Implementing matrix operations for a subclass of block matrices

Siddharth Bhat
<siddu.druid@gmail.com>
github.com/bollu

August 16, 2018

1 Introduction

We study the problem of implementing specialized operations for matrix multiplication and matrix inversion on a subclass of matrices. The original problem statement is available here (click for link).

These matrices are parametrized by two variables, henceforth denoted by S, and D. Such matrices live in the space of $\mathbb{R}^{BD \times BD}$. They consist of non-overlapping diagonal blocks of size $D \times D$, arranged in a grid of S^2 such blocks. An example of such a matrix is below:

$$\begin{bmatrix} D_{11} & D_{12} & \dots & D_{1S} \\ \dots & \dots & \dots & \dots \\ D_{S1} & D_{S2} & \dots & D_{SS} \end{bmatrix}$$

This parametrization will be dubbed the **character** (S, D) of the matrix. S for "size", and D for "diagonal".

If we set $D = 1, S = *$, we regain the usual matrices that we use, of dimension $S \times S$. Similarly, if we set $D = *, S = 1$, we regain diagonal matrices of dimension $D \times D$. These two extreme cases will be used repeatedly to sanity check our formulas, and make for useful examples to look back at.

After all, to quote Paul Halmos, - "...the source of all great mathematics is the special case, the concrete example. It is frequent in mathematics that every instance of a concept of seemingly great generality is in essence the same as a small and concrete special case."

2 Matrix representation

Note that we only need to store the diagonal elements of all the blocks of any given matrix. Therefore, in our representation, we choose to save the diagonal elements per-block, giving us a representation of the matrix as a collection of $S \times S \times D$ numbers, which are indexed as: $\langle$`row block index, column block index, diagonal index`$\rangle = \langle$`r, c, d`$\rangle$

The amount of space required per block will be $\mathcal{O}(D)$. Since the total number of blocks is S^2, the total space complexity of this matrix will be $\mathcal{O}(S^2 D)$. This is better than storing an entire $S \times D$-dimensional matrix which will work out to $\mathcal{O}((S \times D)^2) = \mathcal{O}(S^2 D^2)$. We save space by a factor of D.

3 Matrix multiplication

3.1 Matrices that have the same (S, D) character

Here, we consider multiplying two matrices which have identical D, S. This will be used as a stepping stone in the theory of general matrix multiplication. This is also more faithful to how I understood the problem, so I choose to perform the exposition along a similar path.

3.1.1 A quick aside: Block multiplication

When we are multiplying matrices, we can multiply them as blocks - that is, this formula holds:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Where A_{ij}, B_{ij} are blocks of the original matrix. This also generalizes to any number of blocks we wish to choose. The rough proof is that to compute an element of the output matrix, c_{ij}, we compute it as $c_{ij} = \sum_k a_{ik} b_{kj}$. Thinking computationally, the computation of matrix C only ever reads from A and B. Thus, we can perform the computation of elements of C in any order we want, and computing per-block is a perfectly valid choice.

So now, if we try to analyze how to compute C_{11}, we realize the that the computation of elements of C_{11} can be factored as shown above. A similar argument holds for all blocks of C.

3.1.2 Exploiting block multiplication

Now that we know that block multiplication is possible, an immediate solution arises — we already have blocks of size $D \times D$, which are extremely easy to multiply (since they are diagonal matrices). Hence, we block the two input matrices A and B into their diagonal blocks (they are both of the same character (S, D)).

This leads us to the algorithm (all displayed code will be in python code):

```python
def matmulDiagEqCharacter(m1, m2):
    (S1, D1) = matdiagdim(m1)
    (S2, D2) = matdiagdim(m2)

    # sanity check inputs
    assert S1 == S2 and D1 == D2

    # dimensions of the output matrix
    SNEW = S1
    DNEW = D1
    out = zeromat(S, D)

    for i in range(SNEW):
        for j in range(SNEW):
            for k in range (SNEW):
                for d in range(DNEW):
                    out[i][j][d] += m1[i][k][d] * m2[k][j][d]

    return out
```

Time Complexity: From the `for`-loops in the code, the time complexity of this algorithm works out to $\mathcal{O}(S^3D)$, which in comparison to the naive $\mathcal{O}((S \times D)^3) = \mathcal{O}(S^3D^3)$ saves us a factor of D^2.

3.2 Matrices with different (S, D) character

Let us consider trying to multiply matrices of different (S, D) characters. Let A have character $(S1, D1)$, and B be $(S2, D2)$. Since these are square matrices, we can arrive at the condition that $S1 \times D1 = S2 \times D2$. However, once we have analyzed this, we need some insight: Will the resultant always be a (S', D') matrix? If so, why? And how do we compute the resultant (S', D') values?

Let's try some examples. We already know that if $S1 = S2, D1 = D2$, then $S1' = S1 = S2, D' = D1 = D2$.

The other example we can take is that of $(S1, D1) = (1, N)$ and $(S2, D2) = (N, 1)$. In this case, A is a diagonal matrix, and B is a "normal" matrix. We know the result is another normal matrix, so $(S' = N, D' = 1)$.

We tentatively conjecture that something along the lines of `gcd` or `lcm` should be involved in this process, since we need to find a new blocking size D', which would need to properly divide $D1$, $D2$ intuitively for our arguments to carry over.

I ran some experiments at this point and experimentally verified that $D' = gcd(D1, D2)$ is the correct relation. This fits with our previous examples as well. Once I had this, I began looking for an explanation. Note that we can block the original matrix A into blocks of size D', since

D' divides $D1$. Similarly, we can block B as well into blocks of size D', since D' also divides $D2$.

This is a legal blocking, since the original blocks $D1$ get subdivided into blocks D' which **still only have elements on the diagonal**. Similarly, the "empty" blocks that are created from $D1$ in D' are still legal blocks, just with diagonal $\langle 0\ 0\ \ldots\ 0 \rangle$.

Add image example

We have now reduced the original problem of matrix multiplication into the original block matrix multiplication of matrices of the same character, (S', D').

The code is a little annoying due to coordinate system conversions from the new coordinate system of (S', D') to $(S1, D1)$ and $(S2, D2)$ to be able to index into the old matrix. This leads to some verbosity in error checking. However, I prefer to reproduce the correct code, than incorrect-but-deceivingly-simple code.

```
# (m1, m2: [S][S][d])
def matmulDiagNonEqChunking(m1, m2):
    (S1, D1) = matdiagdim(m1)
    (S2, D2) = matdiagdim(m2)

    assert(S1 * D1 == S2 * D2)

    DNEW = gcd(D1, D2)
    SNEW = S1 * D1 // DNEW

    out = [[[0 for _ in range(DNEW)] for _ in range (SNEW)] for _ in range (SNEW)]

    for i in range(SNEW):
        for j in range(SNEW):
            for k in range (SNEW):
                for d in range(DNEW):
                    maybeOld = newSDCoordToOld(SNEW, DNEW, S1, D1, i, k, d)

                    m1val = None
                    if maybeOld is not None:
                        (iold, kold, dold) = maybeOld
                        m1val = m1[iold][kold][dold]
                    else:
                        m1val = 0
                    assert (m1val is not None)

                    m2val = None
                    maybeOld = newSDCoordToOld(SNEW, DNEW, S2, D2, k, j, d)
                    if maybeOld is not None:
                        (kold, jold, dold) = maybeOld
                        m2val = m2[kold][jold][dold]
                    else:
                        m2val = 0
                    assert (m2val is not None)



                    out[i][j][d] += m1val * m2val

    return out
```

This uses an auxiliary function, `newSDCoordToOld` used to convert from the (S, D) coordinates of the new matrix to the (S, D) coordinates of the old matrices. The implementation is straightforward, as is listed below:

```
def newSDCoordToOld(SNEW, DNEW, SOLD, DOLD, isnew, jsnew, dnew):
    """ return an (isold, jsold, dold) pair in the old coordinate system from
    a point in the new coordinate system (isnew, jsnew, dnew)

    (SNEW, DNEW) are the new sizes
    (SOLD, DOLD) are the old sizes

    returns:
        (isold, jsold, dold) if it lies on the diagonal of the
        old system. **None otherwise**
    """
    # shape preservation
    assert (SNEW * DNEW == SOLD * DOLD)
    # new is smaller than old
    assert (DNEW <= DOLD)
    # new neatly partitions old
    assert (DOLD % DNEW == 0)

    i = isnew * DNEW + dnew
    j = jsnew * DNEW + dnew

    isold = i // DOLD
    jsold = j // DOLD

    idold = i % DOLD
    jdold = j % DOLD

    # if idold == jdold, then it's on the diagonal of the old system,
    # otherwise it isn't, and we should consider it as 0
    # eg: SNEW = 6, DNEW = 1, SOLD = 2, DOLD = 3
    # inew = 0, jnew = 1, dnew = 0

    if idold == jdold:
        return (isold, jsold, idold)
    else:
        return None
```

Time Complexity: We know that $D' = gcd(D1, D2)$. We also know that $S' \times D' = S1 \times D1$. We can therefore derive that $S' = (S1 \times D1)/D' = (S1 \times D1)/gcd(D1, D2)$.

The time complexity works out to $\mathcal{O}(S'^3 D')$ from the previous result.

We can sanity check it by plugging in $(S1 = 1, D1 = N), (S2 = N, D2 = 1)$, which gives us $(D' = 1, S' = N)$, that is, multiplying a diagonal matrix with a full matrix, which gives us $\mathcal{O}(N^3)$. **This seems sub-optimal, because in this situation, we should be able to do this is $\mathcal{O}(N^2)$.** This shows that there **should be some way to gain another factor of D in this case.** I need to think about this more carefully to identify where performance is lost.

4 Matrix inversion

We now look at how to perform matrix inversion. Once again, we try to observe a small system and generalize. Consider a matrix of the character $(S = 2, D = 2)$.

$$\left[\begin{array}{cc|cc} a & 0 & p & 0 \\ 0 & b & 0 & q \\ \hline c & 0 & r & 0 \\ 0 & d & 0 & s \end{array}\right]$$

On making this matrix act on an arbitrary vector, we receive:

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} \left[\begin{array}{cc|cc} a & 0 & p & 0 \\ 0 & b & 0 & q \\ \hline c & 0 & r & 0 \\ 0 & d & 0 & s \end{array}\right] = \begin{bmatrix} a\alpha + p\gamma \\ b\beta + q\delta \\ c\alpha + r\gamma \\ d\beta + s\delta \end{bmatrix}$$

Notice the action of the linear transform on the vector - in essence, the first and third coordinates of the vector are modified by the first and third rows of the matrix, and similarly for the second and fourth.

Hence, we can factor the matrix into two matrices as follows:

$$\begin{bmatrix} \alpha \\ \gamma \end{bmatrix} \begin{bmatrix} a & p \\ c & r \end{bmatrix} = \begin{bmatrix} a\alpha + p\gamma \\ c\alpha + r\gamma \end{bmatrix}$$

$$\begin{bmatrix} \beta \\ \delta \end{bmatrix} \begin{bmatrix} b & q \\ d & s \end{bmatrix} = \begin{bmatrix} b\beta + q\delta \\ d\beta + s\delta \end{bmatrix}$$

This suggests that we can solve smaller sets of independent linear equations, whose solutions we can combine to solve the larger equation. We need to solve the independent linear equations, and then combine them together by ”scattering” the solutions correctly. This is implemented in `matrix.h:invDiagMatrix`, along with an implementation of the Gauss-Jordan algorithm (`matrix.h:invRawMatrixOurs`) to perform the actual inversion of the smaller system of equations.

```
def invDiagMatrix(m):
"""invert an (S, D) matrix. If the matrix is non-invertible, returns None"""
    (S, D) = matdiagdim(m)
    out = zeroDiagMatrix()

    # for every D, we get BxB solutions to solve
    for d in range(D):
        # subproblem is a SxS matrix
        subproblem = matnormal(S, S)

        # gather subproblem
        for i in range(D):
            for j in range(D):
                subproblem[i][j] = m.blocks[i][j][d];

        subsoln = invMatrix(subproblem)
        # system was non-invertible
        if (subsoln is None): return None

        # scatter subsolution
        for i in range(D):
            for j in range(D):
                out[i][j][d] = subsoln[i][j];

    # return the final collected system
    return out
```

Time Complexity: The time complexity of this is $\mathcal{O}((\text{invert } S \times S \text{ matrix}) \times D)$. Since I implemented matrix inversion using Gauss-Jordan, this works out to $\mathcal{O}(S^3 \times D)$. This in contrast to the naive $\mathcal{O}(\text{invert } SD \times SD \text{ matrix}) = \mathcal{O}(S^3 D^3)$ saves us a factor of D^2.

Once again, as a quick sanity check, plugging in $D = 1$ recovers us $\mathcal{O}(S^3)$, which is what we would expect to invert a general matrix. And similarly, plugging in $S = 1$ lets us recover $\mathcal{O}(D)$, which is what we would hope for to invert a diagonal matrix.

5 Experimental Evaluation

5.1 Implementation

The implementation consists of a C++ header-only library, `matrix.h`, which depends on `CML`, a C++ math library for a reference implementation to verify correctness against.

Our C++ library parametrizes the type of the (S, D) matrices over the entries of the matrix into an abstract type T (well, we require T to be a ring but this is something that cannot be expressed in C++ at the moment. Once C++ receives Concepts, we can expressly ask that T has a ring structure).

The library also encodes the S, D values into the **type of the matrix** to allow for better type checking and performance efficiency (the data structures get allocated with the correct sizes on the stack, instead of having to dynamically grow). This style of API provides much more type safety than the untyped APIs present in Python.

There is also another partial Python implementation, present in `python-implementation.py` which was used to independently verify the algorithms present here, mostly because it makes for a better pseudo-code language.

Listing 1 `matrix.h` API

```
// matrix.h API

// D = number of dimensions per diagonal
// S = size / number of blocks
template<int D, int S, typename T>
struct DiagMatrix {
    using Diag = std::array<T, D>;
    std::array<std::array<Diag, S>, S> blocks;
};


// An NxN matrix, which is used internally.
template<int N, typename T>
using RawMatrix = std::array< std::array<T, N>, N>;


// Note the use of compile-type constexpr to automatically evaluate
// the size of the resultant matrix. This requires C++17 I believe
// for the use of `constexpr`
template<int D1, int S1, int D2, int S2, typename T>
DiagMatrix<gcd(D1, D2), S1 * D1 / gcd(D1, D2), T>
       mulDiagMatrixGeneral(DiagMatrix<D1, S1, T> m1, DiagMatrix<D2, S2, T> m2);

// Multiply two matrices of the same (S, D)
template<int D, int S, typename T>
DiagMatrix<D, S, T> mulDiagMatrixSameSize(DiagMatrix<D, S, T> m1,
                                          DiagMatrix<D, S, T> m2);

// Returns the inverse of a matrix. Sets the success to false
// if the inverse does not exist.
template<int D, int S, typename T>
DiagMatrix<D, S, T> invDiagMatrix(DiagMatrix<D, S, T> m, bool &success);
```

5.2 Correctness

We verify the correctness of the implementation by running the implementation against a reference implementation which uses the naive version of matrix multiplication and matrix inversion. Since the C++ library that was being used for matrix math (`CML`) was buggy, another implementation was constructed in python (`python-implementation.py`).

Both of these implementation validate the theory as laid out in the document.

5.3 Performance (WIP)

I have not benchmarked this yet, but I should when I get some downtime. I actually find it interesting, to see if asymptotics will win over hand-optimized `LAPACK` kernels. I feel for small-to-medium problem sizes, `LAPACK` will outperform the given asymptotically better implementation. It would be interesting to have results.

The other thing that I am interested in this space is running Polly (LLVM's loop and data locality optimizer — link here) on the hand-written implementation, to see how much of a performance win we get. In theory, Polly should perform additional tiling and other loop transformations to eke out greater performance. As a Polly developer, it would be interesting to see if this succeeds or not.

6 Closing thoughts

I didn't try very hard to push the boundaries of what's possible, in terms of matrix multiplication. I believe it's totally possible, by using some kind of Strassen-like strategy combined with the shown optimization once the matrices get small enough to fit in cache.

I've also *completely* ignored things that matter in the real world like memory latency and cache sizes — for example, coming up with a good matmul algorithm is non-trivial. One of the best matrix multiplication algorithms I know of, and is used in the BLIS kernel the Goto algorithm depends on non-trivial observations about the cache hierarchy, and a careful choice of blocking to make sure data can be streamed effectively by the processor, and to ensure that data lies in the cache hierarchy. I choose to ignore the real world to focus on the asymptotics here, but this is a very important part of writing real world code.

In general, I enjoyed thinking about the assignment. I'm not fully happy with my proof of matrix multiplication in the general case, and I feel I'm losing some asymptotic performance I should be able to gain, so I wish to think a little longer about this.

I also wanted a nicer proof of the general matrix multiplication, because it feels like it does not come from first principles, but I suppose sometimes one has to settle.

I do strongly resonate with Hardy's opinion, "Beauty is the first test: there is no permanent place in the world for ugly mathematics."