

Parallel Vectorize

The `parallel_vectorize.py` module contains a set of `llvmpy` code generators for creating multithreaded *ufunc*. It depends on the new `numpy.fromfunc` for turning arbitrary function pointers into *ufunc*.

From LLVM Function

The `parallel_vectorize_from_func` method generates multithreaded *ufunc* from LLVM functions.

First, we will implement a workload function:

```
In [1]: from llvm_cbuilder import *
        from llvm_cbuilder import shortnames as C
        from llvm.core import *

        # Implement a workload
        class Square(CDefinition):
            _name_ = 'square'
            _retty_ = C.double          # 1 output: double
            _argtys_ = [('x', C.double)] # 1 input: double

            def body(self, x):
                self.ret(x * x)

        m = Module.new('my_module')
        llvm_square = Square()(m) # Generate a llvm function
        print(llvm_square)

define double @square(double %x) {
decl:
    %0 = fmul double %x, %x
    ret double %0
}
```

Then, we will generate a *ufunc* from `llvm_square`:

```
In [2]: from llvm.ee import *
        engine = EngineBuilder.new(m).create() # Generate JIT engine

        from numbapro.vectorize.parallel import parallel_vectorize_from_func
        import numpy as np

        ufunc_square = parallel_vectorize_from_func(llvm_square,
                                                    [[np.dtype(np.double).num]*2],
                                                    engine) # Generate UFunc
```

We are ready to use `ufunc_square` as a regular *ufunc*.

```
In [3]: A = np.arange(10., dtype=np.double)
        ufunc_square(A)
```

```
Out[3]: array([ 0.,  1.,  4.,  9., 16., 25., 36., 49., 64., 81.])
```

Here's another example that uses three inputs:

```
In [4]: class SumOfThree(CDefinition):
        _name_ = 'sum.of.three'
        _retty_ = C.int
        _argtys_ = [('x', C.int),
                     ('y', C.int),
                     ('z', C.int)]
        def body(self, x, y, z):
            self.ret( x + y + z )

        llvm_sum3 = SumOfThree()(m)
        ufunc_sum3 = parallel_vectorize_from_func(llvm_sum3,
                                                  [[np.dtype(np.int32).num]*4],
                                                  engine)

        A = np.arange(10, dtype=np.int32)
        B = A * 10
        C = A * 100
        ufunc_sum3(A, B, C)
```

```
Out[4]: array([ 0, 111, 222, 333, 444, 555, 666, 777, 888, 999], dtype=int32)
```

Internals

There are four functions behind each multithreaded *ufunc*.

- I. the workload function (user defined);
- II. the thread worker function (UFuncCoreGeneric);
- III. the thread manager function (ParallelUFuncPlatform);
- IV. the ufunc entry point function (SpecializedParallelUFunc).

UFuncCoreGeneric specializes to a llvm function type. **It currently understands simple builtin scalar types (integers, float, double) only as arguments and return-type for the workload function.** It sends work items to the workload function and performs work-stealing when it has finished its own workqueue. Work-stealing uses atomic compare-exchange (or CAS) instruction to acquire ownership of a workqueue. Work-stealing is implemented in the UFuncCore._do_work_stealing. It can be disabled on platform that does not support atomic operations.

ParallelUFuncPlatform specializes to the maximum number of threads. It divides all works equally among all threads. Each thread executes the function generated by UFuncCoreGeneric once.

SpecializedParallelUFunc is the specialized *ufunc* entry point for a specific combination of workload, UFuncCoreGeneric and ParallelUFuncPlatform.

Here's an example that uses SpecializedParallelUFunc directly for the SumOfThree workload.

```
In [5]: import numbapro.vectorize.parallel as pv
        # specialize
        def_spuf = pv.SpecializedParallelUFunc(pv.ParallelUFuncPlatform(num_thread=2),
                                                pv.UFuncCoreGeneric(llvm_sum3.type.pointee),
                                                CFuncRef(llvm_sum3))

        # define
        spuf = def_spuf(m)
        print(spuf.name)

        specialized_parallel_ufunc_2_ufunc_worker.i32.i32.i32.i32_sum.of.three
```

CFuncRef also accepts arbitrary function pointer as long as the function type is provided.

```
In [6]: # specialize
fnty = llvm_sum3.type.pointee
sum3ptr = engine.get_pointer_to_function(llvm_sum3)
print("as function pointer: %x" % sum3ptr)
def_spuf = pv.SpecializedParallelUFunc(pv.ParallelUFuncPlatform(num_thread=2),
                                       pv.UFuncCoreGeneric(fnty),
                                       CFuncRef('sum3.as.ptr', fnty, sum3ptr))

# define
spuf = def_spuf(m)
print(spuf.name)
```

```
as function pointer: 7f3d501666f0
specialized_parallel_ufunc_2_ufunc_worker.i32.i32.i32.i32_sum3.as.ptr
```