Loo.py: A Loop Generation Tool for CPUs and GPUs

ANDREAS KLÖCKNER[†] and TIM WARBURTON[‡]

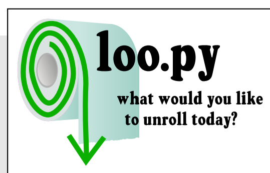
[†] Courant Institute of Mathematical Sciences, New York University, New York, NY
[‡] Computational and Applied Mathematics, Rice University, Houston, TX

Performance-aware implementations of even simple mathematical concepts typically encounter an explosion in code size and complexity. Much of this complexity is artificial, as the ideas driving the implementation are often taken from a rather limited set of tricks. Encoding these tricks in C is tedious, error-prone and leads to redundant, unmaintainable results. This is a failure of the currently available tools. While progress in hardware has been abundant, the situation with tools has been called a "train wreck" (Tim Mattson at a previous edition of this workshop).

Loo.py is our attempt at fighting complexity growth. By extracting just the tuning ideas and leaving their implementation to the machine, visible redundancy is greatly reduced. Our approach is based on program transformation. A user starts with a semi-mathematical algorithm description (see an example shown in Figure 1) and then issues transformation commands (see Figure 2) that gradually make the code suitable for a target machine. The transformations uniquely specify the generated program, which is output in human-readable C. This puts the user in charge of the optimization, eliminates guesswork and helps debuggability. It also differentiates the approach from directive-based compilation, where the compiler has final say and the result cannot easily be inspected. Further, our transformation is driven from Python, a high-level language, making it practical for user code to apply transformations based on hardware and operational considerations at run time.

I will demonstrate the use of Loo.py and its effectiveness for PDE and linear algebra problems, including ones relevant to oil and gas.

This work is based on an ongoing effort to supply high-level language tools for performance programming, realized in the PyOpenCL and PyCUDA packages. These packages are having broad impact across academia and industry, with 40,000+ downloads in the last 12 months. I will briefly cover the use of these packages, and I will show how they enable tools like Loo.py.




```
import loopy as lp

order = "C"
dtype = np.float32

knl = lp.make_kernel(ctx.devices[0],
    "[n] -> {[i,j,k]: 0<=i,j,k<n}",
    [
        "c[i, j] = sum_float32(k, a[i, k]*b[k, j])"
        ],
    [
        lp.ArrayArg("a", dtype, shape="n, n", order=order),
        lp.ArrayArg("b", dtype, shape="n, n", order=order),
        lp.ArrayArg("c", dtype, shape="n, n", order=order),
        lp.ScalarArg("n", np.int32, approximately=1000),
        ],
    name="matmul")
```

Figure 1: Loo.py description of inputs, outputs and mathematical procedure to be carried out, in this simple case matrix multiplication.

```
knl = lp.split_dimension(knl, "i", 16,
        outer_tag=None, inner_tag="l.1")
knl = lp.split_dimension(knl, "j", 16,
        outer_tag="g.1", inner_tag="l.0")

knl = lp.split_dimension(knl, "k", 16)
knl = lp.split_dimension(knl, "i_outer", 4,
    inner_tag="ilp", outer_tag="g.0")
knl = lp.add_prefetch(knl, 'a',
    ["k_inner", "i_inner", "i_outer_inner"])
knl = lp.add_prefetch(knl, 'b', ["j_inner", "k_inner", ])

knl = lp.tag_dimensions(knl,
        {"i_outer_inner_fetch_a": None})
```

Figure 2: Loo.py description of loop transformations to adapt the problem from Figure 1 to Nvidia GPU hardware.