Notes for Math 221, Lecture 4, Sep 17, 2024

Goal: Understand the real cost (in time) of running an algorithm,
so we can design algorithms that run as fast as possible.
Traditionally we just count the number of arithmetic operations,
but in fact multiplication and addition are the cheapest operations
an algorithm performs: it can be orders of magnitude more expensive
to get that data from wherever it is stored in the computer (say main
memory) and bring it to the part of the hardware that actually does
the arithmetic.
(draw pictures of simple models of sequential and parallel computers).
For example, on a sequential computer, the main cost of a naive
algorithm would be moving data between main memory (DRAM) and on-chip
cache, and on a parallel computer, it is moving data between
processors connected over a network. As a shorthand, we call all
forms of moving data "communication", and seek to minimize it if
possible.

In the case of matrix multiplication (matmul), there is a theorem
that gives a lower bound on the amount of communication required
between DRAM and cache, assuming one can do the usual O(n^3)
operations in any order (Hong & Kung, 1981). And there is a well-known
and widely-used algorithm that attains this lower bound.
In 2011 we showed that this lower bound extends to any algorithm that
"smells like" the 3-nested loops of conventional matmul (there is a
formal definition of this, your intuition is fine for now),
essentially covering all the usual linear algebra algorithms, for
solving Ax=b, least squares, etc. (Ballard,Holtz,Schwartz,D. 2011)
It turns out that the usual algorithms for these problem cannot always
attain the lower bound, just by doing the same operations in a
different order; instead one needs new algorithms. Some of these
recently invented algorithms have given very large speedups (O(10x))
and are widely used.

All these results (lower bounds and optimal algorithms) extend to
other kinds of computer architectures, so with multiple layers of
cache (so there is communication between every pair of of adjacent
layers), and with parallel processors (so there is communication
between different processors over a network).

In this lecture we will talk about this lower bound and optimal
algorithm for matmul, in the simplest case of having DRAM and cache.
In later lectures, when we talk about more complicated linear algebra
algorithms, we will just sketch how to redesign them to minimize

communication. There is more detail about the parallel case presented in CS267.

There are many more algorithms, and computer architectures, to which these ideas could be applied, which are possible class projects.

To get started, we want a simple mathematical model of what it costs to move data, so we can analyze algorithms easily and identify the one that is fastest. So we need to define two terms:
bandwidth and latency.

First, to establish intuition, consider the time it takes to move cars on the freeway from Berkeley to Sacramento:
Bandwidth measures how many cars/hour can get from Berkeley to Sacramento:
 #cars/hour = density (#cars/mile in a lane) x velocity(miles/hour)
 x #lanes
Latency measures how long it takes for one car to get get from Berkeley to Sacramento:
 time(hours) = distance(miles) / velocity(miles/hours)
So the minimum time it takes n cars to go from Berkeley to Sacramento is when they all travel in a single "convoy," that is all as close together as possible given the density, and using all lanes:
 time(hours) = time for first car to arrive
 + time for remaining cars to arrive
 = latency + n/bandwidth

The same idea (harder to explain the physics) applies to reading bits from main memory to cache (draw picture): The data resides in main memory initially, but one can only perform arithmetic and other operations on it after moving it to the much smaller cache. The time to read or write w words of data from memory stored in contiguous locations (which we call a "message" instead of a convoy) is
 time = latency + w/bandwidth

More generally, to read or write w words stored in m separate contiguous messages costs m*latency + w/bandwidth.

Notation: We will write this cost as m*alpha + w*beta, and refer to it as the cost of "communication".
We refer to w as #words_moved and m as #messages.
We also let gamma = time per flop, so if an algorithm does f flops our estimate of the total running time will be
 time = m*alpha + w*beta + f*gamma

To reiterate our claim that communication is the most expensive operation:

(1) gamma << beta << alpha on modern machines (factors of 10s or 100s for memory, even bigger for disk or sending messages between processors)

(2) It is only getting worse over time: gamma, beta and alpha are improving year over year, gamma faster than beta faster than alpha

(Show plot of hardware speed trends).

And the same story holds for energy: the energy cost of moving data is much higher than doing arithmetic. So whether you are concerned about the battery in your cell phone dying, the O($1M) per megawatt per year it takes to run your data center, or how long your drone can stay airborne, you should minimize communication.

So when we design or choose algorithms, we need to pay attention to their communication costs.

How do we tell if we have designed the algorithm well wrt not communicating too much? If the f*gamma term in the time dominates the communication cost m*alpha + w*beta

f*gamma >= m*alpha + w*beta

then the algorithm is at most 2x slower than as though communication were free. When f*gamma >> m*alpha + w*beta, the algorithm is running at near peak arithmetic speed, and can't go faster.
When f*gamma << m*alpha + w*beta, communication costs dominate.

Notation: the Computational Intensity is
q = f/w = "flops per word_moved"
This needs to be large to go fast, since f*gamma > w*beta means
q = f/w > beta/gamma >> 1.

We will know we have done as well as possible if we can show that w and m are close to known lower bounds, that we will describe below.

But first to describe how this trend has impacted algorithms over time, we give a little history.

In the beginning, was the do-loop. This was enough for the first libraries (EISPACK, for eigenvalues of dense matrices, in the mid 1960s). People didn't worry about data motion, arithmetic was more expensive, they mostly just tried to get answer reliably, for O(n^3) flops.

BLAS-1 library (mid 1970s) - Basic Linear Algebra Subroutines
This was a standard library of 15 operations (mostly) on vectors, including:
(1) y = alpha*x + y, x and y vectors, alpha scalar ("AXPY" for short)
    Ex: innermost loop in Gaussian Elimination (GE):
      for k=i+1:n, A(j,k) = A(j,k) - A(j,i)*A(i,k)
(2) dot product
(3) 2-norm(x) = sqrt(sum_i x(i)^2)
(4) find largest entry in absolute value in a vector
    (for pivoting in GE)
The motivations for the BLAS1 at the time were:
 easing programming, readability, since these were commonly used
   functions
 robustness (eg avoid over/underflow in 2-norm(x))
 portable and efficient (if optimized on each architecture)
But there is no way to minimize communication in such simple
operations, because they do only a few flops per memory reference,
eg 2n flops on 2n data for a dot product, so with a computational
intensity of q=2n/(2n)=1. So if we implement matmul or GE or anything
else by loops calling AXPY, it will communicate a lot, the same as
the number of flops.

BLAS-2 library (mid 1980s)
This was a standard library of 25 operations (mostly) on
matrix-vector pairs:
(1) y = alpha*y + beta*A*x ("GEMV" for short),
   with lots of variations for different matrix structures
   (symmetric, banded, triangular etc)
   it allowed A^T, A^* to be used instead of A,
   and did obvious optimizations when alpha and/or beta = +1, -1 or 0
(2) A = A + alpha*x*y^T (matrix + rank-1 update) ("GER")
   Ex: It turns out the 2 innermost loops in GE can be written with
      GER (details later):
      A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i)*A(i,i+1:n)
(3) Solve T*x=b where T triangular (used by GE to solve A*x=b)
   ("TRSV")
The motivation was similar to the BLAS-1, plus more opportunities to
optimize on the vector computers of the day. But there is still not
much reduction in communication, for example, GEMV reads
n^2+2*n+2 words, writes n words, and does 2n^2+3*n flops,
for a computational intensity of about q=2.

BLAS-3 library (late 1980s)
This was a standard library of 9 operations on matrix-matrix pairs

(1) C = beta*C + alpha*A*B ("GEMM" for short)
(2) C = beta*C + alpha*A*A^T, C symmetric ("SYRK")
(3) Solve T*X=B where T triangular, X and B matrices ("TRSM")
Finally these have the potential for significant communication optimization, since for example the computational intensity q of GEMM applied to n x n matrices is
  (2n^3 flops)/(3n^2 inputs + n^2 outputs) = n/2.
But as we'll see, the straightforward 3-nested-loop version of GEMM is no better than BLAS-2 or BLAS-1, so a different implementation is required, with very large speedups possible. In fact, there is an optimal way to implement GEMM, that provably does as little communication as possible (under certain assumptions).

Note: BLAS-k does O(n^k) operations, for k=1,2,3, making the names easier to remember. These are supplied as part of the optimized math libraries on essentially all computers: use them as your building blocks! Reference implementations and documentation is at
  www.netlib.org/blas

This led the community to seek algorithms for the rest of linear algebra (solving Ax=b, least squares, computing eigenvalues, SVD, etc) that did much of their work by calling BLAS-3 routines, which would then run at high speed. This led to the LAPACK library (and its parallel version called ScaLAPACK) which is the basis of most dense linear algebra libraries provided by computer vendors, and used in many packages like Matlab.

As stated earlier, we later discovered that the communication lower bounds attained by GEMM also apply to essentially the rest of linear algebra. Next we realized that the algorithms in LAPACK and ScaLAPACK usually did not attain these lower bounds, in fact the algorithms there could do asymptotically more communication than the bounds required. This in turn set off a search for new (or previously invented but ignored) algorithms that would attain these lower bounds. This has led to a number of new algorithms, some with large speedups, and some still of just theoretical interest; the constants hidden in a O(.) analysis are important in practice!

We will talk about GEMM in more detail, and sketch results for other algorithms.

The following theorem for matrix multiplication was first proved in 1981 by Hong and Kung. The proof below is based on one by Irony, Tiskin and Toledo in 2004, which extends both to parallel algorithms, and to other 3-nested-loop-like algorithms:

Thm: Suppose one wants to multiply two nxn matrices C = A*B by doing the usual 2*n^3 multiplies and adds, on a computer with a main memory large enough to hold A, B and C, and a smaller cache of size M. Arithmetic can only be done on data stored in cache. Then a lower bound on the number of words W that need to move back and forth between main memory and cache to run the algorithm is
  Omega(n^3/sqrt(M)).

More generally, one does not need to multiply dense square matrices, the same proof works for rectangular and/or sparse matrices; the only change is that Omega(n^3/sqrt(M)) becomes Omega(#flops/sqrt(M)).

Proof Sketch: Suppose we fill up the cache with M words of data, do as many flops as possible, store the results back in main memory, and repeat until we are done. Suppose we could upper bound by G the number of flops that are possible given M words of data. Then doing G flops would cost at least 2M words moved back and forth between main memory and cache. Since we have to do 2n^3 flops altogether, we would need to repeat this at least 2n^3/G times, for a total cost of moving at least (2n^3/G)*2M words. So we need to find an upper bound G.

We turn this into a geometry problem as follows. We represent each pair of flops (or inner loop iteration)
   C(i,j) = C(i,j) + A(i,k)*B(k,j)
as a lattice point (i,j,k) in 3D space, with 1 <= i,j,k <= n, so an n x n x n cube of points. The flops we can do with M words of data is represented by some subset V of this cube of lattice points. What data is required to execute the flops represented by (i,j,k)? It is C(i,j), A(i,k), and B(k,j), which are represented by the 3 lattice points (i,j), (i,j) and (k,j) on 3 faces of the cube, i.e. the projections of (i,j,k) onto these three faces (draw picture). We know we can have at most M words of data, i.e. M projected points.

We now use a classical geometry theorem by Loomis and Whitney (1949) which says that if the set of lattice points is V whose cardinality |V| we want to bound, and its 3 projections onto the faces of the cube are
   V_C (representing entries C(i,j))
   V_A (representing entries A(i,k))
   V_B (representing entries B(k,j))
then |V| <= sqrt(|V_A| * |V_B| * |V_C|)

Since we can have at most M entries of A, B and C in cache, this means

|V| <= sqrt(M * M * M) = M^(3/2)

yielding our desired upper bound G = M^(3/2). Finally, this yields our lower bound W >= (2*n^3)/M^(3/2) * 2M = Omega(n^3/sqrt(M)) as claimed.

We have not tried to be careful with the constant factors, but just tried to give the main idea. The best (nearly attainable) lower bound is actually 2*n^3/sqrt(M) - 2*M.

The proof also gives us a big hint as to how to design an optimal algorithm: What is the shape of the V that attains its upper bound M^(3/2)? Clearly V must be a cube of side length sqrt(M). This means that we will want to break A, B and C up into square submatrices of dimension at most sqrt(M/3), read 3 submatrices into cache (the factor of 3 means we can fit 3 such submatrices simultaneously), multiply them, and write the results back to main memory:

```
... Multiply n x n matrices C = C+A*B
... Let b be block size, small enough so 3*b^2 <= M
... express C as a block matrix where C[i,j] is a b x b block
   (picture)
... ditto for A[i,k] and B[k,j]
   for i = 1 to n/b  ... number of b x b blocks in each dimension
     for j = 1 to n/b
       read C[i,j] into cache  ... b^2 words moved
       for k = 1 to n/b
         read A[i,k] and B[k,j] into cache  ... 2*b^2 words moved
           ... all 3 b x b submatrices fit in cache since 3*b^2 <= M
         C[i,j] = C[i,j] + A[i,k]*B[k,j]
           ... b x b matrix multiplication, so 3 more nested loops
           ... since everything in cache, no communication
       end for
       write C[i,j] back to main memory  ... b^2 words moved
```

Counting the total number of words moved between cache and memory, we get

(n/b)^2*b^2 = n^2   for reading C[i,j]

(n/b)^3*2*b^2 = 2*n^3/b for reading A[i,k] and B[k,j]

(n/b)^2*b^2 = n^2   for writing C[i,j]

for a total of 2*n^3/b + 2*n^2 words moved. We minimize this by making b as large as possible, which is limited by 3*b^2 <= M, i.e. b = sqrt(M/3), or the number of words moved = O(n^3/sqrt(M)), which attains the lower bound (to within a constant factor).

You might ask "What about multiplying a A*B where B has just a few columns, or just one? It's still 3 nested loops."
Yes, but if B has fewer than b columns, then we can't break it into b x b blocks. But all the above lower bounds and optimal algorithms can be extended to the case of small loop bounds; the (attainable) lower bound becomes
   Omega(max(#flops/sqrt(M), size(input) + size(output))) .

This approach to finding a communication lower bound and corresponding optimal algorithm has recently been extended to any algorithm that can be expressed as nested loops accessing arrays, with any number of loops, arrays, and subscripts, as long as the subscripts are "affine", so of the form i, i+j, 2*i-3*j+4*k-5, etc.
The lower bound is always of the form
  Omega(#loop iterations/M^e),
where e is an exponent that depends on the problem (e=1/2 for matmul).
And the optimal block shapes can be general parallelograms.
For linear algebra, we only need the special case described above.

Here is another optimal matmul algorithm, that we will find simpler to generalize to other linear algebra problems, because it is "cache oblivious", i.e. it works for any cache size M, without needing to know M:

```
   function C = RMM(A,B)  ... Recursive Matrix Multiplication
     ... assume for simplicity that A and B are square of size n
     ... assume for simplicity that n is a power of 2
     if n=1
       C = A*B  ... scalar multiplication
     else
       ... write A = [ A11  A12 ], where each Aij is n/2 by n/2
                     [ A21  A22 ]
       ... write B and C similarly
       C11 = RMM(A11,B11) + RMM(A12,B21)
       C12 = RMM(A11,B12) + RMM(A12,B22)
       C21 = RMM(A21,B11) + RMM(A22,B21)
       C22 = RMM(A21,B12) + RMM(A22,B22)
     endif
     return
```

Correctness follows by induction on n

Cost: let A(n) = #arithmetic operation on matrices of dimension n
  A(n) = 8*A(n/2)   ... 8 recursive calls

  + n^2   ... 4 additions of n/2 x n/2 matrices
and A(1) = 1 is the base case
Claim that A(n) = 2n^3 - n^2, same as the classical algorithm

Solve by changing variables from n=2^m to m:
 a(m) = 8*a(m-1) + 2^(2m),   a(0) = 1
Divide by 8^m to get
 a(m)/8^m = a(m-1)/8^(m-1) + (1/2)^m
Change variables again to b(m) = a(m)/8^m
 b(m) = b(m-1) + (1/2)^m, b(0) = a(0)/8^0 = 1
   = sum_{k=1 to m) (1/2)^k + b(0)
a simple geometric sum

Let W(n) = #words moved between slow memory and cache of size M

W(n) = 8*W(n/2) ... 8 recursive calls
   + 12(n/2)^2 ... up to 8 reads and 4 writes of n/2 x n/2 blocks
  = 8*W(n/2) + 3n^2

Oops - this looks bigger than A(n), rather than A(n)/sqrt(M).
But what is base case? It is not at W(1), rather
 W(b) = 3*b^2 if 3*b^2 <= M, i.e. b = sqrt(M/3)
Solve as before: change variables from n=2^m to m:
 w(m) = 8*w(m-1) + 3*2^(2m), w(log2 sqrt(M/3)) = M
Let mbar = log2 sqrt(M/3)
Divide by 8^m to get v(m) = w(m)/8^m
 v(m) = v(m-1) + 3*(1/2)^m,
 v(mbar) = M/8^mbar = M/(M/3)^(3/2) = 3^(3/2)/M^(1/2)
This is again a geometric sum but just down to m = mbar, not 1
 v(m) = sum_{k=mbar+1 to m} 3*(1/2)^k + v(mbar)
   <= 3* (1/2)^(mbar) + v(mbar)
   = 3/sqrt(M/3) + 3^(3/2)/sqrt(M)
   = 2*3^(3/2)/sqrt(M)
so
 W(n) = w(log2 n) = 8^(log2 n)*v(log2 n) <= 2*3^(3/2)*n^3/sqrt(M)
as desired.

We mention briefly that the lower bound extends to the parallel case
in a natural way: now the "cache" is the memory local to each
processor, that it can access quickly, and "main memory" is the
memory local to all the other processors, which is much slower to
access. We assume each processor is assigned an equal fraction of the
flops to do, 2*n^3/P, where P is the number of processors, and an
equal fraction of the memory required to store all the data, so

M = 3*n^2/P. Then the same proof as above says that for each processor to perform 2*n^3/P flops it needs to move

```
   Omega((n^3/P)/sqrt(M)) = Omega(n^2/sqrt(P)) words
```

into and out of its local memory. This lower bound is indeed attained by known algorithms, such as SUMMA, discussed in more detail in CS267.

One more topic: We can go asymptotically faster than n^3, not just to multiply matrices, but any linear algebra problem. There are many such algorithms, we discuss only the first one, which so far is the most practical one:

Strassen (1967): Matrix multiply is possible in

```
  O(n^(log2 7)) ~ O(n^2.81) operations.
```

The algorithm is recursive, based on remarkable identities for multiplying 2 x 2 matrices:

```
Write A = [ A11 A12 ], each block of size n/2 x n/2, same for B and C
       [ A21 A22 ]
   P1 = (A12-A22)*(B21+B22)
   P2 = (A11+A22)*(B11+B22)
   P3 = (A11-A21)*(B11+B12)
   P4 = (A11+A12)*B22
   P5 = A11*(B12-B22)
   P6 = A22*(B21-B11)
   P7 = (A21+A22)*B11
   C11 = P1+P2-P4+P6
   C12 = P4+P5
   C21 = P6+P7
   C22 = P2-P3+P5-P7
```

This changes naturally into a recursive algorithm:

```
    function C = Strassen(A,B)
    ... assume for simplicity that A and B are square of size n
    ... assume for simplicity that n is a power of 2
    if n=1
      C = A*B  ... scalar multiplication
    else
      P1 = Strassen(A12-A22,B21+B22) ... and 6 more similar lines
      C11 = P1+P2-P4+P6 ... and 3 more similar lines
    end
```

```
A(n) = #arithmetic operations
   = 7*A(n/2)      ... 7 recursive calls
    + 18*(n/2)^2  ... 18 additions of n/2 x n/2 matrices
```

We solve as before: change variables from n=2^m to m, A(n) to a(m):
  a(m) = 7*a(m-1) + (9/2)*2^(2m),   a(0)=A(1)=1
divide by 7^m, change again to b(m) = a(m)/7^m
  b(m) = b(m-1) + (9/2)*(4/7)^m, b(0)=a(0)=1
      = O(1) ... a convergent geometric sequence
so
  A(n) = a(log2 n) = b(log2 n)*7^(log2 n) = O(7^(log2 n))
      = O(n^(log2 7))

What about #words moved? Again get a similar recurrence:
  W(n) = 7*W(n/2) + O(n^2),
and the base case W(nbar) = O(nbar^2) when nbar^2 = O(M),
i.e. the whole problem fits in cache.
The solution is O(n^x/M^(x/2 -1)) where x = log2(7).
Note that plugging in x=3 gives the lower bound for standard matmul.

Again, there is a theorem (with a very different proof), showing that
this algorithm is optimal:
Thm (D., Ballard, Holtz, Schwartz, 2010): Omega(n^x/M^(x/2 - 1)) is a
lower bound on #words_moved for Strassen and algorithms "enough like"
Strassen, where x = log2 7 for Strassen, and whatever it is for
"Strassen-like" methods.

This result was generalized to even more "Strassen-like" methods in
a PhD Thesis (Scott, 2015).

Strassen is not (yet) much used in practice, but it can pay off for
n not too large (a few hundred).
Its error analysis slightly worse than usual algorithm:
   Usual Matmul (Q1.10): |fl(A*B) - (A*B)| <= n*eps*|A|*|B|
   Strassen ||fl(A*B) - (A*B)|| <= O(eps)*||A||*||B||
which is good enough for lots of purposes.
But when can Strassen's error bound be much worse?
(suppose one row of A, or one column of B, is very tiny).
Strassen is also not allowed to be used in the "LINPACK Benchmark"
(www.top500.org) which ranks machines by
   speed = (2/3)*n^3 / time_for_GEPP(n).
and if you use Strassen (we'll see how) in GEPP,
it does << (2/3)n^3 flops so the computed "speed" could exceed
actually peak machine speed. As a result, vendors stopped optimizing
Strassen, even though it's a good idea!

Current world's record: Williams, Xu, Xu, Zhou (2024): O(n^2.371552)

but it is impractical: would need huge n to pay off.
All such algorithms were shown numerically stable (in an asymptotic norm-sense) by D., Dumitriu, Holtz, Kleinberg (2007).
Given any fast matmul running in O(n^w) operations, it is also possible to do the rest of linear algebra (solve Ax=b, least squares, eigenproblems, SVD) in O(n^(w+eta)) for any eta>0, and to do so numerically stably, shown by D., Dumitriu, Holtz (2007)

A similar Strassen-like trick can be used to make complex matrix multiplication a constant factor cheaper than you think. Normally, to multiply two complex numbers or matrices, you use the formula

(A+i*B)*(C+i*D) = (A*C-B*D)+i*(A*D+B*C)

costing 4 real multiplications and 2 real additions.
Here is another way:

T1 = A*C
T2 = B*D
T3 = (A+B)*(C+D)

Then it turns out that

(A+i*B)*(C+i*D) = (T1-T2) + i*(T3-T1-T2)

for a cost of 3 real multiplications and 5 real additions.
But since matrix multiplication costs O(n^3) or O(n^x) for some x>2, and addition costs O(n^2), for large n the cost drops by a factor 3/4.
The error analysis is very similar to the usual algorithm.

Applying the above formula recursively in a different context yields an algorithm for multiplying two n-bit integers in O(n^log2(3)) = O(n^1.59) bit operations, as opposed to the conventional O(n^2). This topic is taught in classes like CS170.