Evaluation Guide

If you've written a kernel and want to get estimates on how well it performs on the decades architecture, this is the guide for you!

We've wrapped up some nice presets, the compilation run, and the simulator run all in a nice package. We've even got baseline numbers from some baseline implementations that you can compare against.

We'll be using the two examples from the numba example documentation (Numba_Decades_Examples.ipynb), i.e. matrix multiplication and reduction.

Specification

Our evaluation tools expect your python tools to have a certain specification; this is so we can keep things straight between application names and inputs more easily. The specification is:

```
<script_name>.py <input_name>
```

That is, your script must take only one argument, and it must be the input name.

Recall

Recall that in the examples (Numba_Decades_Examples.ipynb), we had our kernels take a single file as input even though it seemed quite inconvenient? We did that to fit this specification.

Script and Input Naming

With respect to what you name the script: please include the application name in the script name *if you want to compare against a baseline*.

We will use a simple text analysis to try and automatically determine which baseline you are aiming to implement, but this is not perfect.

For example, if you are programming an application "Scan Statistics", please make sure your script name contains "Scan_Statistics" in it somewhere.

This is similar for input name.

Baselines

Currently we have a few baseline numbers hardcoded in, but we will have to update this on a per-assignment basis.

Please contact us ASAP if you believe you should have a baseline to compare against, but the program is unable to find one.

Finally, our evaluator will automatically set some default presets for you that we think will mirror our 2020 system. To use these presets, please remove all `DEC_Options` settings from your script (see here (Decades_Numba_Pipeline.ipynb)) and add the single line: `DEC_Options.preset_config()` .

Running the Evaluator.

If your script fits the above specification, then you can run it through the evaluator! This program is simply called `decades_numba_presets` (`decades_tf_presets` for TensorFlow scripts) and where you would typically run `python` , you can now run this program instead!

For example, previously we tested the mat_mul.py example by doing:

```
$ python mat_mul.py thirty_two.txt
```

To run through the evalutor, we can now just run:

```
$ decades_numba_presets mat_mul.py thirty_two.txt
```

This script does both a compiler and simulation pass and may take quite some time to run, so *please be patient*. For extremely large datasets, this could take hours. We hope you will try on smaller data sets first to get comfortable with the system.

Luckily our two examples are small enough to run in this framework in about a minute.

Outputs

If we try running our matrix multiplication, i.e. using the command:

```
$ decades_numba_presets mat_mul.py thirty_two.txt
```

We should see some outputs about running the simulator, compiler etc. But the big output at the end should look something like this:

raw results ----- chip GOPS/Watt ---------------- ----------- baseline 1.26087 decades framework 6.79576 DECADES framework improves over baseline by: chip GOPS/Watt ---------------- ----------- decades framework 5.390x

So this means our DECADES architecture got about a 5x increase over a baseline matrix multiplication run on the baseline system! Pretty cool!

You can play with the implementation of matrix multiplication to see if you can get the number higher, or lower based on design decisions.

Note

the metric obtained here is just GOPS/Watt, or giga operations per watt. Its not necessarily fast processing that raises this number. For example, a multithreaded application that does not scale well might actually lower this number (due to more cores using more energy), even if it computes slightly faster.

Our DECADES chip uses a low-power, in-order design to help increase this metric. We encourage you to play with multithreaded examples and different implementation chioces.

Inputs

If you want to run your script on a bigger input, please go ahead and try! The only issue is that we may not have a baseline for the different inputs. In this case, you should record the raw results associated with your script. For example, here is the output associated with running mat_mul.py on matrices of size (16x16), for which we do not have a baseline.

In this case the output is kind of boring (shown below), as we cannot compute these metrics without a baseline.

raw results ----- chip GOPS/Watt ---------------- ----------- baseline NA decades framework NA DECADES framework improves over baseline by: chip GOPS/Watt ---------------- ----------- decades framework NA

Verbose Output

We think you might be interested in other metrics, even if you can't compare to the baseline, so we've added a way for you to do this. Simply use the same evaluator script, but set the environment variable: `DECADES_VERBOSE` the value `1`

This produces a more interesting table, which includes other metrics such as cycles and power measurements:

raw results ----- chip cycles global_energy global_avg_power GOPS/Watt ---------------- -------- -------------- ----------------- ----------- baseline NA NA NA NA decades framework 370742 1.93081e-06 0.0260398 NA DECADES improves over baseline by: ----- chip cycles global_energy global_avg_power GOPS/Watt ---------------- -------- -------------- ------------------ ----------- decades framework NA NA NA NA

There are still a lot of `NA` values in there because a baseline is not available, but this allows you to peek under the hood and see more metrics of what is going on.

Setting Multiple Tiles (Threads)

The safest option for the evaluator is to run your kernel single threaded. That's okay, there's still a lot of transparent optimizations that happen (e.g. different hardware technology and supply/compute decoupling).

If you are adventurous (and we hope you will be!) you can explicitly tell your script to run with multiple tiles. If your application scales well, then you will likely see these metrics change in your favor!

To do this, after the `DEC_Options.preset_config()` , simply add a call to `DEC_Options.set_num_tiles(N)` , where N is the number of tiles you want to run with.

If you don't remember the parallel framework, please jog your memory by looking at the kernel execution documentation (here) [Decades_Numba_Pipeline.ipynb]

Final Thoughts

Using our transparent optimizations (e.g. hardware technology and decoupling), multiple tiles, and clever algorithmic choices, we have been able to see impressive gains over the many baselines (over 1000x in some cases)!

We hope you have a similar experience and please contact us for any question or comments!