Università della Svizzera italiana

Institute of Computing CI

HPC Software Atelier, Fall Semester 2021
Team: Prof. O. Schenk, Dr. P. Bansal,
Dr. J. Kardoš, T. Holt, M. Lechekhab

Project 2
Motivation for Improving Matrix Multiplication

1 Matrix multiplication

Now we will just consider the best way to implement the usual algorithm for matrix multiplication, the one that takes $2n^3$ arithmetic operations for n-by-n matrices. To see that there is something interesting to discuss, consider the plot 1, which shows the speed of n-by-n matrix multiplication on an AMD Opteron Processor 2344 with 1700 MHz for the processor. The horizontal axis is n, and the vertical axis is speed measured in MFlops = millions of floating point arithmetic operations per second; the peak on this machine is 6800 MFlops, or 4 operations per cycle. The top curve, labelled GOTO BLAS library, is a version hand-optimized by a group of researchers at the Texas Advanced Computing Center. More recently this code is being maintained, improved, and extended to new platforms and processors by the OpenBLAS project [1] and the BLIS framwork [2].
The bottom curve is the naive algorithm (3 nested for loops) in C; it is over 100 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 long way to go. Each manufacturer typically supplies a BLAS library (e.g. Intel MKL library), 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 a benchmark to compare the speed of computers.
Instead of using any vendor library method you can also use the BLAS software, which is the result of a research project at the University of Tennessee that aimed to automatically generate code as good as hand-generated code for any architecture. See http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms for a project called BLAS that produces optimized code used widely today. For example, if you have ever used MATLAB you have used BLAS/LAPACK. The goal of this part of the mini-project notes is to explain the basic techniques used both by manufacturers and libraries like GotoBLAS in their hand-optimized libraries 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 several universities in applying these ideas to automatically tune the performance of other important functions such as sparse matrix-vector multiplications.

2 What operations should we count?

So far we have counted arithmetic operations — such as 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 an algorithm, so that if Algorithm 1 did fewer additions and multiplications than Algorithm 2, it would automatically be 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 several other instructions besides additions and multiplications, some of which are executed just as often as the additions and multiplications. Which ones are the 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 bit slower. It turns out that **loads** and **stores**, i.e., moving

[1] `http://www.openblas.net/`
[2] `https://en.wikipedia.org/wiki/BLIS_(software)`

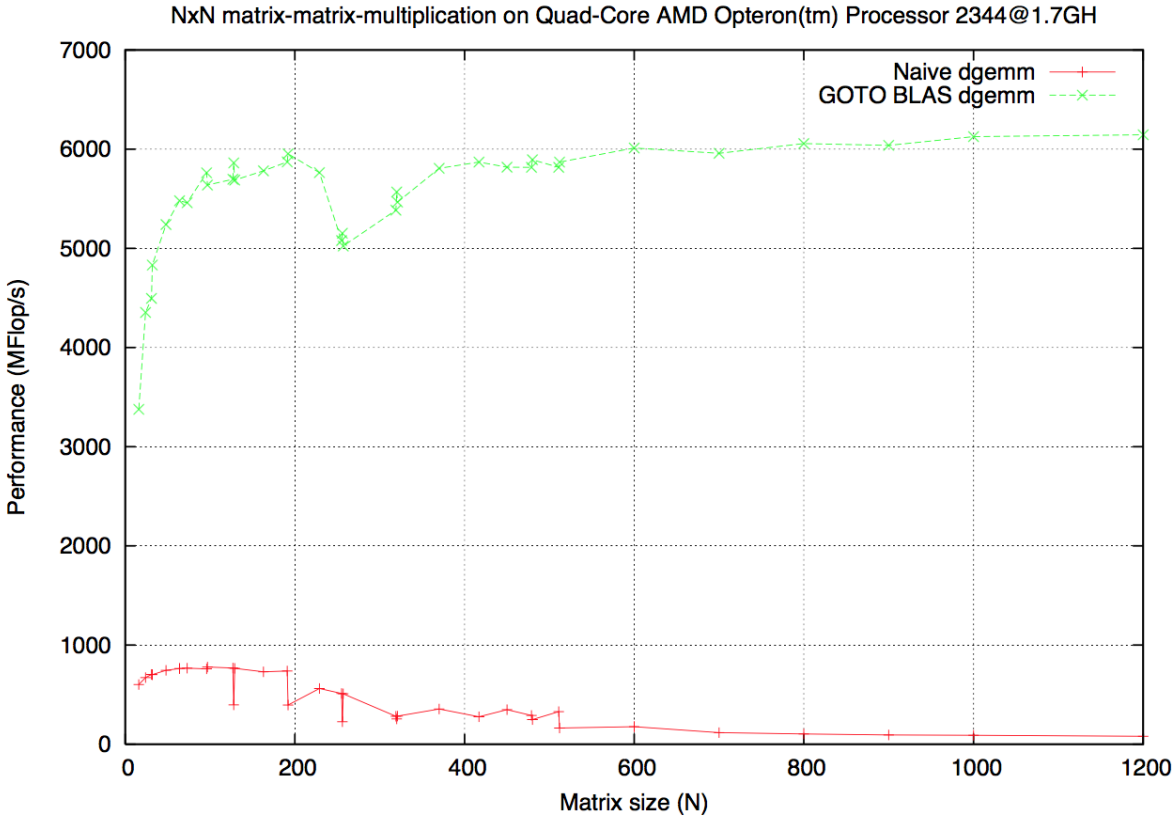

Figure 1: Performance of matrix multiplication on the AMD Opteron.

data from the memory to registers (which is where all the arithmetic and other useful work takes place) are the most expensive operations, 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 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 a larger, slower, cheaper memory. Whenever you want to do arithmetic on data which is not available in registers, you have to go fetch 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, it is common to build intermediate levels in the memory hierarchy, called **caches**, that lie in between registers and the main memory in terms of cost, size and speed. Thus, accessing data not in registers but in caches is not as slow as accessing the 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 overhead. This corresponds to using data in registers. But if the pencil is not on the desk, you have to open the desk drawer and look in it. Since you probably keep more pencils in your drawer than your desktop, 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 a **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 will have them easily available when they are needed next. 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,

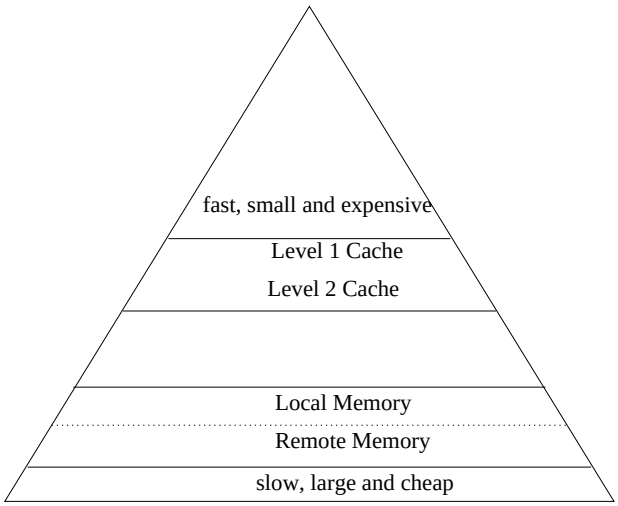

Figure 2: Typical memory hierarchy on HPC architectures.

you will need more of them.

Next, if the supply cabinet is also 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 the 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 the tape, the bottom of the memory hierarchy.

Thus, the principle underlying 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 a main memory, an internal disk, and an 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.

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). For matrix multiplication, if we assume that 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 in 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 the complexity analysis distinguishing memory accesses to the fast memory and to the slow memory. Therefore, when we specify the algorithm, we will not only have to say what operations are performed, but also 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 the fast memory, decreasing the number of slow memory accesses.

3 Modeling memory hierarchies — the complicated truth

Real memory hierarchies are very complicated, so modeling them carefully enough to predict the 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 bytes, reading another 8 bytes Size/s times, s is 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 reliable estimate of the run-time of 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 an AMD Opteron, a 1700 MHz microprocessor. The vertical axis on the plot is time per load in nanoseconds. The horizontal axis is the 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.

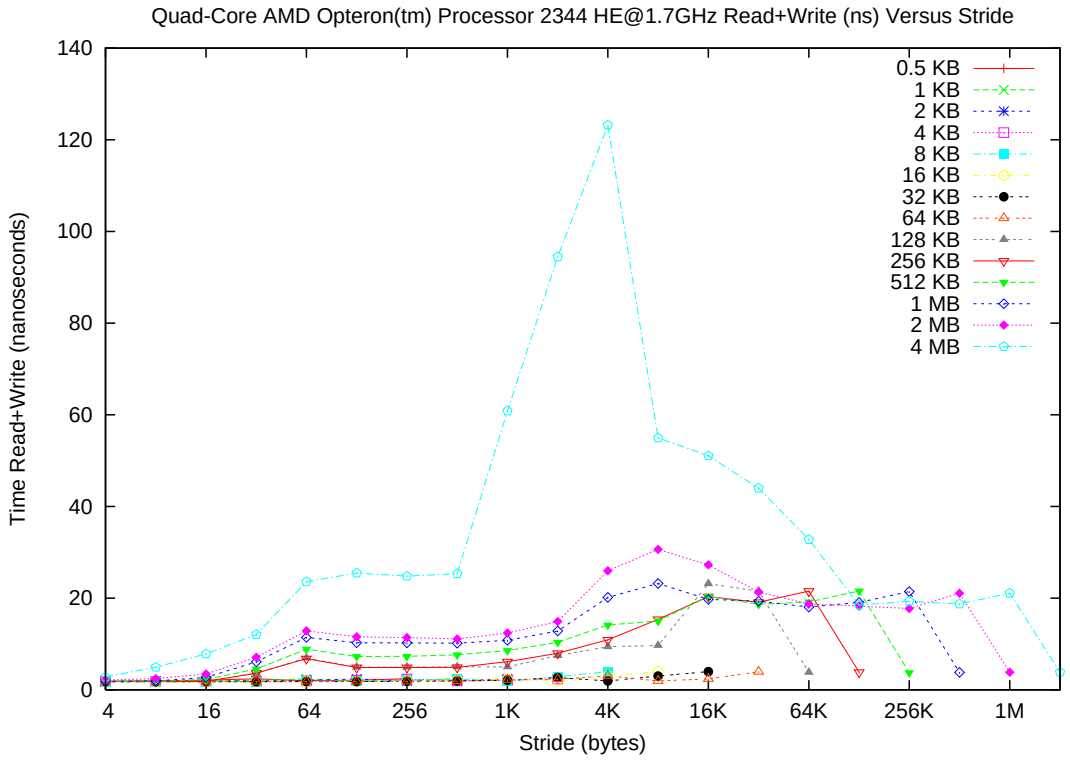

Figure 3: Memory hierarchy on the AMD Opteron.

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 0.58ns (1 cycle at 1700 MHz, the minimum) for the smallest sizes, up to over 50 nanoseconds. It turns out that every curve on this plot can be explained in terms of the behavior of the AMD Opteron memory hierarchy, given enough effort. 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 0.4 nsecs = 1 cycle per load instead of over 140 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 the fastest and the slowest time per load is only getting larger on newer architectures, so the penalty for using a bad algorithm will only increase.

3.1 Modeling memory hierarchies - a useful simplifcation

To make progress, we will use the following simplified 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 fit in the fast memory simultaneously.
4. The programmer explicitly controls when which data moves between the two memories.
5. Arithmetic (and logic) can only be done on data residing in the fast memory. Each such operation takes time t_a. Moving a word of data from one memory to the other takes time $t_m \gg t_a$. Hence, if a program performs m memory moves and a arithmetic operations, the total time it takes is $T = a \cdot t_a + m \cdot t_m$, where it may be that $m \cdot t_m \gg a \cdot t_a$.

The reader may object to bullet (4) above, since it is the 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 if 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 reduces data movement between the main memory and cache.
With this model, let us 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 between algorithms that we permit is the order in which the arithmetic operations are executed. According to our model, the time taken will be at least

$$T = a \cdot t_a + (m_i + m_o) \cdot t_m, \tag{1}$$

no matter which clever order we do the arithmetic in.
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 $p[i] = a[i] \cdot b[i]$. Then $a = 2n$, $m_i = 2n$, and $m_o = 2n$. Thus, a lower bound on the run-time for any algorithm for this problem is $T = 2nt_a + 4nt_m$. Let us look at two different algorithms for this problem, and analyze which will be faster. According to our model, we assume that a and b are initially in the slow memory, and $M \ll n$, so a and b cannot fit in the fast memory. The two algorithms are as given below, we have indicated when loads from and stores to slow memory occur:

```
1 Algorithm 1                    Algorithm 2
2
3 for i = 1, n                   for i = 1, n
4    {load a[i], b[i]}              {load a[i], b[i]}
5    s[i] = a[i] + b[i]             s[i] = a[i] + b[i]
6    {store s[i]}                   {store s[i]}
7 endfor                            p[i] = a[i] * b[i]
8                                   {store p[i]}
9 for i = 1, n                   endfor
```

```
10    {load a[i], b[i]}
11    p[i] = a[i] * b[i]
12    {store p[i]}
13 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 the fast memory to keep all the $a[i]$ and $b[i]$ for a second pass to compute $p[i]$.
As per 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. Thus, for $t_a \ll t_m$, Algorithm 1 takes 50% longer to run than Algorithm 2. In the case of matrix multiplication, we will observe an even more dramatic difference.

3.2 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 the 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 simply be:

```
1 Move A, B and C from slow to fast memory
2
3 Compute C = C + A · B entirely in fast memory
4
5 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, and $m_o = n^2$ stores of C to slow memory), yielding a run-time of $T = 2n^3t_a + 4n^2t_m$. Clearly, no algorithm doing $2n^3$ arithmetic operations can be 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 run-time of atleast $T = 2n^3t_a + (2n^3 + 2n^2)t_m$.

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 run-time is $2n^3t_a + (2n^3 + 2n^2)t_m$, and the best we can hope for is $2n^3t_a + 4n^2t_m$, which would be almost $n/2$ times faster for $t_m \gg t_a$.
We begin by analyzing the simplest matrix multiplication algorithm, which we repeat below, including descriptions of when data moves between the slow and the fast memories. Remember that A, B, and C all start in the slow memory, and that the result C must be finally stored in the slow memory.

```
1 Naive matrix multiplication C = C + A · B
2
3 method NaiveMM(A, B, C)
4
5 for i=1 to n
6    for j=1 to n
7       Load c_{i,j} into fast memory
8       for k=1 to n
9          Load a_{i,k} into fast memory
```

```
10          Load b_{k,j} into fast memory
11          c_{i,j} = c_{i,j} + a_{i,k} * b_{k,j}
12          end for
13       Store c_{i,j} into slow memory
14    end for
15 end for
```

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

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

or about as many slow memory references as arithmetic operations. Thus, the run-time is $T_{\text{naive}} = 2n^3 t_a + (2n^3 + 2n^2)t_m$, the worst possible.
Here, we provide an alternative called **blocked matrix multiplication** (sometimes it is called **tiled** or **panelled** instead of **blocked**). Now, suppose 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 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 s, and represents an s-by-s matrix multiplication and addition. The algorithm becomes:

```
1  Blocked matrix multiplication C = C + A · B
2
3  method BlockedMM(A, B, C)
4
5  for i=1 to n/s
6     for j=1 to n/s
7        Load C_{i,j} into fast memory
8        for k=1 to n/s
9           Load A_{i,k} into fast memory
10          Load B_{k,j} into fast memory
11          NaiveMM (A_{i,k}, B_{k,j},  C_{i,j}) using only fast memory
12       end for
13       Store C_{i,j} into slow memory
14    end for
15 end for
```

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

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

Comparing $m_{\text{naive}} = 2n^3 + 2n^2$ and $m_{\text{blocked}}(s) = 2n^3/s + 2n^2$, it is obvious that we want to pick s as large as possible to make $m_{\text{blocked}}(s)$ 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 the fast memory, which we cannot do. So, s depends on the size M of the 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 the fast memory, which implies $3s^2 \leq M \implies s \leq \sqrt{M/3}$. Therefore, the largest value $s = \sqrt{M/3}$ yields

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

In other words, for large matrices (large n) we decrease the number of slow memory references, the most expensive operation, by a factor $O(M)$. 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

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

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 $\mathcal{O}(n^3/\sqrt{M})$ slow memory references.

In practice, this technique is very important to get matrix multiplication to run as fast as possible. But careful attention must also 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 a part of a library. It may also be produced automatically using GOTO BLAS (`http://www.openblas.net/`). [3]

[3] This document is originally based on a tutorial from Professor Katherine A. Yelick from the Computer Science Department at the University of Berkeley, `http://www.cs.berkeley.edu/~yelick/`