Notes on Setting Up OpenCL for HARK

The module `ConsIndShockOpenCL.py` rewrites the `ConsIndShock` model to use `opencl4py`, a 'heterogeneous computing' toolkit that enables code to choose which available computational resources to use. You can choose either to use a restricted set of the resources of your own CPU, or computational resources on a graphics processing unit (GPU) available to your computer. (You might want to do this because GPU's can massively accellerate the solution of certain problems; you might want to use CPU cores rather than GPUs either for convenient debugging, or because the capabilities of GPU's are more limited).

OpenCL is effectively a restricted subset of C. (The restrictions are part of the reason that not all problems are suited to solution using OpenCL).

`ConsIndShockOpenCL.py` will not run "out of the box"– you must do a bit of extra setup first. The setup has two parts:

1. Install `opencl4py`, which is a python 'wrapper' to interact with OpenCL
2. For the specific machine you are on, configure `opencl4py` so that it knows exactly which compuing resources (CPUs, GPUS) you want to use for your particular project.
    - This reflects an important limitation on the use of OpenCL: By default, OpenCL code is not platform independent, because it requires specific configuration to cater to the particular machine on which it is running

1 Install OpenCL for Your Computer

Different versions (or implementations or "platforms") of OpenCL are required for different devices.

1.1 OpenCL for MacOS

All recent MacOS machines, with recent versions of the OS, include OpenCL installed and properly configured as part of the OS.[1] So, you don't need to install it.

[1] In order to write your own programs directly in OpenCL, you need also to install the Xcode SDK which is free from Apple but requires registration as a developer.

1.2 OpenCL for Windows or Linux

AMD makes a Software Development Kit (SDK) which will work for either AMD or Intel cores.[2] You should read through the installation notes (linked on that page) before installing the SDK, but there isn't much to know. The most important caveat is that, if you have an AMD GPU, you should update your graphics drivers before installing. To run OpenCL on an nVidia GPU, you should install the nVidia CUDA Toolkit from here.

2 Install `opencl4py`

As of this writing,[3] the version of the `opencl4py`package at its PyPi homepage (which is the version that will be installed via 'pip install opencl4py') is outdated **and does not work for MacOS**. The safest way to install `opencl4py` is by retrieving it from its GitHub page. The easiest way to install the package is to copy the directory `src/opencl4py` from the zip archive into the same directory as all of your other packages; if you're using Anaconda, this is something like `.../Anaconda2/lib/site-packages/`.

If you are using anaconda's virtual environments (you *should* be using virtual environments), you need to make sure the package is visible in the virtual environment you are using for the Econ-ARK/HARK tools.[4]

This should be all you need to set up OpenCL, but there's a bit more to do to figure out how it will work on your system.

3 Configuring `opencl4py`

Because OpenCL's *raison d'etre* is to let the user choose the devices (CPU; GPU's) on which code runs, the user needs to specify *which device(s)* will actually run the code; in order to make this choice, you need to figure out the menu of devices available. In Python:

```
import os
os.environ["PYOPENCL_CTX"] = "0:0"
```

[2] There is also an Intel version of OpenCL that seems to work only on Intel CPUs; we do not kow of any advantages of this platform over the AMD SDK.

[3] 2018-03-24

[4] `pip show opencl4py` at the command line.

```
import opencl4py as cl
platforms = cl.Platforms()
print(platforms.dump_devices())
```

The environment variable `PYOPENCL_CTX` must be defined before importing `opencl4py`; the example value here `(0:0)` is guaranteed to work as long as an OpenCL interpreter exists on your system. The final line will print to screen a list of OpenCL platforms, and devices for each platform. This will likely include your CPU, and possibly one or more GPUs on your machine. (The naming of GPUs can be cryptic, as the name often displays as the development codename of that particular model. You might need to look on Wikipedia to translate the GPU names you see here into the name that you know the device by.)

All code that calls `opencl4py` must specify the "context": which devices are to be used. This is done by setting an environment variable *before* `opencl4py` is imported and a context is created:

```
os.environ["PYOPENCL_CTX"] = "0:0,1,2"
```

This sets `PYOPENCL_CTX` to refer to devices 0, 1, and 2 on platform 0, as named by the `dump_devices()` method. You can include in the context any or all devices from the list (for one particular platform). There are two main reasons to exclude a device from the context:

1. The device (GPU) is being used for displaying graphics to a monitor(s). Running OpenCL code on a GPU that's displaying graphics doesn't necessarily result in a crash, but it often does.[5]

2. The device does not have support for double precision floating point operations, and you want to run code that uses double precision. Older or low end GPUs (often chips that are hardwired to micro-ATX form motherboards) do not have native double precision capabilities; this is also possible for *very* old CPUs (486 and lower).

To determine whether a device has double precision capability in OpenCL, run:

[5]Some machines, even some laptops – Mac Retina laptops in particular – have more than one GPU built in, and you can tell the OS to exclusively use one of the GPUs for display purposes, freeing the other GPU for computing.

```
ctx = platforms.create_some_context()
ctx.devices[n].extensions
```

Do this for each device number `n` for this platform (e.g. 0, 1, and 2). If a device has double precision capabilities, its list of extensions should include `cl_khr_fp64` and/or `cl_amd_fp64`. The vast majority of GPUs manufactured since 2011 should have double precision capability; you very likely want to use DP, which requires you to remove any single precision device from the `PYOPENCL_CTX` environment variable.

To run a very simple test kernel, open up the module `OpenCLtest.py` in the `/Testing/` directory of HARK. Set the context to include the appropriate devices, and then run the module. It should display the device list, then give timings for adding two vectors and multiplying by a constant. The structure of how to run a function (or "kernel") using `opencl4py` is as follows:

1. Make an OpenCL *context* as above with `create_some_context()`.

2. Make a *queue* for one of the devices in the context by doing (e.g.) `queue = ctx.create_queue(ctx.devices[0])`. Note that the device number here does not necessarily correspond to the numbering from `cl.dump_devices`. If not all devices were included in this context, then the excluded devices are excluded from the numbering. See `ctx.devices` for the appropriate numbering.

3. Load in a *program* as a `string` with `prg = ctx.create_program(my_code)`. In the test file, we explicitly define the code as a `string`, typed out. In most applications, your OpenCL code will be stored in a separate file which can then be read in to Python as a `string`; see for example `ConsIndShockOpenCL.py` and `ConsIndShockModel.cl`.

4. Define a *kernel* from your program with `krn = prg.get_kernel("test")`. In the test file, the name `test` refers to the name of the kernel in `my_code`.

5. Define memory *buffers* that the kernel will use with the `create_buffer` method. This method must be passed appropriate flags to indicate whether the memory is read-only, write-only, or read-write (these distinctions only matter within a kernel); and for whether the buffer should copy data from a `numpy` array or merely allocate memory space of a given number of bytes.

6. Set the arguments of the kernel to point to the appropriate buffers with `krn.set_args()`. The number of buffers passed to `set_args` should equal the number of inputs listed for that kernel, as in its code.

7. Put the kernel in the queue to be executed with `queue.execute_kernel`. The second argument for this method is the *global work size*, or total number of work items. The third argument is the *work group size*, which can be set to a default with `None`. Setting work group size to something other than a factor of 16 on a GPU can result in extreme slowdowns.

8. Read the buffer back into a `numpy.array` with `queue.read_buffer`.

The command queue works exactly as you expect it would, holding a list of operations to perform. If you write a loop that executes a kernel 100 times, then after that loop you read the buffer, the resulting `array` will be the result after all 100 kernel executions. If you want to edit the contents of a buffer after creating it, you can use `queue.write_buffer`.

In the test module, we have provided two different kernels: a single precision version and a double precision version. You can choose which version is used with the `use_DP` variable near the top of the file. Note that this boolean both sets `my_code` and `my_type`. When passing `arrays` to OpenCL buffers, you must take care that the `dtype` of the `array` corresponds to the data type used in the kernel. If you make a buffer using a double precision array (`float64`, the default in `numpy`), but then use that buffer in a kernel that expects `float`, a 32-bit floating point number, your code will run... but it will generate nonsense. When the kernel executes, it will try to interpret the data in the buffer *as if* it were made up of `float`s, reading in 32 bits per number. These bits actually correspond to *half* the bits of some 64-bit floating point number (the first half or second half) and thus have no relation to the number you actually want to represent!

The module `ConsIndShockOpenCL.py` provides a non-trivial application of OpenCL. This module defines the class `IndShockConsumerTypesOpenCL`, which is constructed by passing a list of `IndShockConsumerTypes` instances. Data that defines the problem of each instance type will be loaded into OpenCL buffers on the chosen device. If the user wants to manually move existing model solutions (found in HARK) into OpenCL for simulation, this can be done with the method `makeSolutionBuffers()`. A user who wants solution to be performed by OpenCL, should then run the method `prepareToSolve()`, which makes buffers that will

hold the solution, as well as additional objects needed to solve the model. All types can then be solved using `solve()`, filling in the solution buffers. As of 2018-03-24, OpenCL can only solve lifecycle models (`cycles=1`), but we anticipate that this will be improved in the near future. Whether the solution is loaded from Python or solved in OpenCL, the model can be simulated using `simNperiods()` (which will be renamed). Simulation variables (like `cNrmNow`) can be extracted from OpenCL buffers into attributes of each agent type with the `readSimVar()` method; `writeSimVar()` moves simulation data in the opposite direction, into a buffer.