

OpenCLV Manual

RaptorView, LLC

Table of Contents

| | |
|---|----|
| 1 Introduction..... | 3 |
| 2 Installation and Requirements..... | 3 |
| 2.1 Requirements..... | 3 |
| 2.2 Installation..... | 3 |
| 3 Definitions..... | 3 |
| 4 OpenCLV Structure..... | 4 |
| 4.1 Icon Description..... | 4 |
| 4.1.1 OpenCLV Header..... | 4 |
| 4.1.2 OpenCLV Function..... | 4 |
| 4.1.3 OpenCLV Function Precision..... | 4 |
| 4.1.4 OpenCLV Function Icon..... | 4 |
| 5 OpenCLV Array/Memory Dimension Orientation and Objects..... | 5 |
| 5.1 Array Orientation..... | 5 |
| 5.2 OpenCLV Controls..... | 5 |
| 5.3 OpenCLV Object..... | 5 |
| 5.4 OpenCLV and Out-of-Bounds Memory Access..... | 6 |
| 6 Getting Started..... | 6 |
| 6.1 Best Practices..... | 6 |
| 6.1.1 Functional Global Variables..... | 6 |
| 6.1.2 Error Handling..... | 6 |
| 6.2 Phase 1 - Allocate..... | 8 |
| 6.2.1 Starting OpenCLV.vi (Illustration 4)..... | 8 |
| 6.2.2 Create Device GUI.vi (Illustration 5)..... | 8 |
| 6.2.3 Load Program.vi (Illustration 7)..... | 9 |
| 6.2.4 Allocate Memory.vi (Illustration 9)..... | 10 |
| 6.2.5 Allocate Image.vi (Illustration 10)..... | 10 |
| 6.3 Phase 2 - Operate..... | 11 |
| 6.3.1 Write Memory.vi (Illustration 11)..... | 11 |
| 6.3.2 Read Memory.vi (Illustration 12)..... | 11 |
| 6.3.3 Copy Memory (Illustration 13: Copy Memory.vi)..... | 12 |
| 6.4 Phase 3 – Delete..... | 12 |
| 6.4.1 Delete Memory (Illustration 14: Delete Memory Objects)..... | 12 |
| 6.4.2 Delete Images (Illustration 15: Delete Image Objects)..... | 12 |
| 6.4.3 Stop OpenCLV.vi (Illustration 16)..... | 13 |
| 7 Custom Kernels..... | 13 |
| 7.1 Memory | 14 |
| 7.2 Work Groups..... | 14 |
| 7.3 Load Kernel.vi (Illustration 19)..... | 15 |
| 7.4 Pre-FFT Kernel Walkthrough..... | 15 |
| 7.5 Set Kernel Arguments (Illustration 21)..... | 17 |
| 7.5.1 Private Memory..... | 18 |
| 7.5.2 Local Memory..... | 18 |
| 7.6 Execute Kernel Simple.vi (Illustration 24)..... | 21 |
| 7.7 Execute Kernel Advanced.vi (Illustration 25)..... | 21 |
| 8 Cleanup OpenCLV..... | 21 |
| 8.1 Error Checking.vi (Illustration 26)..... | 21 |
| 9 Additional Features..... | 22 |
| 9.1 AMD OpenCL FFT and Inverse FFT..... | 22 |
| 9.1.1 Supported Sizes..... | 22 |
| 9.1.2 Allocate (Illustration 27)..... | 22 |

OpenCLV Manual

RaptorView, LLC

9.1.3 Destroy (Illustration 28)..... 23
9.1.4 Compute FFT (Illustration 29)..... 23
10 Troubleshooting..... 23
11 Revision History..... 24

OpenCLV Manual

RaptorView, LLC

Disclaimer: LabVIEW™ is a trademark of National Instruments. Neither RaptorView, nor any software programs or other goods or services offered by RaptorView, are affiliated with, endorsed by, or sponsored by National Instruments.

1 Introduction

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

- Quickly determine OpenCL platforms and devices
- Load any OpenCL program or kernel to a device
- Easily allocate, write, and read from device memory
- Perform FFTs and Inverse FFTs using AMD FFT OpenCL Library

2 Installation and Requirements

2.1 Requirements

- Windows 7 64-bit or later
- LabVIEW™ 2012 64-bit with VI Package Manager installed
- At least one OpenCL 1.1 compatible device or higher
- At least one OpenCL platform that is compatible with the OpenCL device
- For working with the OpenCL image class, OpenCLV requires an OpenCL 1.2 compatible device or higher

2.2 Installation

OpenCLV is only available for installation through the VI Package Manager provided in LabVIEW™ 2012 and beyond. Simply double click on the .vip package provided to start the installation process. Example code can be found by going to **Labview Menu Bar->Help->OpenCLV Examples** or through the example finder. Please read the known issues at **Labview Menu Bar->Help->OpenCLV Known Issues** before contacting technical support.

3 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
- Complex Functions – OpenCLV Polymorphic VIs that consist of a Create, Compute, and Delete.

OpenCLV Manual

RaptorView, LLC

4 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. Let's take a look at a typical OpenCLV VI icon:

Example of a Complex Function:



Illustration 1: Typical Complex Function VI Icon

4.1 Icon Description

4.1.1 OpenCLV Header

Signifies that the VI is associated with the OpenCLV library.

4.1.2 OpenCLV Function




Signifies visually or via text which OpenCL function is implemented.

4.1.3 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.

4.1.4 OpenCLV Function Icon

Shows the currently selected function of the polymorphic VI. Common icons seen in **Complex Functions** are:

-  Compute – Performs the computation portion of the **Complex Function**
-  Create – Creates the Program, Kernel, and Memory Objects needed by the **Complex Function**
-  Delete – Deletes the Program, Kernel, and Memory Objects associated with the **Complex Function**

5 OpenCLV Array/Memory Dimension Orientation and Objects

5.1 Array Orientation

OpenCLV arrays are setup in a 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:

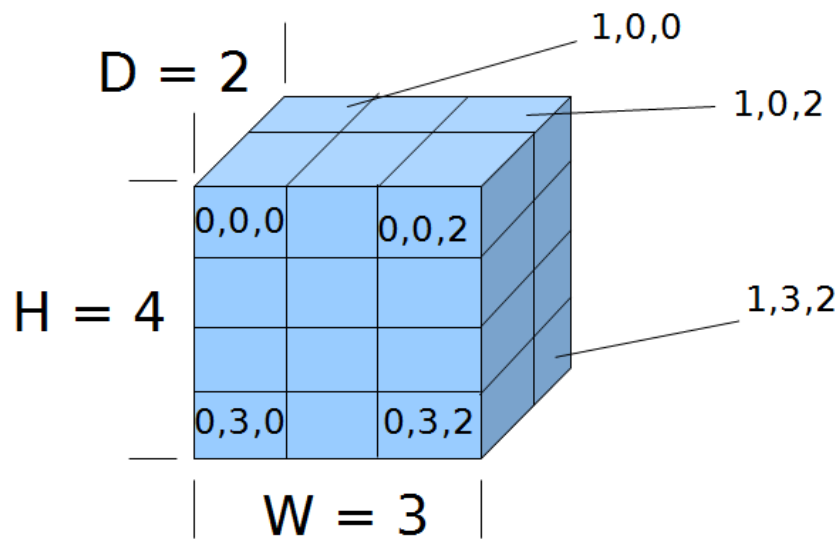


Illustration 2: OpenCLV Array Definition

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

5.2 OpenCLV Controls

OpenCLV provides a handful of front panel controls. They are:

- Array Dimensions – Set the dimensions of array sizes. This control doesn't allow values of 0
- Kernel Object – Set the size of a kernel needed for some of the filters. This control only allows odd values to be selected
- Offset – Used for indicating an offset when reading, writing, and copying data to 3D arrays
- Copy Parameters – A control used primarily for copying one array to another

5.3 OpenCLV Object

OpenCLV relies heavily on Objects that are custom controls. For example, 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 objects when needed.

5.4 OpenCLV and Out-of-Bounds Memory Access

Most LabVIEW™ programmers never worry about reading or writing outside the bounds of a LabVIEW™ allocated memory array. A value is always returned and better yet, no errors occur. Since OpenCLV memory is allocated and deleted based on vendor libraries, behaviour accessing memory outside the bounds has various behaviours.

Usually on video cards, nothing happens. On Intel processors, LabVIEW™ will most likely crash. So make sure that you save often. If a kernel is crashing, check section 10 Troubleshooting for advice on how to avoid common Out-of-Bounds conditions.

6 Getting Started

OpenCLV can be broken into 3 phases

- Allocate – Allocate platforms, devices, complex functions, and memory
- Operate – Write, Read, and Execute OpenCL commands on memory buffers
- Delete – Delete all memory, programs, kernels, and complex functions

6.1 Best Practices

Before diving into the how to use OpenCLV, let's take a moment to discuss some best practices for using OpenCLV in a clean and consistent manner.

6.1.1 Functional Global Variables

OpenCLV works best when used in conjunction with Functional Global Variables. A template, **OpenCLV Functional Global Template.vit** has been provided to help speed up getting OpenCLV into a functional global format.

OpenCLV Functional Global Template.vi provides three default states:

- Allocate – Place all memory and complex function initialization here
- Operate – Place all memory read/write, kernel, and complex function calls here
- Delete – Delete all memory and complex function calls here

Additionally, the **OpenCLV Functional Global Template.vi** provides a shift register that can be used to bundle all memory and complex function references into a nice package. Make sure to write in what the cluster contains so that it is easier to remember what goes where.

Don't feel limited to these states, they are just a guideline for best use. Please see the example code for use of functional global variables and OpenCLV.

6.1.2 Error Handling

OpenCLV provides a few tools to help troubleshoot and diagnose errors that can occur, including a final Error Log that shows when, where, and what occurred. All OpenCLV errors that occur within OpenCLV provided

OpenCLV Manual

RaptorView, LLC

VIs will automatically add to the error log. The **Error Handler.vi** is a polymorphic VI that provides the following states:

- Add Error – Add an error to the error log. Best used to log non-OpenCLV errors.
- Quiet – By default, any time an error occurs a pop-up will display when, where, and what error occurred. Calling the Quiet state will suppress these pop-ups until re-enabled.
- Verbose – Enables the pop-up error information that displays when, where, and what error occurred. Verbose mode is enabled by default.
- View Log – Shows a new window with all the errors that occurred while the program was running (Illustration 3: Error log from the Error Test.vi).

See **Error Test.vi** in the OpenCLDemo and Documents folder for more information on using the **Error Handler.vi**.

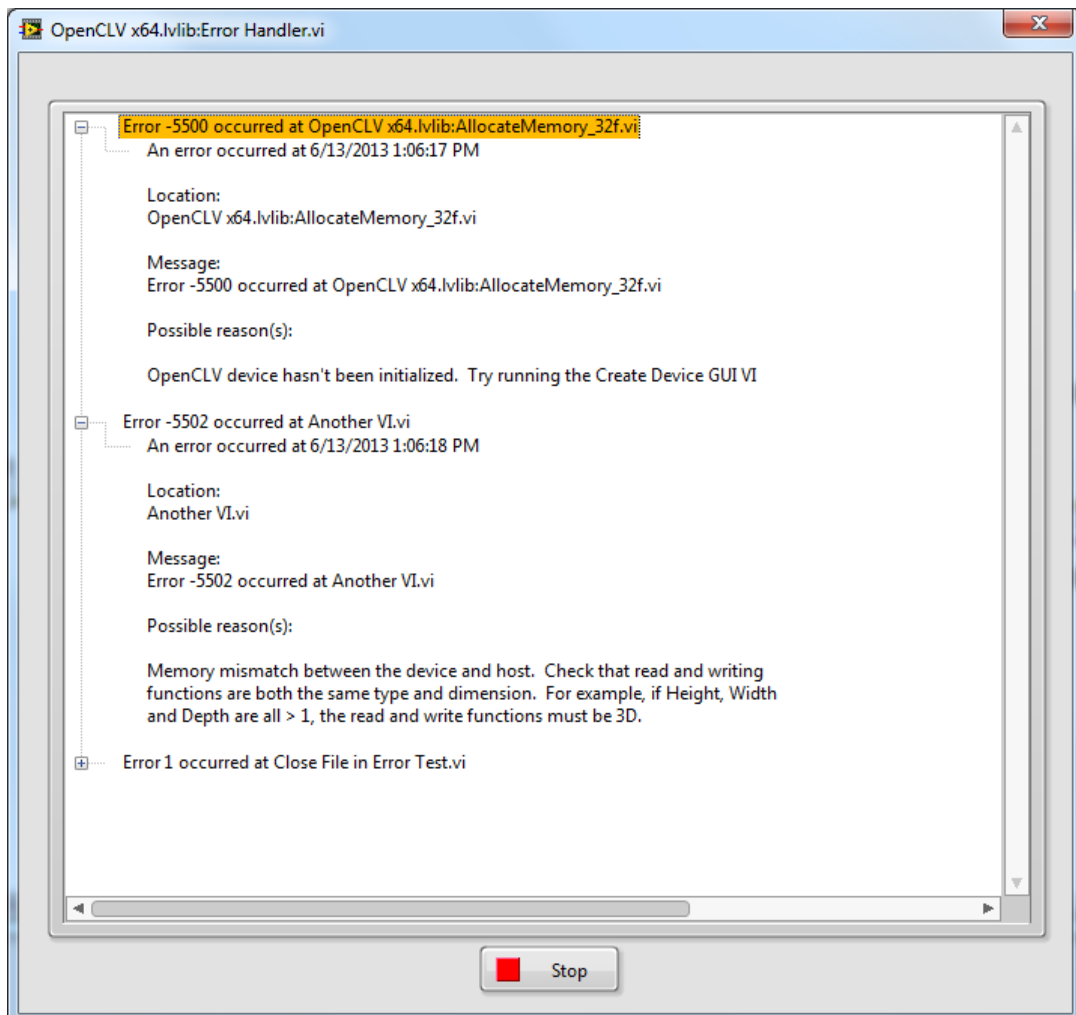


Illustration 3: Error log from the Error Test.vi

OpenCLV Manual

RaptorView, LLC

6.2 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.

6.2.1 Starting OpenCLV.vi (Illustration 4)

OpenCLV makes use of functional globals to track devices and errors. **Start OpenCLV.vi** is vital to ensure that the functional globals are purged, especially if the LabVIEW™ Abort Execution button has been used to prevent OpenCLV from closing properly. This VI should be one of the first VIs called and **should NOT be placed in a functional global**.



Illustration 4: Start OpenCLV.vi

6.2.2 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

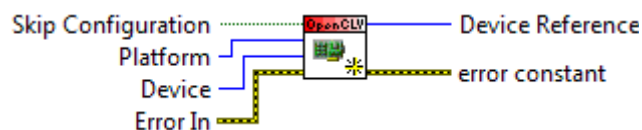


Illustration 5: Configure Platform and Device.vi

once the user is familiar with the platforms and devices available, though these will change from host to host. Illustration 5 shows the block diagram 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.

OpenCLV Manual

RaptorView, LLC

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**.

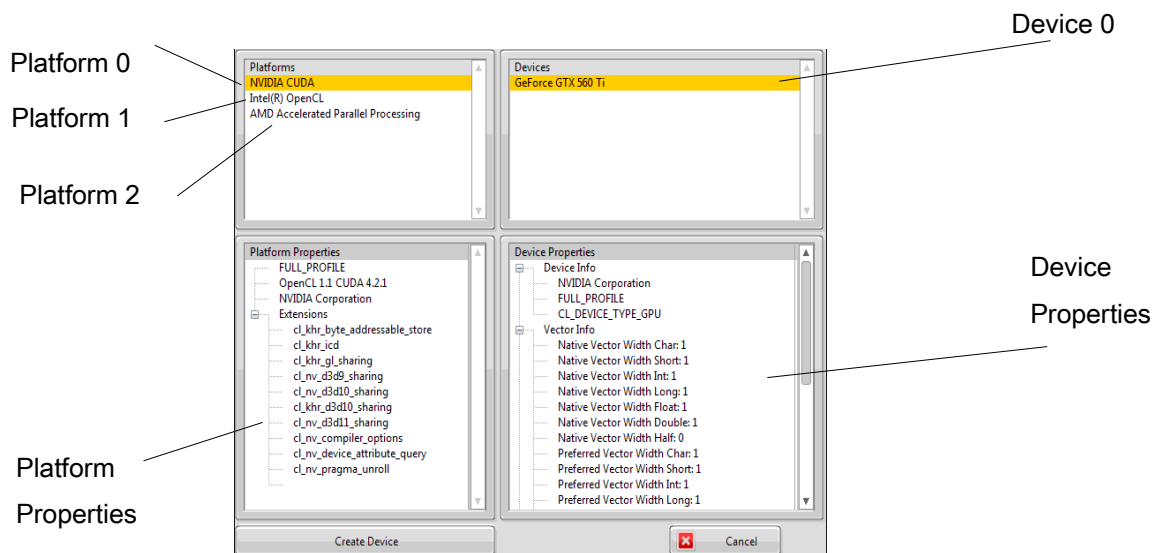


Illustration 6: Front panel for the Create Device GUI.vi

6.2.3 Load Program.vi (Illustration 7)

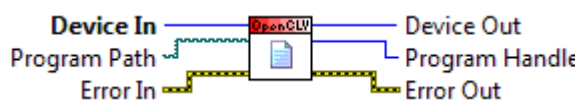
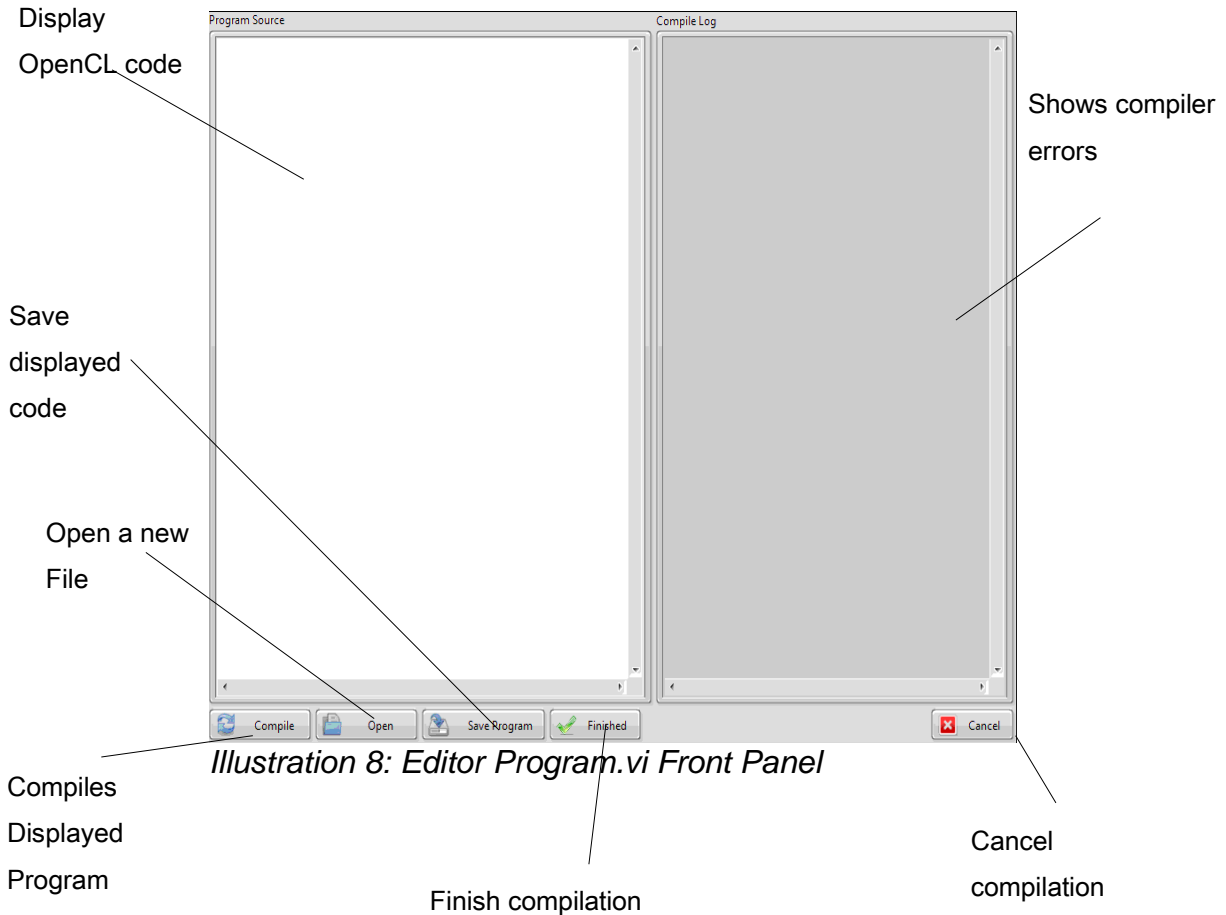


Illustration 7: Load Program.vi

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.vi** in the **Demo and Example Code** folder.

OpenCLV Manual

RaptorView, LLC



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.

6.2.4 Allocate Memory.vi (Illustration 9)

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 memory on the device.

OpenCLV Manual

RaptorView, LLC

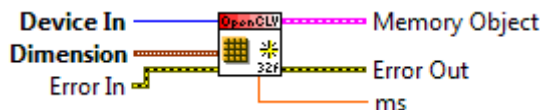


Illustration 9: Allocate Memory.vi for 32f

6.2.5 Allocate Image.vi (Illustration 10)

OpenCLV provides a VI for allocating images. In many cases, images might be preferable to standard memory. Images are:

- Cached in texture memory, which can increase read and write performance
- Supply cases to handle accessing memory outside of the image

Note: Images can't be used with OpenCLV **complex functions** like the Fast Fourier Transform and Anisotropic Filter.

Allocating images is similar to allocating memory. A device, dimensions, and memory type are all required. In addition, the channel order is required to let OpenCLV know how the data is stored. Examples are RGB, ARGB, Luminance (Grayscale), etc.



Illustration 10: Allocate Image

Images are supported in OpenCLV kernels using standard OpenCL calls. Images examples aren't provided with OpenCLV at this point in time.

6.3 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.

6.3.1 Write Memory.vi (Illustration 11)

OpenCLV provides a polymorphic VI, **Write Memory.vi** that writes LabVIEW™ data to the device. **Write Memory.vi** takes input 3D arrays from LabVIEW™ and copies the data from the host to the device. The memory written to the device should be the same size that was allocated.

The offset input can be used to offset the input data with respect to the Memory Object. For example, if the Memory Object is {1000, 1000, 5} and the input data is {1000, 1000, 1} and we wish to replace the 3rd depth memory block of 1000 x 1000 elements, the offset would be set to {0, 0, 2}.

See **Reading and Writing Data with an Offset.vi** example for more information.

OpenCLV Manual

RaptorView, LLC

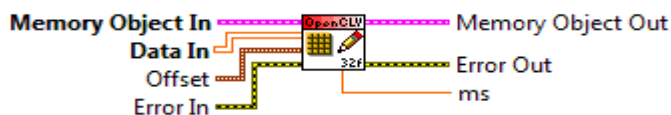


Illustration 11: Write Memory.vi adapted to a 32f input.

Note: It is not acceptable to write more data to the device than allocated. Unknown behaviour and crashes may occur.

6.3.2 Read Memory.vi (Illustration 12)

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.

The offset input can be used to offset the input data with respect to the Memory Object. For example, if the Memory Object is {1000, 1000, 5} and the the 3rd depth memory block of 1000 x 1000 elements needs to be read, the Dimensions would be {1000, 1000, 1} the offset would be set to {0, 0, 2}.

See **Reading and Writing Data with an Offset.vi** example for more information.

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.

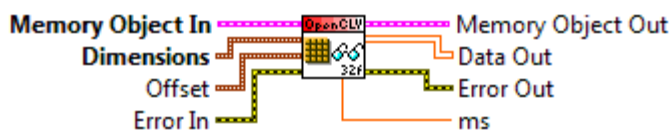


Illustration 12: Read Memory.vi block diagram icon adapted to read 32f data

6.3.3 Copy Memory (Illustration 13: Copy Memory.vi)

Copies one the Source memory object to the Destination memory object. If no Copy Parameters are provided, the entire Source will be copied to the Destination. Use the Copy parameters to copy only subsets of memory objects. See **Copy Buffer Example.vi** for more information.

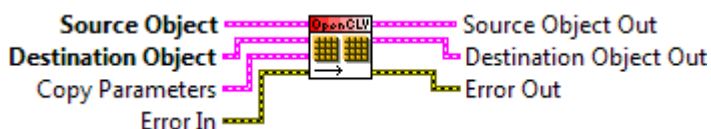


Illustration 13: Copy Memory.vi

6.4 Phase 3 – Delete

OpenCLV Manual

RaptorView, LLC

When **OpenCLV Stop.vi** is called, all allocated memory, programs, and kernels should be deleted. **It is best practice to at least delete memory and complex functions need to be manually deleted.**

6.4.1 Delete Memory (Illustration 14: Delete Memory Objects)

Delete Memory.vi is used to delete Memory Objects.

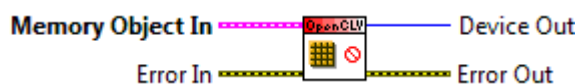


Illustration 14: Delete Memory Objects

6.4.2 Delete Images (Illustration 15: Delete Image Objects)

Delete Image.vi is used to delete Image Objects.



Illustration 15: Delete Image Objects

6.4.3 Stop OpenCLV.vi (Illustration 16)

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. This should be one of the last Vis called and **should NOT be placed in the functional global.**



Illustration 16: Stop OpenCLV.vi

7 Custom Kernels

OpenCL kernels are where the magic happens. OpenCLV has tried to faithfully implement as much of the OpenCL standard as possible. This manual shouldn't be a substitute for reading through the OpenCL documentation and excellent references, such as "*OpenCL in Action*" by Matthew Scarpino.

The example to demonstrate OpenCLV is from an imaging modality called Optical Coherence Tomography (OCT) and is a real-world application. OCT is akin to ultrasound imaging in that a 3D volume of data is created that has a height, width, and depth. Most of the computations on each voxel is independent of the others, so it is a perfect example for OpenCLV.

The data is acquired using a Analog to Digital converter that usually outputs 12 to 16 bits of unsigned data. This data needs to be converted to Voltage with a bias and a gain (based on the card), windowed (usually a

OpenCLV Manual

RaptorView, LLC

Hanning Window), Fourier Transformed, and then the Magnitude and Phase computed from the complex results of the FFT.

In the following examples, there are two kernels, a pre-FFT kernel and a post FFT kernel. The FFT is discussed in 9.1 AMD OpenCL FFT and Inverse FFT. The data will be generated at runtime using the LabVIEW™ random numbers.

7.1 Memory

OpenCL devices have multiple types of memory, some faster than others. OpenCLV has implemented the following types:

- Global Memory – This memory is where large data is allocated and stored and is the slowest memory.
- Local Memory – This is memory that work groups can access and can be as fast as private memory.
- Private Memory (kernel only) – Each thread has a small amount of memory that only it can access and is the fastest memory available.
- Constant Memory – Stores constants, similar to Global memory, but read only.
- Image Memory – Memory that may be cached to improve read/write performance.

OpenCL kernels must minimize reads and writes to Global Memory as much as possible. This can be done by using Local Memory and work groups covered 7.5 Set Kernel Arguments (Illustration 21) and 7.7 Execute Kernel Advanced.vi (Illustration 25).

7.2 Work Groups

OpenCL allows threads to work together and share information. Threads can be bundled into Work Groups which can use Local Memory (7.5.2 Local Memory) at speeds that are similar to Private Memory.

Work Groups sizes can vary from device to device, but they can be composed of 1D, 2D, or 3D groups of threads. Work Groups must also be a multiple of some minimum value, set by the device. To get this information, OpenCLV provides **Kernel Workgroup Information.vi** (Illustration 17) to gather - a Device ID and Kernel are required.

Work Groups require (see Illustration 18: Kernel Information Cluster):

- Dimensions must be a multiple of Work Group Multiple (1 is the default)
- All 3 dimensions must multiply and be less than Max Work Group Size
- Dimension cannot be larger than the Thread Dimensions

For example, the ATI 7970 has a Max Work Group size of 256 threads, and a Work Group Multiple of 64 threads. Therefore, the following Work Groups would be valid for Thread Dimensions of {32, 32, 1024}:

- {2, 2, 2}
- {4, 4, 4}
- {1, 1, 256}

While the following would be invalid:

- {7, 2, 2} – 7 not a multiple of 64
- {64, 1, 1} – Work Group depth dimension is larger than the Thread depth dimension size of 32
- {16, 16, 16} – These multiply to be greater than 256, the Max Work Group Size.

OpenCLV Manual

RaptorView, LLC

This information can be used with 7.7 Execute Kernel Advanced.vi (Illustration 25) to create more advanced and optimized kernels.

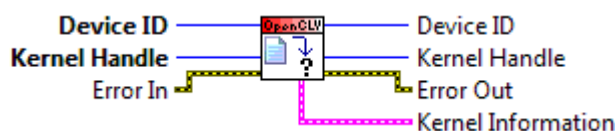


Illustration 17: Kernel Workgroup Information.vi

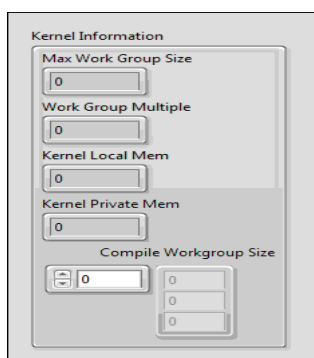


Illustration 18: Kernel Information Cluster

7.3 Load Kernel.vi (Illustration 19)

Once a program has been successfully loaded and compiled 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**).

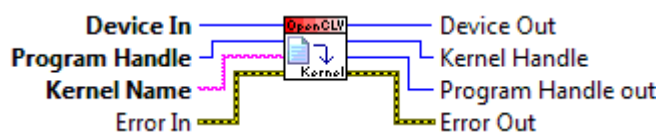


Illustration 19: Load Kernel.vi

7.4 Pre-FFT Kernel Walkthrough

In the example provided in Text 1, the **Kernel Name** would simply be PreFFT_32f. Illustration 20 shows how this would look in LabVIEW™ **OCT Example Program** and **clOCT.cl** kernel provided in the **Demo and Example Code** folder.

In short, this kernel first converts the unsigned short (U16) data to a Voltage, then computes a Hanning window based on the width (float `t_WindowSize = get_global_size(2)`) and position of the thread. Text 2 shows an example where the Hanning Window is stored in Global Memory .

OpenCLV Manual

RaptorView, LLC

```
__kernel void PreFFT_32f(__global unsigned short *Raw,
                        float Gain,
                        float Bias,
                        __global float *Windowed){

    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;

    float in = ((float)Raw[linear_coord])*Gain - Bias;
    float t_WindowSize = get_global_size(2); //Set the window size
    float t = (float)i;

    t = 2*M_PI_F*t / (t_WindowSize - 1); //Compute Hanning Window
    Windowed[linear_coord] = in*(.5 - .5*cos(t)); //Finish Hanning Window and output
}
```

Text 1: OCT PreFFT kernel where Hanning window is computed on the fly

```
__kernel void PreFFT_Global_32f(__global unsigned short *Raw,
                                float Gain,
                                float Bias,
                                __global float *HanningWindow,
                                __global float *Windowed){

    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;

    float in = ((float)Raw[linear_coord])*Gain - Bias;

    Windowed[linear_coord] = in*HanningWindow[j]; //Finish Hanning Window and output
}
```

Text 2: OCT PreFFT Kernel using a Look-up Table in Global Memory

If you aren't used to programming OpenCL, this code may seem a little intimidating. Some observations that may help:

- Since OpenCLV forces the use of 3D matrices, a kernel will always have 3 thread dimensions based on the Depth, Width, and Height. The command `get_global_id()` lets the kernel know which thread is being called.

OpenCLV Manual

RaptorView, LLC

- The data can be thought of as stored in memory as a 1D array. The 3 thread dimensions can be converted into a 1D array using `get_global_size()` which tells the kernel the maximum thread size in a given dimension. Therefore, `linear_coord` can be thought of as:

$$\text{linear_coord} = i + \text{width} * j + \text{width} * \text{height} * k$$

- The Raw and Window pointers are preceded by `__global` to indicate they reside in Global Memory, regardless of the type.
- Note that Gain and Bias are scalar values. These are examples of Private Memory, a small but fast area of memory that used be used for constants that each thread must use.

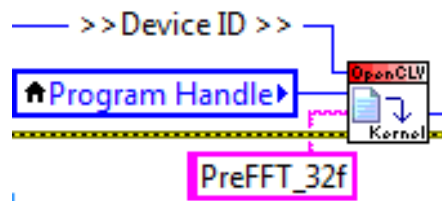


Illustration 20: Loading a Kernel in LabVIEW™

7.5 Set Kernel Arguments (Illustration 21)

After a kernel has been loaded, OpenCL requires that the arguments be set. In our current example, argument 0 is the unsigned short Raw data, argument 1 is the Gain of the ADC, argument 2 is the Bias of the ADC, and argument 4 is the windowed and adjusted data.

There are currently 12 different Set Kernel Arguments functions, one for: I8, U8, I16, U16, I32, U32, I64, U64, 32f, 64f, one for Memory Objects, and one for Image Objects. Of these 12 Set Kernel Argument functions, the first 10 are used for allocating memory within or sending a constant to a kernel. The last 2 are used for accessing Global Memory.

Even though the `*Raw` and `*Windowed` are pointers, they are preceded by `__global` and thus are pointing to Global Memory on the device, and thus **MUST** be referenced using **Set Kernel Arguments Memory Object.vi (Illustration 21)**. The **Set Kernel Argument Image Object.vi** is similar, except requires an Image Object instead of a Memory Object.

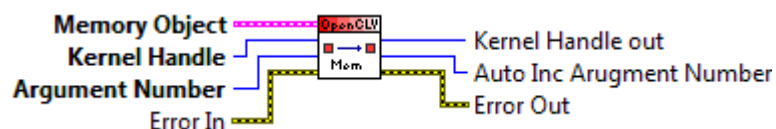


Illustration 21: Set Kernel Argument for Memory Objects

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 for a new

OpenCLV Manual

RaptorView, LLC

kernel, 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 (and hopefully written to) memory object.

In the ongoing example, there would need to be two **Set Kernel Arguments Memory Object.vi**. The Gain and Bias will be set using the **Set Kernel Argument 32f.vi** (Illustration 22). The other data type functions look similar.

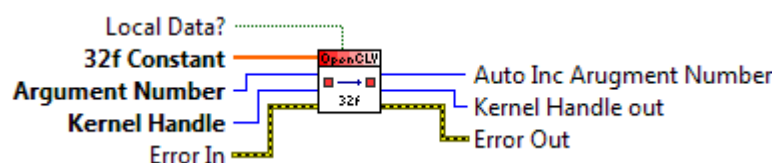


Illustration 22: Set Kernel Argument 32f

7.5.1 Private Memory

This VI is a bit different, in that it requires a 1D array and has the option to declare the memory as Local Data. If setting Kernel Argument to Private Memory, Local Data must be False, and the 32f Constant must be an array of 32f numbers, even if that array is only a single element.

In our case Gain and Bias are single float values, but it is possible to pass kernels float2, float3, float4, float8, and float16 vectors to Private memory.

For example, to pass a float4, declare a 1D array in LabVIEW™ of {1, 2, 3, 4}. Wire this to **Set Kernel Argument 32f.vi** and in the kernel call:

```
__kernel void PrivateVectorExample(float4 OneTwoThreeFour){  
    ...  
}
```

After the 4 arguments have been set, the block diagram looks like Illustration 23:

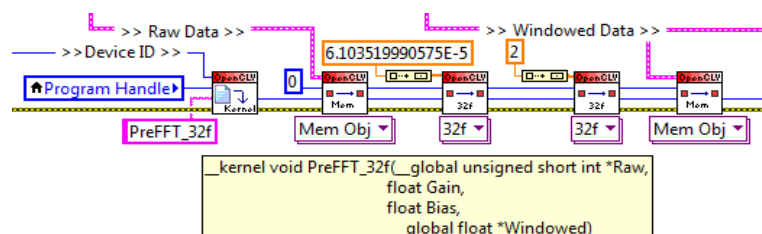


Illustration 23: Setting arguments to the PreFFT_32f Kernel with a note about the kernel for future generations

OpenCLV Manual

RaptorView, LLC

This should be packaged as a sub-VI to keep the block diagram clean and in agreement with the National Instruments LabVIEW™ Style Guide.

7.5.2 Local Memory

Setting Local Data to true lets the host allocate an array that is the size of the 32f Constant on the device that can be used by a Work Group. In the case of Local Data, **no information is passed to the device**, it is simply allocated by the host on the device for Work Groups to use.

For example, a kernel requires Work Groups to have a chunk of memory that is 1 x 1 x 256 32f numbers. In LabVIEW™, a 1D array of 32f would be initialized that is 256 32f elements, which would be wired to 32f Constant. Local Data would be set to true, and memory accessed in the kernel using the following syntax:

```
__kernel void LocalMemoryExample(__local *OneDArray){  
    ...  
}
```

Local Memory has two very big advantages over global memory:

- All threads in a Work Group can access local memory at private memory speeds
- All threads in a Work Group can read/write to global memory in batches

What is the best way to use it? It is very application dependant, but a common example is to use it to buffer global memory into local memory. Text 3 shows an example of how to take the existing OCT example and use local memory.

In the example, the pre-computed Hanning window is buffered to local memory. The device being used to create this example is an ATI 7970 with a Max Work Group size of 256 threads. Therefore, when calling this kernel the Work Group size would be {1, 1, 256} Depth, Height, Width respectively. This allows the kernel to batch read 256 values from Global Memory into local memory.

The command `read_mem_fence(CLK_LOCAL_MEM_FENCE)` lets OpenCL know that the intention is to complete this batch read before continuing to the rest of the kernel code. Once the batch read is complete, the Local Memory can be read and written to as quickly as Private Memory.

The final results of the Hanning Window multiplication are re-stored back into Local Memory, and then a batch write is requested using the command `write_mem_fence(CLK_GLOBAL_MEM_FENCE)`.

OpenCLV Manual

RaptorView, LLC

```
__kernel void PreFFT_Local_32f(__global unsigned short *Raw,
    float Gain,
    float Bias,
    __global float *HanningWindow,
    __local float *WindowMultiple,
    __global float *Windowed){

    //The local_id is usually a base 2 for example 256. Therefore,
    //we can't buffer the entire hanning window into local memory
    //Note: Since the since the size of the Hanning Window is
    //    tied to the global thread count, we never have to
    //        worry about accessing memory outside of the array
    WindowMultiple[get_local_id(2)] = HanningWindow[get_global_id(2)];

    //This forces all the memory reads in the workgroup to
    //execute in a batch instead of individually
    read_mem_fence(CLK_LOCAL_MEM_FENCE);

    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;

    float in = ((float)Raw[linear_coord])*Gain - Bias;

    WindowMultiple[get_local_id(2)] = in*WindowMultiple[get_local_id(2)]; //Finish Hanning Window and
    //output

    Windowed[linear_coord] = WindowMultiple[get_local_id(2)];

    write_mem_fence(CLK_GLOBAL_MEM_FENCE);
}
```

Text 3: OCT PreFFT kernel where the Hanning Window is buffered into Local Memory for batched reads and writes

The LabVIEW™ implementation can be seen in **OCT Example Local Memory.vi**. Running this VI in comparison to the Global Memory only **OCT Example Global Memory.vi** gives the following results on an AMD7970:

- {1, 5000, 4096} - Global Memory average time over 10 runs is ~368ms
- {1, 5000, 4096} – Local Memory average time over 10 runs is ~98ms

This is an increase of almost 3.5x with just a little effort!

OpenCLV Manual

RaptorView, LLC

7.6 Execute Kernel Simple.vi (Illustration 24)

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 default example so far, each element in the array of 1 (Depth) x 1000 (Height) x 1024 (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 1024.

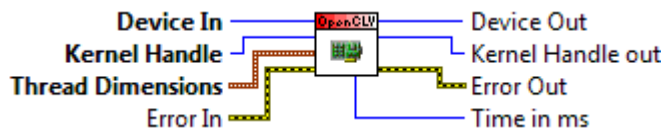


Illustration 24: Execute Kernel Simple.vi

7.7 Execute Kernel Advanced.vi (Illustration 25)

For more advanced users, **Execute Kernel Advanced.vi** gives the ability to create Work Groups, and if the OpenCL device allows for it, give an offset for where the threads start (not available on all devices).

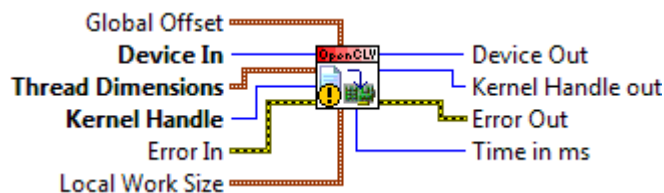


Illustration 25: Execute Kernel Advanced

Some cards are compatible with the Global Offset feature, so keep this in mind if odd behaviour occurs. Please see the OpenCL standard for more information about Global Offset and Work Groups.

8 Cleanup OpenCLV

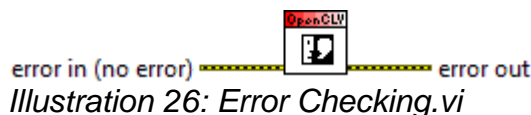
After all the data has been computed and read, it is time to clean-up OpenCLV and OpenCL. The final VI to be called should be **Stop OpenCLV.vi** (see **Illustration 16**). As each program, memory object, and kernel, and device are allocated, they are added to a functional global. When **Stop OpenCLV.vi** is called, these functional global variables are purged and the objects delete. This is not the case with **Complex Functions**.

8.1 Error Checking.vi (Illustration 26)

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.

OpenCLV Manual

RaptorView, LLC



9 Additional Features

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

9.1 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.

9.1.1 Supported Sizes

From the AMD FFT documentation:

*“clAmdFft supports powers of 2, 3 and 5 sizes. This means that the vector lengths that can be configured through a plan can be any length that is a power of two, three, and five; examples include 20, 21*31, 33*55, 22*33*55; up to the limit that the device can support.”*

9.1.2 Allocate (Illustration 27)

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.**

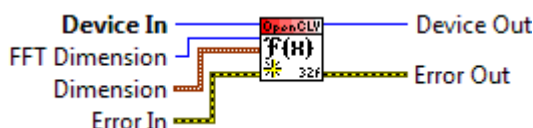


Illustration 27: Create FFT Plan (32f)

OpenCLV Manual

RaptorView, LLC

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.

9.1.3 Destroy (Illustration 28)

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 28: Destroy FFT Plan

9.1.4 Compute FFT (Illustration 29)

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.

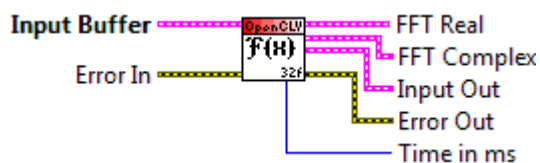


Illustration 29: Compute FFT

10 Troubleshooting

Unfortunately, OpenCLV doesn't have control over LabVIEW™ Runtime errors which means runtime errors will crash LabVIEW™. OpenCLV has tried really hard to minimize crashes, since it is frustrating.

Q: My custom kernel is crashing, what's up?

A: The most common reason the code crashes is a Memory Access Violation. In C++, this normally halts the debugger. Unfortunately, LabVIEW™ is very unforgiving in handling this type of error from external code and crashes.

The most common memory access violations are from reading memory outside of what was allocated and read and writing data at the same time.

For example, we have an array that is {1, 1000, 1000} and would like to perform a divergence calculation. Divergence requires $x + 1$, $x - 1$, $y + 1$, $y - 1$, $z - 1$, and $z + 1$ throughout the calculation, so boundary conditions must be taken into account. There are a few ways to do this:

- Check each thread id to make sure it is > 0 and $< \text{get_global_size}(\dots) - 1$

OpenCLV Manual

RaptorView, LLC

- Use `global_offset(...)` and set the number of threads to be $\{z - \text{global_offset}(0)*2, y - \text{global_offset}(1)*2, x - \text{global_offset}(2)*2\}$. Make sure to check the OpenCL device can use `global_offset`. Intel and AMD are good at supporting it, NVIDIA isn't
- Use the OpenCL Image memory object. The kernel can be set to handle pixels outside of memory in a variety of ways: clamping, mirror, etc.

Furthermore, in the above example, if we are reading and writing to the same set of data, then thread 1 may write to $x + 1$ while thread 2 is reading from $x + 1$. This condition can cause undefined behaviour, and in LabVIEW™ that is probably going to lead to a crash. The solution in this case is to have an input and an output buffer in the case where the algorithm requires surrounding data.

OpenCLV Manual

RaptorView, LLC

11 Revision History

| Revision | Date | Changes | Writer | Quality Manager |
|----------|----------|--|----------------|-----------------|
| 1 | 02/14/13 | Initial Release | Austin McElroy | |
| 2 | 06/14/13 | Updated after feedback from NI and a CLA | Austin McElroy | |