Notebook

January 18, 2015

1 Blocked and unblocked Cholesky factorization

This notebook implements and tests the performance of unblocked and blocked algorithms for *Cholesky factorization*,

$$A = LL^H,$$

where A is Hermitian positive-definite and L is lower-triangular with a positive diagonal (however, please note that the concept generalizes to positive semidefinite matrices). The simplest means of generating a Hermitian positive-definite matrix is to form $A = BB^H$, where B is any non-singular square matrix. We instead form $A = BB^H + I$, with B set to a normally-distributed matrix via the command `randn(n,n)`.

In order to test the accuracy of our factorizations of A, we will solve the linear system

$$Ax = b$$

and test the *relative residual*,

$$\|b - Ax\|_2 / \|b\|_2$$

,

as well as check the relative error in the factorization itself,

$$\|A - LL^H\|_2 / \|A\|_2.$$

For this reason, we will save $ | A |*2 $ and $ | b |2 $ for future use.

```
In [1]: n = 1000;

        # Build a random SPD matrix
        B = randn(n,n);
        AOrig = B*B' + eye(n,n);
        ANorm = norm(AOrig);
        println("|| A ||_2 = $ANorm")

        # Build a random right-hand sidel
        b = randn(n);
        bNorm = norm(b);
        println("|| b ||_2 = $bNorm")
```

```
|| A ||_2 = 4008.613535000883
|| b ||_2 = 31.506582883514458
```

The first algorithm that we implement is an *unblocked right-looking* Cholesky factorization, which can quickly be derived from partitioning the expression $ A = L L^H $ in a manner which exposes the top-left entry of A and the lower-triangular matrix L, say

$$\begin{pmatrix} \alpha_{1,1} & a_{2,1}^H \\ a_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} \lambda_{1,1} & 0 \\ l_{2,1} & L_{2,2} \end{pmatrix} \begin{pmatrix} \lambda_{1,1} & l_{2,1}^H \\ 0 & L_{2,2}^H \end{pmatrix} = \begin{pmatrix} \lambda_{1,1}^2 & \lambda_{1,1} l_{2,1}^H \\ \lambda_{1,1} l_{2,1} & l_{2,1} l_{2,1}^H + L_{2,2} L_{2,2}^H \end{pmatrix},$$

where we have made use of the fact that L is required to have a positive diagonal (and therefore $\lambda_{1,1}$ equals its conjugate).

The unblocked right-looking algorithm for Cholesky factorization can now be derived in a straight-forward manner by equating the exposed quadrants of A with the corresponding quadrants of the product LL^H. In particular, the common convention, such as in LAPACK's zpotf2, is to overwrite the lower triangle of A with the Cholesky factor L (leaving the upper-triangle of A unaccessed and unchanged).

It should then be clear from $\alpha_{1,1} = \lambda_{1,1}^2$ that the first step of the unblocked algorithm should be to overwrite $\alpha_{1,1} := \sqrt{\alpha_{1,1}} = \lambda_{1,1}$, and that the relation $a_{2,1} = \lambda_{1,1} l_{2,1}$ then implies that $a_{2,1}$ can be overwritten with $l_{2,1}$ via the operation $a_{2,1} := a_{2,1}/\lambda_{1,1}$. The original problem can then be reduced to an equivalent subproblem via overwriting $A_{2,2}$ with the Schur complement

$$A_{2,2} := A_{2,2} - l_{2,1} l_{2,1}^H = L_{2,2} L_{2,2}^H.$$

This algorithm is implemented in `CholUnb` below.

In [2]:
```
function CholUnb(A)
    m,n = size(A);
    for k=1:n
        A[k,k] = sqrt(A[k,k]);
        A[k+1:end,k] /= A[k,k];
        A[k+1:end,k+1:end] -= A[k+1:end,k]*A[k+1:end,k]';
    end
end
```

Out[2]:
```
CholUnb (generic function with 1 method)
```

We now test the performance and relative accuracy of the unblocked algorithm via a simple usage of Julia's `tic`, `toq`, and `norm` routines. We can easily convert the timing into a rough measurement of the number of floating-point computations performed per second by recognizing that Cholesky factorization of a real $n \times n$ matrix involves roughly $\frac{1}{3}n^3$ floating-point operations.

In [3]:
```
# Time the unblocked algorithm
A = copy(AOrig);
tic();
CholUnb(A);
unbTime=toq();
GFlops=(1.*n*n*n)/(3.*unbTime*1.e9);
println("unblocked time: $unbTime seconds")
println("unblocked GFlops: $GFlops")

# Check the error of the blocked algorithm
L = tril(A);
decompError=norm(AOrig - L*L');
relDecompError=decompError/ANorm;
println("|| A - L L' ||             = $decompError")
println("|| A - L L' || / || A ||   = $relDecompError")

# A = L L' implies inv(A) b = inv(L L') b = inv(L') inv(L) b
x = L'\(L\b);
residError=norm(b-AOrig*x);
relResidError=residError/bNorm;
println("|| b - A x ||_2             = $residError")
println("|| b - A x ||_2 / || b ||_2 = $relResidError")
```

```
unblocked time: 5.655188571 seconds
unblocked GFlops: 0.05894292102701545
|| A - L L' ||              = 3.984683177125967e-12
|| A - L L' || / || A ||    = 9.940302656601914e-16
|| b - A x ||_2              = 6.516412719395838e-12
|| b - A x ||_2 / || b ||_2 = 2.0682702226033827e-13
```

As of this writing, a typical core of a laptop can be expected to have a theoretical peak performance of roughly 10 billion floating-point operations per second (10 GFlops), and our timings reflect that this is nowhere near achieved by the unblocked algorithm. The inefficiency is easily explained by the fact that the symmetric rank-one update $A_{2,2} := A_{2,2} - l_{2,1} l_{2,1}^H$ asymptotically dominates the work in the algorithm but does not involve any significant data reuse.

We can lift the previous algorithm into a *blocked* algorithm which spends the vast majority of its effort computing a related update of the form

$$A_{2,2} := A_{2,2} - L_{2,1} L_{2,1}^H,$$

where $L_{2,1}$ is a tall and skinny matrix instead of a single column vector. Furthermore, the derivation of the algorithm directly mirrors that of the unblocked algorithm (indeed, it uses the unblocked algorithm as a component!) and begins by exposing a small, square $O(1) \times O(1)$ submatrix in the top-left corners of A and L rather than a single entry:

$$\begin{pmatrix} A_{1,1} & A_{2,1}^H \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{pmatrix} \begin{pmatrix} L_{1,1}^H & L_{2,1}^H \\ 0 & L_{2,2}^H \end{pmatrix} = \begin{pmatrix} L_{1,1} L_{1,1}^H & L_{1,1} L_{2,1}^H \\ L_{2,1} L_{1,1}^H & L_{2,1} L_{2,1}^H + L_{2,2} L_{2,2}^H \end{pmatrix}.$$

The exact dimension of $A_{1,1}$ and $L_{1,1}$ should be chosen based upon architectural considerations, such as L1 and L2 cache sizes, but values near 100 are common when the dimension of A is at least a few thousand.

We can now derive a *blocked* right-looking Cholesky factorization by again equating corresponding blocks of the partitioned matrices A and LL^H, beginning with the relation $A_{1,1} = L_{1,1} L_{1,1}^H$, which can be solved for $L_{1,1}$ via our unblocked Cholesky factorization algorithm, `CholUnb`. The equivalence $A_{2,1} = L_{2,1} L_{1,1}^H$ can be used to compute $L_{2,1}$ from $A_{2,1}$ and the newly-found $L_{1,1}$ via the BLAS routine trsm, which solves a *TR*iangular *S*ystem involving a *M*atrix. Once $L_{2,1}$ has been computed, its outer-product with itself can be subtracted from $A_{2,2}$ using an efficient kernel referred to as a *HE*rmitian *R*ank-k update (herk), which, for real matrices, is called a *SY*mmetric *R*ank-k update (syrk). Its efficiency is derived from the fact that k multiplications and adds are performed for every entry of $A_{2,2}$ that is modified, which masks the fact that memory access speeds tend to be significantly slower than the theoretical peak floating-point performance.

```
In [4]: function Chol(A,bsize)
            m,n = size(A);
            # Blocked Cholesky
            for k=1:bsize:n
                nb = min(n-k+1,bsize);
                ind1 = k:k+nb-1;
                ind2 = k+nb:n;
                A11 = sub(A,ind1,ind1);
                A21 = sub(A,ind2,ind1);
                A22 = sub(A,ind2,ind2);

                # A11 := Chol(A11) = L11
                CholUnb(A11);

                # A21 := A21 inv(L11)^H
                BLAS.trsm!('R','L','C','N',1.,A11,A21);

                # A22 := A22 - L21 L21^H
                # NOTE: 'herk' does not fall through to 'syrk' for real matrices.
```

```
            BLAS.syrk!('L','N',-1.,A21,1.,A22);
        end
    end
```

```
Out[4]: Chol (generic function with 1 method)
```

The above routine, `Chol`, repeatedly exposes diagonal blocks of dimension `bsize` (except for perhaps the last iteration), factors the small diagonal block using an unblocked algorithm, solves a sequence of triangular systems using the Cholesky factor of the diagonal block, and then performs an efficient symmetric update of the remaining bottom-right quadrant using the solutions to the triangular systems.

Two pieces of Julia-specific syntax are worth discussing: * The function sub is used to return a lightweight object which effectively points to a contiguous submatrix of A. * The "!" prefix on `BLAS.trsm` and `BLAS.syrk` denotes the fact that the input arguments `A21` and `A22`, respectively, will be modified.

It is now time to measure the performance of the blocked algorithm relative to the built-in "" operator, which would ideally call LAPACK's `dpotrf` followed by two triangular solves.

```
In [5]: # Time the blocked algorithm
        A = copy(AOrig);
        tic();
        Chol(A,96);
        blockTime=toq();
        println("blocked time: $blockTime")

        # Check the error of the blocked algorithm
        L = tril(A);
        decompError=norm(AOrig - L*L');
        relDecompError=decompError/ANorm;
        println("|| A - L L' ||             = $decompError")
        println("|| A - L L' || / || A ||   = $relDecompError")
        # A = L L' implies inv(A) b = inv(L L') b = inv(L') inv(L) b
        x = L'\(L\b);
        residError=norm(b-AOrig*x);
        relResidError=residError/bNorm;
        println("|| b - A x ||_2             = $residError")
        println("|| b - A x ||_2 / || b ||_2 = $relResidError")

        # Time and check the error from the built-in solve
        tic();
        x = AOrig\b;
        builtinTime=toq();
        println("built-in solve time: $builtinTime")
        residError=norm(b-AOrig*x);
        relResidError=residError/bNorm;
        println("|| b - A x ||_2             = $residError (backslash)")
        println("|| b - A x ||_2 / || b ||_2 = $relResidError (backslash)")
```

```
blocked time: 0.69307987
|| A - L L' ||             = 4.364014655279667e-12
|| A - L L' || / || A ||   = 1.0886593624393143e-15
|| b - A x ||_2             = 6.457948068642932e-12
|| b - A x ||_2 / || b ||_2 = 2.0497138939246873e-13
built-in solve time: 0.291754306
|| b - A x ||_2             = 6.409611260950687e-12 (backslash)
|| b - A x ||_2 / || b ||_2 = 2.0343720817481795e-13 (backslash)
```

As can be seen, without any tuning, our Julia implementation of a right-looking blocked Cholesky factorization, `Chol`, is orders of magnitude faster than the unblocked algorithm and near the performance of the equivalent LAPACK implementation which drives the backslash operator.