Assignment 1 - Matrix multiplication in Numba

Deadline: Monday, 18 October 10am

Print to PDF

Note: You must do this Assignment, including codes and comments as a single Jupyter Notebook. To submit, make sure that you run all the codes and show the outputs in your Notebook. Printout the notebook as pdf and submit the pdf of the Assignment.

We consider the problem of evaluating the matrix multiplication $C = A \times B$ for matrices $A, B \in \mathbb{R}^{n\times n}$. A simple Python implementation of the matrix-matrix product is given below through the function `matrix_product`. At the end this function is checked against the Numpy implementation of the matrix-matrix product.

```python
import numpy as np

def matrix_product(mat_a, mat_b):
    """Returns the product of the matrices mat_a and mat_b."""
    m = mat_a.shape[0]
    n = mat_b.shape[1]

    assert(mat_a.shape[1] == mat_b.shape[0])

    ncol = mat_a.shape[1]

    mat_c = np.zeros((m, n), dtype=np.float64)

    for row_ind in range(m):
        for col_ind in range(n):
            for k in range(ncol):
                mat_c[row_ind, col_ind] += mat_a[row_ind, k] * mat_b[k, col_ind]

    return mat_c

a = np.random.randn(10, 10)
b = np.random.randn(10, 10)

c_actual = matrix_product(a, b)
c_expected = a @ b

error = np.linalg.norm(c_actual - c_expected) / np.linalg.norm(c_expected)
print(f"The error is {error}.")
```

```
The error is 1.0814245296430078e-16.
```

The matrix product is one of the most fundamental operations on modern computers. Most algorithms eventually make use of this operation. A lot of effort is therefore spent on optimising the matrix product. Vendors provide hardware optimised BLAS (Basis Linear Algebra Subroutines) that provide highly efficient versions of the matrix product. Alternatively, open-source libraries sucha as Openblas provide widely used generic open-source implementations of this operation.

In this assignment we want to learn at the example of matrix-matrix products about the possible speedups offered by Numba, and the effects of cache-efficient programming.

- Benchmark the above function against the Numpy dot product for matrix sizes up to 1000. Plot the timing results of the above function against the timing results for the Numpy dot product. You need not benchmark every dimension up to 1000. Figure out what dimensions to use so that you can represent the result without spending too much time waiting for the code to finish. To perform benchmarks you can use the `%timeit` magic command. An example is

  ```python
  timeit_result = %timeit -o matrix_product(a, b)
  print(timeit_result.best)
  ```

- Now optimise the code by using Numba to JIT-compile it. Also, there is lots of scope for parallelisation in the code. You can for example parallelize the outer-most for-loop. Benchmark the JIT-compiled serial code against the JIT-compiled parallel code. Comment on the expected performance on your system against the observed performance.

- Now let us improve Cache efficiency. Notice that in the matrix B we traverse by columns. However, the default storage ordering in Numpy is row-based. Hence, the expression `mat_b[k, col_ind]` jumps in memory by `n` units if we move from k to $k+1$. Run your parallelized JIT-compiled Numba code again. But this time choose a matrix B that is stored in column-major order. To change an array to column major order you can use the command `np.asfortranarray`.
- We can still try to improve efficiency. A frequent technique to improve efficiency for the matrix-matrix product is through blocking. Consider the command in the inner-most loop `mat_c[row_ind, col_ind] += mat_a[row_ind, k] * mat_b[k, col_ind]`. Instead of updating a single element `mat_c[row_ind, col_ind]` we want to update a $\ell\times\ell$ submatrix. Hence, the inner multiplication becomes itself the product of two $\ell\times\ell$ submatrices, and instead of iterating element by element we move forward in terms of $\ell\times\ell$ blocks. Implement this scheme. For the innermost $\ell\times\ell$ matrix use a standard serial triple loop. Investigate how benchmark timings depend on the parameter ℓ and how this implementation compares to your previous schemes. For simplicity you may want to choose outer-matrix dimensions that are multiples of ℓ so that you need not deal in your code with the remainder part of the matrix if the dimensions are not divisible by ℓ. Note that while such schemes are used in practical implementations of the matrix-matrix product it is not immediately clear that a Numba implementation here will be advantageous. There is a lot going on in the compiler in between writing Numba loops and actually producing machine code. Real libraries are written in much lower-level languages and can optimize closer to the hardware. Your task is to experiment to see if this blocked approach has advantages within Numba.

In all your implementations make sure that you write your code in such a way that SIMD code can be produced. Demonstrate if your produced codes are SIMD optimized.

Previous
Further topics

Next
Assignment 2 - GPU Accelerated solution of Poisson problems