Homework 2.1: Sequential Matrix-Matrix Multiply

Kathy Yelick, Mark Hoemmen, and Brian Kazian

11. Sep. 2007

1 Summary

Your job:

- Design and write one or more optimized sequential optimized multiplication routines for square matrices;
- Compare performance with the naive implementation, a simple optimized implementation that we provide, and an optimized library for which we provide a wrapper;
- Discuss performance results in a writeup.

2 Introduction

Matrix-matrix multiplication (which we abbreviate as MMM) computes the following, for matrices $A$, $B$, and $C$:

$$ C := C + A \cdot B. $$

MMM is a basic building block of many scientific computations. Modern codes for solving a linear system $Ax = b$ or doing a least-squares fit of coefficients to data depend on efficient MMM implementations for good performance. MMM is popular for benchmarks, for the following reasons:

- A good implementation can show off the ideal performance of a computer’s floating-point system and memory hierarchy;
- A bad implementation performs much, much worse than a good implementation, so the benefits of applying a new optimization are clear.
Many such benchmarks refer to MMM as xGEMM.\(^1\) xGEMM is the name of the matrix-matrix multiply routine in the BLAS (Basic Linear Algebra Subroutines). The BLAS is an interface for basic operations involving vectors and matrices. Different vendors are free to offer their own implementations of the BLAS that are tuned for specific platforms, and projects such as ATLAS\(^2\) automatically generate a BLAS implementation that is tuned for whatever platform on which you run the ATLAS build process. We will be comparing your MMM routine with a routine from an optimized implementation of the BLAS. (We use the term MMM because xGEMM performs a slightly more general operation \(C := \beta C + \alpha A \cdot B\). Naturally if you set \(\beta = 1\) and \(\alpha = 1\) you get MMM.)

3 Conventions

3.1 Matrices and arrays

We’d like to distinguish between matrices and arrays. A matrix \(A\) is a mathematical object. It has entries \(A_{ij}\) which are indexed by two indices \(i\) and \(j\). Sometimes we may use the Fortran or Matlab notation \(A(i,j)\) to mean \(A_{ij}\); in either case, \(i\) and \(j\) both start counting at one. In contrast, an array is a data structure which may be used for storing a matrix. You may have heard of dense and sparse matrices; we’re working with dense ones in this assignment, which means that all the entries of the matrix are stored explicitly, even if they are zero.

You may have used arrays in Java or C to store matrices. Probably, you were taught to use multidimensional arrays for this purpose. In this case, \(A_{ij}\) would be written as \(A[i][j]\). Multidimensional arrays have an inefficient and inconvenient implementation in Java, and are also inconvenient to use in C.\(^3\) Instead, we use one-dimensional arrays to store matrices in C and Java.

3.2 Row- or column-oriented?

The terms column-oriented and row-oriented refer to how a matrix is stored in an array. Column-oriented means that elements above and below each other in the same column are stored next to each other in the array. (You can imagine stacking up the columns of the matrix to form a single vector.) Row-oriented

\(^1\)If you’re curious about the name, the “GE” stands for “general” (as in general matrices, as opposed to symmetric or Hermitian) and “MM” stands for “matrix-matrix” (multiply). The “x” isn’t actually the name of the routine; there are four different routines (SGEMM, DGEMM, CGEMM, and ZGEMM), and the first letter of each indicates the type of entries in the matrix (S for single-precision real, D for double-precision real, C for single-precision complex, and Z for double-precision complex).

\(^2\)http://math-atlas.sourceforge.net/

\(^3\)Java does not guarantee that the elements of a matrix are stored contiguously in memory, which is bad both for performance and for binary compatibility with existing libraries. In both C and Java, a two-dimensional array is basically an array of pointers, with one pointer for each row of the matrix. You have to allocate an array for each row separately, and in C, you have to deallocate each row separately.
means that elements to the right and left of each other in the same row are stored next to each other in the array.

When you write two-dimensional arrays in C or Java, they are stored in row-oriented fashion. We prefer instead the Fortran column-oriented convention, as it is standard in the numerical linear algebra community, and allows binary compatibility with BLAS implementations and many other existing libraries. In this assignment, the input matrices will be stored in column order, and the output matrix that your routine will compute must be stored in column order.\footnote{You are welcome to do copy optimizations, but the user shouldn’t have to think about them. Copy overhead also be included in the official timings, although you are welcome to measure and report the copy overhead time.}

Note that it doesn’t really matter whether a matrix is row-oriented or column-oriented, because the two formats are transposes of each other.

### 3.3 Accessing column-oriented matrices

If your matrix is stored in column order, you need to know three things about it: the row and column dimensions, and the \textit{leading dimension} (which we sometimes call LDA, for “leading dimension of \( A \)). If your matrix \( A \) is just a submatrix of some other matrix, then the number of rows of \( A \) may be different than the number of rows in the larger matrix. (Submatrices will prove useful in your implementation.) In that case, \( A \) may not be stored contiguously; there are “gaps” in the storage which belong to other elements of the larger matrix. LDA is the key to avoiding these gaps which don’t belong to \( A \).

Let’s say we want to access element \( i, j \) of \( A \), and the matrix \( A \) is stored in some array \( A_{\text{array}} \). Then we write \( A_{\text{array}}[i + j \times \text{LDA}] \).

### 4 Matrix-matrix multiplication

#### 4.1 Mathematical definition

MMM for square matrices has the following mathematical definition (which is independent of any implementation). Given two \( n \times n \) matrices \( A \) and \( B \), the sum and product

\[
C = A + B
\]

is an \( n \times n \) matrix \( C' \) satisfying

\[
C'_{ij} = C_{ij} + \sum_{k=1}^{n} A_{ik} \cdot B_{kj}
\]

for \( 1 \leq i, j \leq n \).

We can define MMM for non-square matrices too; this will prove useful in implementations. You can’t multiply just any old \( A \) and \( B \), though: they must have \textit{compatible} dimensions. That is, if you want to compute \( C = A \cdot B \), then
A must have the same number of columns as the number of rows of \( B \). If \( A \) is \( m \times n \) and \( B \) is \( n \times p \), then \( C = A \cdot B \) is \( m \times p \). In that case,

\[
C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}.
\]

### 4.2 Naive algorithm

The most naive code to multiply square matrices comes straight from the above definition. It is short, sweet, simple, and very slow:

1. \textbf{for} \( i = 1 \) to \( n \) \textbf{do}
2. \hspace{1em} \textbf{for} \( j = 1 \) to \( n \) \textbf{do}
3. \hspace{2em} \textbf{for} \( k = 1 \) to \( n \) \textbf{do}
4. \hspace{3em} \( C(i,j) := C(i,j) + A(i,k) \cdot B(k,j) \)
5. \hspace{2em} \textbf{end for}
6. \hspace{1em} \textbf{end for}
7. \textbf{end for}

Prof. Yelick explains some reasons in lecture for the naive algorithm’s poor performance. Ultimately, it comes down to \textit{temporal locality}: how well does the code exploit reuse of data in the memory hierarchy, from the registers up to the caches and TLB? Taking advantage of temporal locality means the difference between a runtime of \( \Theta(n^3) \) and a runtime that grows more like \( \Omega(n^5) \).

### 5 Assignment

You will optimize a routine to multiply square matrices. We provide an implementation of the naive matrix-matrix multiply algorithm in C, and a very simple optimized implementation in C. We’re also providing a wrapper that calls a highly optimized version of matrix-matrix multiplication which was written by professional tuning experts, so that you have an idea how fast MMM can go on your benchmarking machines. Your job is to write a sequential MMM routine that outperforms the naive algorithm as well as the simple optimized version.

You will provide the following in your solution:

- Source code, Makefile, and PBS script, along with instructions for building and running the code. The executable should take a parameter \( n \), which is the matrix dimension.

- Any code generation scripts you may have used, along with instructions for running them. You are not required to use code generation scripts, but they will probably be useful.

- A written report that details your optimizations, and compares the performance of your code against that of the provided routines. Please save the report as PDF if possible.
• The report should include a graph with the matrix dimension $n$ on the $x$ axis, and the performance (in floating-point operations per second) on the $y$ axis.

5.1 Notes

You are welcome to write all the code you want, and to adapt the provided code as necessary to support your work, as long as (a.) we can compile, link, and run it; and (b.) it’s a fair benchmark.

Your code should compare the numerical result (the matrix $C$) of your routine against the output of the naive algorithm. Such comparisons should not be counted in the runtime. One way to compare two matrices is to subtract them elementwise and find the maximum element (in absolute value) of the resulting matrix. Report the difference; if it’s large, you might have a bug.

We provide some suggested matrix dimensions to try. Trying only powers of two is going to make your lives hard, because they will perform badly! You may even wish to handle powers-of-two-sized matrices by padding them with zeros to a different size.

You should limit the number of trials (different matrix sizes that you try) when you are debugging, but be sure to enable all of them for your final benchmarks.

NOTE: write a sequential code only. We’ll be discussing parallel versions later.

5.2 Due date

This assignment (2.1) will be due at 5pm Pacific Standard Time on Tuesday, 18 Sep 2007. Please keep your code and results as you’ll need them for the next part of the assignment.

6 Implementation hints

Prof. Yelick’s Monday 10 Sep 2007 lecture discusses a number of optimizations. Don’t feel like you need to implement them all! Start simple, and add optimizations incrementally.

6.1 General code structure

Recursion is the right way to think about this problem, even if you decide to use iteration in your solution for the sake of efficiency. Figure out how to make small matrix-matrix multiplications fast, then use that as a building block to make larger matrix-matrix multiplications go fast.

5Think about how cache associativity might cause this problem.
6.2 Code generation

In the discussion section of Friday 07 Sep 2007, we discussed using a code generation script in order to generate a number of alternative implementations. You can find a link on the Homeworks page\textsuperscript{6} to those (working) Python scripts. Code generation is especially useful if you’d like to vary the number and dimensions of the submatrices in a recursive decomposition. This tends to look like “2-D loop unrolling,” which is why we provided (1-D) loop unrolling as an example.

You are not required to use a code generation script, but you may find it useful.

\textsuperscript{6}http://www.cs.berkeley.edu/~mhoemmen/cs194/homeworks.html