
// Read data from FPGA
clEnqueueReadBuffer(queue, dev_dout, CL_TRUE, 0,sizeof(int)*N, host_dout, 0, NULL, NULL);
```

Discussed later

Create a Program

- A program is a set of kernels (functions)
  - Either source code
  - Or precompile binary
- GPU/CPU vendors supports `clCreateProgramWithSource(...)`
  - Online compilation of kernels
- Intel FPGA only support pre-compiled binaries using `clCreateProgramWithBinary(...)`
  - Binary implementation is vendor specific
  - Aocx files supported (FPGA image)

Create programs from a binary (Intel FPGA)

Valid context

```
cl_program clCreateProgramWithBinary(cl_context context,
                                     cl_uint num_devices,
                                     const cl_device_id *device_list,
                                    const size_t *lengths,
                                    const unsigned char **binaries,
                                     cl_int *binary_status,
                                    cl_int *errcode_ret);
```

Number of devices

List of valid devices

Lengths of the binary

Content of the aocx binary

Status of the binary loading

Error code

Create programs from a binary (Intel FPGA)

Error code

Valid program

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

Number of devices in device_list

List of devices

Build option

Callback function

Data for callback function

- Intel FPGA only supports pre-compiled binaries :
    - `clBuildProgram` is not doing anything concrete
    - But mandatory in order to conform to the standards

Create the kernel

Kernel object encapsulates the kernel function

Valid program

```
cl_kernel clSetKernel(cl_program program,
                      const char *kernel_name,
                      cl_int *errcode);
```

Kernel function name

Error code

- Intel FPGA can load any compiled kernels from the aocx file

Set kernel arguments

Error code

Kernel object

```
cl_int clSetKernelArg(cl_kernel kernel,
                      cl_uint arg_index,
                      size_t arg_size,
                      Const void *arg_val);
```

Argument index
From 0 (leftmost arg) to N-1 (N is total number of args)

Argument index
From 0 (leftmost arg) to N-1 (N is total number of args)

Pointer to data used as argument value

- Careful with the arg_index -- no checking is done

Execute kernel

Error code

Kernel object

```
cl_int clEnqueueTask(cl_command_queue command_queue,
                     cl_kernel kernel,
                     cl_uint num_events_in_wait_list,
                     const cl_event *event_wait_list,
                     cl_event *event);
```

Kernel to be executed

Events for synchronization

- Alternative `clEnqueueNDRangeKernel` for NDRange kernel

UNIVERSITÉ DU LUXEMBOURG

Kernel execution example

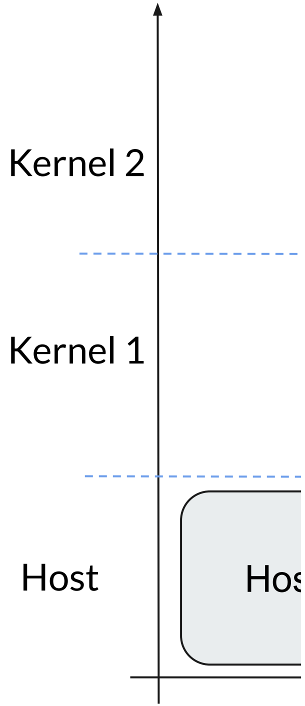
```c
// Read FPGA binary
size_t length = 0x10000000;
unsigned char *binary = (unsigned char *)malloc(length);
FILE *fp = fopen("first_kernel.aocx", "rb");
fread(binary, length, 1, fp);
fclose(fp);

// Create program
cl_program program = clCreateProgramWithBinary(context, 1, &fpga_device, &length, (const unsigned char **)&binary, NULL, NULL);

// Build program
clBuildProgram(program,1,&fpga_device,NULL,NULL,NULL);

//Create kernel.
cl_kernel kernel = clCreateKernel(program, "first_kernel", NULL);
```

Kernel execution example

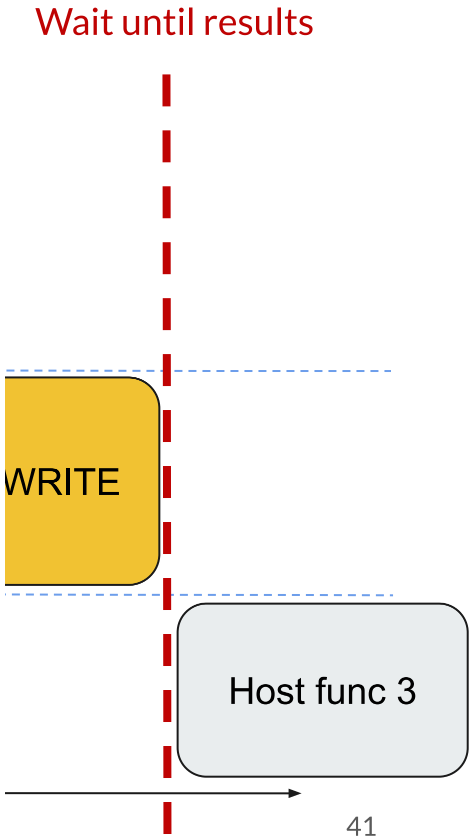
```
// Creating buffers
…
// Execute kernel
clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_din);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_dout);
cl_event kernel_event;
clEnqueueTask(queue, kernel, 0, NULL, &kernel_event);
clWaitForEvents(1, &kernel_event);
```

- `clEnqueue…` creates asynchronous tasks
  - Non-blocking functions
  - Need explicit synchronization

Need for synchronization




```
cl_command_queue queue1, queue2;
cl_event kernel_event1, kernel_event2;

clEnqueueTask(queue1,kernel1,...,&kernel_event1);
clEnqueueTask(queue2,kernel2,...,&kernel_event2);

cl_event waiting_events_list[2];
waiting_events_list[0]=e1;
waiting_events_list[1]=e2;

clEnqueueTask(queue1, kernel3,2,waiting_events_list,NULL);

clFlush(queue1);
clFlush(queue2);

clFinish(queue1);
clFinish(queue2);
```



Synchronization (host)

- `clFinish(queue)` blocks until all commands have finished their execution
    - Use `clFlush(queue)` to empty it
- Blocking memory commands (Read/Write Buffer)
- Each `clEnqueue` tasks can wait on a list of events and generate a new event id as prerequisite

```
cl_int clEnqueueTask(cl_command_queue command_queue,
                     cl_kernel kernel,
                     cl_uint num_events_in_wait_list,
                     const cl_event *event_wait_list,
                     cl_event *event);
```

Number of events to wait

Wait for these events to finish

Generate a new event

Clean up everything

```
// Free all dynamic memory objects
clReleaseEvent(kernel_event);
clFlush(queue);
clFinish(queue);

clReleaseMemObject(dev_din);
clReleaseMemObject(dev_dout);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
free(host_din);
free(host_dout);
```

Writing OpenCL programs

On the device

Kernel code

```
__kernel void my_kernel (__global float *data){...}
```

- Starts with the keyword : `__kernel`
- Returns `void`
- Data qualifiers can be `__global`, `__private`, `__local` or `__constant`
- Restrictions:
  - No pointers to functions
  - No recursion
  - No predefined identifiers
  - No writable static variables
- Types:
  - Scalars (device): char, ushort, uint, long, float, etc ... (for the host, it is recommended to prefix with `cl_`)
  - Image: image2d_t, image3d_t, sampler_t
  - User-defined structures

Vector type

```
__kernel void my_kernel (__global int8 *data){...}
```

- Supported size are: 2, 3, 4, 8, 16
- Vector types are available on both host and device
  - Ex: (for kernel: float16; for host: cl_float16)
- Aligned at vector length
- Use vector or components operations

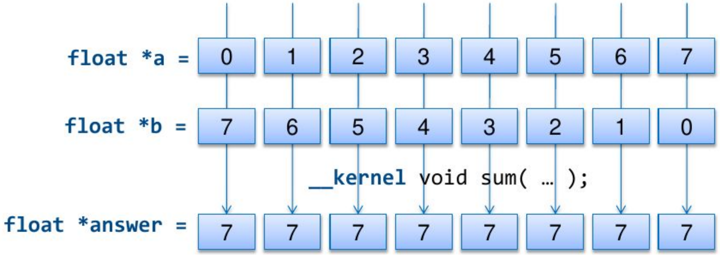
```
int8 x,y,z;
z = x + y;
```

```
int4 x,y,z;
z.s0 = x.s0 + y.s0; z.s1 = x.s1 + y.s1;
z.s2 = x.s2 + y.s2; z.s3 = x.s3 + y.s3;
z.s4 = x.s4 + y.s4; z.s5 = x.s5 + y.s5;
z.s6 = x.s6 + y.s6; z.s7 = x.s7 + y.s7;
```

NDRange kernels

- Multiple Data-parallel threads
- Kernel is executed as a single program multiple data (SPMD) fashion
    - One thread calls a work-item
- Aligned at vector length
- Fine-grained parallelism

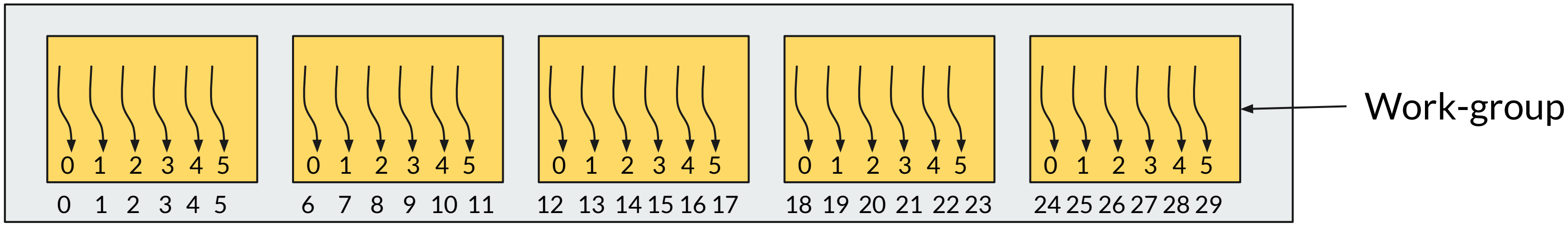
```
__kernel void
sum(__global const float *a,
    __global const float *b,
    __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```



(source:Intel)

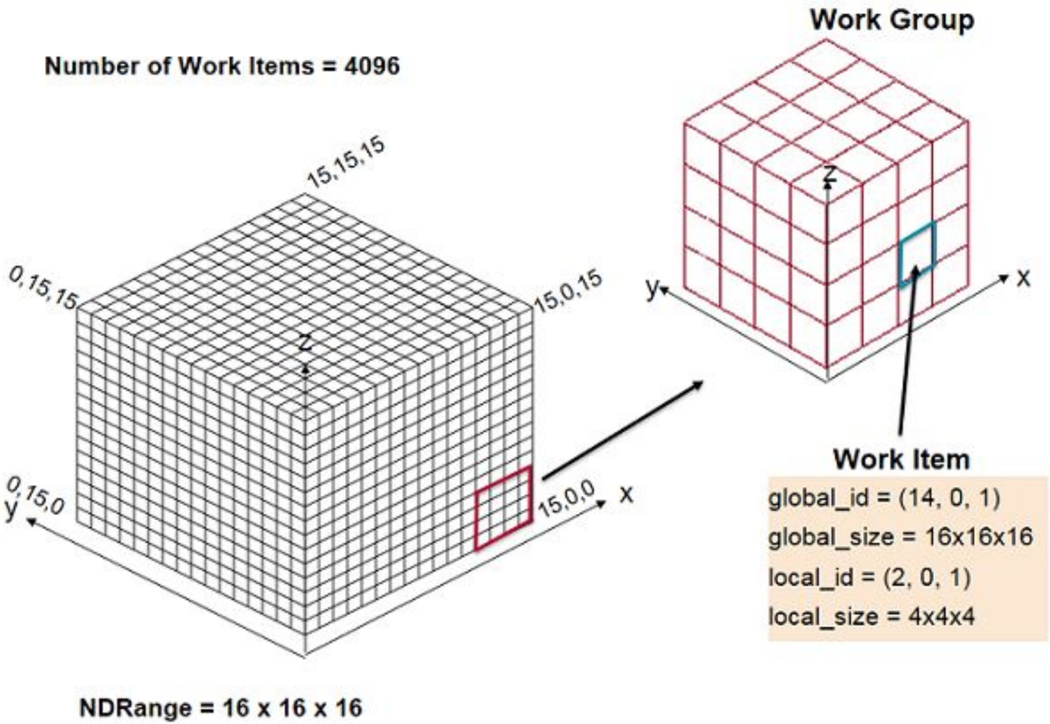
Thread IDs

- Each thread has information about its ids (global, group, local)
  - Global → `get_global_id(0)`
  - Group → `get_group_id(0)`
  - Local→ `get_local_id(0)`
- Threads/work-items are partitioned into work-groups
  - Only threads within the one work-group can share local memory




```
get_local_id(0) = get_group_id(0) * get_local_size(0) + get_local_id(0)
```

NDRange in 3D



(source:xilinx)

Workgroup

- Partitioned the NDRange into small chunks of work-items sharing local memory
- Can have maximum 3 dimensions like NDRange
- Global Dim (NDRange) must be evenly divisible by workgroup size
- Synchronization between work-items only possible inside workgroups

clEnqueueNDRangeKernel

Error code

Command queue

```
cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,
                              cl_kernel kernel,
                              int 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);
```

Kernel to be executed

Number of dimensions

Starting global ID (NULL={0,0,0})

Number of total work-items in each dimensions

Workgroup size in each dimension

Example

```
//1D Work-Group
int err;
size_t const globalWorkSize = 1920;
size_t const localWorkSize = 8;
err=clEnqueueNDRangeKernel(queue, 1dkernel, 1, NULL, &globalWorkSize, &localWorkSize,
            0, NULL, NULL);

//3D Work-Group
size_t const globalWorkSize[3] = {512,512,512};
size_t const localWorkSize[3] = {16, 8, 2};
err=clEnqueueNDRangeKernel(queue, 3dkernel, 3, NULL, globalWorkSize, localWorkSize,
            0, NULL, NULL);
```

(source:Intel)

Single Work-item

- Equivalent to launch kernels with NDRange of (1,1,1)
- Define as a Task in OpenCL
- Loops in single-work items are automatically parallelized by the Intel FPGA compiler
- Why:
  - NDRange difficult for problems with dependencies (e.g. filters, compression algorithms)
  - Difficulties to partitioned into workgroups
- More similar as a C program
- Ideal to port CPU applications

Loop pipelining

```
#define N 64
#define M 256
__kernel void loop_kernel ( __global const int * restrict a,
             __global int * restrict b, __global int * restrict res)
{
   local int shift_register[N]
   #pragma unroll
   for(unsigned int i=0; i<N; i++)
     shift_register[i]=0
   for(unsigned int k=0; k<M; k++){
       #pragma unroll
       for(unsigned int i=0; i<N; i++)
           shift_register[i+1]=shift_register[i]
       shift_register[N-1] =  a[k]
       for(unsigned int i=0; i<N;i++)
           res+= shift_register[i]*b[i]
   }
}
```

Dependency

Independent reduction