CS170 – Fall 2011 – Lecture 4 – Sep 6

1 Motivation for Improving Matrix Multiplication

Now we will just consider the best way to implement the usual algorithm for matrix multiplication, the one that take $2n^3$ arithmetic operations for n-by-n matrices. To see that there is something interesting to discuss, consider the plot below, which shows the speed of n-by-n matrix multiplication on a Sun Ultra-1/170 (an old machine, but the plots would be similar on modern machines). The horizontal access is n, and the vertical access is speed measured in Mflops = millions of floating point arithmetic operations per second; the peak on this machine is 330 Mflops, or 2 per cycle. The top curve, labelled Sun Perf Lib, is a version hand-optimized by the manufacturer especially for this machine. The bottom curve is the naive algorithm (3 nested loops) in C; it is over 10 times slower for large n. Thus we see that while we might hope that the compiler would be smart enough take our naive algorithm and make it run fast on any particular machine, we see that compilers have a ways to go.

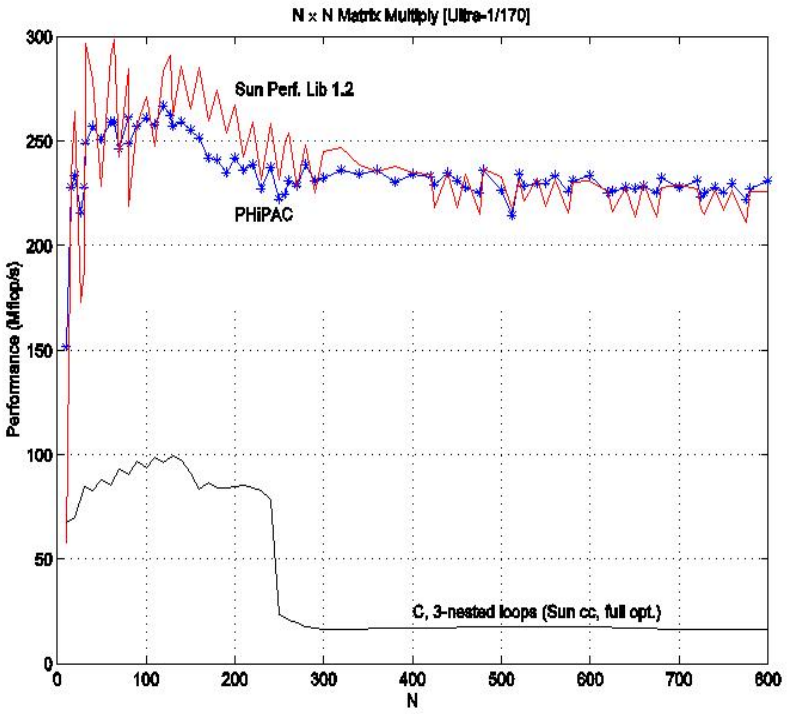
Each manufacturer typically supplies a library like Sun Perf Lib, carefully hand-optimized for its machines. It includes a large number of routines, not just matrix multiplication, although matrix multiplication is one of the most important, because it is used as part of benchmark to compare the speed of computers.

Finally, the middle curve, labeled PHIPAC, is the result of a research project here at Berkeley that aimed to automatically generate code as good as hand-generated code for any architecture, and so automate the alternative tedious hand-optimization process. See `www.icsi.berkeley.edu/~bilmes/phipac` for information about PHIPAC, and `math-atlas.sourceforge.net` for a later project called ATLAS inspired by PHIPAC that produces widely code used. For example, Matlab has used ATLAS.

The goal of these lecture notes is to explain the basic techniques used both by manufacturers in their hand optimizations and by PHIPAC and ATLAS to obtain these speedups. We remark that there is also active research in the compiler community on automatically applying optimizations of the sort we discuss here, as well as active research at Berkeley in applying these ideas to automatically tune the performance of other important functions.

In addition, the tuning of matrix multiply has long been used as an assignment in the graduate class CS267 to teach students the important of various optimization technique. For example, see `www.cs.berkeley.edu/~demmel/cs267_Spr11` for the course, `www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/` for the assignment from 2009, and `www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/` for the results showing how many

student teams used which optimizations (not just the ones described here), wrote how much code (up to thousands of lines!), and got how close to the vendor optimized code in performance. (The optimization described here gives a 7x speedup over the obvious 3 nested loops, but other optimizations make the fastest code 6x faster on top of this.) Now we also use a version of this assignment in CS61C.



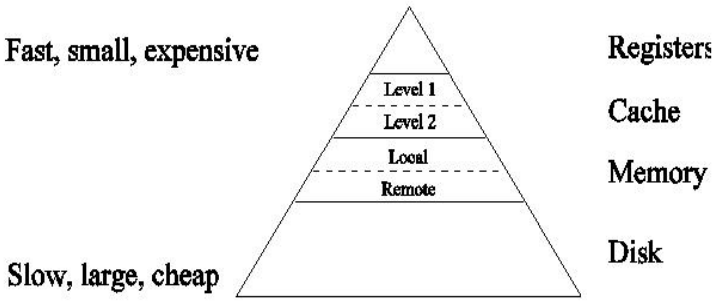
2 What operations should we count?

So far we have counted arithmetic operations – additions and multiplications – in our complexity analysis. This is because our underlying assumption has been that these are the most expensive and most frequent operations in the algorithm, so that if Algorithm 1 did fewer additions and multiplications than Algorithm 2, it was automatically faster. Let us examine this assumption, which is not entirely true.

If you look at the assembly language version of matrix multiplication, which is what the machine executes, then there are many kinds of instructions besides additions and multiplications, some of which are executed just as often as the additions and multiplications. Which ones

are most expensive? It turns out that on most modern processors, the instructions that take two floating point numbers (like 3.1416) from two registers, add or multiply them, and put them back in registers, are typically the *fastest* the machine performs. Addition is usually 1 cycle, the least any instruction can take, and multiplication is usually as fast or perhaps a little slower.

It turns out that *loads* and *stores*, i.e. moving data from memory to registers (which is where all the arithmetic and other "useful work" takes place) are the most expensive, sometimes costing *hundreds* of cycles or more. The reason is that any computer memory, from the cheapest PC to the biggest supercomputer, is organized as a *memory hierarchy*:



The idea is that the it is only possible to do arithmetic on data in a few expensive memory cells, usually called *registers*, that are right next to the arithmetic unit. They are too expensive and therefore too few to store all your data, so most data must be stored in larger, slower cheaper memory. Whenever you want to do arithmetic on data not in registers, you have to go get the data from this slower memory, which takes more time.

If all the data were either in registers or in the *main memory*, then there would be an enormous penalty for accessing data not in registers, since main memory is hundreds of times slower. So

people almost always build intermediate levels in the memory hierarchy, called *caches*, that are intermediate in cost, size and speed between registers and main memory. Thus accessing data not in registers but in cache is not as slow as accessing main memory.

Here is an everyday analogy. Suppose you are sitting at your desk, and you need a pencil to write something. The first place you look is the top of your desk, and if a pencil is there, you pick it up and use with a minimum of overhead. This corresponds to using data in registers.

But if the pencil is not in the desk, you have to open the desk drawer and look there. Since you probably keep more pencils in your drawer than your desk top, you will likely find one, and it will only take a little more time to open the drawer to get it, than it would if the pencil were on the desk. This corresponds to using the *cache*.

Now suppose your desk drawer had no pencils. You have to get up and go to the supply cabinet to get pencils, which takes yet more time. And you should not just get one pencil from the supply cabinet, you should get a small box of pencils, so you'll have them the next time. This corresponds to going to main memory, and fetching a whole *cache line* of consecutive words of memory, in the expectation that if you need one, you'll need more of them.

Next, if the supply cabinet it empty, you have to go to the store to buy pencils. This takes significantly longer, and you will probably buy a bigger box of pencils, so you don't have to go to the store very often. This corresponds to taking a *page fault*, and getting a whole page (thousand of bytes) of your data off of disk.

In the worst case, if the store is sold out, they have to go to a wholesale distributor to get a whole carton of pencils, which corresponds to getting your data off of tape, the bottom of the memory hierarchy.

Thus the principle behind memory hierarchies is a perfectly normal economic principle we encounter in everyday life. A typical computer, including a PC, may have two levels of cache (on-chip and off-chip), as well as main memory, internal disk and external disk (instead of tape). A large parallel computer may have many memory modules, some *local* to each processor, and so faster to access, and some *remote*, and so slower.

So it is clear that merely counting arithmetic operations is not the whole story; we should try to minimize memory accesses instead. In previous analyses, when we counted the number of, say, accesses to entries in an array, we (quite reasonably) assumed that each such access involved a memory access, so were already counting the *maximum* number of memory accesses (in the $O(.)$ sense). In the case of matrix multiplication, if we assume there is some small number of memory accesses for each statement like $c_{i,j} = c_{i,j} + a_{i,k} \cdot b_{k,j}$ (load $a_{i,k}$, $b_{k,j}$, $c_{i,j}$; store $c_{i,j}$), then counting arithmetic is nearly the same as counting memory accesses.

But it turns out that in matrix multiplication, as well as some other codes, if there is a memory hierarchy then one can organize the algorithm to minimize the number of accesses to the *slower*

levels of the hierarchy. In other words, we will do complexity analysis distinguishing memory accesses to fast memory and to slow memory. Therefore, when we specify the algorithm, we will not only have to say what operations are performed, but when data is moved among the levels of the memory hierarchy, since these are the most expensive operations. It turns out that we can organize matrix multiplication to *reuse* data already in fast memory, decreasing the number of slow memory accesses.

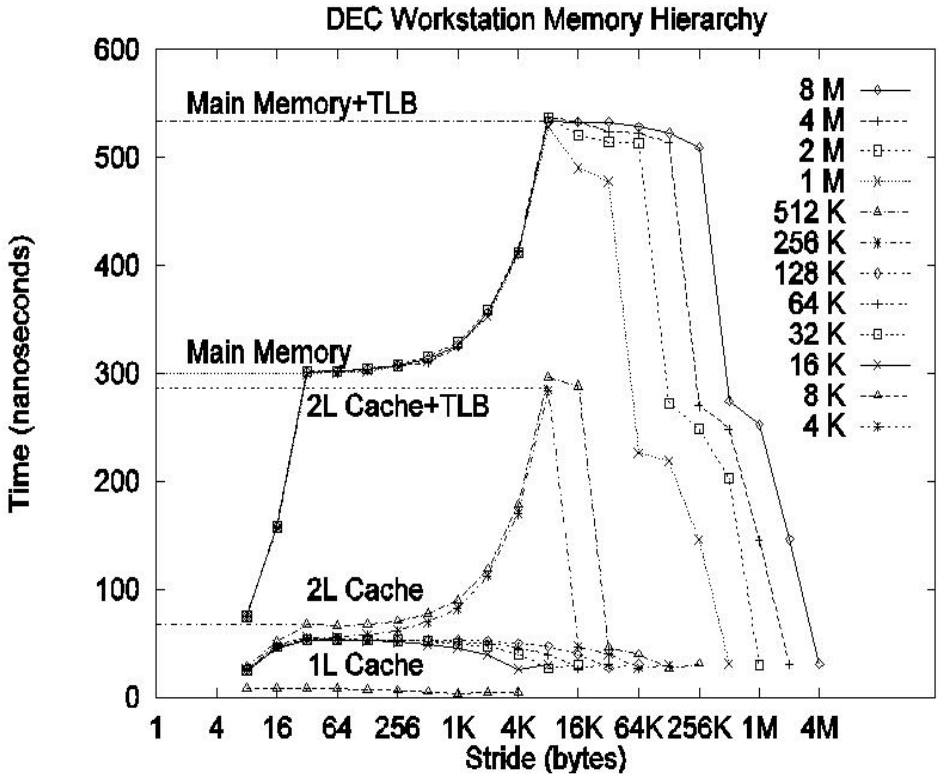
3 Modeling Memory Hierarchies - The Complicated Truth

Real memory hierarchies are very complicated, so that modeling them carefully enough to predict performance of an algorithm is hard. After a simple example to illustrate this point, we will introduce a very simple model that we will use to devise a better matrix multiplication algorithm.

To see how complicated the execution time of even a very simple program can be, consider the following program:

```
array A[Size] of doubles  ...   8 bytes per element
for i=0 to Size by s
   load A[i]
```

All this program does is step through memory, loading 8 bytes from memory into a register, skipping 8s-8 bytes, reading another 8 bytes, etc., Size/s times. s is also called the *stride*. We run this program many times and divide the total elapsed time by the number of times we ran it, in order to get a relialble estimate of the time to run the program. Then we take this time and divide by Size/s, the number of loads, to get the time per load. The time per load is plotted on the plot below, for various values of Size and s, on a DEC Alpha 21064, a 150 MHz microprocessor from the early 1990s; modern machines can be even more complicated. The vertical axis on the plot is time per load in nanoseconds. The horizontal axis is stride s. For each Size from 4K= 2^{12} to 8M= 2^{23} (powers of 2 only), values of s from 8 up to Size/2 were tried (again, powers of 2 only). The times per load for a fixed Size and varying s are connected by lines.



The most obvious fact about the time per load for this simple program is that it is a rather complicated function of Size and s, and varies from 6.7 nanoseconds (1 cycle at 150 MHz, the minimum) for the smallest Sizes, up to over 500 nanoseconds. It turns out that every curve on this plot can be explained in terms of the behavior of the DEC Alpha's memory hierarchy, given enough effort (see `www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps` for details). But that is a problem for a different class.

The second fact is that we obviously would prefer to organize our algorithms to take the minimum 6.7 nsecs = 1 cycle per load instead of over 500 nsecs.

The third fact is that we need a simple model of the memory hierarchy so we can think about algorithms without getting buried under all the details of this picture. Not only will the performance be even more complicated for more interesting programs, but the details will vary from machine to machine. The gap between fastest and slowest time per load is only getting larger on newer architectures, so the penalty for using a bad algorithm will only increase.

4 Modeling Memory Hierarchies - A Useful Simplification

To make progress, we will use the following very simple model:

1. There are only two levels in the memory hierarchy, fast and slow.
2. All the input data starts in the slow level, and the output must eventually be written back to the slow level.
3. The size of the fast memory is M, which is large but much smaller than the main memory. In particular, the input of large problems will not all fit in fast memory simultaneously.
4. The programmer explicitly controls when which data moves between fast and slow memory.
5. Arithmetic (and logic) can only be done on data residing in fast memory. Each such operation takes time t_a. Moving a word of data from fast to slow memory, or slow to fast, takes time $t_m \gg t_a$. So if a program does m memory moves and a arithmetic operations, the total time it takes is $T = a \cdot t_a + m \cdot t_m$, where $m \cdot t_m$ may be $\gg a \cdot t_a$.

The reader may object to bullet (4) above, since it is hardware that automatically decides when to move data between main memory and cache. However, we know how the hardware works: it moves data to the cache precisely when the user first tries to load it into a register to perform arithmetic, and puts it back in main memory when the cache is too full to get the next requested word. This means that we can write programs *as though* we controlled the cache explicitly by doing arithmetic operations in different orders. Thus two programs that do the same arithmetic in different orders may run at very different speeds, because one will cause less data movement between main memory and cache.

With this model, let's get a *lower bound* on the speed of any algorithm for a problem that has m_i inputs and m_o outputs, and does a arithmetic operations. The only difference among algorithms that we permit is the order in which arithmetic operations are done. By our model, the time taken will be at least

$$T = a \cdot t_a + (m_i + m_o) \cdot t_m$$

no matter what clever order in which we do the arithmetic.

For example, suppose the problem is to take two input arrays of n numbers each, $a[i]$ and $b[i]$ for $i = 1, .., n$, and produce two output arrays s and p where $s[i] = a[i]+b[i]$ and $p[i] = a[i]*b[i]$. Then $a = 2n$, $m_i = 2n$ and $m_o = 2n$, so a lower bound on the running time for any algorithm for this problem is $T = 2nt_a + 4nt_m$. Let us look at two different algorithms for this problem,

and see which will be faster. According to our model, we assume a and b are initially in slow memory, and $M \ll n$, so we cannot fit a and b in fast memory. The two algorithms are as follows. We have indicated when the loads from and stores to slow memory occur.

```
Algorithm 1                    Algorithm 2

for i = 1, n                   for i = 1, n
  {load a[i], b[i]}              {load a[i], b[i]}
  s[i] = a[i] + b[i]             s[i] = a[i] + b[i]
  {store s[i]}                   {store s[i]}
endfor                           p[i] = a[i] * b[i]
for i = 1, n                     {store p[i]}
  {load a[i], b[i]}            endfor
  p[i] = a[i] * b[i]
  {store p[i]}
endfor
```

The point is that Algorithm 1 loads a and b twice whereas Algorithm 2 only loads them once, because there is not enough room in fast memory to keep all the $a[i]$ and $b[i]$ for a second pass to compute $p[i]$.

Thus, in our simple model Algorithm 1 takes time $T_1 = 2nt_a + 6nt_m$ whereas Algorithm 2 takes only $T_2 = 2nt_a + 4nt_m$, the minimum possible. So when $t_a \ll t_m$, Algorithm 1 takes 50% longer than Algorithm 2. In the case of matrix multiplication, we will see an even more dramatic difference.

5 Minimizing Slow Memory Accesses in Matrix Multiplication

We will only consider different ways to implement n-by-n matrix multiplication $C = C + A \cdot B$ using $2n^3$ operations, i.e. not Strassen's method. Thus the only difference between algorithms will depend on the number of loads and stores to slow memory. We also assume that the slow memory is large enough to contain our three n-by-n matrices A, B and C, but the fast memory is too small for this.

Otherwise, if the fast memory were large enough to contain A, B and C simultaneously, then our algorithm would be

Move A, B and C from slow to fast memory

Compute $C = C + A \cdot B$ entirely in fast memory

Move the result C back to slow memory

The number of slow memory accesses for this algorithm is $4n^2$ ($m_i = 3n^2$ loads of A, B and C into fast memory, plus $m_o = n^2$ stores of C to slow memory), yielding a running time of $T = 2n^3 t_a + 4n^2 t_m$. Clearly, no algorithm doing $2n^3$ arithmetic operations can run faster.

At the extreme where the fast memory is very small ($M = 1$), then there will be at least 1 memory reference per operand for each arithmetic operation involving entries of A and B, for a running time of at least $T = 2n^3 t_a + (2n^3 + 2n^2) t_m$.

So when the size of fast memory M satisfies $1 \ll M \ll 3n^2$, what is the fastest algorithm? This is the practical question for large matrices, since real caches have thousands of entries. As we just saw, the worst case running time is $2n^3 t_a + (2n^3 + 2n^2) t_m$, and the best we can hope for is $2n^3 t_a + 4n^2 t_m$, which can be almost $n/2$ times faster when $t_m \gg t_a$.

We begin by analyzing the simplest matrix multiply algorithm, which we repeat below, *including* descriptions of when data moves from slow to fast memory and back. Remember that A, B and C all start in slow memory, and that the results C must be finally stored in slow memory.

```
Naive matrix multiplication C = C + A*B
procedure NaiveMM(A,B,C)
for i = 1 to n
    for j = 1 to n
        Load c_{i,j} into fast memory
        for k = 1 to n
            Load a_{i,k} into fast memory
            Load b_{k,j} into fast memory
            c_{i,j} = c_{i,j} + a_{i,k} · b_{k,j}
        end for
        Store c_{i,j} into slow memory
    end for
end for
```

Let m_{naive} denote the number of *slow memory references* in this naive algorithm. Then

$$\begin{aligned} m_{naive} &= n^3 \text{ ... for loading each entry of } A \text{ } n \text{ times.} \\ &\quad + n^3 \text{ ... for loading each entry of } B \text{ } n \text{ times.} \\ &\quad + 2n^2 \text{ ... for loading and storing each entry of } C \text{ once.} \\ &= 2n^3 + 2n^2 \,, \end{aligned}$$

or about as many slow memory references as arithmetic operations. Thus the running time is $T_{naive} = 2n^3 t_a + (2n^3 + 2n^2)t_m$, the worst case.

Here is an alternative, called *blocked matrix multiplication* (sometimes it is called *tiled* or *panelled* instead of *blocked*). Suppose now that, as in Strassen, each a_{ij} is not 1-by-1 but s-by-s, where s is a parameter called the *block size* that we will specify later. We assume s divides n for simplicity. Matrices B and C are similarly partitioned. Then we can think of A as an n/s-by-n/s *block matrix*, where each entry $a_{i,j}$ is an s-by-s block. The inner loop $c_{i,j} = c_{i,j} + a_{i,k} \cdot b_{k,j}$ now runs for $k = 1$ to n/s, and represents an s-by-s matrix multiplication and addition. The algorithm becomes

```
Blocked matrix multiplication C = C + A*B
procedure BlockedMM(A,B,C)
for i = 1 to n/s
    for j = 1 to n/s
        Load c_{i,j} into fast memory
        for k = 1 to n/s
            Load a_{i,k} into fast memory
            Load b_{k,j} into fast memory
            NaiveMM(a_{i,k}, b_{k,j}, c_{i,j}) ... using only fast memory
        end for
        Store c_{i,j} into slow memory
    end for
end for
```

The inner loop NaiveMM($a_{i,k}$, $b_{k,j}$, $c_{i,j}$) has all its data residing in fast memory, and so causes no slow memory traffic at all. So expanding NaiveMM(), the algorithm consists of 6 nested loops. Redoing the count of slow memory references yields

$$\begin{aligned} m_{blocked}(s) &= (n/s)^3 \cdot s^2 \text{ ... for loading each block } a_{i,k} \ (n/s)^3 \text{ times.} \\ &\quad + (n/s)^3 \cdot s^2 \text{ ... for loading each block } b_{k,j} \ (n/s)^3 \text{ times.} \\ &\quad + 2(n/s)^2 s^2 \text{ ... for loading and storing each block } c_{i,j} \text{ once.} \\ &= 2n^3/s + 2n^2 \ , \end{aligned}$$

Comparing $m_{naive} = 2n^3 + 2n^2$ and $m_{blocked}(s) = 2n^3/s + 2n^2$, it is obvious that we want to pick s as large as possible to make $m_{blocked}(s)$ as small as possible. But how big can we pick

s? The largest possible value is obviously $s = n$, which corresponds to loading *all* of A, B and C into fast memory, which we cannot do. So s depends on the size M of fast memory, and the constraint it must satisfy is that the three s-by-s blocks $a_{i,k}$, $b_{k,j}$ and $c_{i,j}$ must simultaneously fit in fast memory, i.e. $3s^2 \leq M$. The largest value s can have is therefore $s = \sqrt{M/3}$, yielding

$$m_{blocked}(\sqrt{M/3}) = \sqrt{12}\frac{n^3}{\sqrt{M}} + 2n^2$$

In other words, for large matrices (large n) we decrease the number of slow memory references, the most expensive operation, by a factor $\Theta(\sqrt{M})$. This is attractive, because it says that cache (fast memory) helps, and the larger the cache the better.

In summary, the running time for this algorithm is

$$T_{blocked} = 2n^3 t_a + \left(\sqrt{12}\frac{n^3}{\sqrt{M}} + 2n^2\right) t_m$$

There is a theorem, which we will not prove, that says that up to constant factors, we cannot do fewer slow memory references than this (while doing the usual $2n^3$ arithmetic operations):

Theorem. (Hong + Kung, 1981, 13th Symposium on the Theory of Computing). *Any implementation of matrix multiplication using* $2n^3$ *arithmetic operations performs at least* $\Omega(n^3/\sqrt{M})$ *slow memory references.*

In practice, this technique is very important to get matrix multiplication to go as fast as possible. But careful attention must be paid to other details of the instruction set, arithmetic units, and so on. If there are more levels of memory hierarchy (two levels of cache) then one might use this technique *recursively*, dividing s-by-s blocks into yet smaller blocks to exploit the next level of memory hierarchy. Matrix multiplication is important enough that computer vendors often supply versions optimized for their machines, as part of a library It may also be produced automatically using ATLAS (`math-atlas.sourceforge.net`). called the *BLAS*, or Basic Linear Algebra Subroutines.