

Run calculations on your CPU or GPU using IgorCL

Peter Dedecker
peter.dedecker@hotmail.com

April 11, 2014

1 Introduction

IgorCL is an “external operation” for Igor Pro (www.wavemetrics.com) that allows you to perform calculations on your computer’s CPU or GPU using OpenCL. OpenCL is a software framework that aims at providing a uniform way to perform computation on heterogeneous devices. The authoritative source on OpenCL is the official website, <http://www.khronos.org/opencv/>, and particularly the OpenCL specification (<http://www.khronos.org/registry/cl/specs/opencv-1.2.pdf>). This XOP is intended for advanced programmers only.

Using IgorCL is very different from writing plain Igor code, and requires a very thorough understanding of programming. Ideally you should have experience with low-level languages such as C or C++. You will probably not get much out of IgorCL unless you are a very advanced Igor programmer with knowledge of C.

The main goal of OpenCL is to run code across a wide range of devices. In today’s computers this is usually the CPU, the main processor, and the GPU, the graphics processor or ‘video card’. Because your code should run on a variety of devices, most of which Igor doesn’t know anything about, OpenCL code must be written in OpenCL-C, a subset of the C programming language. OpenCL-C is very different from Igor programming and requires a much more detailed understanding of how computers work.

2 Target audience

The biggest draw of OpenCL is usually the possibility to run calculations on the GPU. I have found that my analysis needs do not lend themselves well to GPU processing. There is a reason why CPUs are known as ‘general-purpose’ processors. Hardware vendors have been very successful in painting general-purpose GPU programming as a computational revolution, but conveniently forget to mention that these speedups are only possible on a very select group of problems.

My main use for IgorCL is in running code on the CPU, not on the GPU. This often provides a speedup over native Igor code because OpenCL compiles directly to machine code, which executes much faster. However, this is only worthwhile if you can identify specific bottlenecks that cannot be rewritten in terms of built-in Igor operations.

OpenCL is hard. Don’t bother unless you feel competent of your programming abilities.

3 OpenCL basics

It is entirely outside the scope of this document to provide an introduction to OpenCL. There are plenty of resources on the web, and plenty of books out there. I read *CUDA by Example: An Introduction to General-Purpose GPU Programming* by Jason Sanders and Edward Kandrot, and *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition* by Benedict Gaster et al. While the first book is about CUDA, not OpenCL, the concepts are sufficiently similar that the insights can be applied directly to OpenCL. In what follows we will focus on the Igor-specific aspects of OpenCL.

A unit of executable OpenCL code is known as a ‘kernel’. These are functions, written in OpenCL-C, that are executed on a compute device. When execution starts the input data is passed into a kernel via its arguments. These arguments are either scalars or pointers to memory buffers. Kernels never return a value -

calculation results must be returned by having the device write to one or more memory buffers also passed in as an argument.

You must write these kernels in order to have OpenCL do something for you. They have nothing to do with the Igor programming language – you cannot use the same syntax nor can you make use of Igor functionality from your OpenCL code. Igor doesn't know anything about OpenCL. You need to look at the OpenCL reference to find out what you can do in OpenCL-C.

Since OpenCL is expected to run across a wide range of devices, kernels must be provided as plain-text source code. At runtime you select the device the calculation will run on, and OpenCL will automatically compile your kernels for that particular architecture. This 'just-in-time' compilation is a defining feature of OpenCL, and it is the reason why it can work across a wide range of devices. Since compiling can be an expensive operation, IgorCL allows you to reuse compiled kernels, though you cannot use compiled kernels across different devices, computers, or OpenCL driver versions. You should not reuse kernels across Igor runs.

4 Installing IgorCL

To use IgorCL, simply download the IgorCL.xop file appropriate to your platform (unzipping it if necessary) and install it in the "Igor Extensions" folder in your Igor Pro User Files. To find the location of your Igor Pro User Files, launch Igor and choose Help – "Show Igor Pro User Files". Then restart Igor.

Depending on your hardware and installed software, you may also need to install the OpenCL framework. Implementations of OpenCL are available from a number of different vendors, including the AMD, NVidia, and Intel OpenCL framework. The AMD runtime provides support for both AMD GPUs and x86 CPUs (from Intel and AMD), so it is a one-stop solution for computers running Intel/AMD hardware. NVidia drivers are specific to the GPU, and will not execute on the CPU. AMD and NVidia include the runtime with their device drivers. Multiple OpenCL frameworks ('platforms') can be installed on the same computer, and IgorCL will allow you to choose between them at runtime.

On Macintosh the OpenCL runtime is included in recent versions of OS X, so OpenCL should work out of the box.

On Windows I have experimented with the AMD and Intel OpenCL drivers. Overall I suggest the AMD drivers since it supports both AMD video cards and x86 (AMD and Intel) CPUs. I did have difficulties on some systems, which would report missing libraries when OpenCL was invoked. I resolved this by replacing OpenCL.dll (in the win32 folder) with the OpenCL.dll that is shipped with the AMD Stream SDK.

5 Features and limitations of IgorCL

IgorCL makes OpenCL callable from Igor. It provides a means for you to select a particular OpenCL kernel, platform and device, specify the input and output data, and start the computation.

To pass data between Igor and OpenCL, IgorCL connects Igor waves to OpenCL buffers. At runtime, you must specify one or more Igor waves, and IgorCL will convert these to OpenCL memory buffers of the same numeric type, size, and contents. Igor waves are the *sole* means of data exchange between Igor and OpenCL. All arguments to your kernel must be put into one or more waves, with one wave per argument. You must also create and pass in Igor waves that will receive the output of the calculation.

From OpenCL's point of view, the Igor waves consist of linear arrays. It doesn't know, nor does it care, that this memory is part of an Igor wave. It is also unaware of the dimensionality of the wave. All that it sees is a pointer to memory, with your promise that the array contains all the . You are responsible for passing in the correct number of waves, with the correct dimensions, and of the correct number type.

Here's an example: suppose that your kernel takes three arguments: `float*`, `float*`, and `double*`. Each of these can be input, output, or both input and output. You are responsible for passing three waves to IgorCL, the first two of type float (32-bit floating point; single precision), and the last one of type double (64-bit floating point; double precision). You are also responsible for ensuring that the waves are large enough so that OpenCL does not generate an out-of-bounds access. IgorCL will *not* help you here.

For scalar arguments (such as `int`) you still need to pass a wave containing one or more points and of the appropriate numeric type. You must also let IgorCL know that this is a scalar argument by passing an appropriate memory flag (discussed in section 6.2.1).

If your kernel executes on the CPU and undertakes an action that the operating system considers to be invalid, such as writing to or reading from a memory location that does not belong to the program, the operating system will probably crash Igor and you (or the users of your code) will lose all unsaved data. The results of performing an invalid operation on the GPU are less well-defined. Nothing may happen, Igor may crash, or the computer may lock up. Don't do it. You should test all of your code on the CPU before passing it on to the GPU.

Kernels can be specified in two ways: as a single Igor string containing all of the kernel code, or as a wave that was previously created using the `IgorCLCompile` operation. The second way exists so you can compile a kernel once, and call it many times, thus avoiding the work associated with repeated compilation.

The main limitation of IgorCL is that there is no way to define new data types for the Igor programming language. This means that it is impossible to add an entity representing OpenCL-allocated memory in the Igor environment, or any other associated functionality. For this reason IgorCL does not support resident OpenCL memory: when you start an IgorCL calculation, IgorCL will typically allocate OpenCL memory for each wave you pass in, and will free this memory before returning. The unfortunate consequence of this is that the performance of your OpenCL calculations may be reduced by the allocating and copying involved. It also means that more involved calculation strategies are out of reach.

6 Anatomy of IgorCL

IgorCL adds three operations to Igor:

`IgorCLInfo` allows you to query the hardware available on the computer and find out about its capabilities.

`IgorCL` executes OpenCL kernels.

`IgorCLCompile` compiles OpenCL-C code to a compiled binary that you can then pass to IgorCL.

6.1 IgorCLInfo

OpenCL distinguishes between *platforms* and *devices*. A platform is an implementation of OpenCL that knows how to communicate with one or more hardware devices. Platforms are typically provided by the hardware manufacturer. A device is simply a symbolic representation of a particular piece of hardware attached to your computer.

`IgorCLInfo` lets you query the OpenCL runtime for supported platforms and devices, and provides some information on the abilities of each of these devices. It does so by creating two or more text waves. `M_OpenCLPlatforms` lists the available platforms on the device. For each of these platforms, another text wave called `M_OpenCLDevicesX` will be created, where X is the index of the platform that manages these devices. In some cases multiple platforms will be available for the same device. For example, the Intel and AMD OpenCL implementations both support x86 processors. In this case you can freely choose which platform you select for that particular device, though the platforms may differ in supported capabilities and efficiency.

The wave dimension labels provide descriptions of each attribute that is listed in these waves. To see them, try executing `Edit M_OpenCLDevices0.lid` in Igor.

IgorCL identifies platforms and devices by numeric index. If there are two platforms, Igor will refer to the first platform as index 0, and the second platform as index 1. The order is given by the order in `M_OpenCLPlatforms` and the suffix in `M_OpenCLDevicesX`. Likewise devices are identified by a numeric index. These must be passed to IgorCL and IgorCLCompile using the `/PLTM` and `/DEV` flags.

6.2 IgorCL

The IgorCL operation consists of a series of flags and a variable number of wave arguments.

```
IgorCL [flags] wave0, [wave1, ...]
```

6.2.1 Flags

/PLTM=platform The index of the OpenCL platform that this calculation should be executed on. If this flag is omitted then the first platform (index 0) is automatically used.

/DEV=device The index of the device that this calculation should be executed on. If this flag is omitted then the first device (index 0) within the selected platform is automatically used. This flag can not be combined with the **/DTYP** flag.

/DTYP=deviceTypeStr Selects the first device of the specified type for execution. *deviceTypeStr* is one of “CPU”, “GPU”, or “ACCELERATOR”. This flag can not be combined with the **/DEV** flag.

/SRCT=sourceStr *sourceStr* must contain valid OpenCL-C code, specifying one or more kernels. Exactly one of the **/SRCT** or **/SRCB** flags must be specified.

/SRCB=sourceBinaryWave *sourceBinaryWave* must be an unsigned byte wave containing a compiled OpenCL program, previously obtained using the `IgorCLCompile` operation. Exactly one of the **/SRCT** or **/SRCB** flags must be specified.

/KERN=kernelNameStr The name of the kernel (as provided in the OpenCL-C code) that will be executed. This flag is required.

/GSZE={g1,g2,g3} The global work size associated with the calculation. Must be specified in three dimensions. This flag is required.

/WGRP={w1,w2,w3} The local workgroup size (or work size) associated with the calculation. Must be specified in three dimensions, though the second and third argument can be zero. If you omit this argument a default size will be selected by the OpenCL platform.

/MFLG=memoryFlagsWave This flag lets you pass in various flags that control how IgorCL will manage the OpenCL memory buffers. Flags must be specified in *memoryFlagsWave*, which must be numeric. The first point in *memoryFlagsWave* will be applied to the first wave argument, the second to the next, etc.

/Z=[quiet] When used as plain **/Z** or with non-zero *quiet*, prevents OpenCL errors from aborting procedure execution. Instead the error will be reflected in a nonzero value of `V_flag`. Igor errors will still cause an abort. Setting *quiet* to zero is the same as omitting this flag.

6.2.2 Arguments

You must provide one wave for each argument to the OpenCL kernel, and each wave must be of the appropriate numeric type. In addition, you must ensure that each wave contains sufficient points to store all of the input and/or output values. This is solely your responsibility. More information can be found in section 5.

6.2.3 Memory flags

If you specify a memory flags wave using the **/MFLG** flag, each point in *memoryFlagsWave* can be a bit-wise combination of the values listed in table 1. The first four of these are wrappers for OpenCL flags, and are explained in the OpenCL reference. However `IgorCLIsLocalMemory` is particularly useful because it causes OpenCL to use the contents of the Igor wave directly, without copying, if the device supports it. This can be a significant speedup if the OpenCL calculations are executed on a device that shares the host memory space. The last two flags are specific to IgorCL. `IgorCLIsLocalMemory` notifies IgorCL that the value of the associated wave is the size *in bytes* of a local memory buffer to be allocated on the device. `IgorCLIsScalarArgument` notifies IgorCL that the first point of the associated wave should be passed as a scalar argument to the kernel.

Symbolic name	bit	OpenCL flag
IgorCLReadWrite	0	CL_MEM_READ_WRITE
IgorCLWriteOnly	1	CL_MEM_WRITE_ONLY
IgorCLReadOnly	2	CL_MEM_READ_ONLY
IgorCLUseHostPointer	3	CL_MEM_USE_HOST_PTR
IgorCLIsLocalMemory	4	specific to IgorCL
IgorCLIsScalarArgument	5	specific to IgorCL

Table 1: Memory flags supported by the IgorCL operation.

6.3 IgorCLCompile

6.4 IgorCL

The IgorCL operation consists of a series of flags and a variable number of wave arguments.

```
IgorCLCompile [flags] string
```

6.4.1 Flags

/PLTM=platform The index of the OpenCL platform that this calculation should be executed on. If this flag is omitted then the first platform (index 0) is automatically used.

/DEV=device The index of the device that this calculation should be executed on. If this flag is omitted then the first device (index 0) within the selected platform is automatically used. This flag can not be combined with the **/DTYP** flag.

/DTYP=deviceTypeStr Selects the first device of the specified type for execution. *deviceTypeStr* is one of “CPU”, “GPU”, or “ACCELERATOR”. This flag can not be combined with the **/DEV** flag.

/DEST=destination Causes the wave containing the compiled binary to be created at the given destination.

/Z=[quiet] When used as plain **/Z** or with non-zero *quiet*, prevents OpenCL errors from aborting procedure execution. Instead the error will be reflected in a nonzero value of **V_flag**. Igor errors will still cause an abort. Setting *quiet* to zero is the same as omitting this flag.

6.4.2 Arguments

IgorCLCompile takes a single string argument containing the full OpenCL-C code for one or more kernels. This can be the string that you would otherwise pass to IgorCL using the **/SRCT** flag.

6.4.3 Output

By default the compiled binary will be created in the wave **W_CompiledBinary**, but this can be changed using the **/DEST** flag. The resulting wave can then be passed to IgorCL using the **/SRCB** flag.

7 Examples

7.1 Adding numbers

Let’s follow the canonical example and add a bunch of numbers together. Here is the kernel:

```
kernel void VectorAdd(global float* A, global float* B, global float* C) {
    int unitID = get_global_id(0);
    C[unitID] = A[unitID] + B[unitID];
}
```

And here is some Igor code to run it on an example dataset:

```

Function TestAdd()
    string source = "kernel void VectorAdd(global float* A, global float* B, global
        float* C) {\n"
    source += "int unitID = get_global_id(0);\n"
    source += "C[unitID] = A[unitID] + B[unitID];\n"
    source += "}\n"

    Make /S/N=(128)/O W_A = 1, W_B = 1, W_C = 0 // data
    IgorCL /SRCT=source /DTYP="CPU" /GSZE={128,1,1} /WGRP={64,1,1} /KERN="VectorAdd"
        W_A, W_B, W_C
End

```

Executing TestAdd() will run the calculation on the first platform on your system, and on the CPU. You will get an error if the CPU is not supported. I have also included some code that times the operation.

7.2 Adding many numbers

The previous example could only handle situations where the number of points is multiple of the work group size. What if we wanted to add numbers where this is not the case, or if there are a lot of numbers? Let's expand the code to handle this situation. Here's the new kernel:

```

kernel void VectorAdd2(global float* A, global float* B, global float* C, int nValues) {
    int unitID = get_global_id(0);
    int globalSize = get_global_size(0);
    for (int i = unitID; i < nValues; i+=globalSize) {
        C[i] = A[i] + B[i];
    }
}

```

And here's the Igor code:

```

Function /S Source2()
    string source = "kernel void VectorAdd2(global float* A, global float* B, global
        float* C, int nValues) {"
    source += "int unitID = get_global_id(0);\n"
    source += "int globalSize = get_global_size(0);\n"
    source += "for (int i = unitID; i < nValues; i+=globalSize) {\n"
    source += "C[i] = A[i] + B[i];\n"
    source += "}\n"
    source += "}\n"
    return source
End

```

```

Function TestAdd2()
    variable IgorCLIsScalarArgument = 2^5
    variable nValues = 1e6

    Make /S/N=(nValues)/O W_A = 1, W_B = 1, W_C = 0 // data
    Make /I/O/N=(1) W_nValues = nValues // number of points
    Make /I/U/O /N=(4) W_MemFlags = 0 // memory flags
    W_MemFlags[3] = IgorCLIsScalarArgument
    variable globalSize = ceil(nValues / 64) * 64
    IgorCL /SRCT=Source2() /DTYP="CPU" /MFLG=W_MemFlags /GSZE={globalSize,1,1}
        /KERN="VectorAdd2" W_A, W_B, W_C, W_nValues
End

```

Note that things are getting more complex simply because we added a scalar value. We could also have used a buffer with just a single point instead of a scalar.

7.3 Avoiding memory copying

It's possible to avoid unnecessary copy operations if you're calculating on the CPU or if the device and the host have a unified memory space. Here are the necessary modifications:

```

Function TestAdd3()
    variable IgorCLUseHostPtr = 2^3

```

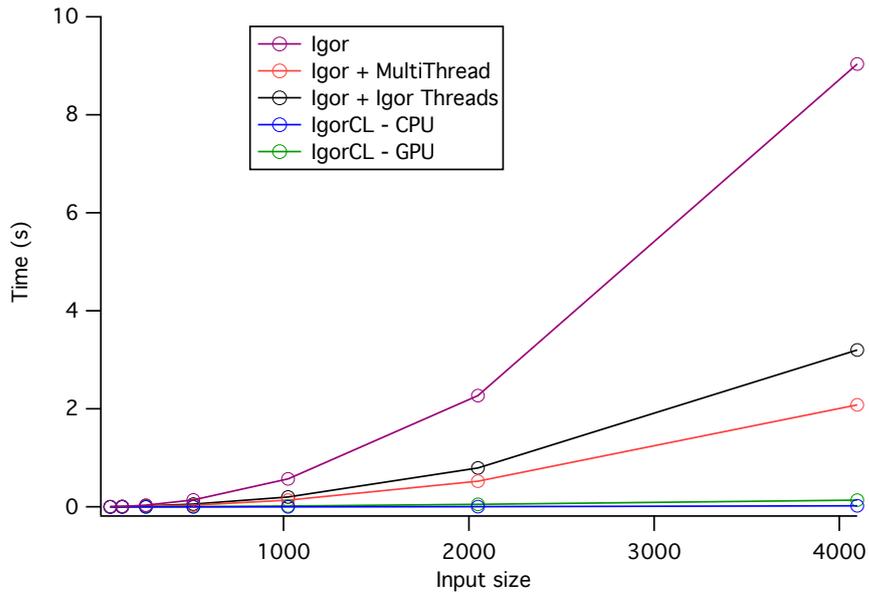


Figure 1: Time taken to calculate a distance matrix for the given number of points with random coordinates. Tested on a Macbook Pro with an Intel i7-3720QM CPU and Intel HD Graphics 4000 GPU.

```

variable IgorCLIsScalarArgument = 2^5
variable nValues = 1e6

Make /S/N=(nValues)/O W_A = 1, W_B = 1, W_C = 0
Make /I/O/N=(1) W_nValues = nValues
Make /I/U/O /N=(4) W_MemFlags = IgorCLUseHostPtr
W_MemFlags[3] = IgorCLIsScalarArgument
variable globalSize = ceil(nValues / 64) * 64
IgorCL /SRCT=Source2() /DTYP="CPU" /MFLG=W_MemFlags /GSZE={globalSize,1,1}
/KERN="VectorAdd2" W_A, W_B, W_C, W_nValues
End

```

On my computer the IgorCL operation executes about six times faster in this version. This large speedup occurs because we do very little work in our kernel, and much of the execution time is taken up simply shifting the data back and forth. This code will also work if the host and device do not have a uniform memory space, but in that case it may be slower compared to the previous version.

7.4 Using the GPU

You can run all of these calculations on the GPU by changing the `/DTYP="CPU"` flag to `/DTYP="GPU"`. It should just work. That's the beauty of OpenCL!

7.5 Benchmarks

Figure 1 shows a simple benchmark. Given a set of (x, y) coordinates, I wrote code that calculates the distance matrix between these points. The kernel was pre-compiled for each device to avoid compilation overhead. As you can see, the IgorCL-based solution performed much faster than the native Igor code. An Igor experiment, 'Benchmark.pxp', implementing this code is included with the IgorCL binaries.