PARALLELISM IN PYTHON*: DIRECTING VECTORIZATION WITH NUMEXPR*

Boosting Performance for Computing with Arrays and Numerical Expressions

Fabio Baruffa, PhD, Technical Consulting Engineer, Intel Corporation

Python* has several pathways to vectorization (i.e., instruction-level parallelism), ranging from just-in-time (JIT) compilation with Numba*[1] to C-like code with Cython*. One interesting way of achieving Python parallelism is through NumExpr*, in which a symbolic evaluator transforms numerical Python expressions into high-performance, vectorized code. NumExpr achieves this by vectorizing in chunks of elements instead of compiling everything at once—thus creating accelerated object kernels that are usable from Python code. In this article, we'll explore how to refactor Python code to take advantage of NumExpr's capabilities.

Parallelization of Numerical Expressions

The flexibility of Python, with its easy syntax, allows developers to rapidly prototype numerical computations with the help of libraries like NumPy* and SciPy*. But the Python language wasn't developed with parallelism in mind—although it's a key requirement to get performance out of modern vector and multicore processors. So how is it possible to vectorize numerical expressions using Python?

A numerical expression is a mathematical statement that involves numbers and mathematical symbols to perform a calculation (e.g., `11*a-42*b`). In Python, this expression can also operate on arrays `a` and `b` defined from the NumExpr package. In this case, similar expressions working on arrays are accelerated, making use of intrinsic parallelism and vectorization, compared to the same calculation in standard Python.

To boost performance, NumExpr can use the optimized Intel® Vector Mathematical Function Library (Intel® VML), included in **Intel® Math Kernel Library (Intel® MKL)**. This makes it possible to accelerate the evaluation of mathematical functions (e.g., sine, exponential, or square root) that operate on vectors stored contiguously in memory.

Refactoring Common NumPy Calls for NumExpr

To make use of the NumExpr package, you only need to pass the computational string to the `evaluate` function. Then it's compiled into an object, leaving the entire computation at low-level code before completion. After that, the result is returned to the Python layer, avoiding too many calls to the Python interpreter.

Let's look at an example where we compute a simple expression for NumPy arrays:

```
import numpy as np
import numexpr as ne
a = np.arange(1e6)
b = np.arange(1e6)
%timeit 11*a-42*b
14.2 ms ± 826 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
%timeit ne.evaluate("11*a-42*b")
3.51 ms ± 248 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In this case, we have a 4x speedup due to the intrinsic vectorization enabled by Intel VML. The library can also perform in-place operations, where the copying overhead is negligible.

Now let's evaluate the speedup from NumExpr when we use a mathematical function, where the benefit of Intel VML becomes more evident:

```
import numpy as np
import numexpr as ne
a = np.arange(1e6)
b = np.arange(1e6)
%timeit np.exp(a)**2 + np.sqrt(b)**3
498 ms ± 11.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
%timeit ne.evaluate("exp(a)**2 + sqrt(b)**3")
26.4 ms ± 2.23 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In this case, we achieve higher performance due to the optimize `sqrt` function in Intel MKL. The speedup is close to 19x. This indicates that the NumPy library doesn't provide the acceleration we expect for some expressions. Also, the NumExpr implementation circumvents memory allocations for intermediate results, which gives better cache utilization and reduces memory overhead. We can really see the benefit of these optimizations in computations with large arrays.

Controlling the NumExpr Evaluator

Since NumExpr uses the Intel VML library internally, it computes the mathematical functions only for the types the library allows. It also operates on real and complex vector arguments with unit increment, integer, and Boolean. In cases where the types of arrays don't match in the evaluate expression, they're cast according to the usual inference rules.

The performance depends on a number of factors, including vectorization and memory overhead. For this reason, you can use some of Intel VML's functions to tune performance and control numerical accuracy (and eventually the number of threads).

To get information about the Intel VML library version, you can call the function `get_vml_version()`, which might be useful for checking the installation. All the vector functions support the following accuracy modes through the function `set_vml_accuracy_mode(mode)`. The mode can be set to:

- **High**, equivalent to High Accuracy (HA), the default mode.
- **Low**, equivalent to Low Accuracy (LA), which improves performance by reducing accuracy of the two least significant bits.

- **Fast**, equivalent to Enhanced Performance (EP), which provides better performance at the cost of significantly reduced accuracy. Approximately half of the bits in the mantissa are correct.

For more information, see the Intel MKL Developer Reference[2] and the official documentation of NumExpr.[3]

NumExpr can also be used to control the number of threads. The function `set_num_threads(nthreads)` sets the maximum number of threads to be used by Intel VML operations. The return value is the previous setting of the number of threads in the current environment. Let's modify the previous example to use threads to improve performance even further:

```
import numpy as np
import numexpr ad ne
a = np.arange(1e6)
b = np.arange(1e6)
ne.set_num_threads(4)
%timeit ne.evaluate("exp(a)**2 + sqrt(b)**3")
7.2 ms ± 137 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The speedup is 3.7x, with 93% parallel efficiency. In this example, more threads equal better performance.

```
ne.set_num_threads(8)
%timeit ne.evaluate("exp(a)**2 + sqrt(b)**3")
3.94 ms ± 191 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Using NumExpr as alternative to NumPy can give significant performance benefits for computing with arrays and numerical expressions, thanks to the Intel VML performance library. The syntax is very similar to NumPy and, with a couple of easy function calls, you can transition your code to NumExpr.

References

1. ***The Parallel Universe*, issue 36**
2. **Developer Reference for Intel® Math Kernel Library**
3. **NumExpr 2.0 User Guide**