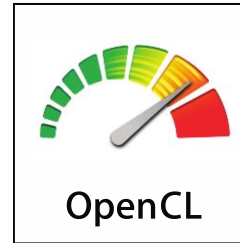


TOWARDS "WRITE ONCE, RUN ANYWHERE"

Miguel Astrain Etxezarreta
Universidad Politécnica de Madrid (UPM)

Acknowledgements:
Mariano Ruiz, Antonio Carpeño and Sergio Esquembri of the Universidad Politécnica de Madrid (UPM)

INSTRUMENTATION & APPLIED ACOUSTICS RESEARCH GROUP

miguel.astrain@i2a2.upm.es

UPM BIG SCIENCE

POLITÉCNICA

CERN School of Computing

Agenda

Introduction (1h)
- Context
- The OpenCL Models
- OpenCL Host Programming
- OpenCL Kernel Programming
- FPGA Oriented Kernel Design

Advanced (1h)
- More on OpenCL host programming
- Example: Matrix Multiplication
- More on FPGA Oriented kernel design
- Memory Hierarchy
- Synchronization in OpenCL
- The OpenCL Event Model
- Building OpenCL

Additional materials

In addition to these slides, and the set of examples, it is useful to have:

- **OpenCL in Action. Matthew Scarpino. Manning 2011. (Very recomended)**
- Heterogeneous Computing with OpenCL. Gaster, Howes, et al Morgan Kaufmann 2011.
- OpenCL Parallel Programming Development CookBook. Raymond Tay. PACKT 2013.

Vendor specific documentation for each platform:

- Xilinx VITIS Softwate platform
  - https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/index.html
- Intel FPGA SDK for OpenCL
  - https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html

Before we start…

Three main profiles that will benefit from this course:

1. **Scientific background**: Scientific experiments, Data Acquisition, some hardware…
2. **Programmer background:** Programming frameworks, some parallelization, GPUs….
3. **Hardware background:** HDL, FPGAs, DSP, some programming….

No knowledge is required from the other fields.

The goal… **understanding the heterogeneous programming model.**

1. Lower learning curve to write high-performance algorithms for hardware.
2. Apply programming concepts to hardware. Integrate many hardware in a unified framework.
3. Optimizations thanks to hardware knowledge, provide tools for the above.

CERN School of Computing

The very basics

Very recommended to review the CSC2019 lectures on these topics:

1. Software Design in the Many-Cores era; A. Gheata, E. Tejedor.
2. Base Concepts of Parallel Programming: A Pragmatic Approach; A. Gheata, E. Tejedor.

Over-summarized, the frameworks for high-performance computing help on:

1. Dividing tasks into smaller problems.
2. Managing the tasks (and memory!) to execute efficiently.

But OpenCL offers one more perk, managing heterogeneous hardware:

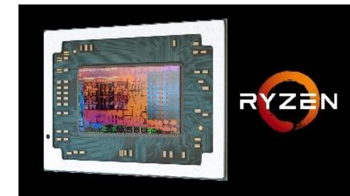
1. Additional to parallelism I can select optimal hardware! → More performance!
2. A single computing language can manage many devices! → Ease of maintenance, portability!
3. An open standard, which is manufacturer agnostic →Write once, run anywhere!

It’s a Heterogeneous world

- OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform

A modern computing platform includes:

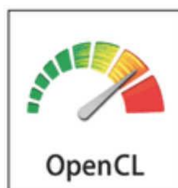
- One or more CPUs
- One or more FPGAs
- One of more GPUs
- DSP processors
- FPGA with AI Cores + DSP + PCIe +DDR
- ADCs?! (Xilinx RFSOC) …



OpenCL – Open Computing Language

Implementers
Desktop/Mobile/Embedded/FPGA

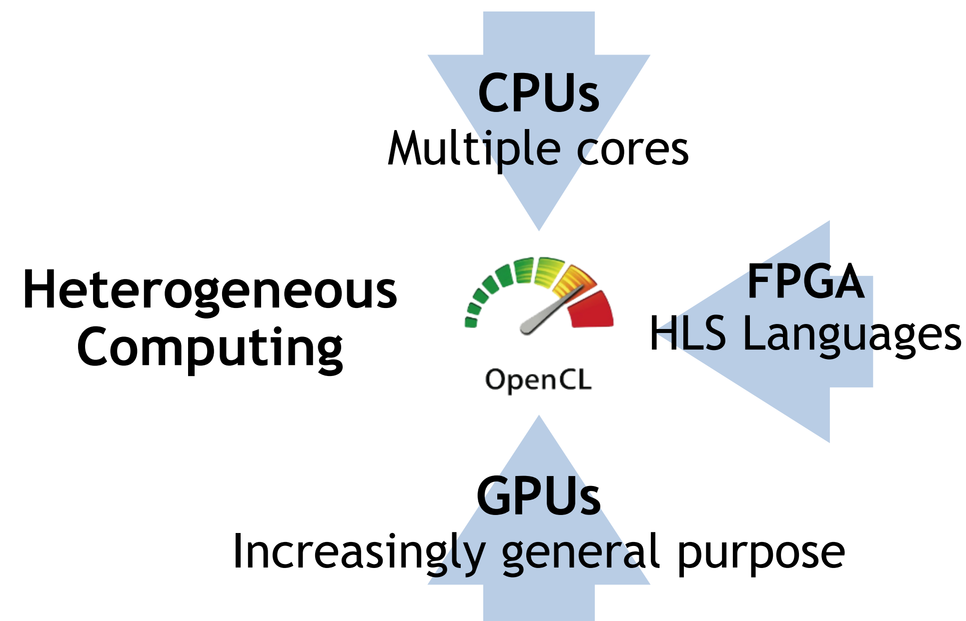
Single Source C++ Programming



Core API and Language Specs

Portable Kernel Intermediate Language

Working Group Members
Apps/Tools/Tests/Courseware

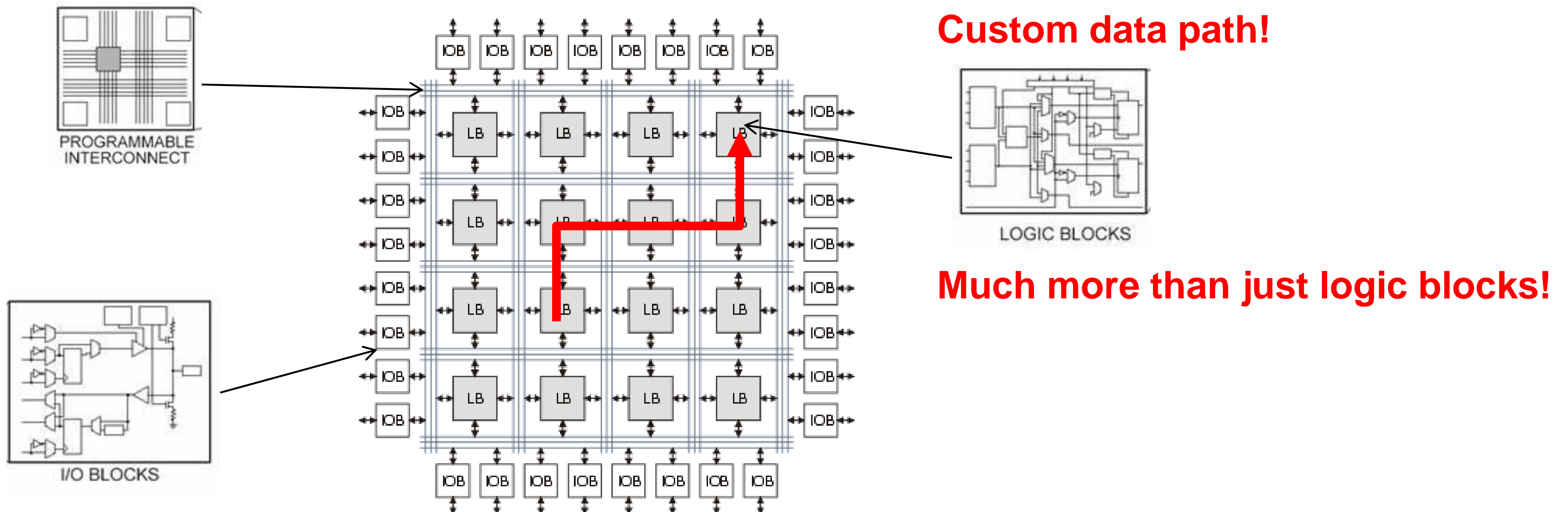


Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, FPGAs, DSPs,…

OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group.

FPGAs?

- Very interesting hardware, with real-time capabilities.
- While often losing to GPUs in raw computation muscle, FPGAs are more energy efficient!
- The learning curve for HDLs is steep, what do I need to know about FPGAs to use OpenCL?
  - Actually, very little! Knowledge helps, of course, but the tools provided with OpenCL SDKs help us!



THE OPENCL MODELS

There are quite a few.

CERN School of Computing

OpenCL Models

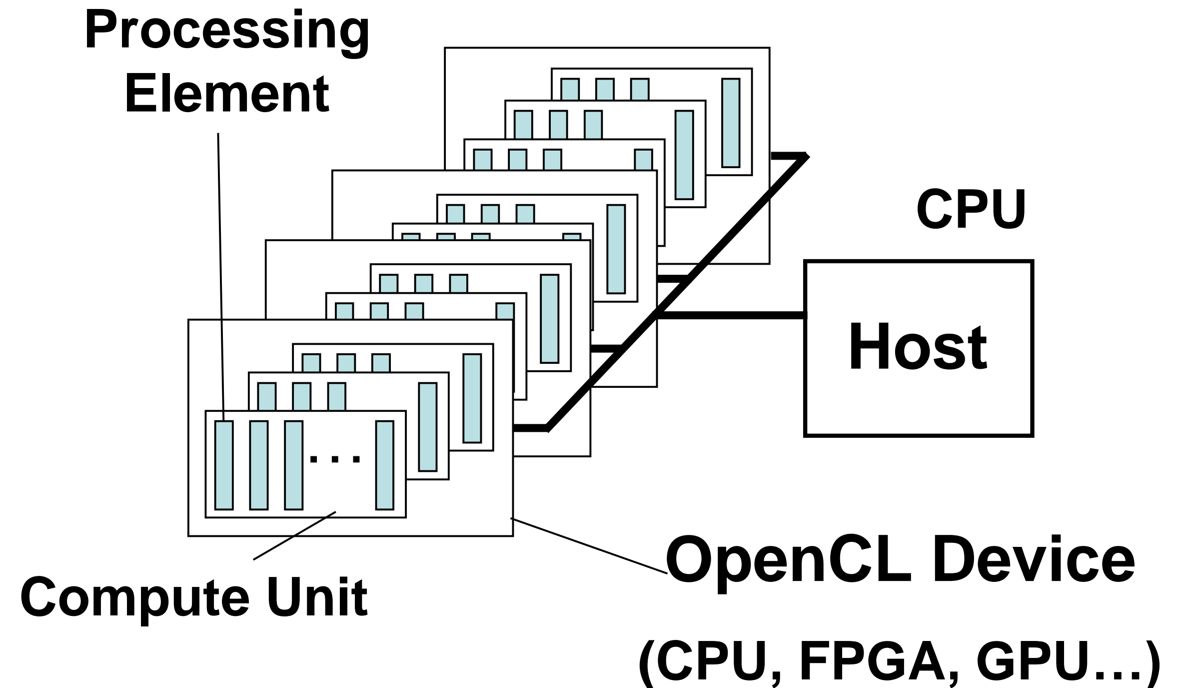
OpenCL defines a set of models to organize the core ideas:

- Platform Model
- Execution Model
- Memory Model
- Programming Model

from: Khronos registry, OpenCL specification 2.2
276 pages of condensed information!

OpenCL Platform Model

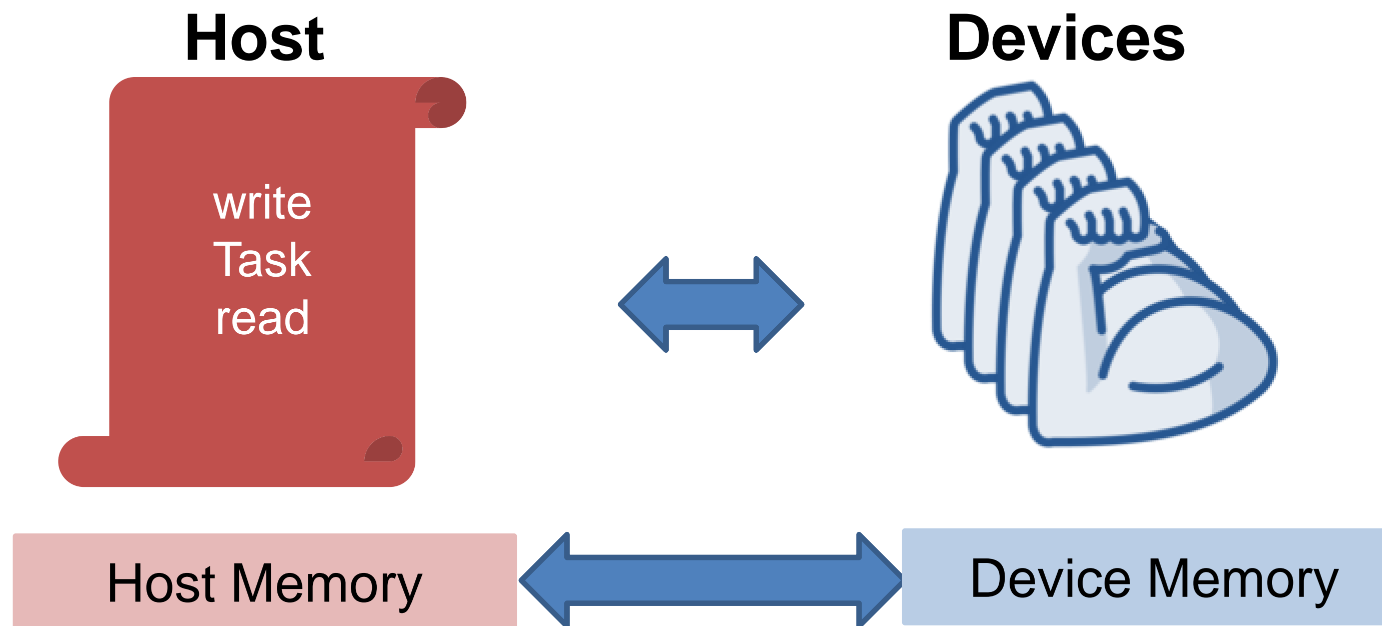
- One *Host* and one or more OpenCL *Devices.*
  - Each OpenCL Device is composed of one or more *Compute Units.*
    - Each Compute Unit is divided into one or more *Processing Elements.*



OpenCL keywords are high-lightened in RED hereinafter.

Host and Device

- The host has the recipe on how to perform the computation
  - It uses commands to the device to do so.
- The device has the power to perform the computation.
  - Can only understand kernel code.
- Memory divided into *Host Memory* and *Device Memory.*



OpenCL Execution Model

- Define a problem domain and execute an instance of a **kernel** for each point in the domain.
  - The smallest unit is called a work-item
- If the problem needs synchronization or has dependencies, manage them into work-groups

C/C++ code:

```
void times_two( float* input,
float* output)
{
   int i =
some_identifier_routine();
   output[i] = 2.0f * input[i];
}
```

OpenCL code:

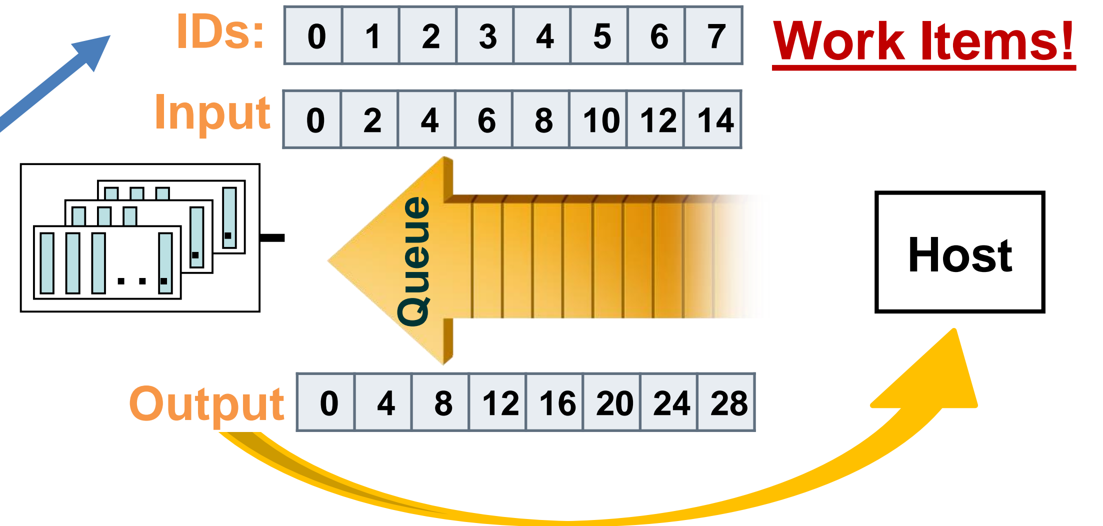
```
__kernel void times_two(
    __global float* input,
    __global float*
output)
{
   int i =
get_global_id(0);
   output[i] = 2.0f *
input[i];
}
```

OpenCL

OpenCL qualifiers

OpenCL Execution Model

```
__kernel void times_two(
        __global float* input,
        __global float*
output)
{
    int i = get_global_id(0);
    output[i] = 2.0f *
input[i];
}
```



- Note there is no for loop.
- This is very GPU like programming.
- No imposed order of execution

The idea behind OpenCL

- Computation divided into simpler functions (a kernel) executing at each point in a problem domain.
- Most of the computing muscle is usually needed in a few lines of code.
- Typical example: 1024x1024 image with one kernel invocation per pixel or 1024x1024 = 1,048,576 kernel executions

Traditional loops

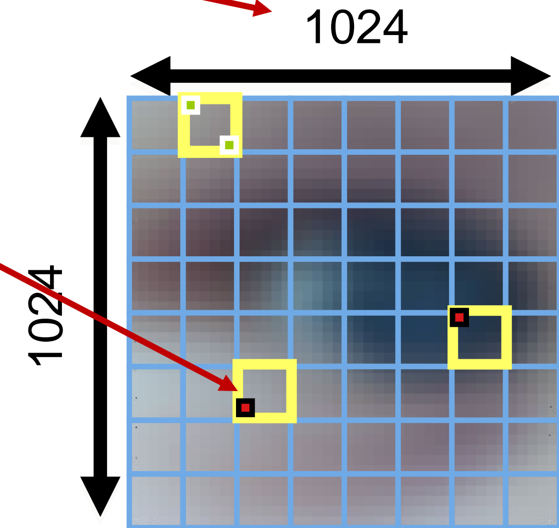
```
void multiply(const int n, const float *a,
            const float *b, float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
__kernel void multiply (__global const float *a,
                        __global const float *b,
                        __global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
// many instances of the kernel
// called work-items execute
in parallel, but how much parallelism? ...
```

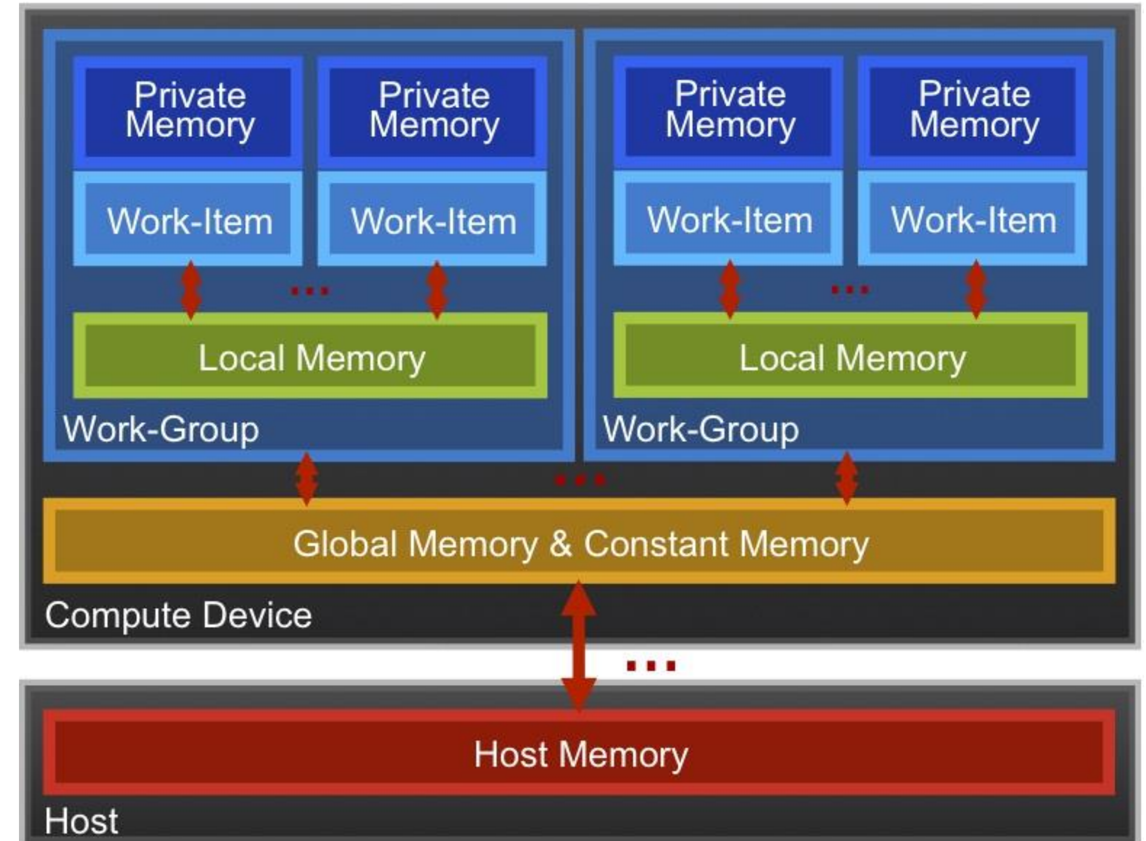
Example: N-dimensional domain

- Typical example, an image:
- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (**work-group**, executes together)
- The work-item is the pixel.
- You can tune the dimensions that are “best” for your hardware.
- Remember Platform Model “CU” and “PE”.



OpenCL Memory model

- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory*
  - Visible to all work-groups
- *Host Memory / Constant*
  - On the CPU

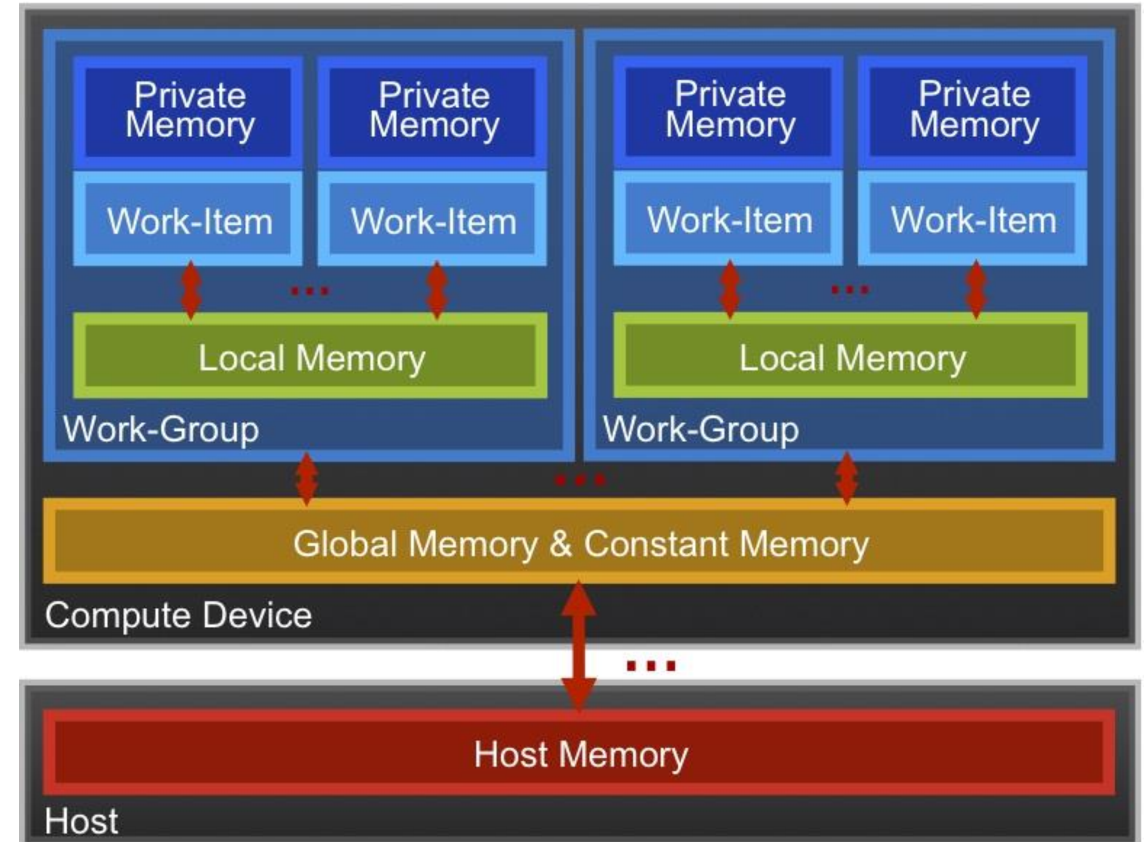


https://www.khronos.org/

OpenCL Memory model

Memory management is **explicit!**.

More on this and SYCL later…



https://www.khronos.org/

OpenCL Programming Model

Definitions, lots of definitions… What does it all mean?

- They help keep ideas clear.
- Divide and manage the problem the OpenCL way!
- The programming model is … the rest …
- Your problem, your algorithm, your hardware

OpenCL Programming Model

- Data Parallelism
- Task Parallelism
- Single Instruction Multiple Data
- Single Program Multiple Data

- kernels and indexes
  - Work-items
- kernels and queues
  - Work-Group
- OpenCL C
  - Vector instructions.
- Platforms and devices
  - Deploy to multiple devices

OPENCL HOST PROGRAMMING

What is your favorite language?

CERN School of Computing

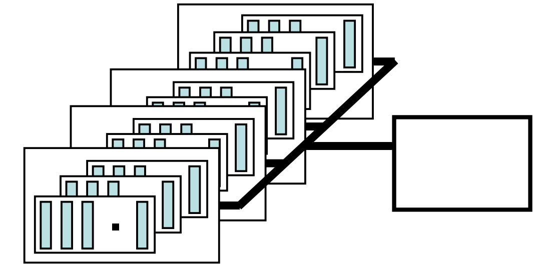
Host Programming

- There is now a full specification in C++.
- Host bindings are available for C, C++, Java, Python.
- Kernels are written in **OpenCL C subset of C99** with specific extensions and restrictions.

- **Recommend using C/C++. Most examples are written in C.**
- **Lots of development effort in C++, SYCL,…**

Host Programming

- The host program is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels



- 5 simple steps in a basic host program:
  1. Define the ***platform*** … platform = devices + context + queues
  2. Create the ***program*** (dynamic library for kernels)
  3. Setup ***memory*** objects
  4. Define the ***kernel*** (attach arguments to kernel functions)
  5. Submit ***commands*** … transfer memory objects and execute kernels

CERN School of Computing

Host Program

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
                        CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetConte                                          &cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);
```

Define platform and queues

```
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_                                          cb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                              sizeof(cl_float)*n, NULL, NULL);
```

Define memory objects

```
// create the program
program = clCreateProgramWithSource(context, 1,
                                                NULL, NULL);
```

Create the program

```
// build the program
err = clBuildProgra                                L,NULL);
```

Build the program

```
// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                          sizeof(cl_mem));
err |= clSetKer                                    [1],

err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                          sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;
```

Create and setup kernel

```
// execute kernel
err = clEnqueueNDR                                 1, NULL,
          global_work_size, NULL,0,NULL,NULL);
```

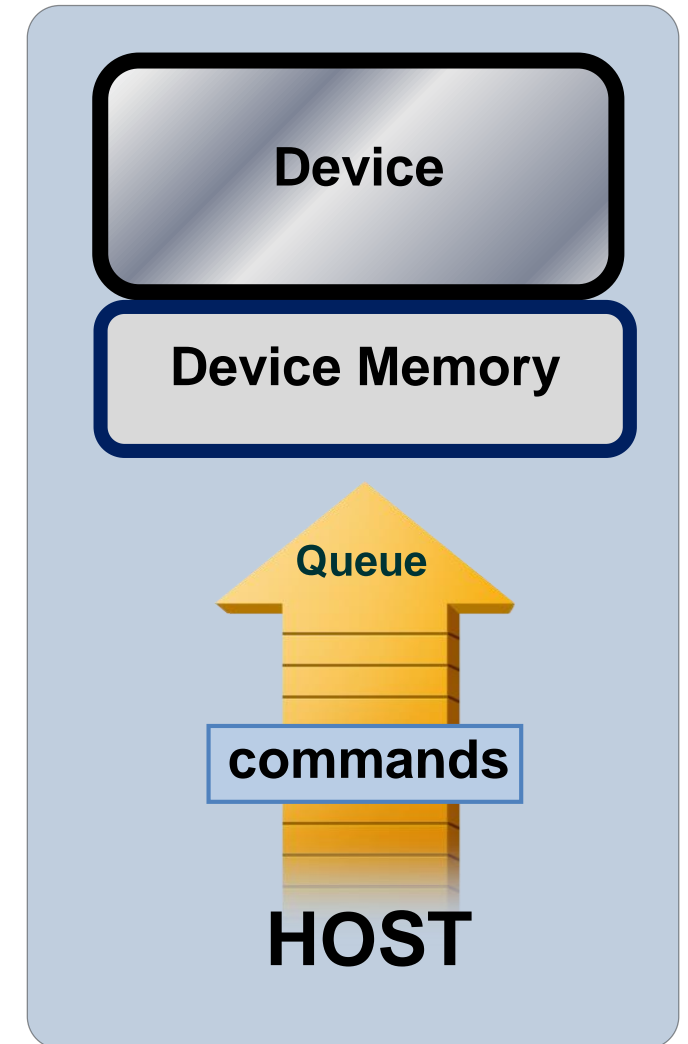
Execute the kernel

```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
          CL_TRUE, 0,

          0, NULL, NULL);
```

Read results on the host

Command-Queues

- The computation "recipe" is scheduled through the command-queue.
- ***Commands*** for a device include kernel execution, synchronization, and memory transfer operations.



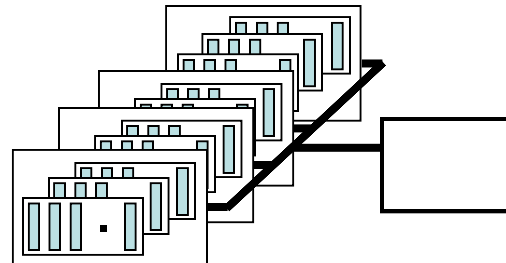
Example: step by step

Lets analyse a host program:

1. **Platform:**

Ex:
1.
2.
3.
4.
5.

Device:
CPU, Intel x86, OpenCL C



Host:
CPU, Linux, C language

The code ahead might be simplified or wrong to keep it shorter and readable.

CERN School of Computing

Example: step by step

```
cl_uint num_of_platforms = 0;
clGetPlatformIDs(0, 0, &num_of_platforms);
cl_platform_id* platforms = new cl_platform_id[num_of_platforms];
clGetPlatformIDs(num_of_platforms, platforms, 0);
----------------------------------------------------------------------------
clGetPlatformInfo( platforms[i], CL_PLATFORM_NAME, 0, 0,  platform_name_length);
clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, platform_name_length,
platform_name, 0);
cl_platform_id platform = platforms[selected_platform_index];
```

Ex:

1.
2.
3.
4.
5.

```
Number of available platforms: 1
Platform names:[0] Intel(R) OpenCL [Selected]
```

CERN School of Computing

Example: step by step

```
cl_uint cur_num_of_devices;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 0, 0, &cur_num_of_devices);

cl_device_id* devices_of_type = new cl_device_id[cur_num_of_devices];
clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, cur_num_of_devices,
devices_of_type, 0);

cl_uint device_index = 0;
cl_device_id device = devices_of_type[device_index];
```

Ex:
1.
2.
3.
4.
5.

```
CL_DEVICE_TYPE_CPU: 1
```

```
CL_DEVICE_TYPE_CPU[0]
CL_DEVICE_NAME: Genuine Intel(R) CPU @ 2.60GHz
CL_DEVICE_AVAILABLE: 1
CL_DEVICE_VENDOR: Intel(R) Corporation
```

CERN School of Computing

Example: step by step

- Create a simple context with a single device:

```
cl_context clCreateContext(cl_context_properties *properties, cl_uint
num_devices, cl_device_id *devices, (void CL_CALLBACK
*notify_func)(…), void user_data, cl_int *error);
```

Ex:

1. ✓
2.
3.
4.
5.

```
context = clCreateContext(NULL, 1, &device_id, NULL,
NULL, &err);
```

- Create a simple command-queue to feed our device:

```
cl_command_queue clCreateCommandQueue(cl_context context,
cl_device_id device_id, cl_command_queue_properties properties,
cl_int *error);
```

```
q_commands = clCreateCommandQueue(context, device_id, 0,
&err);
```

CERN School of Computing

Example: step by step

```
const char* raw_text = &program_text_prepared[0];
cl_int err;
cl_program program = clCreateProgramWithSource(context,
1, &raw_text, 0, &err);

clBuildProgram(program, (cl_uint)num_of_devices, devices,
build_options.c_str(), 0, 0);
```

Ex:

1. ✓
2. ✓
3.
4.
5.

```
Build program options: "-DT=float -DTILE_SIZE_M=1 -DTILE_GROUP_M=16
-DTILE_SIZE_N=128 -DTILE_GROUP_N=1 -DTILE_SIZE_K=8"
```

CERN School of Computing

Example: step by step

```
cl_kernel krnl = 0;
string kernel_name = “Multiply”
krnl = clCreateKernel(program, kernel_name.c_str(), &err);
```

Ex:
1. ✓
2. ✓
3.
4. ✓
5.

- As we will see later, kernels are really like functions.
- They have arguments. But must return void.
- They are identified by name for OpenCL.
- Remember that kernels are compiled for the device architecture.

CERN School of Computing

Example: step by step

- HOST MEMORY BUFFER:

```
cl_float* data_ptr = (cl_float *) malloc(sizeof(cl_float) * count);
cl_mem array_of_floats = clCreateBuffer(context, CL_MEM_READ_WRITE,
sizeof(cl_float)*count, data_ptr, NULL);
```

- KERNEL ARGUMENTS :

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &array_of_floats);
```

Ex:

1. ✓
2. ✓
3. ✓
4. ✓
5.

Example: step by step

Remember the variables:

```
cl_command_queue q_commands;
cl_kernel krnl;
cl_mem array_of_floats;
```

Ex:

1. ✓
2. ✓
3. ✓
4. ✓
5. ✓

```
1. clEnqueueWriteBuffer(q_commands, array_of_floats, CL_FALSE, 0,
   sizeof(cl_float)*count, data_ptr,0, NULL, NULL);

2. clEnqueueTask(q_commands, krnl, 0, NULL,NULL);
```

NO dimensions!

```
3. clEnqueueReadBuffer(q_commands, array_of_floats, CL_TRUE,
   sizeof(cl_float)*count, data_ptr, 0, NULL, NULL);
```

OPENCL KERNEL PROGRAMMING

CERN School of Computing

OpenCL Kernel Programming

- Derived from **ISO C99 + ISO C11**
  - A few ***restrictions***: no recursion, function pointers,...
  - Preprocessing directives defined by C99 are supported (#include etc.)
- Built-in data types
  - Scalar and vector data types, pointers
  - Image types:
    - image2d_t, image3d_t and sampler_t
- OpenCL #pragmas added to guide the compiler.
- The return type of a kernel function must be void

CERN School of Computing

OpenCL Kernel Programming

- Function qualifiers
  - **__kernel** qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued.
  - Kernels can call other kernel-side functions
- Address space **QUALIFIERS**
  - **__global, __local, __constant, __private**
  - Pointer kernel arguments must be declared with an address space qualifier
- Work-item functions
  - get_work_dim(), get_global_id(), get_local_id(), get_group_id()
- Synchronization functions
  - **Barriers** - all work-items within a work-group must execute the barrier function before any work-item can continue
  - **Memory fences** - provides ordering between memory operations

CERN School of Computing

OpenCL Kernel Programming

- Pointers to functions are ***not*** allowed
- Pointers to pointers allowed ***within*** a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are optional in OpenCL v1.1, but the key word is reserved

(note: most implementations support double)

CERN School of Computing

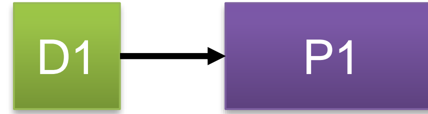
OpenCL Kernel Programming

- Built-in functions — ***mandatory***
  - Work-Item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
  - printf ()
- Built-in functions — ***optional*** (called "extensions")
  - Double precision, atomics to global and local memory
  - Selection of rounding mode, writes to image3d_t surface

FPGA ORIENTED KERNEL DESIGN

FPGA Oriented Kernel Design

- Some general rules for FPGAs:
- Work-Item and a kernel.

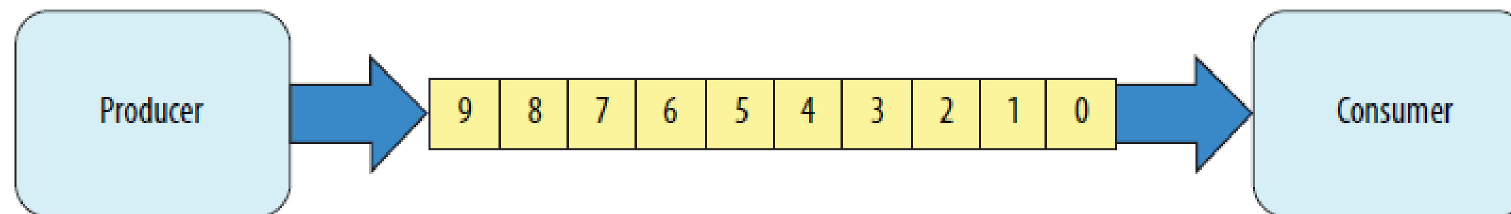


Strategy	Scheme	AREA	FREQ	THROUGHPUT	LATENCY
Parallelizing	D1 → P1 D2 → P1	++	=	++	=
Pipelining	D1 P1.1 P1.2 D2 P1.1	+	++	+	+
Complex op.	D1 → P1+P2+P3	=	--	++	--
Divide op.	D1 → P1 → P2 → P3	++	++	=	++

FPGA Oriented Kernel Design

The most important FPGA design patern

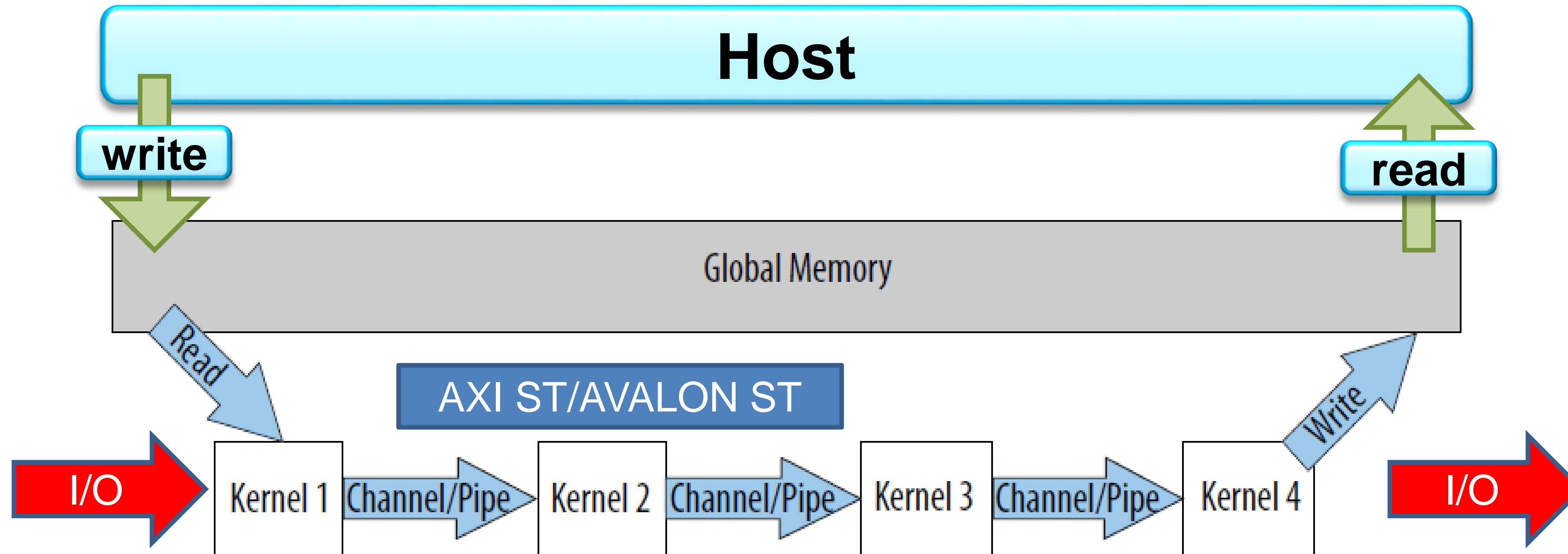
```
#pragma OPENCL EXTENSION cl_intel_channels : enable
channel int c0;

__kernel void producer() {
    for (int i = 0; i < 10; i++) {
        write_channel_intel (c0, i);
    }
}

__kernel void consumer (__global uint * restrict dst) {
    for (int i = 0; i < 5; i++) {
        dst[i] = read_channel_intel(c0);
    }
}
```



from IntelFPGA programming guide

CERN School of Computing

FPGA Oriented Kernel Design



- Send Data from one kernel to another without host intervention
- Send Data from I/O to kernel or from kernel to I/O
- Send Data from host to kernel and vice versa without using global memory
- Data remains in a channel as long as the kernel remains loaded on the FPGA device, persistence among NDRange invocations and among work-groups
- Blocking and Non-Blocking behavior
- Pipes are OpenCL 2.0 standard, contrary to channels

CERN School of Computing

Comments on SYCL

- SYCL. cross-platform abstraction C++ programming model for **OpenCL**.
  - Adding much of the ease of use and flexibility of single-source C++.
- SYCL implements a single-source multiple compiler-passes (SMCP).
  - Simplifying the **Device-Host** separation of OpenCL.
- Easier to handle for programmers. But OpenCL concepts remain, i.e.

- The SYCL Platform Model
- SYCL Execution Mode (command groups)
- Memory Model (same 4 layers)
- The SYCL programming model

- cl::sycl::platform
- cl::sycl::context
- cl::sycl::device
- cl::sycl::program
- cl::sycl::queue
- class kernel

As a summary, is just simpler …

for programers.

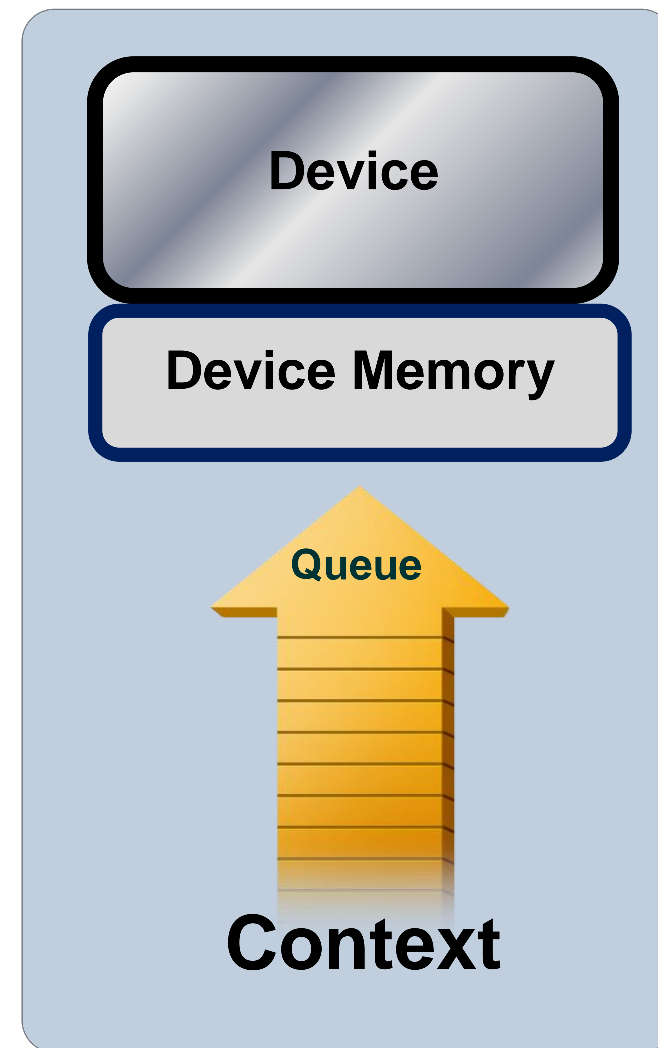
END OF INTRODUCTION

ADVANCED:

MORE ON OPENCL HOST PROGRAMMING

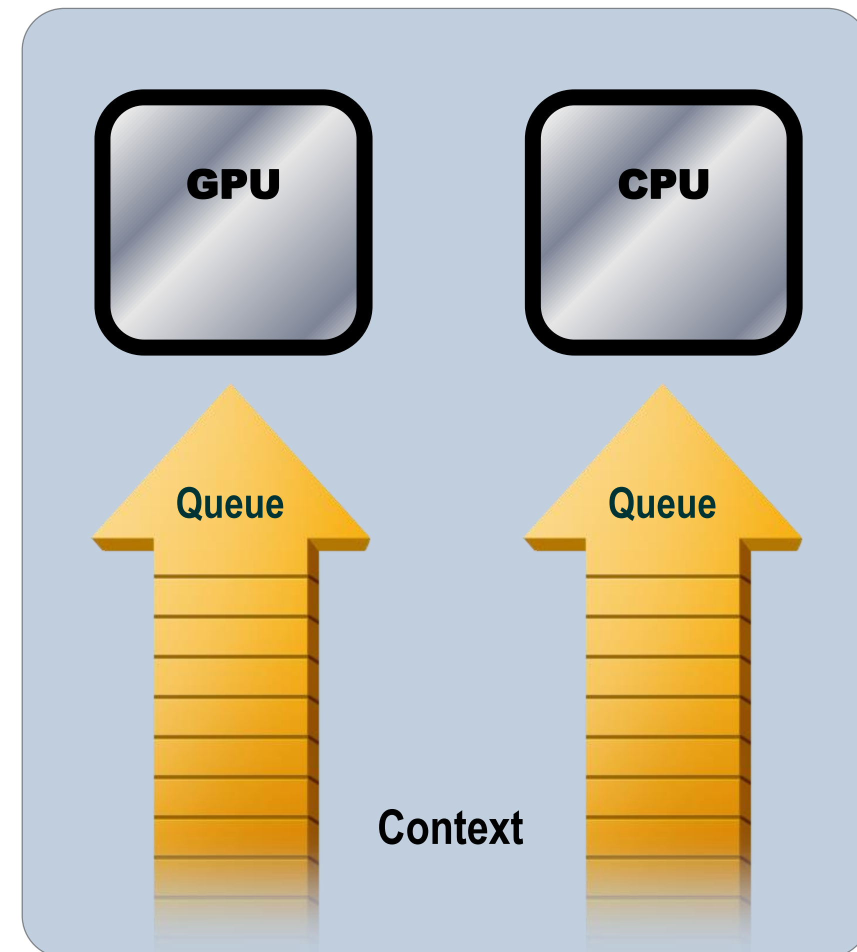
Context and Command-Queues

- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



Command-Queues

- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Independent streams of commands that don’t require synchronization.



CERN School of Computing

Command-Queue execution details

Command queues can be configured in different ways to control how commands execute

- ***In-order queues***:
  - Commands are enqueued and complete in the order they appear in the program (program-order)
- ***Out-of-order queues***:
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.
- Execution of commands in the command-queue are guaranteed to be completed at synchronization points
  - Discussed later

What do we put in device memory?

Memory Objects:
- A handle to a reference-counted region of global memory.

There are two kinds of memory object
- ***Buffer*** object:
  - Defines a linear collection of bytes ("*just a C array*").
  - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
- ***Image*** object:
  - Defines a two- or three-dimensional region of memory.
  - Image data can only be accessed with read and write functions, i.e. these are opaque data structures. The read functions use a sampler.
- ***Pipe*** (Channel) object:
  - A pipe is a memory object that stores data organized as a FIFO.
  - Pipe objects are not accessible from the host.

Memory Object Options

Flag value	Meaning
CL_MEM_READ_WRITE	The memory object can be read from and written to.
CL_MEM_WRITE_ONLY	The memory object can only be written to.
CL_MEM_READ_ONLY	The memory object can only be read from.
CL_MEM_USE_HOST_PTR	The memory object will access the memory region specified by the host pointer.
CL_MEM_COPY_HOST_PTR	The memory object will set the memory region specified by the host pointer.
CL_MEM_ALLOC_HOST_PTR	A region in host-accessible memory will be allocated for use in data transfer.

- These are from the point of view of the **device.**

CERN School of Computing

Conventions for naming buffers

- **It can get confusing about whether a host variable is just a regular C array or an OpenCL buffer**

- **A useful convention is to prefix the names of your regular host C arrays with "h_" and your OpenCL buffers which will live on the device with "d_"**

Tasks and NDRange

- Enqueue the kernel for execution:

```
clEnqueueTask(cl_command_queue commands, cl_kernel kernel, cl_uint
num_events, const cl_event *wait_list, cl_event event);

clEnqueueNDRangeKernel (cl_command_queue commands, cl_kernel kernel,
cl_uint work_dims, size_t *global_work_offset, size_t
global_work_size, size_t *local_work_size,
 0, NULL, NULL);

clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local,
 0, NULL, NULL);
```

Events later.

MATRIX MULTIPLICATION EXAMPLE

Example: Linear Algebra

Analize C=A·B

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{pmatrix}$$

C(i,j) = A(i,:) x B(:,j)

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
              // C(i, j) = sum(over k) A(i,k) * B(k,j)
              C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Matrix multiplication: OpenCL kernel (1/2)

```
__kernel void mat_mul(
 const int N,
__global float *A, __global float *B, __global float *C)
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        // C(i, j) = sum(over k) A(i,k) * B(k,j)
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Mark as a kernel function and specify memory qualifiers

Matrix multiplication: OpenCL kernel (2/2)

```
__kernel void mat_mul(
  const int N,
  __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
        for (k = 0; k < N; k++) {
            // C(i, j) = sum(over k) A(i,k) * B(k,j)
            C[i*N+j] += A[i*N+k] * B[k*N+j];
        }

    }
}
```

Remove outer loops and set work-item co-ordinates

CERN School of Computing

Matrix multiplication: OpenCL kernel

```
__kernel void mat_mul(
 const int N,
 __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    // C(i, j) = sum(over k) A(i,k) * B(k,j)
    for (k = 0; k < N; k++) {
      C[i*N+j] += A[i*N+k] * B[k*N+j];
    }
}
```

Matrix multiplication: OpenCL kernel improved

- Rearrange and use a local scalar for intermediate C element values (a common optimization in Matrix Multiplication functions)

```
__kernel void mmul(
   const int N,
   __global float *A,
   __global float *B,
   __global float *C)
{
  int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tmp = 0.0f;
  for (k = 0; k < N; k++)
   tmp += A[i*N+k]*B[k*N+j];
  }
  C[i*N+j] += tmp;
}
```

This Accumulation is recognized by the compiler !

MORE ON FPGA ORIENTED KERNEL DESIGN

Single Work-Item vs NDRange Kernel

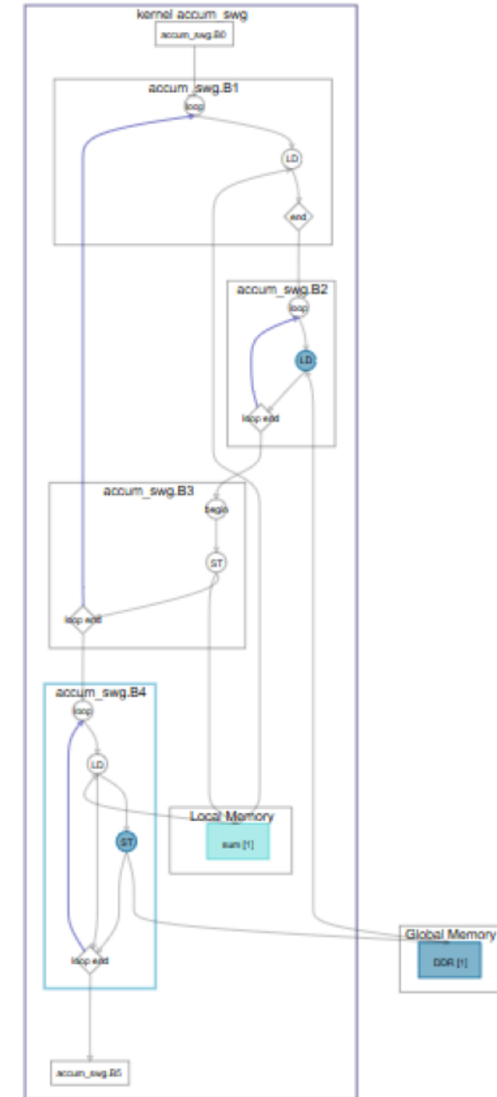
- Intel recommends single work-Item kernels, when possible
- Use NDRange when the code does not have memory dependencies and loops. If data must be shared among WI this structure is not efficient.
- High throughput achieved by using multiple pipelines stages at any time. Parallelism by pipelining the loop iterations.

- Some strategies are common to FPGAs, others depend on the family of FPGA, consult the programming guide for each one.

Single Work-Item vs NDRange Kernel

Single Work-Item kernel

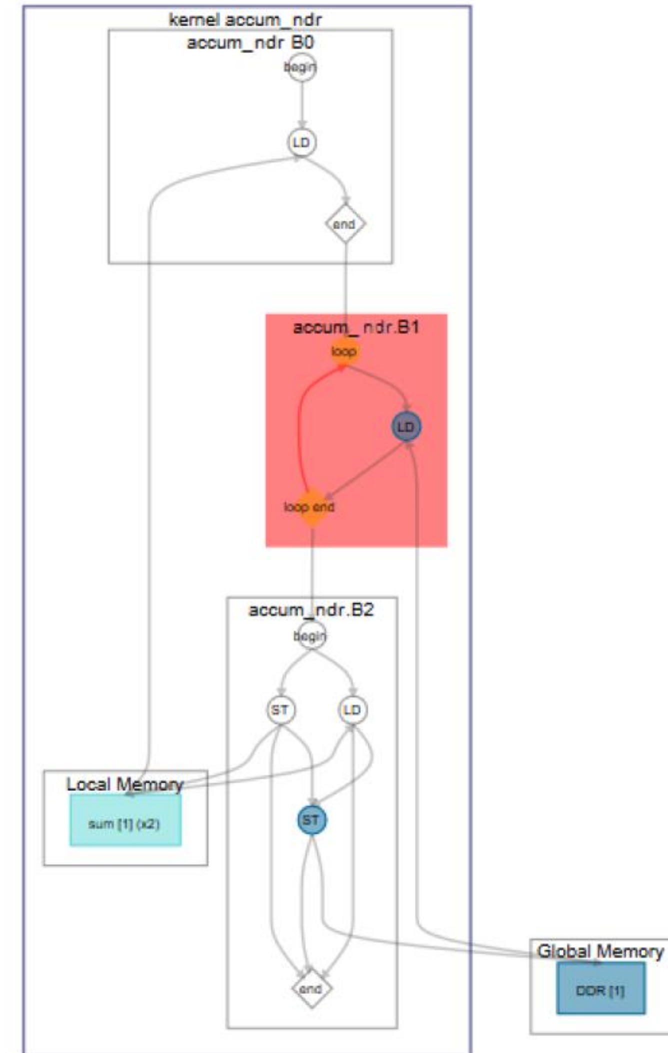
```
1 kernel void accum_swg (global int* a,
                         global int* c,
                         int size,
                         int k_size) {
2     int sum[1024];
3     for (int k = 0; k < k_size; ++k) {
4        for (int i = 0; i < size; ++i) {
5            int j = k * size + i;
6            sum[k] += a[j];
7        }
8     }
9     for (int k = 0; k < k_size; ++k) {
10       c[k] = sum[k];
11    }
12 }
```



Single Work-Item vs NDRange Kernel

NDRange Kernel

```
kernel void accum_ndr (global int* a,
                       global int* c,
                       int size) {
   int k = get_global_id(0);

   int sum[1024];
   for (int i = 0; i < size; ++i) {
     int j = k * size + i;
     sum[k] += a[j];
   }
   c[k] = sum[k];
}
```



CERN School of Computing

Optimizing Data Processing Efficiency

Strategy 1: Unrolling a Loop

#pragma unroll <N>

```
#pragma unroll 2
for(size_t k = 0; k < 4; k++)
{
   mac += data_in[(gid * 4) + k] * coeff[k];
}
```

Optimizing Data Processing Efficiency

Strategy 2: Coalescing Nested Loops

#pragma loop_coalesce <loop_nesting_level>

The OpenCL compiler hates nested loops. Try to avoid its use!!

```
#pragma loop_coalesce
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    sum[i][j] += i+j;
```

```
int i = 0;
int j = 0;
while(i < N){

  sum[i][j] += i+j;
  j++;

  if (j == M){
    j = 0;
    i++;
  }
}
```

Optimizing Data Processing Efficiency

Strategy 3: Specifying a Loop Initiation Interval

#pragma ii <desired_initiation_interval>

Define the number of clock cycles to wait among successive loop iterations

Strategy 4: Loop Concurrency

#pragma max_concurrency <N>

Define the number of iterations to be in progress at one time

Optimizing Data Processing Efficiency

Strategy 5: Specifying the work group size

```
__attribute__((max_work_group_size(512,1,1)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```

```
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```

Optimizing Data Processing Efficiency

Strategy 6: Specifying the Number of Compute Units

```
__attribute__((num_compute_units(2)))
__kernel void test(__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
   size_t gid = get_global_id(0);
   answer[gid] = a[gid] + b[gid];
}
```

Strategy 7: Specifying the Number of SIMD Work-Items

However. Use vectors explicitly as frequently as possible

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void test(__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
   size_t gid = get_global_id(0);
   answer[gid] = a[gid] + b[gid];
}
```

Optimizing Data Processing Efficiency

Strategy 8: Removing Loop-Carried Dependencies

#pragma ivdep

When each loop access different parts of the array there may be fictitious dependencies. This directive commands the compiler to forget the dependencies and remove the extra initiation cycles in the loop immediately after the pragma directive.

CERN School of Computing

Compilation Report

HLD FPGA Reports (Beta) View reports...

Report menu

Analysis pane

Source code pane

Details

fft1d:

- Kernel type: Single work-item
- Required workgroup size: (1, 1, 1)
- Maximum workgroup size: 1

Details pane

Report about code structures that prevent the loops from being fully pipelined.
Report about area usage.
Report about wrong memory management.

More…

Strategy 9: Implementing Arbitrary Precision Integers

Sometimes, optimizing the code when working with FPGAs demands adjusting the size of data to the size strictly needed.

Strategy 10: Inferring Registers and Shift Registers.

When variables are defined as “private” and the access to arrays are statically inferable, they are implemented as FFs in LEs, or in blocks RAM (if their size is larger than 64 bytes). They are the fastest hardware for loop execution.

Strategy 11: Inferring Single-Cycle Floating-Point Accumulator.

Only for Arria10 devices.

… and more strategies

Create good and efficient code for FPGA kernels is a complex task. The results depends heavily on the designer’s expertise.

However, it’s a perfect beginning to read two very useful manuals.

“Intel FPGA SDK for OpenCL Programming Guide” and
“Intel FPGA SDK for OpenCL Best Practices Guide”

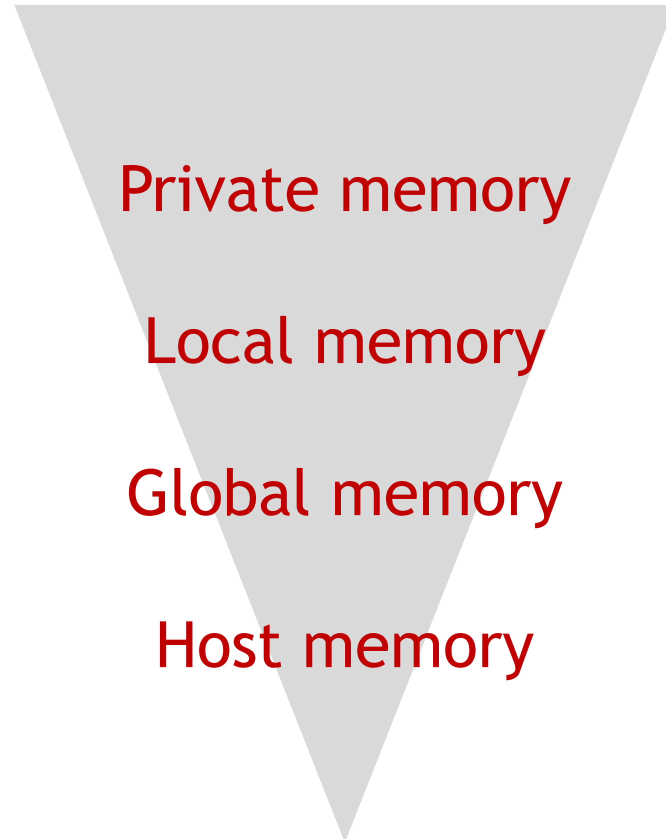
They are hard to deal with though they will give you priceless help for optimizing the area and speed of your application parting from the information given by the compilation report.

They will show you how to work with loop-carried dependency, how to carry out proper memory management to improve access, how to use channels or pipes when needed, etc.

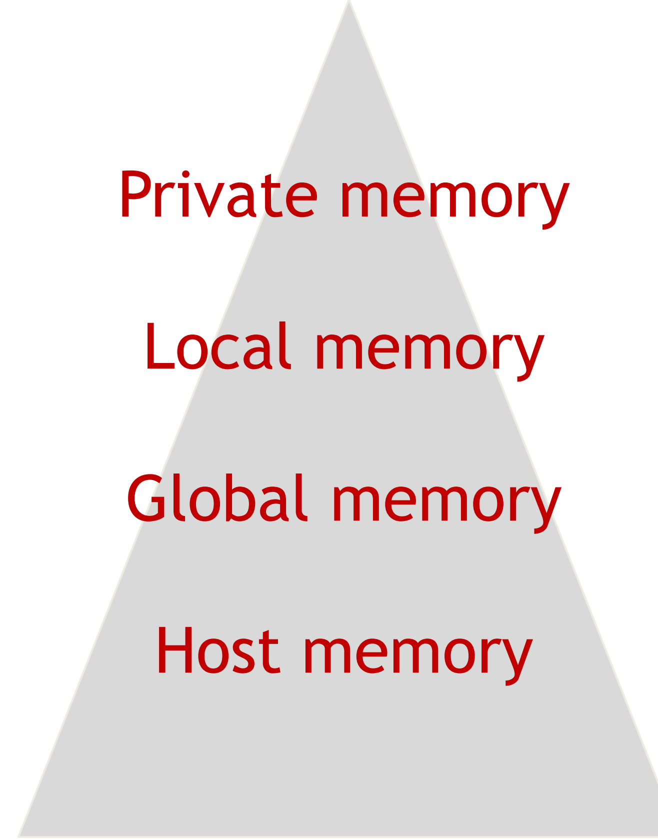
THE OPENCL MEMORY HIERARCHY

The Memory Hierarchy

Bandwidths



Sizes



Easy rule x10

Private Memory

- Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance.
- Remember memory transfers are explicit!

- Private Memory:
  - A **very scarce** resource, only a few thousands of 32-bit words per Work-Item at most
  - If you use too much it spills to global (or local) memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance
  - These is the closest-to-hardware memory. The actual realization varies from one to another. (CPU registers, FPGA registers,…)

Local Memory

- The memory for the work-groups:
  - Close to the hardware, but shared between work-items. Each device realizes it in a different way.
  - Your kernels are responsible for transferring data between Local and Global/Constant memories
- Access patterns to Local Memory affect performance in a similar way to accessing Global Memory.
- Due to their architecture, managing local memory is most important to GPUs.
- FPGA local memory is still very fast (private), but being shared means access patterns affect it.
- CPUs do not have specialized hardware for this….

CERN School of Computing

Global Memory

- The host accessible memory.
- The access pattern to global memory should be the easiest possible.
  - Move data to faster memories, think about dependencies of the algorithm.
- Constant memory is an specialization of global.
- In FPGAs global memory is RAM outside the chip.
  - Constant memory might get replicated or cached to chip memory to satisfy reading needs.
  - While the bandwidth is good, the FPGA can easily overwhelm it with read and write operations.

CERN School of Computing

Memory Consistency

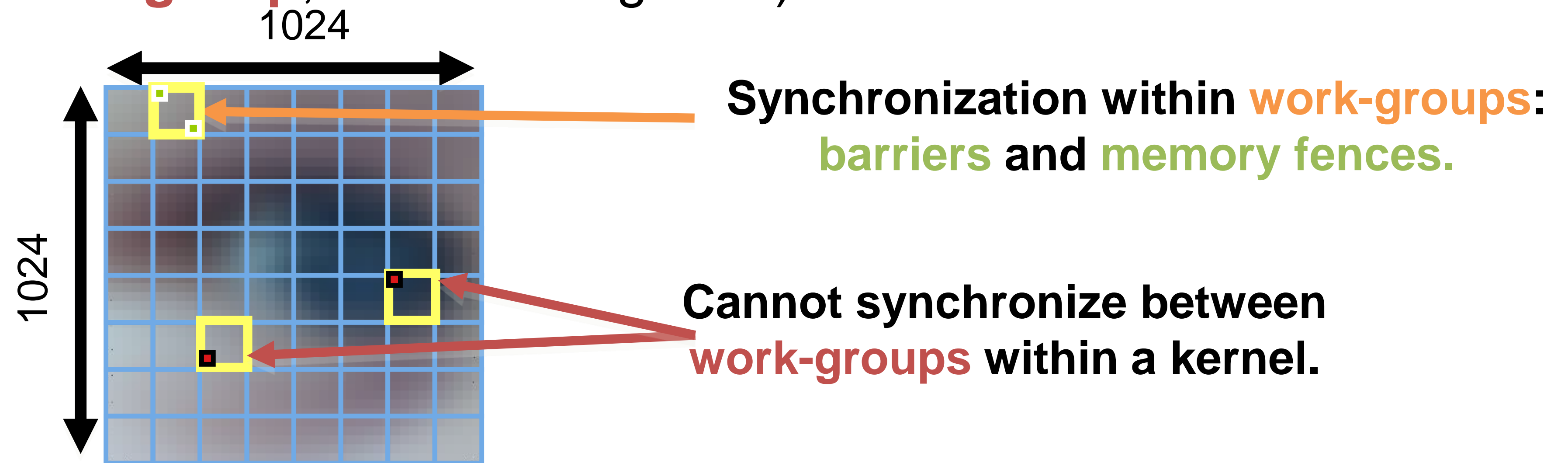
- OpenCL uses a **relaxed consistency** memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
  - Memory has load/store consistency to the work-item’s private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group:
  - Local memory is consistent between work-items at a barrier.
- Global memory is consistent within a work-group at a barrier, **but *not* guaranteed across different work-groups!!**
  - This is a common source of bugs!
- Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)

SYNCHRONIZATION IN OPENCL

CERN School of Computing

Consider N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 64x64 (**work-group**, executes together)



Synchronization: when multiple units of execution (e.g. work-items) are brought to a known point in their execution. Most common example is a barrier … i.e. all units of execution "in scope" arrive at the barrier before any proceed.

CERN School of Computing

Work-Item Synchronization

- Use barrier to synchronize work items inside a work-group.
- **barrier**(CLK_LOCAL_MEM_FENCE) **or barrier**(CLK_GLOBAL_MEM_FENCE)
- Careful with branching! All the work items must take the same branch.
- Across work-groups
  - No guarantees as to where and when a particular work-group will be executed relative to another work-group
  - Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)
  - Only solution: finish the kernel and start another
- The FPGA being hardware, does have other means to synchronize (pipes).

Where might we need synchronization?

- Consider a reduction … reduce a set of numbers to a single value
  - E.g. find sum of all elements in an array
- Sequential code

```
int reduce(int Ndim, int *A)
{
  int sum = 0;
  for (int i = 0; i < Ndim; i++)
    sum += A[i];
  return sum;
}
```

CERN School of Computing

Simple parallel reduction

- A reduction can be carried out in three steps:
  1. Each work-item sums its private values into a local array indexed by the work-item's local id
  2. When all the work-items have finished, one work-item sums the local array into an element of a global array (indexed by work-group id)
  3. When all work-groups have finished the kernel execution, the global array is summed on the host

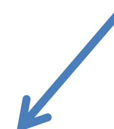
Again, the dimensionality of the problem regarding performance depends on the hardware!

THE OPENCL EVENT MODEL

OpenCL Kernel life cycle

- An event is an object that communicates the status of commands in OpenCL … legal values for an event:
  - CL_QUEUED: command has been enqueued.
  - CL_SUBMITTED: command has been submitted to the compute device
  - CL_RUNNING: compute device is executing the command
  - CL_COMPLETE: command has completed
  - ERROR_CODE: a negative value indicates an error condition occurred.
- Can query the value of an event from the host … for example to track the progress of a command.

```
cl_int clGetEventInfo (
          cl_event event,    cl_event_info param_name,
          size_t param_value_size, void *param_value,
          size_t *param_value_size_ret)
```



Examples:
- CL_EVENT_CONTEXT
- CL_EVENT_COMMAND_EXECUTION_STATUS
- CL_EVENT_COMMAND_TYPE

CERN School of Computing

Generating and consuming events

- Consider the command to enqueue a kernel. The last three arguments optionally expose events (NULL otherwise).

```
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)
```

Number of events this command is waiting to complete before executing

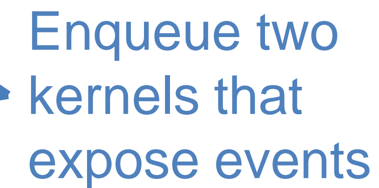
Array of pointers to the events being waited upon … Command queue and events must share a context.

Pointer to an event object generated by this command

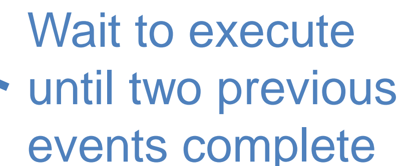
Event: basic event usage

- Events can be used to impose order constraints on kernel execution.
- Very useful with out-of-order queues.

```
cl_event    k_events[2];

err = clEnqueueNDRangeKernel(commands, kernel1, 1,
 NULL, &global, &local, 0, NULL, &k_events[0]);

err = clEnqueueNDRangeKernel(commands, kernel2, 1,
 NULL, &global, &local, 0, NULL, &k_events[1]);

err = clEnqueueNDRangeKernel(commands, kernel3, 1,
 NULL, &global, &local, 2, k_events, NULL);
```

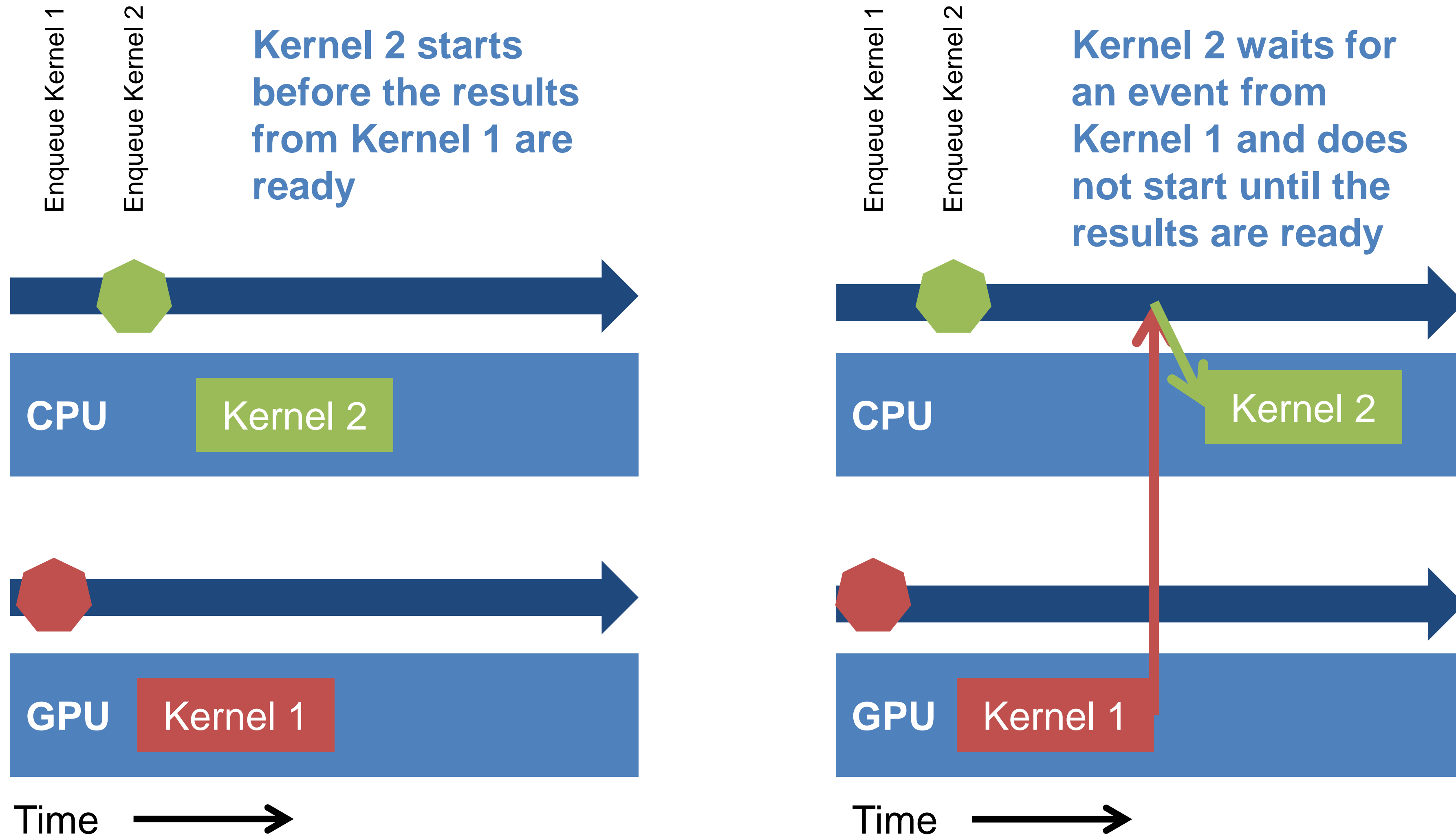


Enqueue two kernels that expose events



Wait to execute until two previous events complete

OpenCL synchronization: queues & events



CERN School of Computing

Profiling with Events

- OpenCL is a performance oriented language … Hence performance analysis is an essential part of OpenCL programming.
- The OpenCL specification defines a portable way to collect profiling data.
- Can be used with most commands placed on the command queue … includes:
  - Commands to read, write, map or copy memory objects
  - Commands to enqueue kernels, tasks
- Profiling works by turning an event into an opaque object to hold timing data.

CERN School of Computing

Using the Profiling interface

- Profiling is enabled when a queue is created with the CL_QUEUE_PROFILING_ENABLE flag set.
- When profiling is enabled, the following function is used to extract the timing data

```
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)
```

Profiling data to query (see next slide)

Expected and actual size of profiling data.

Pointer to memory to hold results

cl_profiling_info values

- CL_PROFILING_COMMAND_QUEUED
  - the device time in nanoseconds when the command is enqueued in a command-queue by the host. (cl_ulong)
- CL_PROFILING_COMMAND_SUBMIT
  - the device time in nanoseconds when the command is submitted to compute device. (cl_ulong)
- CL_PROFILING_COMMAND_START
  - the device time in nanoseconds when the command starts execution on the device. (cl_ulong)
- CL_PROFILING_COMMAND_END
  - the device time in nanoseconds when the command has finished execution on the device. (cl_ulong)

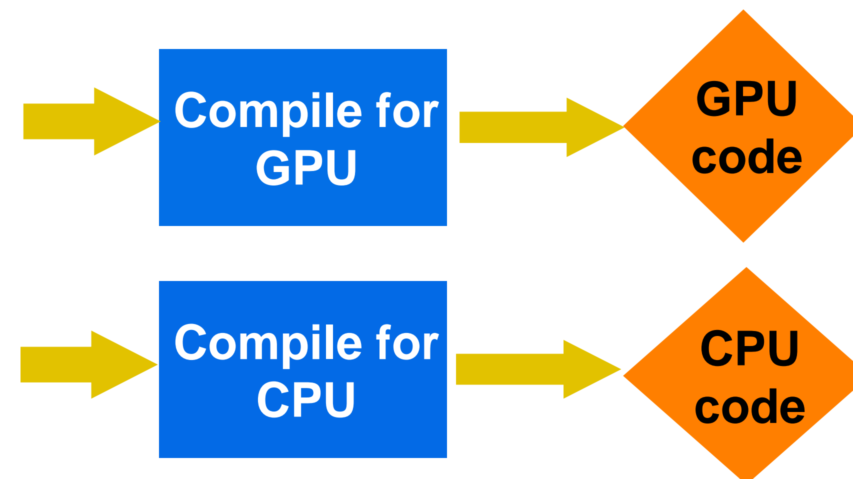
BUILDING OPENCL FOR FPGA

Building Program Objects

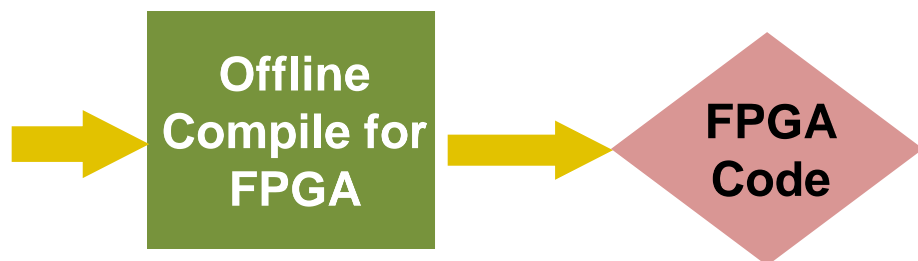
- The program object encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options

- The C API creates a program object:
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

CPUs and GPUs:
OpenCL uses runtime compilation … because in general you don't know the details of the target device when you ship the program

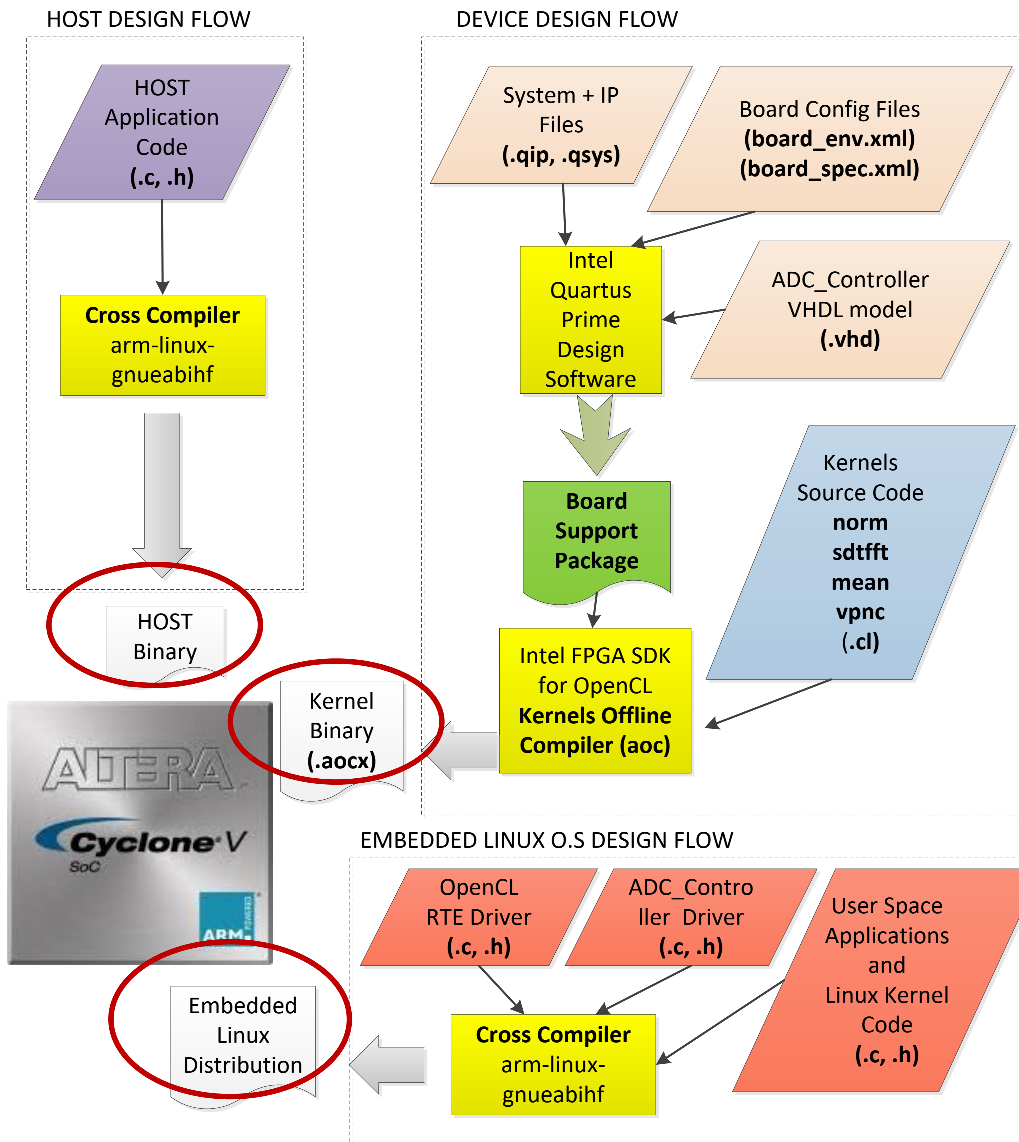


FPGAs:
OpenCL DOESN'T use runtime compilation



OpenCL FPGAs Development

- Example of heterogeneous system:
- SoC with:
- FPGA (Altera)
- CPU (arm).



Final remarks

- Remember the models to divide the computation the OpenCL way!
- The Host is controlling the computation of one or multiple heterogeneous devices.
- The host communicates using commands in command-queues.
- The 4 layers of memory. Memory transfers are explicit!
- There are lots of layers of parallelism
- Synchronize your work-items.
- Use events and profile them to monitor performance.
- Building an OpenCL application requires multiple compilations (min. 2)

Thank you!

Special thanks to Dr. Antonio Carpeño and Professor Mariano Ruiz for their guidance.