Kernel Execution API

If you've written a computation kernel that you now want to execute through the DECADES framework, this is the right document for you!

If you haven't written your kernel, or have questions about how to write a kernel, please see the kernel implementation API (Numba.ipynb) documentation.

If you are unsure what a *kernel* is, please read the DECADES executive summary in the quick guide documeent (Quick_Guide.ipynb)

Remember!!! You can only have one kernel function per script! But feel free to write as many scripts as you'd like.

Kernel Signature

The *signature* of a kernel, i.e. its arguments, has a few restrictions. We don't think you'll find them too difficult, and may even be helpful down the road for you.

No return value

A kernel should not return a value, or a tuple, or anything.

Calm down. We know what you're thinking. A kernel *needs* to return something, otherwise why do any computation at all!

You can return as many values as you want through arguments!

Thus, a Numba function signature that looks like this: `int32(int32[:])` , i.e. a function that takes an integer array and returns an integer, would be changed to: `(int32[:],int32[:])` and you'd store the return value in 0th location of the second arg. See our example on Numba_Decades_Examples (Numba_Decades_Examples.ipynb) to see how this is done.

If you are not comfortable with Numba function signatures, please revisit their documenation here (https://numba.pydata.org)

example

Consider this example that returns the length of an array

```
@njit(int32(int32[:]))
def length_of_array(s):
    return len(s)
```

Change the function from returning a value, to returning through an argument.

```
@njit((int32[:], int32[:]))
def length_of_array(s, return_array):
    return_array[0] = len(s)
```

Keep in mind that you will have to declare your return argument before calling the function, e.g. `return_arg = np.zeros(1, dtype=np.int32)`

Tile ID and number of tiles

A kernel should take in all arguments required for computation and return. *But* a DECADES kernel must take 2 additional special arguments. Both are of type int32 and represent a unique Tile ID and the number of tiles.

Recall that DECADES is a fundamentally parallel system. When you launch a DECADES kernel, you actually launch many instances of the same kernel in parallel. These kernels can access different data or branch to distinct computation using their unique tile ID and the total number of tiles.

You can set the number of tiles to launch your kernel with using the function `DEC_Options.set_num_threads` , described later on this page. You will never call the kernel directly. It will be called using decades kernel launcher described later in this document.

Shared Memory

Our shared memory model is simple.

- For arguments: Anything that is an array is shared. Anything that is a scaler is private.
- local variables:
  - scalars: not shared
  - arrays: *sharing is current undefined*

Note

This time of parallelism might remind you of popular GPU programming models: CUDA and OpenCL. Where a single kernel is executed by many threads and shared memory is passed in through arguments.

Important!

Even if you write a single-threaded kernel, the kernel *must* still take these two arguments.

Example

Consider this example that computes `c = a + b` for two int32 arrays: `a` and `b`

```
@njit((int32[:],int32[:],int32[:]))
def array_add(a,b,c):
    for i in range(len(a)):
        c[i] = a[i] + b[i]
```

to be a DECADES kernel, this must take two additional arguements, even if they are unused.

```
@njit((int32[:],int32[:],int32[:], int32, int32)
)
def array_add(a,b,c, tid, num_tiles):
    for i in range(len(a)):
        c[i] = a[i] + b[i]
```

Now you have access to multiple tiles, you may as well use them! This program is *embarassingly* parallel and can easily be made parallel using the tid and num_threads

```
@njit((int32[:],int32[:],int32[:], int32, int32)
)
def array_add(a,b,c, tid, num_tiles):
    for i in range(tid, len(a), num_tiles):
        c[i] = a[i] + b[i]
```

DECADES Pipeline

We're getting closer to actually executing our kernel! We now need to tell Numba to compile the computation kernel through the DECADES compiler framework. This is conveniently provided as the `DEC_Pipeline` which can be imported and given in the 'njit' signature.

We also recommend using the `nogil=True` option in the `@njit` signature. *But* please do not use `Parallel=True` . We handle our parallelism explicitly through the tile id and number of tiles described above.

Any function that you expect to call in the kernel must also be given with an `@njit` signature with the `DEC_Pipeline`

Example

Consider our array_add kernel from above. We need to pass the DEC_Pipeline to ensure it is compiled through our framework.

```
from DEC_Pipeline import DEC_Pipeline

@njit((int32[:],int32[:],int32[:], int32, int32)
, nogil=True, pipeline_class=DEC_Pipeline)
def array_add(a,b,c, tid, num_tiles):
    for i in range(tid, len(a), num_tiles):
        c[i] = a[i] + b[i]
```

We now know that the kernel will be compiled through the DECADES framework!

Compiler Options

Before compiling the kernel, we can set a variety options. These can accessed in through the DEC_Options class, part of the DEC_Pipeline. Its best to include this when including the DEC_Pipeline. Please note that these must be called before the kernel launcher (described below)

```
from DEC_Pipeline import DEC_Pipeline, DEC_Optio
ns
```

Number of Tiles

This options sets the number of tiles to launch the kernel with. Probably best to start with `1` for development and then scale up as you become comfortable. The kernel launcher (described below) will then launch the kernel with the number tiles and provide each kernel with a unique tile id and the number of tiles.

```
DEC_Options.set_num_threads(1)
```

The next command can get the number of threads:

```
int_num_threads = DEC_Options.get_num_threads()
```

Decoupling Supply and Execute

This options sets the compiler to generate a supply and execute kernel that communicate through a specialized API. This implements the latency-hiding techniques described in this paper (http://mrmgroup.cs.princeton.edu/papers/taejun_micro15.pdf):

It is likely that this setting will not provide a speedup at all when executing, and likely will provide a slowdown. It is a hardware/software co-design feature and should be executed through the simulator, which simulates the necessary hardware additions to see improvements from this feature. Please see the next section (simulator backend) for more details.

Executing a kernel with this feature enabled will dump emulation stats about the number and type of communications between the compute and supply tile.

```
DEC_Options.set_decoupled_mode()
```

Simulator Backend

This option compiles the kernel with the appropriate tracing instrumentation. This is likely not useful in early stages of development. Additionally, it causes *significant* overhead, because of the trace generation.

If you are interested in running the simulator, please see the documentation here (Low_Level_DECADES_Doc.pdf)

The other option to running the simulator (without this option) is to use an *evalution* script, described in our evaulation documentation (Evaluation_Guide.ipynb)

```
DEC_Options.set_simulator_target()
```

Preset Options

It is also possible to run the kernel through a series of DECADES presets and automatically compare execution. This is documented in our evalutation documentation (Evaluation_Guide.ipynb). To enable the kernel to pickup these presents, you must

1. Not include any of the above options
2. Include this instruction: `DEC_Options.preset_config()`

Kernel Launcher

It is finally time to launch your kernel! We have provided a special mechanism to do this, in which the number of tiles, tile ids, simulator, etc. is taken care of behind the scenes for you (depending on which of the above options were set).

This is done through a special function that is part of the DEC_Pipeline called `decades_launch_kernel` . It is recommended to include this when including everything else from this file:

```
from DEC_Pipeline import DEC_Pipeline, DEC_Optio
ns, decades_launch_kernel
```

Now if you've set up your kernel and options above, you can run your kernel using this function.

It takes as arguments:

1. The kernel function name
2. All of the kernel arguments *except* tile ID and number of tiles

The function returns the time, in seconds it took to execute the kernel. Because compilation can take some time, this value reports the actual kernel compuation time.

So if we use our array_add kernel above, we launch it like so:

```
a = np.zeros(1024, np.type=int32)
# Populate a with data
b = np.zeros(1024, np.type=int32)
# Populate b with data
c = np.zeros(1024, np.type=int32)
# Populate c with data

t = decades_launch_kernel(array_add, a,b,c)

print("Time to run array_add kernel (seconds): "
+ str(t))
```

During this time, you will see the DEC++ (DECADES Compiler) command line call. This is useful to capture for bug reports or when asking for help. For example, depending on your compiler settings, you might see:

executing DEC++: DEC++ -fn 1 --target numba -t 1 -m db

DECADES_Numba_out >

DECADES_Numba_out/dec_compile_log_std.txt 2> DECADES\

_Numba_out/dec_compile_log_stderr.txt

There will also be print outs letting you know when compilation is finished and when kernel execution has started.

Although you should never have to directly deal with the DEC++ compiler directly when using the DECADES Numba flow, documentation can be found here (Low_Level_DECADES_Doc.pdf). DEC++ is used directly for dealing with C++ and LLVM and is not a recommended way to program for DECADES.

Summary

Once you've written a kernel, using numba, numpy and our associated kernel implementation documentation (here (Numba.ipynb), then you can launch your kernel with various options using the API described here. Please remember:

- Only one kernel launch per script.
- Kernels cannot directly return values
- Kernel *must* contain a tile id and number of tiles argument (both int32), even if they are single threaded
- Do not call a kernel directly. Use the `decades_kernel_launch` function.

Example.

Here is an example of the array_length kernel from above, all put together:

```
import numpy as np
# include necessary packages
from DEC_Pipeline import DEC_Pipeline, DEC_Optio
ns, decades_launch_kernel

# Even though it is single threaded, still need
extra two parameters (tid and num_tiles)
# Make sure to compile with DEC_Pipeline
@njit((int32[:], int32[:]), int32, int32, nogil=
True, pipeline_class=DEC_Pipeline)
def length_of_array(s, return_array, tid, num_ti
les):
    # it may be called with multiple tiles, but
we only want one tile to do this work
    if tid == 0:
        # return values through arg arrays
        return_array[0] = len(s)

#set kernel to be launched with 2 threads
DEC_Options.set_num_threads(2)

my_array = np.zeros(1024, dtype = np.int32)
return_array = np.zeros(1, dtype = np.int32)

t = decades_launch_kernel(length_of_array, my_ar
ray, return_array)
assert(return_array[0] == 1024)
print("Kernel execution time (seconds): " + str(
t)
```