OpenCLV Manual

RaptorView, LLC

Table of Contents

OpenCLV Manual

RaptorView, LLC

1 Introduction

OpenCL for LabVIEW (OpenCLV) is a suite of tools to combine the power of LabVIEW with the power or OpenCL. OpenCLV gives users the power to:

- Quickly determine OpenCL platforms and devices
- Load any OpenCL program or kernel
- Easily allocate, write, and read from device memory
- Perform FFTs and Inverse FFTs using AMD FFT OpenCL Library
- Vector optimized functions usable without any OpenCL knowledge

2 Definitions

Common definitions:

- VI – Virtual Instrument
- OpenCL – Open Computing Language
- Platform – OpenCL implementation, usually from a company (Intel, AMD, NVIDIA, etc)
- Device – OpenCL compatible device (CPU, GPU, or Accelerator device)
- Host – The device that runs the operating system and allocates resources to devices and programs
- Program – File that contains OpenCL kernels
- Kernels – Functions that can be compiled to run on an OpenCL device
- Vectors – Certain OpenCL devices can group numerical data into vectors of 1, 2, 3, 4, 8, or 16 units that can be operated on in one clock cycle. Please see the OpenCL standard for more information.
- Memory Object – An LabVIEW control used to pass vital information about OpenCL memory buffers

3 OpenCLV Structure

OpenCLV's goal is to simplify OpenCL functions and combine them with the ease of LabVIEW. Work flow for OpenCLV tries to mimic LabVIEW when possible. OpenCLV comes in 2 main flavors, x64 or x86 and both flavors support single and double precision floating point numbers. Let's take a look at a typical OpenCLV VI icon:

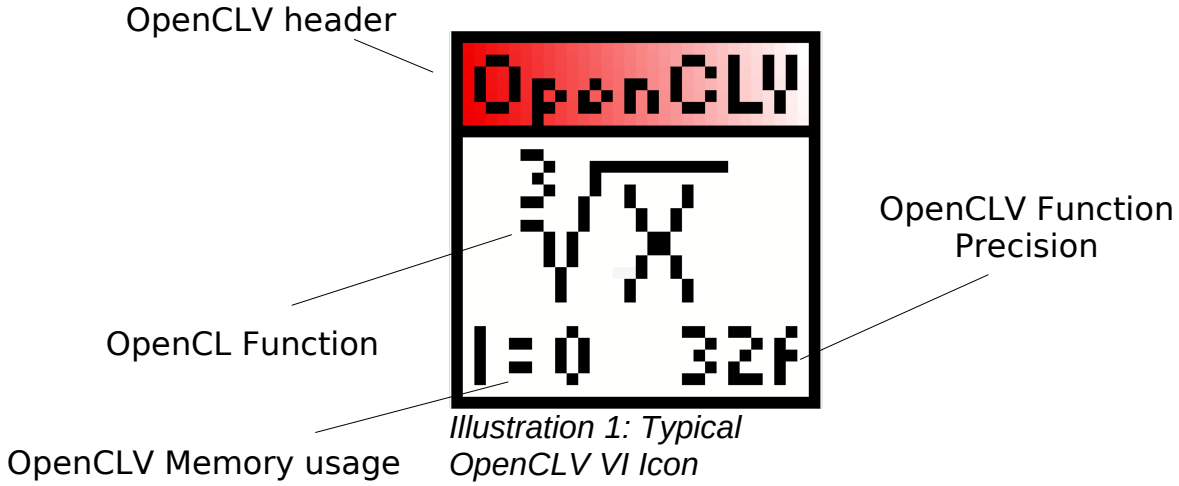

Illustration 1: Typical OpenCLV VI Icon

3.1 Icon Description

3.1.1 OpenCLV Header

Signifies that the VI is associated with the OpenCLV library.

3.1.2 OpenCLV Function

Signifies visually or via text which OpenCL function is implemented.

3.1.3 OpenCLV Memory Usage (Math Functions Only)

Signifies how the VI uses OpenCL memory. If the VI has an **I = O** icon, that means the output results are stored in the input memory. A VI without this text will store the output results in a separate, user defined Memory Object.

3.1.4 OpenCLV Function Precision

Signifies the type of numerical data that the OpenCLV VI operates on. This can be I8, U8, I16, U16, I32, U32, I64, U64, 32f, or 64f.

4 OpenCLV Array/Memory Dimension Orientation and Objects

4.1 Array Orientation

OpenCLV arrays are setup in a consistent, column-major format. A block of memory has a Depth, Height, and Width, all of which must be defined to perform even the simplest operation. Illustration 2 shows how OpenCLV defines Depth, Height, and Width of 2, 4, 3, indexing the first element at 0:

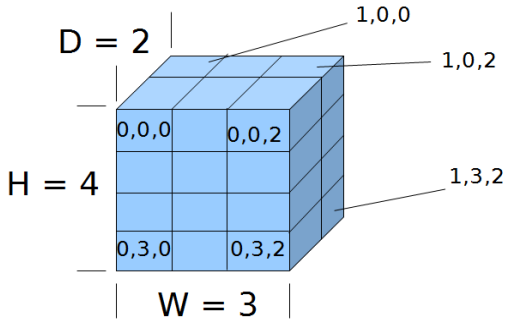

Illustration 2: OpenCLV Array Definition

For example, a 1D array of 1000 units would be defined as 1 x 1 x 1000. OpenCLV provides a **Array Dimension** control that users should use when working with arrays. Values of 0 are not acceptable when defining an array.

4.2 OpenCLV Memory Object

OpenCLV relies heavily on Memory Objects. Memory Objects are a special control that bundles a pointer to device memory, the device ID where the memory is allocated, and the size of the array. OpenCLV VIs will automatically create these memory objects, and it is **NEVER** acceptable to pass user defined Memory Objects to OpenCLV VIs.

5 Getting Started

OpenCLV can be broken into 3 phases

- Allocate – Allocate platforms, devices, and memory
- Operate – Write, Read, and Execute OpenCL commands on memory buffers
- Decimate – Deallocate memory, devices, and programs

5.1 Phase 1 - Allocate

OpenCL, and by default, OpenCLV is a bit different than regular LabVIEW and C/C++ programming. The main differences are during the Allocate phase. Before OpenCL code can be executed, a platform and device must be configured. Next, a program must be loaded and compiled, depending on the device chosen. OpenCL requires that programs be compiled at runtime, so this compilation only needs to be done once at the beginning. Finally, memory must be allocated.

OpenCL is much better at handling memory buffers that are static and not dynamic. What does this mean? It is better practice pre-allocate several buffers and to re-use these throughout the course of the program than it is to dynamically allocate and de-allocate memory on the fly. Constant allocation and de-allocation can cause memory fragmentation on the device eventually causing errors – it is best avoided by careful planning.

5.1.1 Starting OpenCLV.vi (Illustration 3)

OpenCLV makes use of functional globals to track devices, programs, kernels, and memory objects. **Start OpenCLV.vi** is vital to ensure that the funtional globals are refreshed, especially if the LabVIEW Abort Execution button has been used to prevent OpenCLV from closing properly.

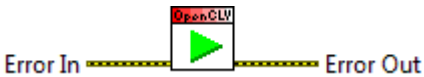

Illustration 3: Start OpenCLV.vi

5.1.2 Stop OpenCLV.vi (Illustration 4)

Just as OpenCLV needs to be started, it should be stopped as well. This tries to catch and delete any device, program, kernels, or memory leftover at the end of the program.

Illustration 4: Stop OpenCLV.vi

5.1.3 Create Device GUI.vi (Illustration 5)

OpenCLV provides an easy to use GUI interface to interrogate platforms and devices on the host. The GUI can be bypassed if needed once the user is familiar with the platforms and devices available, though these will change from host to host. Illustration 5 shows the block digram for platform and device configuration, while Illustration 6 shows the GUI that pops up when the VI is called. If **Skip Configuration** is **True,** the front panel is not displayed.

Platform and **Device** are integer values depending on the host, platform, and devices. For example, Illustration 6 shows three platforms, NVIDIA CUDA, Intel OpenCL, and AMD Accelerated Parallel Processing. It also shows one device for the NVIDIA platform, a GeForce GTX 560 Ti.

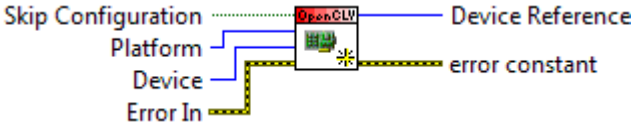

Illustration 5: Configure Platform and Device.vi

To bypass the GUI and configure the OpenCL device for the NVIDIA Platform with the GTX 560, **Skip Configuration** would be **True**, **Platform** would be **0**, and **Device** would be **0**.

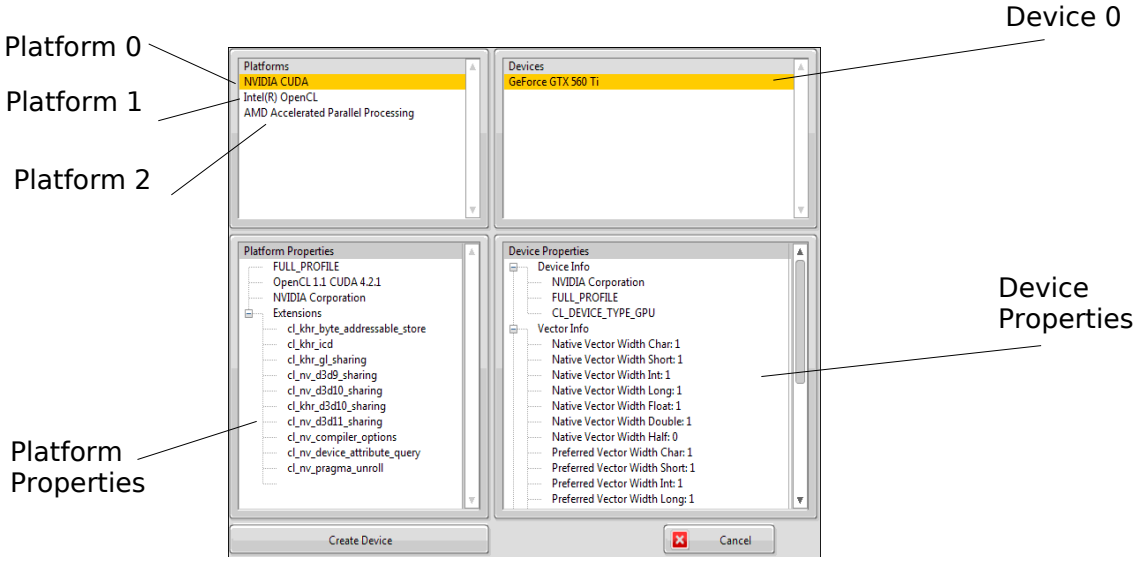

Illustration 6: Front panel for the Create Device GUI.vi

5.1.4 Load Program.vi (Illustration 7)

Loads a file from a specified path for the device specified in the **Create Device GUI.vi**. If the file compiles without an error, no further user action is required. If the code fails compilation, a front panel opens up that lets the user fix errors, save the modified file, open files, or cancel the compilation (Illustration 8). This VI can be run as a basic OpenCL editor and compiler, as seen in the **Basic Program Editor.v**i in the **Demo and Example Code** folder.

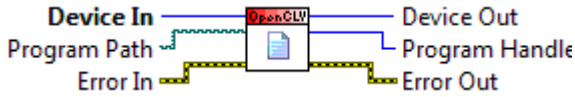

Illustration 7: Load Program.vi

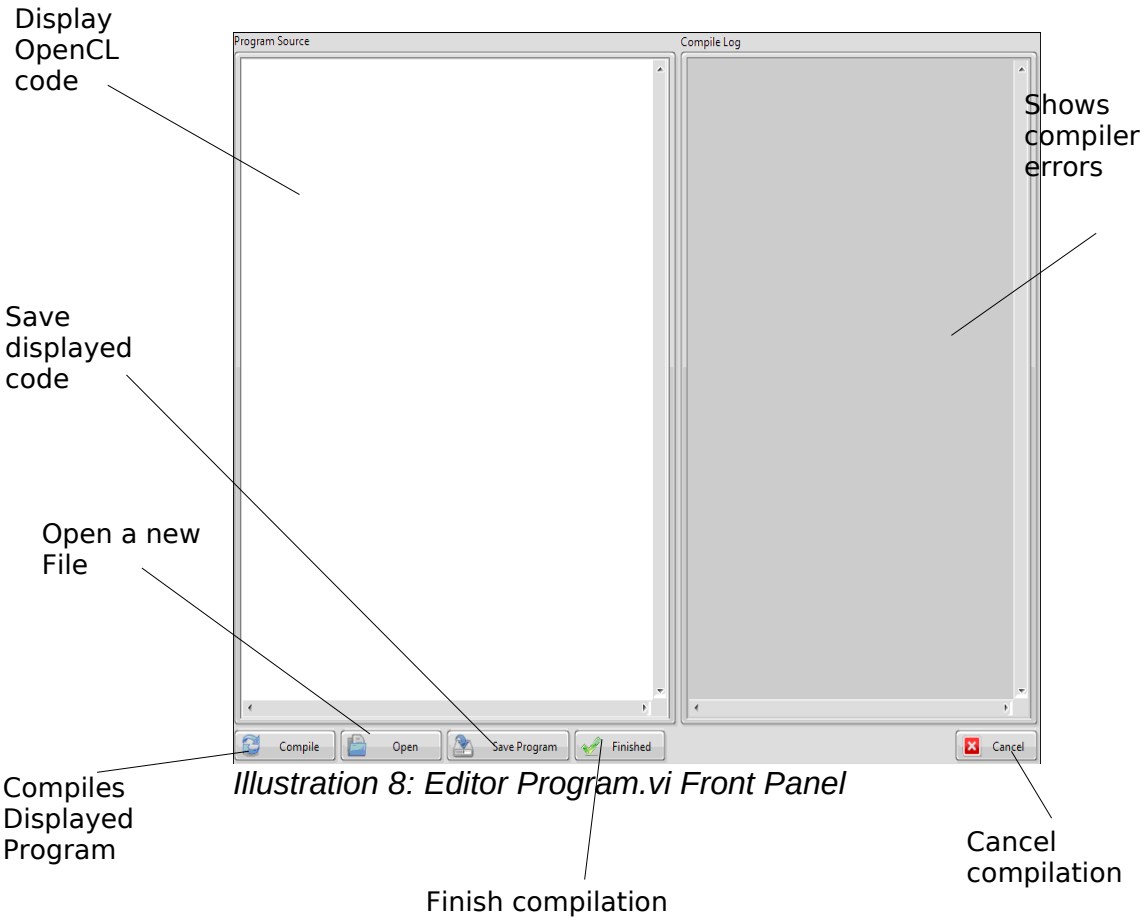

Illustration 8: Editor Program.vi Front Panel

The buttons on the **Editor Program.vi** front panel provide the following abilities:

- Compile – Compiles the text in the Program Source box for the input device
- Open – Opens a new file, the contents which are displayed in the Program Source box

- Save Program – Saves the text displayed in the Program Source box
- Finished – If the program compiles without an error, the Finished button becomes available. Pushing this button will load and compile the source code for the selected device.
- Cancel – Cancel the program editing. Throws an error so other OpenCL functions won't run.

5.1.5 Allocate Memory.vi (Illustration 9)

OpenCL handles memory a bit differently than LabVIEW and C/C++. It is best to allocate blocks of memory early and re-use the memory as needed, instead of allocating and de-allocating memory on the fly. Device memory can easily become fragmented, leading to device errors.

OpenCLV provides a single function to allocate memory, the **Allocate Memory.vi.** The **Allocate Memory.vi** is a polymorphic VI that requires a device and a vector dimension to properly allocate either a chunk of 32f or 64f memory. Other numerical data types will be supported in future releases, including images.

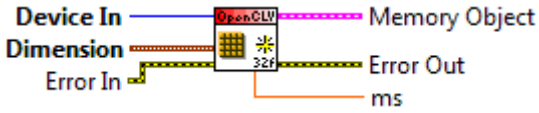

Illustration 9: Allocate Memory.vi for 32f

5.2 Phase 2 - Operate

Now that a device has been configured and memory allocated, it is finally time to operate. The operate phase will be the repetitive portion of program: writing to memory, reading to memory, and operating on memory. The memory may have been allocated, but remains empty until data is transferred from the host to the device. Once data is on the device it can be easily manipulated using built in OpenCLV libraries or custom OpenCL kernels. Data can also be read from the device back to the host for display user feedback.

5.2.1 Write Memory.vi (Illustration 10)

OpenCLV provides a polymorphic VI, **Write Memory.vi** that writes LabVIEW data to the device. **Write Memory.vi** takes input arrays from LabVIEW of size and type 1D, 2D, or 3D in either 32f or 64f numerical types. The memory written to the device should be the same size that was allocated.

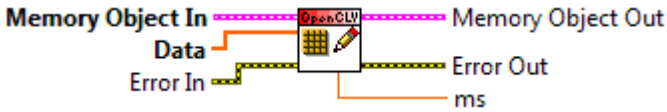

Illustration 10: Write Memory.vi showing polymorphic adaptation to a 1D array

Note: It is not acceptable to write more data to the device than allocated.

5.2.2 Read Memory.vi (Illustration 11)

OpenCLV provides a polymorphic VI, **Read Memory.vi**, that helps get the data off the device and back into LabVIEW. LabVIEW memory is automatically allocated based on the numerical type and array dimensions passed.

Note: Unfortunately, this VI doesn't automatically adapt to the type of memory object passed and requires the user to select the proper output format and numerical type.

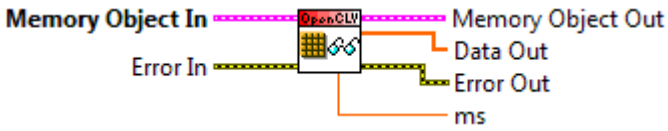

Illustration 11: Read Memory.vi block diagram icon

5.2.3 Perform computations on the device memory

The framework of OpenCLV is designed to get to OpenCL computation with as little difficulty as possible. Now that a device has been configured, a program loaded and compiled, and data written to memory, it is time to process the data.

OpenCLV implements the following functions as LabVIEW VIs in either 32f or 64f numerical types:

• add (scalar or array) • subtract (scalar or array) • multiply (scalar or array) • divide (scalar or array) • ceil • fabs • floor • pow • cbrt • copy • memset	• sqrt • rsqrt • log • log2 • log10 • exp • exp2 • atan • acos • asin • atan2	• exp10 • cosh • sinh • tanh • acosh • asinh • atanh • cos • sin • tan • hypot

These functions can write to either a new output memory (allocated in Phase 1) or store the output in the input memory location. These are provided as a convenience to get started as soon as possible. These functions are also vector optimized depending on the vector length the device prefers. For example, a 3rd generation Core i7 can perform four 32f operations in a single clock cycle.

6 Custom Kernels

The convenience functions provided in OpenCLV are no substitute for a custom kernel written specifically for a certain task. The device has multiple types of memory, some faster than others:

- Global Memory – This memory is where data is allocated in stored and is the slowest memory
- Local Memory – This is memory that work groups can access and is of moderate speed
- Private Memory – Each thread has a small amount of memory that only it can access and is the fastest memory available.
- Constant Memory – Stores constant, similar to Global memory, but read only

The speed is all relative and is based on the device. The OpenCLV convenience functions work almost exclusively with Global memory. For example, using the OpenCLV convenience functions to take the log10 of Memory 1, the log10 of Memory 2, and summing the output back to Memory 1 would require the following Global reads and writes:

- 2 – Read and Write Memory 1 for the log10 computation
- 2 – Read and write Memory 2 for the log10 computation
- 3 – Read from Memory 1, read from Memory 2, and a write the sum back to Memory 1

A custom kernel could perform the same operation in 3 Global memory reads and writes:

- 1 – Read from Memory 1
- 1 – Read from Memory 2
- 1 – Write to Memory 1

} The two log10 and one sum is done in private memory in each thread

OpenCLV provides the tools needed to write custom kernels for optimal results.

6.1 Load Kernel.vi (Illustration 12)

If a program has been successfully written, loaded, and compiled code for a certain device, loading the kernel is as easy as using the **Load Kernel.vi.**

The **Load Kernel.vi** requires a configured device (**Device In**), a compiled program (**Program Handle**), and a string of the name of the kernel (**Kernel Name**).

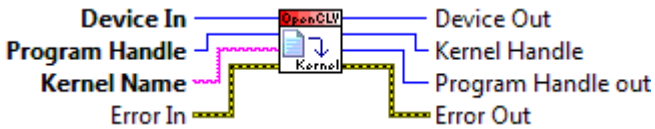

Illustration 12: Load Kernel.vi

```
__kernel void LogSum_32f(__global float* Memory1,
                         __global float *Memory2){

int k = get_global_id(0); //depth
int j = get_global_id(1);  //height
int i = get_global_id(2);  //width

int linear_coord = i + get_global_size(2)*j + get_global_size(1)*get_global_size(2)*k;

Memory1[linear_coord] = log10(Memory1[linear_coord]) +
                        log10(Memory2[linear_coord]);
}
```

Text 1: Example Kernel

In the example provided in Text 1, the **Kernel Name** would simply be LogSum_32f. Illustration 13 shows how this would look in LabVIEW **Example Program** and kernel provided in the **Demo and Example Code** folder.

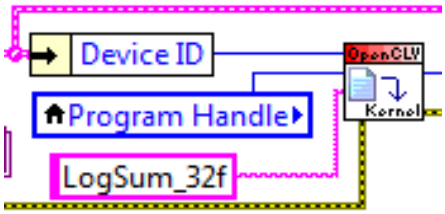

Illustration 13: Loading a Kernel in LabVIEW

6.2 Set Kernel Arguments

After a kernel has been loaded, OpenCL requires that the arguments be set. In our current example, argument 0 is Memory1 and argument 1 is Memory 2. Very simple in this case. Again, OpenCLV provides convenience functions to set the function arguments, whether the data be a single value or a vector.

There are currently nine different Set Kernel Arguments functions, one for: I8, U8, I16, U16, I32, U32, I64, U64, 32f, 64f, and one for Memory Objects. Of these nine Set Kernel Argument functions, 8 are used for sending constant data to a kernel. The ninth, by far the one that will be used the most with custom kernels, can be found in Illustration 12.

Even though the Memory 1 and Memory 2 are float pointers, they are pointing to memory on the device, and thus **MUST** be referenced using **Set Kernel Arguments Memory Object.vi (Illustration 14)**.

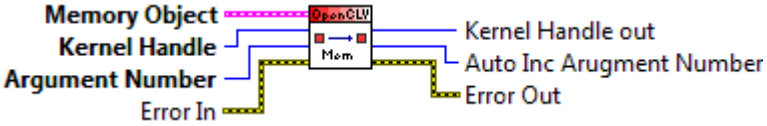

Illustration 14: Set Kernel Argument for Memory Objects

There are three required inputs for **Set Kernel Argument**, the input data (**Memory Object** in this case), a **Kernel Handle**, and an **Argument Number**. The first time **ANY** Set Kernel Argument VI is used, a **0** must be wired to the **Argument Number**. Each **Set Kernel Argument** call, regardless of the type, automatically increments the input number by one and outputs this number for the next Set Kernel Argument VI to keep work flow clean.

The **Kernel Handle** input will come from the **Load Kernel.vi**, while the **Memory Object** will come from an allocated memory object.

In the ongoing example, there would need to be two **Set Kernel Arguments Memory Object.vi** calls as seen in Illustration 15:

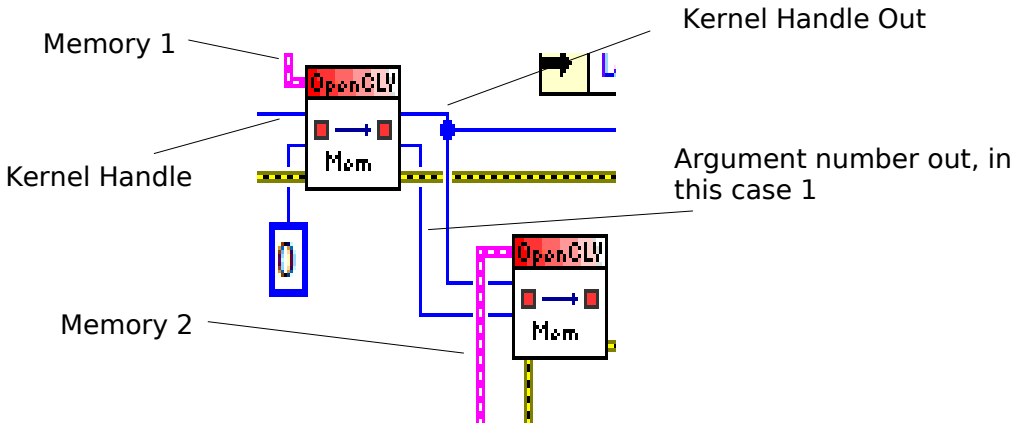

Illustration 15: Setting arguments to LogSum_32f

6.3 Execute Kernel Simple.vi (Illustration 16)

Now it is time to use the **Execute Kernel Simple.vi** and actually get some processing done. To execute a kernel, OpenCLV needs a **Device**, a **Kernel Handle**, and the **Thread Dimension** of the array. **Thread Dimensions** are directly related to the global_work_size variable in the OpenCL function clEnqueueNDRangeKernel and can be thought of as the number of threads that will spawn.

In the example so far, each element in the array of 1 (Depth) x 1000 (Height) x 1000 (Width) is having the same operation performed on it regardless of the location of the element being operated on. The **Thread Dimensions** in this case would simply be the dimension of the array, 1 x 1000 x 1000.

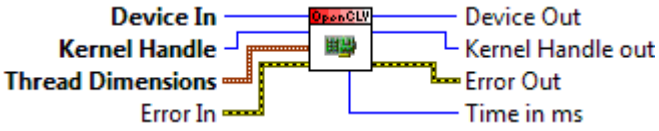

Illustration 16: Execute Kernel Simple.vi

If the standard deviation needed to be computed for each of the 1000 rows, the Thread Dimensions would be 1 x 1 x 1000, so 1000 threads would be spawned, each thread computing the standard deviation for 1 row.

Another example using a 1 x 1000 x 1000 array, except the device can handle a vector of four (float4). The custom kernel would look like Text 2. Because each thread is reading four floats at a time, **Thread Dimensions** in this case would be changed to 1 x 1000 x 250.

```
__kernel void LogSum_32f_v4(__global float* Memory1,
                            __global float *Memory2){

int k = get_global_id(0); //depth
int j = get_global_id(1);  //height
int i = get_global_id(2);  //width

int linear_coord = i + get_global_size(2)*j + get_global_size(1)*get_global_size(2)*k;

float4 Memory1Tmp = vload4(linear_coord, Memory1);
float4 Memory2Tmp = vload4(linear_coord, Memory2);

Memory1Tmp = log10(Memory1Tmp) + log10(Memory2Tmp);

vstore(Memory1Tmp, linear_coord, Memory1);

}
```

Text 2: Example Kernel using float4

For more advanced users, **Execute Kernel Advanced.vi** gives access to the global_work_offset and local_work_size variables in clEnqueueNDRangeKernel. Please see the OpenCL specification for more information on using the additional features.

Illustration 17 shows the final OpenCLV implementation of the custom kernel. It should be packaged in a sub-vi and documented to make it easy to use for future programmers.

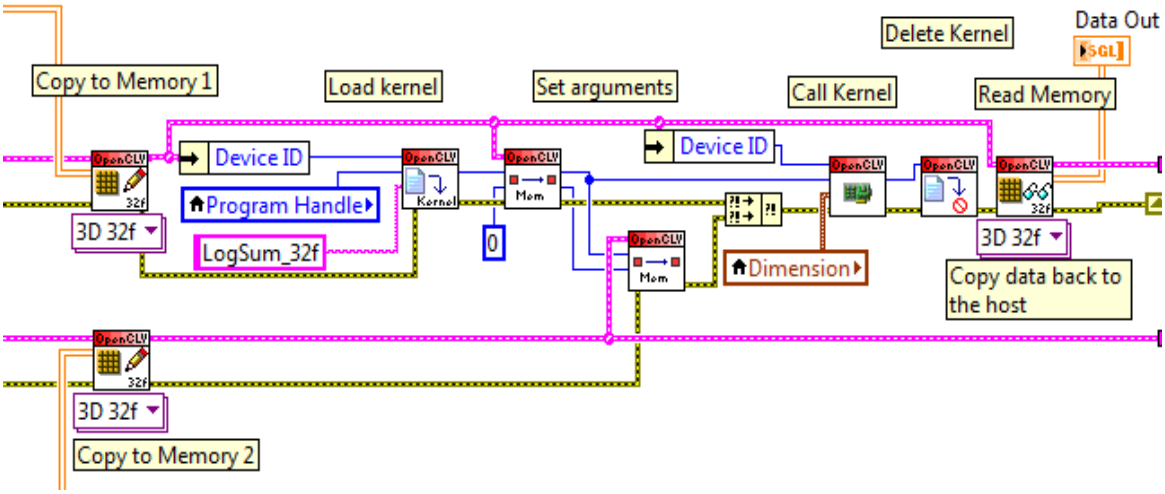

Illustration 17: Finished OpenCLV custom kernel

6.4 Delete Kernel.vi (Illustration 16)

After a kernel has been called and run, it should be deleted until the next time it is needed. Delete Kernel.vi provides this functionality.

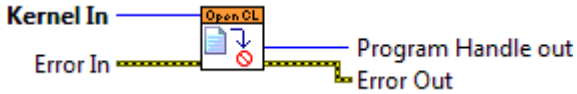

Illustration 18: Delete Kernel.vi

6.5 Results of Custom vs. Convenience

After running the custom kernel, it takes ~33ms on an NVIDIA GTX 560 Ti, while the convenience functions take ~50ms to perform the same operations. This should help demonstrate now the OpenCLV convenience VIs can be useful for quick programs, but for optimal efficiency, as much as possible should be put inside custom kernels and loaded using the OpenCLV suite of tools.

7 Phase 3 – Decimate

After all the data has been computed and read, it is time to clean-up OpenCLV and OpenCL.

7.1 Delete Memory.vi (Illustration 19)

Each memory buffer that was allocated in Phase 1 should be destroyed at this point. Regardless of the type of memory allocated, **Delete Memory.vi** will request that the device delete the memory onject that is passed into it.

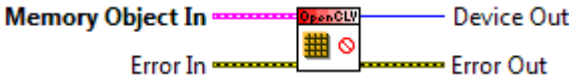

Illustration 19: Delete Memory.vi

7.3 Error Checking.vi (Illustration 20)

This VI explains the miscellaneous errors that may occur during use. Where possible, OpenCL errors have been used and will be displayed. Otherwise, custom errors message will be displayed that will try and help resolve different issues. This VI can be used anywhere to aid in troubleshooting.

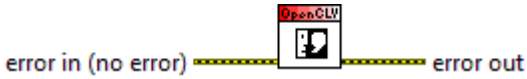

Illustration 20: Error Checking.vi

8 Additional Features

Below are a list of additional functionality that OpenCLV has implemented to speed up development.

8.1 AMD OpenCL FFT and Inverse FFT

OpenCLV has included the AMD FFT (and Inverse FFT) algorithm. This algorithm can handle 1D, 2D, and 3D FFTs with either 32f or 64f precision. OpenCLV handles the creation and deletion of additional memory arrays when needed. To simplify its use, the OpenCLV implementation has wrapped the AMD FFT into a polymorphic VI with 3 options:

- Create FFT (IFFT) Plan
- Destroy FFT (IFFT) Plan
- Compute FFT (IFFT)

The FFT algorithm can handle Real to Hermitian Complex or Hermitian Complex to Real, both are Out of Place operations.

See **2D FFT Example.vi** for more details on use.

8.1.2 Allocate (Illustration 21)

The AMD FFT algorithm has buffers and tables created to greatly speed up the FFT. **This only needs to be done once per plan**. Simply set the polymorphic menu to **Create FFT Plan.**

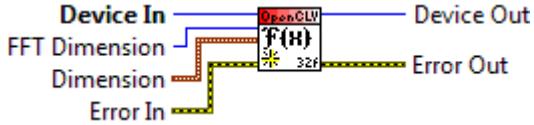

Illustration 21: Create FFT Plan (32f)

The FFT Size is the Width of the input array dimensions. For an array of 1 x 1000 x 1024 (Depth x Height x Width) with the FFT Dimensions set to 1D, 1000 FFTs of size 1024 would be performed when the **Compute FFT.vi** is called. Two Memory Objects would be created automatically when the plan is allocated to store the Real and Complex buffers, in this case two 1 x 1000 x 513 Memory Objects, one for the Real and one for the Complex.

8.1.3 Destroy (Illustration 22)

Any plan created must be destroyed. The memory allocated during the **Create FFT Plan** phase will be de-allocated here and does not need to be done manually. Simply select the polymorphic menu to **Destroy FFT Plan.**

Illustration 22: Destroy FFT Plan

8.1.4 Compute FFT (Illustration 23)

After the plan has been created, the **Compute FFT** polymorphic can be used to invoke the FFT. The Memory Object dimensions in should be identical in size to the dimensions used to create the plan or the results will be incorrect.

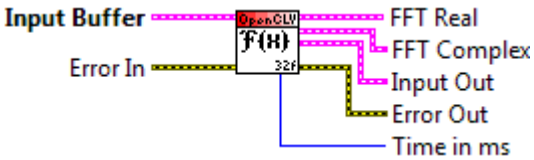

Illustration 23: Compute FFT

7 Revision History

Revision	Date	Changes	Department Head	Quality Manager
1	02/14/13	Initial Release	Austin McElroy	