# FlowCL Application Developers Perspective – FlowCL.hpp

Sytse van Geldermalsen

July 8, 2013

## 1 Motivation

The principles of FlowCL lie with simplicity, abstraction, ease of application development and prototyping. With the dataflow model, a complex algorithm can be seen as a subset of possible independent operations. These operations may require an input and optionally produce an output. It is possible to model algorithms as a dataflow graph, which explicitly exposes parallelism allowing to execute a subset of independent operations concurrently. FlowCL uses dataflow model to create programs both in concept and execution strategy. The framework can mix different levels of granularity. It can be fine grain, since the operations that run on the device can be split at the highest level of granularity, and it can also be coarse grain, because there can be multiple discrete operations executing concurrently, performing a larger task. By presenting an intuitive API to program closely with the dataflow concept, an application developer can rapidly prototype and develop applications.

## 2 Features

The framework has the following features:

1. Increase ease of application development with OpenCL.

   The framework completely hides all low level OpenCL library API from the application developer. Only the OpenCL kernel code that is designed to run the selected device must be provided to the framework. Prototyping, experimentation and ease of construction of different flows is paramount.

2. Provide object oriented declarative API to easily build an application with the concept of dataflow

The programmer simply declares a set of memory objects and operations. The operation runs a single kernel function on any available device, with kernel arguments either constants or previously defined memory. Dependencies between operations are simply declared to enforce memory consistency. With these simple constructs, a graph can be constructed to run on a heterogeneous platform.

3. Automatically apply optimization strategies

   After designing a graph, the framework inherently applies optimization strategies such as overlapping communication and computation, asynchronous data transfers and kernel executions. The framework runs independent threads for each operation and memory transfer needed.

4. Support multiple operating systems

   The framework will support both the Windows and Linux operating system, providing operating system heterogeneity.

# 3 Object Oriented Declarative API

This approach to hiding OpenCL API is creating an abstraction layer that matches the dataflow model terminology.

The FlowCL cardinality diagram is shown in Figure 1. These four objects are enough to create an application with the dataflow style of execution on a heterogeneous platform.

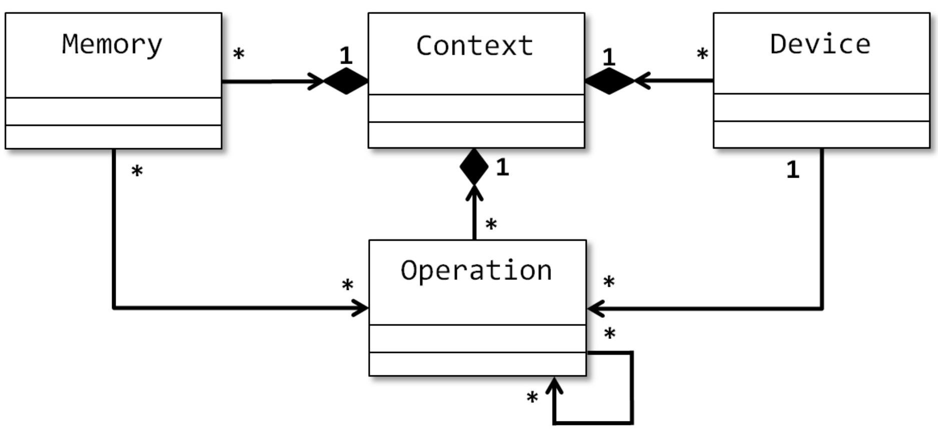


Figure 1: The FlowCL cardinality diagram

**Context**

The `Context` is instantiated with kernel source code from a file or a string, where functions in that kernel will be usable on all of devices found on all of the available platforms on the system. All other objects are created from this context.

**Operation**
The `Operation` runs a user provided kernel function on a selected device. This kernel can have arguments such as constants or memory. To create dependencies between `Operations`, the application developer declares the dependency of a `Memory` object used as an argument by another `Operation`.

**Memory**
`Memory` is created by the context with a given size, and will be available to all of the devices in that context.

**Device**
The `Device` represents an accelerated device found on the system. This `Device` can be used in one or more `Operations` to execute the user specified kernel function, together with any `Memory` arguments or constants.

## 3.1 Context

Figure 2 presents the important functions to utilise the `Context` class. With only these few functions, a developer has sufficient control over heterogeneous computing environment.

```
Context
-------------------------------------------------
...
-------------------------------------------------
void CompileFile( filename )
void CompileSource( source )
Memory CreateMemory( size )
Operation CreateOperation( Device, funcname )
Operation CreateOperation( Device, userfunc )
list<Device> GetDevices()
void Run()
...
```

Figure 2: The Context Class

`CompileFile(filename), CompileSource(source)` create an instance of `Context` that accepts a user specified file path, or source string of the source code to compile for all the devices on the heterogeneous platform.

`GetDevices()` return a list of all the available devices on the heterogeneous platform that can be used by the `Context`.

`CreateMemory(size)` create memory that will be usable on all devices and return the memory object.

`CreateOperation(Device, funcname)` return an `Operation` object that executes the given `funcname` found in the kernel code, on the given `Device`. `funcname` can also be a native C/C++ function.

`Run()` run the created graph made up of `Operations` and their dependencies. This function returns once the graph has finished executing.

## 3.2 Memory

The `Memory` class in Figure 3 is the simplest class, representing reserved memory that is available to all devices. The memory consistency between devices doesn't have to be managed by the user, the framework automatically handles this.

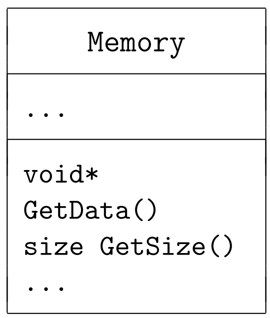


Figure 3: The Memory Class

`GetData()` return a reference to the raw data that is resident on the host. This pointer to data can be modified to be the input data, or contains the result of an execution.

`GetSize()` return the size of the reserved memory

## 3.3 Operation

In Figure 4, the `Operation` class represents an operation that runs on a given `Device` with a given `kernelname` or if the device supports it, a native C/C++ function `userfunc`. This operation is the highest form of abstraction, as it can be run on any device, with any memory, while the underlying framework handles the rest.

```
Operation
---
...
---
void SetArg( index, Memory )
void SetArgConstant( index, const )
void SetArgInput( index, Memory, CopyInterval )
void SetArgOutput( index, Memory, CopyInterval )
void SetArgDependency( Operation, index, Memory,
CopyInterval )
void SetArgCopyInterval( index, CopyInterval )
void SetWorkSize(...)
...
```

Figure 4: The Operation Class

`SetArg(index, Memory)` set the argument at the `index` of the `Operation`'s `kernelname` or `userfunction` to be a `Memory` object. Here, `Memory` explicitly does not need to be copied from host memory to the `Operation`'s device.

`SetArgConstant(index, Const)` set the argument at `index` to be a constant value.

`SetArgInput(index, Memory, CopyInterval)` set the argument at `index` to be a `Memory` object. Here, `Memory` is explicitly copied from the host to the `Device` prior to execution. Optional parameter `CopyInterval` will set how many times this `Memory` must be copied per graph execution: `CopyNever`, `CopyOnce`, or `CopyAlways`. This is very important for iterative flows; for example, if `Memory` is already resident on the `Device`, then there is no need to transfer data to that Device. This way the `Context.Run()` can be run multiple times, without unnecessary data transfers to and from that device.

`SetArgOutput(index, Memory, CopyInterval)` is analogous to `SetArgInput()`, except instead of copying `Memory` from host do `Device`, copies `Memory` from `Device` to host.

`SetArgDependency(Operation, index, Memory, CopyInterval)` set an argument which is a dependency of a parent `Operation` at `index` with a `Memory` object in question. The operation will now only execute after the parent operation(s) have completed execution, and updated their respective `Memory` objects.

`SetCopyInterval(index, CopyInterval)` set the `CopyInterval` of an arguments `index` `Memory` object. This enables modification of the copy intervals during various `Context.Run()`s.

`SetWorkSize(...)` set the granularity of the `Operation`. The granularity depends on how the work size is distributed over the "threads" on the device. If the granularity number is the same as the number of elements in the memory that the device will work on, this represents the finest granularity.

### 3.4 Device

The `Device` class in Figure 5 represents a device on the heterogeneous platform. Interopability of `Devices` of different vendors and their `Memory` are automatically handled by the framework.

```
Device
---------------------------
...
---------------------------
string GetName()
string GetVendorName()
bool IsCPUDevice()
bool IsGPUDevice()
bool IsDedicatedDevice()
...
```

Figure 5: The Device Class

`GetName()` return the name of the `Device`.

`GetVendorName()` return the vendor name of the `Device`.

`IsCPUDevice()`, `IsGPUDevice()`, `IsDedicatedDevice()` return whether it is a CPU device, a GPU device, or a Dedicated Device (not CPU) respectively.

# 4 Visual with Programming Example

The following scenario brings both a visual and programming example of a simple graph constructed in FlowCL. This gives a better perspective how the declarative API closely ties in with the visual representation. Suppose the following scenario: a large amount of random numbers are to be generated, and then sorted accordingly. On a heterogeneous platform, sorting of large datasets can be computationally expensive for the CPU, we can speed this up by adding the computing power of the GPU into the solution. This enables us to split the work between the CPU and the GPU. Figure 6 shows the graph of a possible solution.

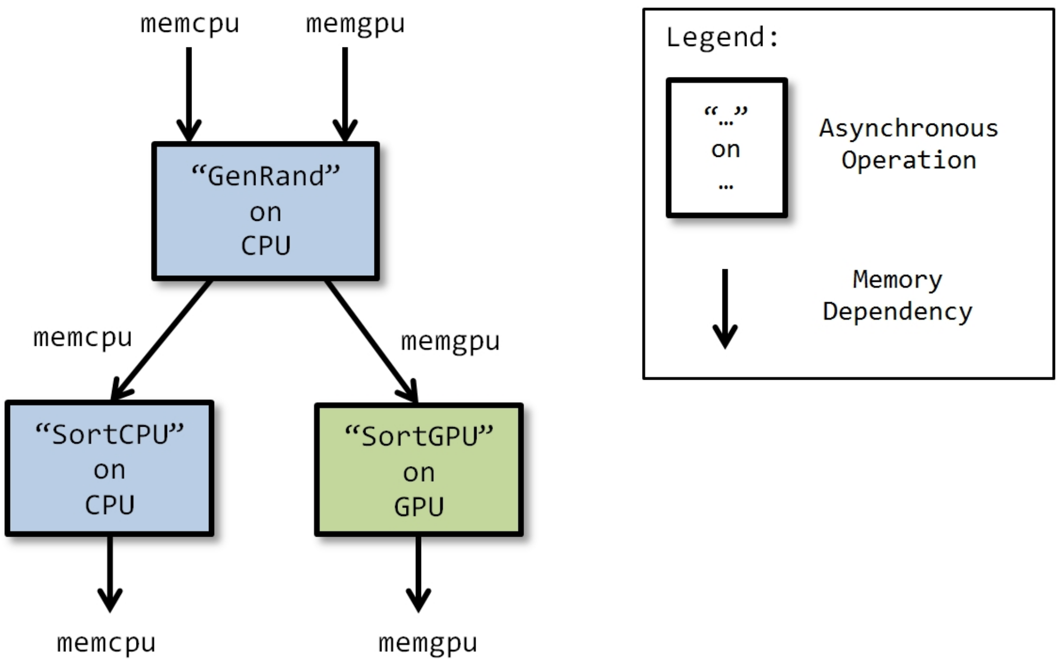


Figure 6: Sample code visual representation

The graph has three `Operations`, with their required memory dependencies. Firstly, the "GenRand" operation generates random numbers on the CPU, and stores them in the memory objects `memcpu` and `memgpu`. These two memory objects will subsequently be required by the following `Operations` "SortCPU" and "SortGPU", their content to be sorted on their respective devices. Finally, the two `Operations` will output the sorted data.

By visualising the graph it becomes easy to see how the CPU and GPU will sort data concurrently and independently. The memory objects and operations are all independent, and handled concurrently. The data transfers and execution on the devices run as soon as the data is readily available. This is a simple way of exploiting parallelism on a heterogeneous platform. For the actual code, see the following programming example in listing 1.

```cpp
#include "FlowCL.hpp"

int main()
{
  using namespace FlowCL;
  Context con;
  con.CompileFile("source.cl");

   // Create 389mb test size
  Memory memcpu = con.CreateMemory( 1e8 * sizeof(int) );
  Memory memgpu = con.CreateMemory( 1e8 * sizeof(int) );

  Operation genrand = con.CreateOperation( con.GetCPUDevice(), "GenRand" );
  genrand.SetArg( 0, memcpu ); // CPU already has access to memory
  genrand.SetArg( 1, memgpu );
  genrand.SetWorkSize( 1e8 ); // Set finest granularity

  Operation sortcpu = con.CreateOperation( con.GetCPUDevice(), "SortCPU" );
  sortcpu.SetArgDependency( genrand, 0, memcpu ); // Wait for genrand
  sortcpu.SetWorkSize( 1e8 );

  Operation sortgpu = con.CreateOperation( con.GetGPUDevice(), "SortGPU" );
  sortgpu.SetArgDependency( genrand, 0, memgpu ); // Wait for genrand
  sortgpu.SetArgOutput( 0, memgpu ); // Copy memory to host
  sortgpu.SetWorkSize( 1e8 );

  con.Run(); // Blocking run

  // Compare results
}
```

Listing 1: Sample code of sorting random numbers on CPU and GPU

By making use of the functions previously explained in section 3.1 - 3.4, it becomes clear how these constructs are used to create a graph. Note: the contents of the kernel source file `"source.cl"` are not shown here, the idea is that the source has three functions, `"GenRand"`, `"SortCPU"`, and `"SortGPU"`.