Fast Matrix Multiplication

Avery Faller, Abishek Malali, Haosu Tang, Thomas Seah

May 7, 2016

1. Introduction

Matrix multiplication, in its simplest form, is fundamentally important for solving numerous linear algebra problems in science and engineering. Dot product of two large matrices had been a challenge in numerical computing, especially in sizable project with large dimensions. The development of modern technology that spawns on such explosion of data, including machine learning, pattern recognition and physical simulations, calls for fast and optimized methods to get accurate numerical calculation. In this context, we discuss the algorithms people came up with to improve the matrix multiplication performance. We use Python 2.7 as our experiment environment, with packages Cython, Numpy, and Scipy.sparse. We compare the multiplication performance on both dense and sparse matrices, using elementwise (naive), Cython optimized, Cython with nogil, Numpy dot and Scipy.sparse dot implementations.

2. Algorithm

2.1 Elementwise

The product of two matrices, $\mathbf{A}$ and $\mathbf{B}$, of size $n \times m$ and $m \times p$:

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1p} \\ B_{21} & B_{22} & \dots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \dots & B_{mp} \end{bmatrix}$$

is written as:

$$\mathbf{AB} \triangleq \mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}$$

, where the product $\mathbf{C}$ is $n \times p$ and:

$$\mathbf{C}_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

The python code for this naive approach is:

```
def dot_elementwise(matA, matB):
    result = []
    for i in xrange(len(matA)):
        thisrow = []
        for j in xrange(len(matB[0])):
            element = 0
            for k in xrange(len(matB)):
                element += matA[i][k] * matB[k][j]
            thisrow.append(element)
```

```
        result.append(thisrow)
    return result
```

This method does not include any optimization; for each element of the resulting matrix $\mathbf{C}$, it loops over all the elements in the corresponding row of $\mathbf{A}$ and column $\mathbf{B}$. The time complexity for this approach is $O(nmp)$.

We test this method on:

1. a $(100, 200)$ dense matrix dot a $(200, 300)$ dense matrix, where every element is a random number between $(-20000, 20000)$. The best out of 3 runs, 1 loop is 2.53 seconds.
2. a $(300, 300)$ sparse matrix with 500 non-zero elements (full rank), dot another $(300, 300)$ sparse matrix with 600 non-zero elements (full rank). The best out of 3 runs, 1 loop is 11.5 seconds.
3. a $(10000, 10000)$ sparse matrix (full rank) dot a $(10000, 10000)$ sparse matrix (full rank). Unable to finish in 2 minutes.

2.2 Cython

Cython converts python code to C code and gets pre-compiled (gcc on my machine) and runs on lower C level to have a speed advantage. We write the above python code in cython:

```
%%cython
cimport cython
import numpy as np
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef dot_cython(double [:, :] A, double [:, :] B):
    cdef int A_r = A.shape[0]
    cdef int A_c = A.shape[1]
    cdef int B_c = B.shape[1]
    cdef int i,j,k
    cdef double [:, :] out = np.zeros((A_r, B_c), dtype = np.float64)

    for i in xrange(A_r):
        for j in xrange(B_c):
            for k in xrange(A_c):
                out[i,j] += A[i,k]*B[k,j]

    return np.asarray(out)
```

Still the same algorithm, but this time in cython. We test the same cases:

1. Dense matrices, the best out of 3 runs, 100 loop is 7.51 milliseconds per loop.
2. Sparse matrices, the best out of 3 runs, 100 loop is 33.9 milliseconds per loop.
3. Large sparse matrix. Unable to finish in 2 minutes.

The speed gain is huge - more than 300 times comparing to non-Cython naive implementation.

2.3 Cython Parallel

And next we notice we can release the Global Interpreter Lock (GIL) as the calculations for each element are independent and therefore can be parallized. Note that if we use the cython inplementation above and we

want to avoid using atomic add, we need to move the inner loop for k to the outside. Then the loops for i and j can be parallelized using `prange` with `nogil`. The code follows:

```
%%cython
cimport cython
from cython.parallel import prange
import numpy as np
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef dot_cython_parallel(double [:, :] A, double [:, :] B):
    cdef int A_r = A.shape[0]
    cdef int A_c = A.shape[1]
    cdef int B_c = B.shape[1]
    cdef int i,j,k
    cdef double [:, :] out = np.zeros((A_r, B_c), dtype = np.float64)

    for k in xrange(A_c):
        for i in prange(A_r, nogil=True):
            for j in prange(B_c):
                out[i,j] += A[i,k]*B[k,j]

    return np.asarray(out)
```

The test results:

1. Dense matrices, the best out of 3 runs, 100 loop is 5.27 milliseconds per loop.
2. Sparse matrices, the best out of 3 runs, 100 loop is 25 milliseconds per loop.
3. Large sparse matrix. Unable to finish in 2 minutes.

Putting the calculation in parallel speeds up the calculation by about 1.5 times.

2.4 NumPy

We then use the NumPy module, which supports efficient linear algebra computations, in hope that it can outperform our cython code. This simple code is shown below, our discussion follows in the next section:

```
def dot_numpy(A_np, B_np):
    return A_np.dot(B_np)
```

The test results:

1. Dense matrices, the best out of 3 runs, 1000 loop is 0.213 milliseconds per loop.
2. Sparse matrices, the best out of 3 runs, 1000 loop is 0.934 milliseconds per loop.
3. Large sparse matrix, 1 run, 28.2 s.

This `numpy` implementation is much faster (more than 20 times) than Cython parallel implementaion.

2.5 Scipy.sparse

Scipy sparse is specially designed data structure to store high-dimension sparse matrix. We use row-based linked list sparse matrix (`spicy.sparse.lil_matrix`) to store the matrix and calculate the dot product as:

```
A_sp = scipy.sparse.lil_matrix(A_np)
B_sp = scipy.sparse.lil_matrix(B_np)
def dot_scipy(A_sp, B_sp):
    return A_sp.dot(S_sp)
```

We expect `scipy.sparse` will perform better in the sparse matrix cases. The test results:

1. Dense matrices, the best out of 3 runs, 100 loop is 21.7 milliseconds per loop.
2. Sparse matrices, the best out of 3 runs, 1000 loop is 0.734 milliseconds per loop.
3. Large sparse matrix, the best out of 3 runs, 100 loop is 10.9 milliseconds per loop.

As expected, `scipy.sparse` is not optimum for dense matrix due to the linked list structure it uses - it is more than 100 time slower than `numpy`. However, for sparse matrices, especially very high dimension matrix, it can gain even more than 2000 times boost in speed.

3. Performance

We list the performance by elementwise, cython, cython-parallel, numpy and scipy.sparse on dense, sparse and large sparse matrices in the following table.

1e-6 s	Dense	Sparse	Sparse (large)
Elementwise	2,530,000	11,500,000	-
Cython	7,510	33,900	-
Cython-parallel	5,270	25,000	-
Numpy	213	934	28,200,000
Scipy.sparse	21,700	734	10,900

From the comparison above, we can see that `numpy` indeed speeds up matrix multiplication by significant amount. On the other hand, if we know the matrices are sparse, `spicy.sparse` gives us better performance as a result of using a more tailored data structure.

Numpy matrix multiplication is faster due to the following reasons:

1. **Data storage**: One crucial feature in Numpy is the implementation of arrays (or ndarrays). Numpy stores vectors and matrices in a more compact and efficient way than regular python lists, which are essentially general-purpose containers. In addition to the actual data, a Numpy array contains metadata specifying shape and data type, while python lists are flexible and types of the elements can be inhomogeneous. For example:
   `numpy.array(['a',1])` converts the integer to string while python list `['a',1]` holds one string object and one integer.
   Python generic lists are also dynamic and therefore might become scatterly stored in the memory. The compact and homogeneous Numpy arrays are moved efficiently from RAM to CPU, allowing for faster access.
   NumPy also takes advantage of vectorized instructions on modern CPUs, like Intel's SSE and AVX.
2. **Data access**: As Numpy arrays are stored in contiguous memory block of fixed size. Allocation of such Numpy arrays are done in C/Fortran and are typed memoryviews that allow efficient access to memory buffer. This avoids loading python interpreter overhead without copying data around.
3. **Optimized vector calculation**: Many Numpy operations are written efficiently in lower-level languages such as C and carried out in faster C loops (avoiding typechecks and other bookkeeping).

Moreover, numpy's matrix calculations are linked to highly optimized linear algebra libraries like *BLAS* and *LAPACK*.

Matrix multiplication, in particular, is written in C code. From `core/ src/ multiarray/ arraytypes.c.src`, there are multiple versions of dot product calculation, the simplest one without BLAS optimization (on vectors):

```
static void
@name@_dot(char *ip1, npy_intp is1, char *ip2, npy_intp is2, char *op, npy_intp n,
           void *NPY_UNUSED(ignore))
{
   @out@ tmp = (@out@)0;
   npy_intp i;

   for (i = 0; i < n; i++, ip1 += is1, ip2 += is2) {
       tmp += (@out@)(*((@type@ *)ip1)) *
              (@out@)(*((@type@ *)ip2));
   }
   *((@type@ *)op) = (@type@) tmp;
}
```

is the same as our elementwise implementation, but in C. The actual matrix multiplication uses BLAS divides the matrix and find out whether each row or column is contiguous, and try to iterate over the contiguous part first. Moreover, in ATLAS it uses Coppersmith–Winograd algorithm to multiply two $(n \times n)$ matrices, that optimizes in $O(n^{2.375477})$ time, comparing to our elementwise $O(n^3)$.

Sparse matrix multiplication in Scipy takes the advantage in storing data in a more efficient structure. A linked list of data indexed by row and column serves better when most matrix elements are zero.

Parallel matrix multiplication is another way to boost up the speed. Through divide and conquer and by calculating the result of each block will result an algorithm that runs in $O(\log^2 n)$ time in parallel.

References:

https://en.wikipedia.org/wiki/Matrix_multiplication
https://en.wikipedia.org/wiki/NumPy
https://github.com/numpy/numpy/blob/master/numpy/core/src/multiarray/arraytypes.c.src
http://docs.scipy.org/doc/numpy-1.10.1/user/whatisnumpy.html
http://kilon.blogspot.com/2010/02/cython-probably-best-programming.html
http://stackoverflow.com/questions/10442365/why-is-matrix-multiplication-faster-with-numpy-than-with-ctypes-in-python
http://ipython-books.github.io/featured-01/
https://en.wikipedia.org/wiki/Coppersmith%E2%80%93Winograd_algorithm