Faculty of Engineering and Information Sciences
University of Wollongong

CSCI976 - Summary

Anton Kullberg

October 31, 2017

Introduction

This is a summary of the course CSCI976 - Multicore and GPU programming at the University of Wollongong during spring session 2017. The summary is based on lectures given by Dr. Casey Chow. It is not in the same chronological learning order as in the course itself but is instead divided up into "natural" sections. Each section treats some part of the course. For specific function details see the OpenCL 1.0 specification.

This summary does in no way replace the lecture material provided by Dr. Casey Chow, but can assist in getting a big picture view of parallel programming and OpenCL.

Contents

1 Heterogeneous computing

Heterogeneous computing refers to systems that uses more than one type of processor. A modern computing platform includes:

- One or more CPUs
- One or more GPUs
- DSP processors
- Others

The main purpose of heterogeneous computing is to utilise multiple processor types (e.g. CPUs and GPUs). Basically to use the "best man for the job". E.g. you wouldn't want a painter to build your house when you could have a carpenter.

1. **GPU**
    - Handle 3D graphics rendering
    - Mathematically intense computations on very large datasets
    - One task performed on every element of an array
    - Vector processing capabilities (perform an operation on a complete vector at once)
    - Thousands of cores that can operate simultaneously
    - Requires setup by a CPU (e.g.)
2. **CPU**
    - Run the operating system
    - Perform serial tasks which require more control
    - Certain scalar algorithms and problems run better on CPU

1.1 Benefits of heterogeneous computing

By 2010 nearly all new desktops had multicore processors. Multicore systems has an advantage over singlecore systems by:

- Adding cores
- Incorporating specialised processing capabilities to handle particular tasks (e.g. vector processing in GPUs)

1.2 Challenges of heterogeneous computing

Challenging to develop software for a wide array of architectures. Applications can have different workload behaviours:

- **Control intensive** - Searching, sorting, parsing.
- **Data intensive** - Image processing, simulation, modelling, data mining
- **Compute intensive** - Iterative methods, numerical methods, financial modelling

Each type of workload works best on a certain hardware architecture.

1.3 Parallel computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel). To exploit parallel computing one must have the physical resources (hardware).

Table 1: Distributed computing vs Heterogeneous computing

Distributed computing	**Heterogeneous computing**
• Two or more nodes work with each other in a coordinated manner • Distributed network of computers • Each node/computer has its own private memory • Message-parsing communication (e.g. MPI - Message Parsing Interface)	• All processors have access to shared memory • Single multicore/superscalar device

Table 2: Concurrency vs Parallelism

Concurrency	**Parallelism**
• A single system performing multiple tasks independently • Concurrent tasks may be executed at the same time (in parallel), but they don't have to	• Running two ore more activities in parallel (i.e. at the same time) • Explicit goal of increasing performance

"Parallel programs must be concurrent but concurrent programs need not be parallel". Amdahl's law depicts the maximum theoretical speedup we can achieve from exploiting inherent parallelism in a problem:

$$S = \frac{1}{B + (\frac{1}{n})(1 - B)} \tag{1}$$

S = speedup

B = serial fraction of the algorithm

n = number of processors (cores)

1.4 Current trends

One of the strongest trends in heterogeneous computing is the concept of heterogeneous uniform memory access. This technology completely removes the need for data copies between a CPU and a GPU to achieve full memory coherence. It basically reduces the overhead for communication between processors in a heterogeneous system. It is at this very moment used in APUs (Accelerated Processing Units) which combine a CPU and a GPU on the same chip. Such chips are already used in e.g. Xbox One and Playstation 4. Example of newly released chips are Intel i7-8550U or AMD Ryzen 2700U.

2 OpenCL

OpenCL is a framework for developing writing programs that execute across heterogeneous platforms. It is an industry standard for programming heterogeneous platforms and is maintained by the non-profit organisation Khronos Group. There are many alternatives to OpenCL such as OpenMP, MPI, POSIX threads, CUDA, C++ AMP, OpenACC, Renderscript etc.

OpenCL launched June 2008 as a collaboration between AMD/ATI, NVIDIA, Intel and Apple to push for an industry standard for writing heterogeneous computing software. It is intended simplify the writing of software for multiple platforms from cellphones to supercomputers. There are three SDKs:

- AMD APP (Accelerated Parallel Processing) - Works for all CPUs and GPUs
- CUDA (Compute Unified Device Architecture) - Works only for NVIDIA GPUs
- Intel SDK for OpenCL Applications - Works only for CPUs

2.1 OpenCL Architecture

Specification is defined in four different parts or "models":

- Platform model
- Execution model
- Memory model
- Programming model

2.1.1 Platform model

- The host - one processor coordinates execution
- The devices - one or more processors capable of executing OpenCL code
- Kernels - An abstract hardware model used by programmers when writing OpenCL functions that execute on devices

Defines a host connected to one or more computing devices. Memory is divided between host memory and device memory. Every device consists of one or more compute units. Compute units are divided into one or more processing elements. The partitioning of data becomes important to fully utilise all these compute units and processing elements.

2.1.2 Execution model

- How the OpenCL environment is configured on the host and kernels executed on the device(s)
- Includes setting up the context

Execution of an OpenCL program occurs in two parts:

- Kernels that execute on one or more OpenCL devices
- A host program that executes on the host - defines the context for the kernels and manages their execution

To execute a kernel it is placed in the command queue. Figure 1 illustrates what would be the kernel code in the case of three nested loops. Each kernel would basically only consist of the innermost loops calculations. This gets rid of all the addition and comparison operations made in the loops.

```
for(i = 0; i < Z; i++) {
    for(j = 0; j < X; j++) {
        for(k= 0; k < Y; k++) {
            process(point[i][j][k]);
        }
    }
}
```

Figure 1: Sequential nested loops

When a kernel is executed on a device, they are executed by something called work-items. A work-item is a single implementation of of the kernel on a specific set of data, e.g. process(point[0][1][2]). A work-item has a uniquely identifying global-id as well as local-id within a work-group. A work-group is a collection of work-items who can all access the same processing resources. Advantages of work-groups is that they have access to "local memory", which is a high-speed memory only accessible by the specific work-group. A work-group is (in OpenCL) run on a single compute unit and has a unique group-id. Each work-item also has a local-id within its work-group. For a work-item to access its designated data to process it uses these three: global-id, group-id and local-id to find the data in the memory-objects passed to the kernel.

2.1.3 Memory model

- Abstract memory hierarchy that kernels use, regardless of actual memory architecture
- Closely resembles current GPU memory hierarchies
- Four distinct memory regions: global memory, constant memory, local memory and private memory
- **Global memory:**
  - Permits read/write access to all work-items in all work-groups.
  - Work-items can read from or write to any element of a memory object.
  - Reads and writes to global memory may be cached depending on device.
- **Constant memory:**
  - A region of global memory that remains constant during the execution of a kernel
  - The host allocates and initialises memory objects placed into constant memory
- **Local memory:**
  - A memory region local to a work-group
  - Can be used to allocate variables that are shared by all work-items in a work-group
  - It may be implemented as dedicated regions of memory on the OpenCL device or it may be mapped onto sections of the global
  - Can't be read or written to by the host
- **Private memory:**
  - A region of memory private to a work-item

  - Variables defined in one work-item's private memory are not visible to another work-item
  - Can be initialised by the host

Data in global and constant memory is easy to work with but is slower than local or private memory. A good idea to load global memory to local memory and let the work-item process it in local memory instead. Local memory is also available to all work-items in a work-group so if data is dependent on other data, it is preferable to load it to local memory.

2.1.4 Programming model

- Defines how concurrency model is mapped to physical hardware
- OpenCL supports data parallel and task parallel programming models

2.2 OpenCL Framework

The OpenCL framework is divided into the following components:

- **OpenCL platform API** - Defines functions used by the host to discover devices and their capabilities and to create the context for the application. Platform, device, context.
- **OpenCL runtime API** - This API manipulates the context to create command-queues and other operations that occur at runtime. Command queues, buffers, program, kernels etc.
- **OpenCL programming language** - Programming language to write the code for the kernels

The first step of coding an OpenCL application is to code the host application:

- Identify and select a platform
- Identify and select a device
- Create a context associated with a list of devices
- Create a command queue to connect the context to the device
- Create memory objects - associated with a context
- Create a program from OpenCL source code - associated with the context
- Compile/build the program
- Create kernel objects from the compiled program
- Set kernel arguments
- Enqueue a kernel object in a command queue associated with a context

When creating memory objects, they must be associated with some sort of input or output data. Memory objects are then transferred to the device and set as kernel arguments. When the kernel is executed they are executed on the data associated with the memory objects. A kernel argument can be:

- Pointer to primitive data
- Pointer to a buffer object (memory object)
- Pointer to an image object (memory object)
- NULL (to reserve memory on the device)

2.2.1 Memory objects

Memory objects are OpenCL data that can be moved on and off devices. They are classified as:

- **Buffers** - Contiguous sequence of addressable elements similar to a C array. Read/write capable
- **Images** - Opaque objects (2D or 3D). Can be accessed in the kernel via read_image() and write_image()

Buffer objects can be a scalar data type, vector data type or a user defined structure. They package any data that is not an image. Certain attributes are associated with a buffer object, they can be read-only, write-only or read-write. Image objects are used for image processing (naturally) and can have the same attributes as buffer objects.

2.3 Kernel programming

- Must begin with keyword __kernel
- Must have return type void
- Must have arguments (for some platforms)
- All pointers passed to a kernel must have access space qualifiers (__global, __local, __constant or __private)
- Use API calls (such as get_global_id) to determine data

When programming a kernel, there are many built-in functions to use for e.g. vector comparison, data transfer, shuffling, selecting etc. Operators work as in C99 and CAN work for vectors if applicable. There are comparison functions such as all or any for vectors.

2.4 Kernel execution

To execute a kernel, it's arguments must be set explicitly. If the arguments are memory objects, data must be transferred to them. This can be done either at creation or by calling "clEnqueueWrite(Buffer/BufferRect/Image)". After this the kernel is enqueued and executed. When finished, data must be transferred back to the host before it is accessible. This is done by calling "clEnqueueRead(Buffer/BufferRect/Image)". Data transfers can be set to block the application until finished or they can be done asynchronously to allow the application to continue while the transfer is being done. Data can also be "mapped" instead of being written/read. This allows the memory object to be modified on the host, as well as allowing just a small piece of the entire memory object to be read.

2.4.1 Global and local size

OpenCL can execute applications with thousands of work-items. The upper limit of the number of work-items is set by the maximum value of size_t, which typically would be 65535. Therefore, to operate on larger data sets, one work-item per piece of data is not possible. One way to get around the problem is to use vectors such as float4. This effectively divides the total number of work-items by 4.

If the host application specifies the local-size the kernels should use, it must also specify the global-size to be a multiple of the local-size. The maximum number of work-items in a work-group is constant and depends on the device resources and capabilities. The more memory a kernel requires, the fewer work-items can be contained within a work-group. The preferred work-group size can be polled by the host application. The local-size should be set to a multiple of this polled size to be as efficient as possible. Further, a program should strive to have as many work-items in a work-group as possible.

2.4.2 Memory banks

Memory banks are hardware units that store data. Local memory is organised into memory banks. The sizes of these banks depend on the device architecture but there are typically 16 or 32 banks per local memory block. The data in these banks is interleaved so that sequential 32-bit values are placed into adjacent memory banks. Each bank can be accessed independently, so if work-items access different banks at the same time, their read/write operations will be executed in parallel. If two work-items try to access the same bank at the same time, a bank conflict occurs. This means that one of the work-items stalls until the other has finished reading/writing. The only exception is when all work-items try to access the same memory bank, then this memory bank can "broadcast" the data element to all work-items which doesn't incur any additional latency.

2.5 Events and synchronisation

Events are data structures that correspond to an occurrence. There are two types of events:

- **Command events** - associated with a command's execution
- **User events** - associated with an occurrence on the host application

An event object can be used to track the execution status of a command and can also be used to synchronise between commands (to create a dependency graph of commands). They are also used to store timing information returned by the device. The main uses of events are:

- **Host notification** - Notify the host that a command has finished execution and to schedule asynchronous data transfers between the device and the host.
- **Command synchronisation** - Force commands to delay execution until another event's occurrence has taken place
- **Profiling** - Monitor how much time a command takes to execute and to profile an application to see the execution flow.

After the host enqueues a command, it has no control over how the command will be processed. With events, it can receive notification when it is done. The event can be associated with a callback function which then executes as soon as the event is triggered. A callback function for an event must be set **after** the command-enqueueing function. To declare that a function is a callback function, it must have return type void and is preceded by the keyword "CL_CALLBACK".

2.5.1 Command queue

By default, command queues will operate in an "in-order" fashion. This means that the commands are executed in the order in which they are enqueued. A command queue can also be specified to be "out-of-order". Then there is no guarantee for which order the commands are executed. See table 3 for attributes associated with the different queue types.

Table 3: In-order vs Out-of-order command queues

In-order	Out-of-order
• Each command executes after the previous one has finished • Memory transactions have consistent views	• Commands are executed as soon as possible without ordering guarantees • All memory operations occur in a single memory pool • Out-of-order queues result in memory transactions that will overlap and clobber data without some form of synchronisation

Every enqueueing command can specify how many and what events a certain command must wait for before starting to execute. With sufficient synchronisation and events, an order can be established for an out-of-order command queue. Thus, one can realise a dependency graph. There are also commands that can be enqueued on a command queue to synchronise execution. These are:

- **Marker** - Is not completed until all commands preceding it are completed. "Marks" this specific location in the queue with an event (must be associated with an event)
- **Wait** - Stalls commands until the events in its wait list have reached completed state
- **Barrier** - Prevents any commands following it from executing until all commands preceding it are completed

The host application can also use "clFinish" and "clFlush" to ensure commands are executed before enqueueing more. "clFinish" blocks until all commands in the queue are completed while "clFlush" flushes all commands to the device but does not block until completion.

Events can also be used to profile the application. In this case, an event is associated with a command. When the command is finished, timing information is stored in the event and can be retrieved by the host application.

2.5.2 Work-item synchronisation

Work-items are typically executed in disordered, non-deterministic fashion. This is fine when work-items access completely different data, but causes problems as soon as work-items need to access the same data. There are two methods to synchronise work-items in the same work-group:

- **Fences and barriers** - A barrier forces a work-item to wait until all other work-items have reached the barrier. This can be used if work-items depend on intermediate results from other work-items. Barriers and fences share the same types of flags - CLK_LOCAL_MEM_FENCE and CLK_GLOBAL_MEM_FENCE. The local variant specifies that the barrier/fence will affect memory operations in the local memory. Corresponding applies for the global.
- **Atomic operations** - An atomic operation cannot be interrupted. As soon as a work-item uses an atomic operation, it forces other work-items wanting to access the same memory to wait until it is done.

Work-items within a work-group can do something called asynchronous data transfer to transfer large amounts of data between global and local memory. This allows the work-item to continue execution while data is being transferred but also requires synchronisation so the work-item does not access the memory until it has finished transferring.

2.6 Performance issues

It is hard to construct a large-scale OpenCL application. Performance can be improved by e.g.

- **Loop unrolling** - If number of iterations is known in advance, the iterations can be coded separately. This removes the need for comparison and addition operations associated with the loop statements.
- **Private variables** - Store small variables in private memory instead of local memory as long as sharing is not an issue. Reuse the same private variables throughout the kernel instead of creating new ones all the time. To avoid naming confusion one can create coding macros which all point to the same variable. Example:

```
//Macros pointing to the same variable tmp.
#define EXP tmp;
#define SINE tmp;
#define COUNT tmp;

int tmp = 2;
```

- **Avoid bank conflicts** - Access local memory sequentially to avoid local memory bank conflicts.
- **Modulo operator** - Avoid the modulo operator if possible. It requires a significant amount of processing time on a GPU and other OpenCL devices. It is fast on a CPU, but not a GPU.
- **FMA function** - For multiply-and-add operations use the fma function if it's available. If the FP_FAST_FMAF macro is defined one can compute $a * b + c$ with greater accuracy by calling the function fma(a, b, c). The processing time of this function is less than or equal to computing $a * b + c$.
- **Inline non-kernel functions** - The inline modifier tells the compiler that each call to the function should be replaced with the complete function code. Not memory efficient, but saves processing time by removing the context switching and stack operations.
- **Avoid branch miss** - By coding conditional statements to be true more often than they are false, one can avoid a branch miss penalty. Many processors predict that a statement will be true and prefetch the address corresponding to a true result. If the statement is false it needs to clear the pipeline and load instructions from another address.

2.7 OpenCL motivation

The big idea with OpenCL is to replace loops with kernels. E.g. to process each pixel of a 1024x1024 image, a sequential approach would require a loop to run $1024 * 1024 = 1048576$ times. To make this more efficient, a kernel can be constructed and run on e.g. a GPU. It would require the same number of kernel invocations, but since a GPU has thousands of cores which execute simultaneously and as an addition can utilise SIMD hardware, the execution time is drastically reduced.

3 Parallel programming

There are different types of parallelism important for GPU computing:

- **Task parallelism** - Ability to execute different tasks within a problem at the same time
- **Data parallelism** - Ability to execute parts of the same task (i.e. different data) at the same time

The concepts of task parallelism and data parallelism are not separate. They are often combined to achieve better performance. E.g. to find the number of occurrences of a string in a body of text. Some types of hardware are better suited for a certain type of parallelism.

- Multicore superscalar processor - Task parallelism
- Vector or SIMD (single instruction multiple data) processors - Data parallelism
- Multicore SIMD processor - Data parallelism

3.1 Task parallelism

E.g. to filter a set of images using fast fourier transform. Consists of three separate tasks: an FFT, a filtering and an inverse FFT. Each task can execute on different compute nodes simultaneously.

3.2 Data parallelism

E.g. multiply two vectors. Each element must be multiplied individually $\Rightarrow$ no dependence on other elements.

3.3 Decomposition

To be able to construct a parallel program the concept of decomposition is often used to deconstruct a problem into different tasks (task decomposition) or divide a data set into discrete chunks that can be operated on in parallel (data decomposition). The choice of how to decompose a problem is based solely on the algorithm.

3.3.1 Task decomposition

Reduces an algorithm to functionally independent parts. Tasks may have dependencies on other tasks (e.g. if the input of task B is dependent on the output of task A, then B is dependent on A). Tasks that are not dependent can be executed in parallel (naturally). To assist with decomposing an algorithm into different tasks, one can construct a task dependency graph. In such a graph, tasks that are not connected can be executed in parallel, see figure 2.

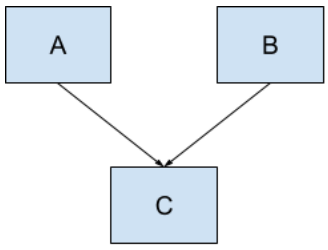

Figure 2: Dependency graph

Here task A and B are independent, but C is dependent on both A and B.

3.3.2 Data decomposition

Breaking down work into multiple independent tasks, but where each tasks has the same responsibility (i.e. each task processes a different piece of data). Data decomposition can be thought of as output decomposition or input decomposition. Output decomposition is useful when (for example):

- 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

Valid when the algorithm is based one-to-one or many-to-one functions. Input decomposition is similar to output but makes sense when the algorithm is a one-to-many function, e.g.:

- Histogram created by placing each input datum into one of many fixed number of bins
- Search function may take a string as input and look for the occurrence of various substrings

3.4 Parallel computing

Example applications:

- Video games
- Computer graphics
- Computer vision
- Machine learning
- Database operations

Some parallelism is automated - instruction-level parallelism. This is when multiple CPU instructions are performed at the same time. Covered here: higher-level parallelism (threading). A program may consist of several subprograms that are allowed to run concurrently in a separate thread. Each thread has its own local memory and can see the same set of global variables as all other threads. This requires synchronisation so no two threads operate on the same data at the same time.

3.5 Parallel software

GPU programs are called kernels. They are written using the Single Program Multiple Data (SPMD) model. SPMD executes multiple instances of the same program independently, each program works on a separate portion of the data. One way of realising this would be to create one thread for each portion of data. For a CPU, the overhead of creating a thread is very large so it is better to let each thread get a big chunk of data. For GPUs on the other hand, thread creation is very cheap so one can be created per iteration of a loop (e.g.).

3.6 Parallel hardware

A SIMD processor can have several processing elements. Each processing element can execute the same instruction with different data at the same time. A single instruction is executed simultaneously on many ALUs (Arithmetic Logic Units). The number of ALUs per elements is the SIMD width. SIMD processors are efficient for data parallel algorithms. A SIMD unit with a width of four could execute four iterations of a loop at the same time. All current GPUs are based on SIMD hardware. The GPU maps each SPMD thread to a SIMD "core", of no concern to the programmer.

3.7 Parallel patterns

Serial iteration patterns maps to several different parallel patterns. It depends on whether and how iterations depend on each other. Most patterns require a fixed number of invocations known in advance.

3.7.1 Map

Replicates a function over every element of an index set. The index set may be abstract or associated with the elements of an array. Map replaces independent operation iterations in serial programs.

3.7.2 Reduction

Combines every element in a collection into one element. Reordering of the operations is often needed to allow for parallelism. A reordering requires the operations to be associative, i.e. $(a + b) + c = a + (b + c)$ must hold.

3.7.3 Geometric Decomposition

Breaks an input collection into several sub-collections. Partitioning is a special case where sub-collections do not overlap. The operation does not move data, just provides an "alternative" view of its organisation.

3.7.4 Stencil

Applies a function to neighbourhoods of a collection. Neighbourhoods are given by set of relative offsets. Boundary conditions need to be considered, but majority of computation is in interior.

3.7.5 Pipeline

Parallelise pipeline by

- Running different stages in parallel
- Running multiple copies of stateless stages in parallel

Running multiple copies of stateless stages in parallel requires reordering of outputs. Need to manage buffering between stages.

4 Parallel Image Processing

Image processing is one application of OpenCL. Image objects are a desirable data structure as opposed to buffers for some reasons:

- On GPUs, image data is stored in special global memory called texture memory. Texture memory is cached for rapid access.
- Read/write functions for image data can be invoked without regard to how the pixel data is formatted.
- OpenCL build-in functions that return image-specific information such as image dimensions, pixel format and bit depth.
- Optimised memory access - some GPUs store images in a Z-order. This increase the spatial locality of reference. Basically, this results in adjacent pixels are also stored in adjacent memory locations to optimise memory usage (cached).

If a device supports images, it **must** support the RGBA as well as the BGRA format. Data types for the pixels vary between channel orders. A basic example of image processing done with OpenCL is image convolution. This modifies each pixel by using information from neighbouring pixels.

4.1 Samplers

Samplers are a data structure that store information about how colour information is read. Kernels need certain information such as:

- How coordinates are formatted (normalised or non-normalised)
- How to interpret coordinates that go beyond the image's size
- How to interpolate colours between pixel values

If a sampler is configured incorrectly, the kernel will read the pixels from the image incorrectly. As stated, coordinates can be either normalised or non-normalised. If they are normalised, they have a floating point value range of [0.0 , 1.0]. If they are non-normalised, they have a range of [0 , DIM_MAX-1], where DIM_MAX is the width, height or depth of the image respectively.

Colours can also be normalised. This removes the colours intensity level by dividing each component by the sum of the components. They have the same [0.0 , 1.0] range as coordinates. If they are non-normalised they have a range of [0 , 255].

4.1.1 Addressing mode

Samplers also need an addressing mode to determine how it interprets coordinates that go beyond the image's size. There are five different modes:

- **CL_ADDRESS_NONE** - Colour values beyond max undefined
- **CL_ADDRESS_CLAMP** - Colours beyond max set to a specific border colour, black by default
- **CL_ADDRESS_CLAMP_TO_EDGE** - Colours beyond max are set equal to the pixels at the edge of the image
- **CL_ADDRESS_REPEAT** - In-range coordinates are repeated. An out-of-range pixel X is set to X mod N, N = DIM_MAX. Only works for normalised coordinates
- **CL_ADDRESS_MIRRORED_REPEAT** - Out-of-range coordinates set equal to the reflection of their corresponding in-range values. Only works for normalised coordinates

4.1.2 Interpolation and filter mode

If coordinates are specified with integers, i.e. non-normalised, the sampler always gets the colour value that corresponds to the given pixel. However, if coordinates are normalised, results will need to interpolated, i.e. computing an unknown data point between known data points. OpenCL supports two methods of image interpolation:

- **CL_FILTER_NEAREST** - Nearest-neighbour interpolation
- **CL_FILTER_LINEAR** - Bilinear interpolation (2D) or Trilinear interpolation (3D)

4.2 Image processing functions

Basic image processing functions are in-built such as read, write and information functions. Read and write are implemented in three different ways depending on the data type of the image.

- **Read_imagei** - returns an int4 vector
- **Read_imageu** - returns a uint4 vector
- **Read_imagef** - returns a float4 vector with values determined by the pixel format. SNORM - [-1.0 , 1.0], UNORM - [0.0 , 1.0], FLOAT - regular floating point values.