Modern architectures are heterogeneous

- A modern computer architecture includes
  - One or more CPUs
  - One or more GPUs
  - Optionally other HW accelerators and FPGAs

- Architectures may have various computing unit per type
- This aspect enables parallelism!

GMCH = graphics memory control hub
ICH = Input/output control hub
Programmability issues

- Each type of resource is generally programmed by using different languages and paradigms

- CPU
  - A huge variety of languages is available: C, C++, Java, ...
  - Various APIs for expressing parallelism: OpenMP, pthreads, ...

- GPU
  - CUDA is a C-based language for NVIDIA GPUs
  - C++ AMP is a programming model based on DirectX defined by Microsoft

- FPGA
  - Verilog and VHDL are the two commonly used HDL to define accelerators
  - Some vendors define high-level proprietary languages for their boards (e.g. MaxJ defined by Maxeler)

- OpenMP and OpenACC are two directive-based languages for parallel computing that recently started supporting heterogeneous devices (CPU, GPU, further accelerators...). Not very flexible in the management of the device

- Huge heterogeneity in system programming!
OpenCL

- OpenCL (Open Computing Language) is a programming framework for heterogeneous compute resources
- Original issue has been to solve heterogeneity between CPUs and GPUs
OpenCL

- OpenCL is a programming framework for heterogeneous compute resources
- Nowadays OpenCL targets many kinds of devices
OpenCL implementations
OpenCL working group

- Diverse industry participation – many industry experts
  - Processor vendors, system OEMs, middleware vendors, application developers
- Apple made initial proposal and is very active in the working group
  - Serving as specification editor
OpenCL timeline

- OpenCL initially developed by Apple
- Meanwhile managed by Khronos Group, a non-profit technology consortium
- Widely agreed-upon industry standard

- We will present OpenCL 1.2 and provide some details on the novelties introduced by OpenCL 2.0 and 2.1
Design goals of OpenCL

• Low-level abstraction for compute
  – Avoid specifics of the hardware (but not hiding hardware management)
  – Enable high performance
  – Support device independence

• Enable all compute resources
  – CPUs, GPUs, and other processors
  – Data- and task-parallel compute models
  – Efficient resource sharing between processors
Design goals of OpenCL

• Efficient parallel programming model
  – ANSI C99 based kernel language
  – Minimal restrictions and language additions
  – Straight-forward development path

• Consistent results on all platforms
  – Well-defined precision requirements for all operations
  – Maximal-error allowances for floating-point math
  – Comprehensive conformance suite for API + language
Supported parallelism models

• Single-Instruction-Multiple-Data (SIMD)
  – The kernel is composed of sequential instructions
  – The instruction stream is executed in lock-step on all involved processing elements
  – Generally used on GPU

• Single-Program-Multiple-Data (SPMD)
  – The kernel contains loops and conditional instruction
  – Each processing element has its own program counter
  – Each instance of the kernel has a different execution flow
  – Generally used on CPU
Supported parallelism models

- Data-parallel programming model
  - The same sequential instruction stream is executed in lock-step on different data
  - Generally used on GPU

- Task-parallel programming model
  - The program issues many kernels in a out-of-order fashion
Problem decomposition

- When we think about how to parallelize a program we use the concepts of decomposition:
  - *Task decomposition*: dividing the algorithm into individual tasks (don’t focus on data)
  - *Data decomposition*: dividing a data set into discrete chunks that can be operated on in parallel
Problem decomposition

• Task decomposition reduces an algorithm to **functionally independent parts**

• Tasks may have dependencies on other tasks
  – If the input of task B is dependent on the output of task A, then task B is dependent on task A
  – Tasks that don’t have dependencies (or whose dependencies are completed) can be executed at any time to achieve parallelism
  – *Task dependency graphs* are used to describe the relationship between tasks

A and B are independent of each other

C is dependent on A and B

B is dependent on A
Problem decomposition

• Data decomposition is a technique for breaking work into multiple independent tasks, but where each task has the same responsibility
  – Each task can be thought of as processing a different piece of data
• Using data decomposition allows us to exploit data parallelism
• Using OpenCL, data decomposition allows us to easily map data parallel problems onto data parallel hardware
Problem decomposition

- For most scientific and engineering applications, data is decomposed based on the output data.
- Examples of output decomposition:
  - Each output pixel of an image convolution is obtained by applying a filter to a region of input pixels.
  - Each output element of a matrix multiplication is obtained by multiplying a row by a column of the input matrices.
- This technique is valid any time the algorithm is based on one-to-one or many-to-one functions.
Problem decomposition

• **Input data decomposition** is similar, except that it makes sense when the algorithm is a **one-to-many function**

• Examples of input decomposition
  – A histogram is created by placing each input datum into one of a fixed number of bins
  – A search function may take a string as input and look for the occurrence of various substrings

• For these types of applications, each thread creates a “partial count” of the output, and synchronization, atomic operations, or another task are required to compute the final result
OpenCL platform model

- The OpenCL platform model is a high-level description of the heterogeneous systems
  - Abstract from peculiarities of specific vendors to have a unified model
OpenCL platform model

- One host connected to a set of compute devices
  - Each compute device is composed of one or more compute units
  - Each compute unit is further divided into one or more processing elements
  - The single unit of computation is executed by a processing element
OpenCL platform model

• One host connected to a set of compute devices
  – Each compute device is composed of one or more compute units
  – Each compute unit is further divided into one or more processing elements
  – The single unit of computation is executed by a processing element

In ARM chips, the GPU may be partitioned in various devices.
OpenCL application

**Device code**
- Written in OpenCL C
- Organized in kernels
- Executes on the device
- Data-parallel execution

**Host code**
- Written in C/C++
- Executes on the host
- Mainly sequential execution
- May use task-parallel execution

Host code sends commands to the devices:
- To transfer data between host memory and device memories (kernel codes and data)
- To execute kernels on the device
OpenCL application

- Serial code executes in a host thread on the host (i.e. a CPU)
- Parallel code (that is the kernel) executes in many device threads across multiple processing elements on the device (e.g. a GPU)
Decompose a task into parallel work-items

- The kernel is actually a data-parallel C function executed on a single instance of the problem
- Parallelization approach:
  - Define an N-dimensional integer index space (N=1,2,3) called NDrange
  - Execute a kernel at each point in the integer index space

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

```c
__kernel void
dp_mul(__global const float *a,
       __global const float *b,
       __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over n "work items"
```
Decompose a task into parallel work-items

- The kernel is actually a data-parallel C function executed on a single instance of the problem
- Parallelization approach:
  - Define an N-dimensional integer index space (N=1,2,3) called NDrange
  - Execute a kernel at each point in the integer index space

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

Traditional loop-based C function

```c
__kernel void dp_mul
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
} // execute over n "work items"
```

OpenCL parallel function
Decompose a task into parallel work-items

- **Work-item**: single execution of the kernel on a data instance (i.e. the basic unit of work)
- **Work-group**: group of contiguous work-items
- Work-groups have the same size and exactly span the NDrange
- The NDrange is actually an N-dimensional grid with a local and global indexing
Decompose a task into parallel work-items

- Example of 1 dimensional problem

```c
kernel void square(global float* input, global float* output)
{
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

- `get_work_dim = 1`
- `get_global_size = 16`
- `get_num_groups = 2`
- `get_group_id = 0`
- `get_local_size = 8`
- `get_local_id = 3`
- `get_global_id = 11`
Decompose a task into parallel work-items

- Example of 3 dimensional problem
Kernel execution on platform model

- Each work-item is executed by a compute element
- Each work-group is executed on a single compute unit
- A wavefront or warp (depending on the implementation) is the subgroup of work-items of a single work-group executed concurrently in lockstep
- Each kernel is executed on a compute device

- Several concurrent work-groups can reside on one compute unit depending on work-group’s memory requirements and compute unit’s memory resources
OpenCL memory model

- Shared memory model
  - With relaxed consistency
  - Explicit synchronization is required
- Private memory
  - Per work-item
- Local memory
  - Shared within a work-group for sharing data
- Global/constant memory
  - Visible to all work-items in any work-groups
  - Constant memory is initialized by the host and is read-only for the kernel
- Host memory
  - On the CPU

- Implementation maps this hierarchy to the available physical memories
OpenCL memory model

• Memory management is EXPLICIT
• Programmer has to move data from host -> global -> local … and back
  – With a memory copy, or
  – With a map of an OpenCL memory objects into the host memory address space
• Programmer has to take care of memory consistency between the kernel and the host and among work-items
OpenCL memory model

• OpenCL uses a “relaxed consistency memory model”:
  – State of memory visible to a work-item **not** guaranteed to be consistent across the collection of work-items at all times

• Memory has load/store consistency within a work-item
• Local memory has consistency across work-items within a work-group at an explicit synchronization point
• Global memory is consistent within a work-group at a an explicit synchronization point, but not guaranteed across different work-groups
• Global memory consistency between host and kernel and between different kernels must be enforced at explicit synchronization points

• The relaxed memory consistency model allows devices scalability
OpenCL framework

Compiled code
Create data & arguments
Send to execution
OpenCL framework

• A **device** represents an OpenCL compute device
A program object contains the source code that implements the kernel and its compiled binary code.
OpenCL framework

- A **kernel** represents an instance of the execution of a program object
- It contains also the actual parameters
OpenCL framework

- **A memory object** is a buffer (or a more advanced structure) allocated in the host or device memory to exchange data between the application and the kernel.
The command queue is used by the host application to submit work to a single device (e.g., kernel execution instances).

- Work is queued in-order, one queue per device.
- Work is executed asynchronously w.r.t. the host application.
- Work can be executed in-order or out-of-order.
OpenCL framework

• The **context** is the environment within a kernel is defined and executed
• It includes devices, kernels, memory objects and command queues
Execution flow

- The application discovers and setups the execution context
  - Discover compute devices
  - Compile kernels
Execution flow

• The application copies the data from the host memory to the device memory by using memory objects
Execution flow

• The application sets the kernel parameters and launches the execution of the kernel on the GPU.
Execution flow

- The GPU executes the kernel on the data stored in its device memory
Execution flow

• The application copies back the data from the device memory to its memory by using the memory objects
OpenCL stack

- Many vendors provide OpenCL implementations
  - There are also open-source implementations
- The OpenCL stack has
  - The front-end equal for all implementations
  - The back-end is vendor-specific
OpenCL stack

• In order to support multiple OpenCL implementations
  – Each implementation provides an Installable Client Driver (ICD)
  – The application is dynamically linked to the ICD loader that manages available ICDs and forward requests
OpenCL Compilation System

- OpenCL uses a Dynamic/Runtime compilation model for the kernel code
  - The code is compiled to an Intermediate Representation (IR)
  - The IR is compiled to a machine code for execution
- LLVM is used for the front-end (replaced/integrated with SPIR-V in OpenCL 2.1)
- Vendor-specific back-ends are used for the second phase
- The host code is still compiled statically
- Runtime compilation cannot be used for FPGA
OpenCL language and API

• Let’s see the source code...

... of the OpenCL kernel
Definition of an OpenCL kernel

• The kernel is implemented in OpenCL language, an extension of C99

• Restrictions w.r.t. C99
  – NO Function pointers
  – NO recursion
  – NO variable-length arrays
  – … few others
Definition of an OpenCL kernel

• New data type
  – Scala data types
    • half, ...
    • Implicit/explicit casts
  – Image types
    • image2d_t, image3d_t, sampler_t
  – Vector data types
    • char2, ushort4, int8, float16, double2, ...
    • Vector lengths 2, 3, 4, 8, 16
    • Endian safe
    • Aligned at vector length
    • Vector operations (+, -, *, -, access to fields, implicit/explicit casts...)
Definition of an OpenCL kernel

- **Address space qualifiers**
  - __global, __local, __constant, __private
- **Built-in functions**
  - Work-items functions: get_global_id, get_local_id, ...
  - Math functions: sin, cos, exp, ...
  - Common functions: min, clamp, max, ...
  - Geometric functions: cross, dot, normalize, ...
  - Vector load/store functions: vload4, vstore4, ...
  - Synchronization functions: barrier
  - Atomic functions: atomic_add, atomic_sub, ...
Definition of an OpenCL kernel

- Example:

```c
__kernel void dp_mult(__global const float* a,
                      __global const float* b,
                      __global float* c)
{
    int i = get_global_id(0);
    c[i] = a[i] * b[i];
}
```
Definition of an OpenCL kernel

- Address space qualifiers
  - __global, __local, __constant, __private

```c
__constant float myarray[4] = {...};

__kernel void pDFA (
const int * InputLength,
local int * Matched,
global int * InputStr,
int OtherThanThat)
{
  int k, j;

  // Declares a pointer variable p in the // private address space. The pointer // points to an int object in address // space __global:
  __global int *p;
}
```

- Variables outside of the kernel scope must be constant.
- Kernel arguments that are pointers can point to constant, local or global memory.
- All non-pointer kernel arguments are private.
- All variables declared inside of a kernel function are private.
- But pointers declared inside of a kernel function can point to other memory spaces!
Work-item synchronization

- Work-items within the same work-group can be synchronized by using a barrier
  - Necessary when work-items cooperate

- Work-items belonging different work-groups CANNOT be synchronized
  - To achieve global synchronization it is necessary to split the kernel in two subsequent ones and put a barrier in the host program

- `barrier()` sets a barrier within the code of a kernel. When all work-items reaches the barrier, the memory (local or global) is flushed and execution is resumed
Work-item synchronization

- This relaxed synchronization mechanism is due to the desire for high scalability on different types of devices (CPUs, GPUs) which manage/execute threads in different ways.
OpenCL language and API

• Let’s see the source code...

... of the host application
Workflow of n OpenCL application

1. Discovery the platforms and devices
2. Creating a context
3. Creating command-queue per device
4. Creating memory objects to hold data
5. Copying the input data onto the device
6. Creating and compiling a program from the OpenCL C source code
7. Generating a kernel of the program and specifying parameters
8. Executing the kernel
9. Copying output data back to the host
10. Releasing the OpenCL resources
Get platform

• A platform is a vendor-specific implementation of the OpenCL API
  – It contains a set of devices

• `clGetPlatformIDs()` is used for accessing the available platforms

• `clGetPlatformInfo()` is used for querying the information about the available platforms
Get platform

• Example:

```c
//get all platforms
cl_int err;
cl_uint numPlatforms;
cl_platform_id *platformIds;
err = clGetPlatformIDs(0, NULL, &numPlatforms);
platformIds = (cl_platform_id *)
    malloc(sizeof(cl_platform_id) * numPlatforms);
err = clGetPlatformIDs(numPlatforms, platformIds, NULL);
```

• All `clGet*` functions can be used to access either the number of items or the list of items
  – Specific parameter lists distinguish the two queries
Get platform

• Example:

```c
//get the name of a platform given its id
err = clGetPlatformInfo(id, CL_PLATFORM_NAME, 0,
                         NULL, &size);
char * name = (char *) malloc(sizeof(char) * size);
err = clGetPlatformInfo(id, CL_PLATFORM_NAME, size,
                         name, NULL);
std::cout << "Platform name: " << name << std::endl
```

• Various kinds of information may be queried:

  - `CL_PLATFORM_NAME`, `CL_PLATFORM_VENDOR`, `CL_PLATFORM_PROFILE`, ...
Get device

• Each platform may contain one or more compute device

• `clGetDeviceIDs()` can be used for querying the available devices

• `clGetDeviceInfo()` can be used for querying the information about the devices
Get device

• Example:

```c
// get all devices of a given platform
cl_uint numDevices;
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
cl_device_id* deviceIds = (cl_device_id *) malloc(sizeof(cl_device_id) * numDevices);
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, numDevices, deviceIds, NULL);
```

• It is possible to filter on the device type
  - `CL_DEVICE_TYPE_GPU`, `CL DEVICE_TYPE_CPU`, `CL_DEVICE_TYPE_ACCELERATOR`
Get device

• Example:

```cpp
//get the maxComputeUnits of a specified device
cl_uint maxComputeUnits;
err = clGetDeviceInfo(deviceID, CL_DEVICE_MAX_COMPUTE_UNITS,
                    sizeof(cl_uint), &maxComputeUnits,
                    &size);
std::cout << "Device has max compute units: "
          << maxComputeUnits << std::endl;
```

• Various kinds of information may be queried:
  
  - CL_DEVICE_MAX_WORK_ITEM_SIZES,
    CL_DEVICE_MAX_WORK_ITEM_ITEM_DIMENSIONS,
    CL_DEVICE_MAX_WORK_ITEM_GROUP_SIZE
Create context

- A context provides a container for associated devices, program objects, kernels, memory objects and command queues
- We may instantiate multiple contexts within a program
- A context may contain various devices of the same platform

- Multiple contexts enable dynamic mapping of the kernels to be executed
  - Data transfers among different contexts have to be explicitly managed by the programmer
Create context

• A context may be created in two different ways
  – `clCreateContext()` creates a context from a specified list of devices of a single platform
  – `clCreateContextFromType()` creates a context by using a specified type of platform and device
Create context

• Example 1:

```c
//select the first device from the specified platform
cl_context context;
cl_context_properties properties[] =
    {CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};
context = clCreateContext(properties, 1, devices[0],
    NULL, NULL, NULL);
```

• Example 2:

```c
//select all devices of any type from the specified platform
cl_context context;
cl_context_properties properties[] =
    {CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};
context = clCreateContextFromType(properties,
    CL_DEVICE_TYPE_ALL, NULL, NULL, NULL);
```
Create command queue

- A command queue is used for issuing commands to a device
- One command queue per device needed
  - Dispatching among devices within the same context explicitly managed by the programmer
  - A single device may have multiple independent queues
- Available commands
  - Copy data to/from device
  - Execute a kernel
  - Synchronize

- Within a single command queue:
  - Commands can be synchronous or asynchronous
  - Commands can be executed in-order or out-of-order
Create command queue

- `clCreateCommandQueue()` creates a command queue on a specified device within a context
- Example:

```c
// create a command queue on the first device within the related context
cl_command_queue queue;
queue = clCreateCommandQueue(context, devices[0], 0, &err);
```

- Commands issued in the queue are executed in order
  - The execution of a command starts when the previous one is completed
  - `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` enables out of order execution of the commands within the queue
Build program

- The source code of each kernel is a text string
  - It can be stored in a C/C++ string or in a text file
- The source code has to be loaded and compiled within the context it needs to be used

- `clCreateProgramWithSource()` creates a program object from the source code
- `clBuildProgram()` compiles the program to obtain the executable code on each device type within the selected context
Build program

• Example:

```c
//source code is created in the specified context and it
//is compiled for the first device in the list
const char *program_source =
    "__kernel void dp_mult (__global const float *a,"
    " __global const float *b, __global float *c)\n",
    "... ";

cl_program program;
program = clCreateProgramWithSource(context, 1,
    program_source, NULL, &err);
clBuildProgram(program, 1, devices, "", NULL, NULL)
```

• Further options:
  – Compiling for all devices within the context
  – Loading an already compiled binary (for a specified device)
Create a kernel

• In OpenCL language a kernel represents an instance of the execution of a program object

• `clCreateKernel()` creates a kernel from a program object

• Example:

```c
//create a kernel
cl_kernel kernel = clCreateKernel(program, "dp_mult", &err);
```
Build program and create a kernel

- There is a high overhead for compiling programs and creating kernels
- Each operation only has to be performed once (at the beginning of the program)
- The kernel objects can be reused any number of times by setting different arguments

Read source code into an array

- `clCreateProgramWithSource`
- `clCreateProgramWithBinary`
- `clBuildProgram`
- `clCreateKernel`
Create memory objects

• Memory objects are used to transmit data to/from a device
  – The buffer is the basic 1-dimensional memory object
  – OpenCL provides also advanced 2/3-dimensional memory objects for managing images

• Memory objects are visible within a device but memory consistency has to be managed explicitly among different devices

• Memory objects cannot be shared among different contexts since they are potentially associated to different platforms

• Memory objects are accessed in the kernel through pointers
Create memory objects

- `clCreateBuffer()` creates a memory object
- Buffers may be read-only, write-only or read-write
- It is possible to initialize buffers with data

**Example:**

```c
//create buffers for input and output
cl_mem bufA, bufB, bufC;
bufA = clCreateBuffer(context, CL_MEM_READ_ONLY,
                      sizeof(float)*NUM_DATA, NULL, &err);
bufB = clCreateBuffer(context, CL_MEM_READ_ONLY,
                      sizeof(float)*NUM_DATA, NULL, &err);
bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                      sizeof(float)*NUM_DATA, NULL, &err);
```
Write into memory objects

- clEnqueueWriteBuffer() transfers data from the host memory to the device global memory

- Example:

```c
//transfer input data
float a[NUM_DATA], b[NUM_DATA];
//write something in a and b
clEnqueueWriteBuffer(queue, bufA, CL_TRUE, 0,
    sizeof(float)*NUM_DATA, a, 0, NULL, NULL));
clEnqueueWriteBuffer(queue, bufB, CL_TRUE, 0,
    sizeof(float)*NUM_DATA, b, 0, NULL, NULL));
```

- An alternative is to map the memory object into the host address space so that the host application can access it
Pass parameters

• Parameters have to be specified one by one
• `clSetKernelArg()` is used to specify a single parameter per time

• Example:

```c
//specify the three parameters for the dp_mult function
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
```
Execute the kernel

• `clEnqueueNDRangeKernel()` launches the execution of a kernel on a specified NDRange

• Two possibilities for the work-group size
  – Do not specify anything and OpenCL will manage it
  – Explicitly specify it
    • Best performance when the maximum work-group size supported by the device is used
    • The considered problem may lead to a different value

• Example:

```c
//executes the dp_mult kernel on the specified NDRange
size_t global_work_size[1] = { NUM_DATA };
size_t local_work_size[1] = { 32 };
clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
           global_work_size, local_work_size, 0, NULL, NULL);
```
Read results

- `clEnqueueReadBuffer()` transfers data from the device global memory to the host memory

- Example:

```c
//read results
float results[NUM_DATA];
clEnqueueReadBuffer(queue, bufC, CL_TRUE, 0,
                     sizeof(float)*NUM_DATA, results, 0,
                     NULL, NULL));
```
Release resources

• At the end of the program, all OpenCL resources have to be released

• Example:

```c
//release resources
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseKernel(kernel);
clReleaseProgram(program);
cReleaseContext(context);
```
Further notes on the code

• Every OpenCL instruction returns an error code
  – Error codes have to be checked after every instruction to assess the correct execution
  – Otherwise the application will continue under an error condition

• Most of the source code is almost the same in every OpenCL application
More on work-items and work-groups sizing

- Do note: the work-groups must span exactly the global index space on every dimension.
- How to manage a problem with a size that is not divisible by a predefined work-group size?
- E.g. `DATA_SIZE=109` and `workgroupsize=16`
More on work-items and work-groups sizing

- A parameter specifying the problem size is added to the kernel

```c
__kernel void dp_mult(__global float* a,
                     __global float* b, __global float* c,
                     int N)
{
    int i = get_global_id(0);
    if(i<N)
        c[i] = a[i] * b[i];
}
```
More on work-items and work-groups sizing

• The global work size is round up to the first multiple of the work-group size greater than the problem size

```c
size_t local_work_size[1] = { 16 };
size_t global_work_size[1] =

   { NUM_DATA +16 - NUM_DATA % 16 };
```

• Therefore the kernels with id>=NUM_DATA will not compute any multiplication

• **NOTE**: OpenCL 2.0 has solved this issue by defining a remainder work-groups
  – They are defined on each dimension on the upmost boundaries
  – Global work size may have any value
Command synchronization

- Explicit synchronization management has to be performed by the programmer to assess the desired precedencies in the kernel execution when using
  - an out-of-order command queue
  - different queues
Command synchronization

- **clEnqueueBarrier()** sets a barrier on the queue specified as parameter
  - Commands after the barrier start executing only after all commands before the barrier have completed
- **clFinish()** blocks the execution of the host program until all commands in the queue specified as parameters are completed
- Example:
  ```c
  //... send commands
  clEnqueueBarrier(queue);
  //... send commands
  clFinish(queue);
  ```
Command synchronization

• Events
  – Commands return events and obey event waitlists
  – A cl_eventobject can be associated with each command (last three parameters)

    clEnqueue*(..., num_events_in_waitlist, *event_waitlist, *event);

  – clEnqueueReadBuffer, clEnqueueReadBuffer can be optionally blocking
    Any commands (or clWaitForEvents()) can wait on events before executing
    Event object can be queried to track execution status of associated command and get profiling information
Command synchronization

• In conclusions
  – Synchronization has to be explicitly managed by the programmer
  – Coherence of data at synchronization points implicitly managed by the runtime layer

• Synchronization using events is possible only in the same context

• The only way to synchronize and share data between different contexts is to act in the host application by explicitly copying data between memory objects
An example

- Let’s consider as example the multiplication of two matrixes $A(M,K)$ and $B(K,N)$ producing $C(M,N)$

$$C[i,j] = \text{sum} (A[m,k]*B[k,n], k = 0, 1, \ldots, K)$$
An example

• Serial solution:

```c
void SerialMultiply (float A[P][Q], float B[Q][R],
    float C[P][R])
{
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++){
            C[i][j] = 0;
            for (int k = 0; k < Q; k++)
                C[i][j] += A[i][k] * C[k][j];
        }
}
```

Just for sake of simplicity dimension management is performed by means of constant macros
An example

• Parallel solution:

```c
__kernel void parMultiply1 (  
    __global float *A, __global float *B,  
    __global float *C, int K)  
{
    // Vector element index
    const int m = get_global_id(0);  
    const int n = get_global_id(1);  
    const int M = get_global_size(0);  
    C[i + M * j] = 0;  
    for (int k = 0; k < K; k++)  
        C[i+M*j] += A[i+M*k] * B[k+M*j];
}
```
An example

- Parallel solution using local memory:
  - Idea: split the matrix multiplication problem into submatrixes that fit in the local workgroup size
  - We can compute each 3x3 submatrix of the answer as the sum of products of the first submatrixes, as if the submatrixes were elements of a matrix
An example

- Parallel solution using local memory:

```c
__kernel void parMultiply2 (__global float *A, __global float *B,
__global float *C, int K)
{
    const int row = get_local_id(0); // Local row ID (max: TS)
    const int col = get_local_id(1); // Local col ID (max: TS)
    const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)
    const int M = get_global_size(0);
    const int N = get_global_size(1);

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[TS][TS];
    __local float Bsub[TS][TS];
```
An example

- Parallel solution using local memory: (continue…)

```c
// Initialize the accumulation register
float acc = 0.0f;

// Loop over all tiles
const int numTiles = K/TS; //K must be a multiple of TS
for (int t=0; t<numTiles; t++) {
    // Load one tile of A and B into local memory
    const int tiledRow = TS*t + row;
    const int tiledCol = TS*t + col;
    Asub[col][row] = A[tiledCol*M + globalRow];
    Bsub[col][row] = B[globalCol*K + tiledRow];
    // Synchronize to make sure the tile is loaded
    barrier(CLK_LOCAL_MEM_FENCE);
}
```
An example

- Parallel solution using local memory: (continue…)

```c
// Perform the computation for a single tile
for (int k=0; k<TS; k++) {
    acc += Asub[k][row] * Bsub[col][k];
}

// Synchronize before loading the next tile
barrier(CLK_LOCAL_MEM_FENCE);
}

// Store the final result in C
C[globalCol*M + globalRow] = acc;
```
Dissecting OpenCL execution

• An example showing buffer creation and initialization

```c
cl_int err;
int a[16];

for (int i = 0; i < SIZE; i++) {
    a[i] = i;
}

// Create the buffer
cl_mem buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, 
SIZE*sizeof(int), a, &err);

if( err != CL_SUCCESS ) {// Handle error as necessary }

// Initialize the buffer
err = clEnqueueWriteBuffer ( 
    queue, 
    buffer, // destination 
    CL_TRUE, // blocking write 
    0, // don’t offset into buffer 
    SIZE*sizeof(int), // number of bytes to write 
    a, // host data 
    0, NULL, // don’t wait on any events 
    NULL); // don’t generate an event
```
Dissecting OpenCL execution

- We can create and initialize a buffer, and use it in a kernel, without explicitly writing the buffer.

```c
cl_mem buf = clCreateBuffer(...,
   CL_MEM_COPY_HOST_PTR,data,...);
clSetKernelArg(..., &buf);
clEnqueueNDRangeKernel(gpuQueue,...);
```

a) Creating and initializing a buffer in host memory (initialization is done using CL_MEM_COPY_HOST_PTR).

b) Implicit data transfer from the host to device prior to kernel execution. The runtime could also choose to have the device access the buffer directly from host memory.
Dissecting OpenCL execution

- As an alternative, a runtime may decide to create the buffer directly in device memory

```
cl_mem buf = clCreateBuffer(...);
```

a) Creating a buffer in device memory (at the discretion of the runtime)

```
clEnqueueWriteBuffer(...,buf,...,data,...);
```

b) Copying host data to the buffer directly in GPU memory
Dissecting OpenCL execution

- The complimentary call to writing a buffer, reads the buffer back to host memory
  - The following diagram assumes that the buffer data had been moved to the device by the runtime

```c
clEnqueueReadBuffer(..., buf,..., data,...);
```

![Diagram showing CPU Memory and GPU Memory, with DMA and Context]

c) Reading the output data from the buffer back to host memory (continued from the previous slide)
Dissecting OpenCL execution

• OpenCL provides API calls to map and unmap memory objects from the host’s memory space
• When a memory object is mapped, a pointer valid on the host will be returned
  – The parameters to the map function should look similar to reading/writing a buffer

```c
void * clEnqueueMapBuffer (cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_map,
    cl_map_flags map_flags,
    size_t offset,
    size_t size,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event,
    cl_int *errcode_ret)
```

```c
cl_int clEnqueueUnmapMemObject (cl_command_queue command_queue,
    cl_mem memobj,
    void *mapped_ptr,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```
Dissecting OpenCL execution

• The example below shows one possible scenario for the map and unmap functions

  a) Create an uninitialized buffer in device memory

  cl_mem buf = clCreateBuffer(…);

  b) Map the buffer into the host’s address space. Access the buffer via a host pointer.

  void *mappedBuf = clEnqueueMapBuffer(..., buf, ...);
  memcpy(data, mappedBuf, ...);

  c) Unmap the buffer from the host’s address space

  clEnqueueUnmapMemObject(..., buf, mappedBuf, ...);
Dissecting OpenCL execution

- Execution of a kernel on a (AMD) CPU
  - Each CPU thread executes in sequence a workgroup (for performance reasons)
  - One thread per core

- Do note that the explicit usage of vector data types and operations maps on CPU Streaming SIMD Extensions (SSE) and Advanced Vector Extension (AVX) technologies

- Execution on GPU has been already discussed …
Dissecting OpenCL execution

- Execution of a kernel on an (AMD) CPU
  - Each CPU thread executes in sequence a workgroup

- Do note that the explicit usage of vector data types and operations maps on CPU Streaming SIMD Extensions (SSE) and Advanced Vector Extension (AVX) technologies

- Execution on GPU has been already discussed during the last class…
Dissecting OpenCL execution

- Memory mapping on GPU
  - Global Memory
    - Maps to cache hierarchy
    - GDDR5 video main memory
  - Constant Memory
    - Maps to scalar unit reads
    - Possibly on texture or constant memory (NVIDIA)
  - Local Memory
    - Maps to the LDS
    - Shared data between work-items of a work group
    - High Bandwidth access from SIMIDs
  - Private memory
    - Maps to vector registers
Dissecting OpenCL execution

- Memory mapping on GPU
  - Global Memory
    - Maps to cache hierarchy
    - GDDR5 video main memory
  - Constant Memory
    - Maps to scalar unit reads
    - Possibly on texture or constant memory (NVIDIA)
  - Local Memory
    - Maps to the LDS
    - Shared data between work-items of a work group
    - High Bandwidth access from SIMIDs
  - Private memory
    - Maps to vector registers
Dissecting OpenCL execution

• Memory mapping on CPU
  – Everything maps on the cache memory
Dissecting OpenCL execution

- On a GPU, within a warp/wavefront, control-flow should not diverge among work items
  - Otherwise, a so-called branch divergence occurs
- GPU cannot follow two instruction streams at the same time within warp
  - Divergence between warps/wavefronts is ok!
To avoid branch divergence, conditions of if-statements must be re-worked in such a way that all work items within a warp/wavefront take the same branch.
Dissecting OpenCL execution

- Simultaneous accesses of the memory to work-items of the same work-groups should be to subsequent addresses to exploit memory coalescence

```c
__kernel void vector(__global float *input, int a, int b, int c) {
    float4 a = input[get_global_id(0)*4];
    input[get_global_id(0)*4+a],
    input[get_global_id(0)*4+b],
    input[get_global_id(0)*4+c])
    ...
}
```
OpenCL on FPGA

• OpenCL can also target FPGA devices!

• CPU/GPU scenario:
  – Architecture is fixed
  – OpenCL program is designed to exploit HW peculiarities

• FPGA scenario:
  – Architecture is not configurable
  – OpenCL program is designed to define the overall optimized architecture of the accelerator!
    • Necessity to define additional parameters to customize architectural design

• OpenCL guarantees the functional portability but not the performance portability
  – Coding style does matter especially when targeting FPGAs
OpenCL on FPGA

- Motivations (from Xilinx whitepaper)
  - OpenCL allows a easier programmability of FPGAs
  - FPGAs provides higher performance/Watt than CPU/GPU
OpenCL on FPGA

- Xilinx provides SDAccel tool to support OpenCL programming on FPGA
  - It is based on Vivado HLS
OpenCL on FPGA

• Target architecture
OpenCL on FPGA

- Architecture of the SDAccel design
OpenCL on FPGA

- OpenCL memory organization

- Global memory is implemented both in the off-chip memory and in the BRAMs
- Local and private memories are implemented both in BRAMs and registers
OpenCL on FPGA

- SDAccel design flow
OpenCL on FPGA

- The kernel has to be synthesized offline
- The bitstream is then configured at runtime when creating a program
OpenCL on FPGA

• The tool requires to specify how many compute units to instantiate
• The configuration of the single compute unit is performed by using specific attributes in the kernel code
OpenCL on FPGA

• Specifying the workgroup size to be used to implement the compute unit

```c
__kernel __attribute__((reqd_work_group_size(4,4,1)))
void mmult32(global int *A, global int *B, global int *C) {
    ...
}
```

• Unroll loops

```c
kernel void
vmult(local int* a, local int* b, local int* c) {
    int tid = get_global_id(0);
    __attribute__((opencl_unroll_hint(2)))
    for (int i=0; i<4; i++) {
        int idx = tid*4 + i;
        a[idx] = b[idx] * c[idx];
    }
```
OpenCL on FPGA

- Pipeline loops

```c
kernel void foo(...) {
  ...
  __attribute__((xcl_pipeline_loop))
  for (int i=0; i<3; i++) {
    int idx = get_global_id(0)*3 + i;
    op_Read(idx);
    op_Compute(idx);
    op_Write(idx);
  }
  ...
}```
OpenCL on FPGA

• Pipeline work items

```c
kernel
__attribute__((reqd_work_group_size(3,1,1)))
void foo(...)
{
    ...
    __attribute__((xcl_pipeline_workitems)) {
        int tid = get_global_id(0);
        op_Read(tid);
        op_Compute(tid);
        op_Write(tid);
    }
    ...
}
```
OpenCL on FPGA

- Memory access needs to be configured as well
- E.g.: usage of burst transmissions

```c
__attribute__((reqd_work_group_size(1, 1, 1)))
kernel void smithwaterman (global int *matrix, global int *maxIndex, global const char *s1, global const char *s2) {
    short north = 0;
    short west = 0;
    short northwest = 0;
    int maxValue = 0;
    int localMaxIndex = 0;
    int gid = get_global_id(0);

    // Local memories using BlockRAMs
    local char localS1[N];
    local char localS2[N];
    local int localMatrix[N*N];

    async_work_group_copy(localS1, s1, N, 0);
    async_work_group_copy(localS2, s2, N, 0);
    async_work_group_copy(localMatrix, matrix, N*N, 0);
```
Conclusions

• OpenCL – Open Computing Language
  – Open, royalty-free standard
  – For cross-platform, parallel programming of modern processors
  – An Apple initiative
  – Approved by NVIDIA, Intel, AMD, IBM, ...
  – Specified by the Khronos group

• Intended for accessing heterogeneous computational resources
  – CPUs (Intel, AMD, IBM Cell BE, ...)
  – GPUs (Nvidia GTX & Tesla/Fermi, AMD/ATI 58xx, ...)
  – HW accelerators and FPGAs (Xilinx)
  – Processors with integrated graphics chip (Sandy Bridge, Llano)

• Difference to CUDA
  – Code hardware agnostic, portable
Conclusions

• Opportunities offered by OpenCL
  – Program with a single language an heterogeneous processing system
  – Decide at run-time where to execute each kernel according to the current working conditions
References

• https://www.khronos.org/
• https://developer.nvidia.com/opencl
• http://developer.amd.com/tools-and-sdks/opencl-zone/
• http://elc.yonsei.ac.kr/courses/csi2110/PP-L07-OpenCL.pdf
• https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html#documentation
• David Kaeli, Perhaad Mistry, Dana Schaa, Dong Ping Zhang, Heterogeneous Computing with OpenCL 2.0, Morgan Kaufmann, 2015

• Some OpenCL source code examples:
  – http://booksite.elsevier.com/9780128014141/online_materials.php
  – http://www.heterogeneouscompute.org/?page_id=7