St. Francis Xavier University
Department of Computer Science

CSCI 554: Matrix Computation
Lecture 1: Introduction
Winter 2022

1 Enter the Matrix

Matrices are a common and concise way to represent numbers and variables that are related in some way. Indeed, matrices and linear algebra are so fundamental to the study of mathematics and computer science that students are often required to take an introductory course in this area at some point in their studies, as the material is more likely than not to appear in one's technical career. Consider, for example, the study of computer graphics, and more specifically playing a 2D video game on a computer. To the uninitiated, it may appear that the sprites are moving around the screen by some kind of digital magic. But those who understand matrices will recognize a common pattern underlying each graphical transformation: rotating a sprite by θ degrees can be done using the matrix

$$R_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix},$$

a reflection about a line $\ell = (\ell_x, \ell_y)$ can be achieved using the matrix

$$R_\ell = \frac{1}{\|\ell\|_2} \begin{bmatrix} \ell_x^2 - \ell_y^2 & 2\ell_x\ell_y \\ 2\ell_x\ell_y & \ell_y^2 - \ell_x^2 \end{bmatrix},$$

and so on. Indeed, these examples only scratch the surface; the matrices for 3D transformations are even wilder!

Where these introductory courses often fall short, however, is in teaching computer science students only how to operate on and manipulate matrices by hand. Computers are faster, more accurate, and more precise than humans—so why don't we use computers to manipulate matrices? Taking all that we learned in an introductory linear algebra course and converting it into the language of algorithms is an excellent exercise in both applying mathematics to computer science and using computers to perform complex mathematical calculations, and these dual principles form the basis of the present course.

In this course, we will study matrices and linear algebra from the perspective of computer science. We will write algorithms to apply common operations, transformations, and techniques to matrices, and we will analyze the performance of such algorithms. And ultimately, we will come away from this course with a better understanding of how matrices can model the computational (and physical) world around us.[1]

2 Motivation: Matrix Multiplication

One of the most common things one does with matrices, and one of the earliest things a student learns to do with matrices, is perform multiplication. *Matrix multiplication* is a very well-studied operation, and indeed, it is the bedrock of many of the results we will see in this course. If we didn't know how to multiply matrices, we couldn't do any of the other interesting stuff we'll see in later lectures!

Here, we will motivate our study of matrix computation by studying and comparing various methods of matrix multiplication.

[1] PS. I promise that the title of the first section will be the only *The Matrix* pun I make in these notes.

2.1 Matrix × Vector

Suppose we are given an $m \times n$ matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

where each of the entries a_{ij} are real numbers, and we are also given a column vector

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

where each of the entries x_i are again real numbers. To take the product of the matrix A with the vector x, we follow the standard procedure where, intuitively, we multiply each row of A with the single column of x and sum the n products. In other terms,

$$\begin{aligned} Ax &= \sum_{i=1}^{m} a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \\ &= \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}x_j. \end{aligned}$$

The product of the ith row of A and x is called the *inner product* or *dot product*. Observe that, even though A is a matrix and x is a vector, the inner product of A and x gives us a single numeric value.

Looking at this process algorithmically, each iteration of the outer sum (i.e., the sum involving i) produces a term that contributes to the overall value; let's call this term b_i. Then we have that $Ax = b$, where $b = b_1 + b_2 + \cdots + b_m$. With this insight, we can write our first algorithm.

Algorithm 1: Matrix-vector multiplication

```
b ← 0
for i from 1 to m do
    for j from 1 to n do
        b_i ← b_i + a_ij x_j
    b ← b + b_i
return b
```

This now raises the natural question: how well does our first algorithm perform? To analyze the algorithm, we require the notion of a *flop*, or a "floating point operation". Since the entries in both A and x were taken to be real numbers, and since real numbers are stored in a computer using a floating point representation, one flop corresponds to a computer performing one arithmetic operation involving floating point numbers. If we assume that all of the matrices we deal with contain real numbers, then counting flops is a great way to approximate the efficiency of algorithms for those matrices.[2]

In Algorithm 1, observe that our inner loop (i.e., the loop involving j) involves two flops: one to multiply a_{ij} and x_j, and one to add $a_{ij}x_j$ to b_i. Moreover, our inner loop runs a total of j times. Our outer loop (i.e., the loop involving i) involves one flop to add b_i to b, and this loop runs a total of i times. Altogether,

[2]Keep in mind, however, that counting flops only gives us an approximation of an algorithm's performance; this technique doesn't take into account other tasks the computer must perform, like allocating/deallocating memory or returning values.

then, the total number of flops used by Algorithm 1 is

$$\begin{aligned}\sum_{i=1}^{m} 1 \cdot \sum_{j=1}^{n} 2 &= \sum_{i=1}^{m} 1 \cdot 2n \\ &= 2mn,\end{aligned}$$

and the algorithm therefore performs $O(mn)$ operations using asymptotic notation—or, if we restrict ourselves to square matrices, $O(n^2)$ operations.

If we wanted to, we could exchange the order of the outer and inner loops of Algorithm 1. In doing so, we would effectively be multiplying the jth column of A with x on each iteration, thus producing a vector b where the ith entry is equal to b_i. That is, we would be calculating

$$\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} x_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} x_2 + \cdots + \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} x_n.$$

However, since we only swapped the order of the loops, we would not be using any additional flops to calculate in this way, and so this modified version of our algorithm would still perform $O(mn)$ operations. The only difference is that we're now performing a *column-oriented* matrix-vector multiplication, instead of the *row-oriented* matrix-vector multiplication of Algorithm 1.

2.2 Matrix × Matrix

Let's now consider the multiplication of the same matrix A from the previous section with some $n \times p$ matrix

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}.$$

Naturally, if $p = 1$, then we return to the exact same scenario of matrix-vector multiplication that we discussed in the last section. Thus, let's assume that $p > 1$.

Similar to matrix-vector multiplication, where we were computing the value $b = Ax$, with matrix-matrix multiplication we are computing a matrix $B = AX$ with dimension $m \times p$. In this matrix B, the entry at row i and column j is equal to the inner product of row i of A and column j of X; that is,

$$b_{ij} = \sum_{k=1}^{n} a_{ik} x_{kj}.$$

Effectively, what we're doing when we compute B is we're performing a total of p matrix-vector multiplications! This insight leads us to our second algorithm.

Algorithm 2: Matrix-matrix multiplication

```
B ← 0
for i from 1 to m do
    for j from 1 to p do
        for k from 1 to n do
            b_ij ← b_ij + a_ik x_kj
return B
```

So, how well does this algorithm perform? Let's count flops once again. As before, the innermost loop (i.e., the loop involving k) involves two flops: one to multiply a_{ik} and x_{kj}, and one to add $a_{ik}x_{kj}$ to b_{ij}. This

innermost loop is nested within two further loops. Therefore, the total number of flops used by Algorithm 2 is

$$\sum_{i=1}^{m}\sum_{j=1}^{p}\sum_{k=1}^{n} 2 = 2mpn,$$

and the algorithm therefore performs $O(mpn)$ operations. Again, restricting ourselves to square matrices, this asymptotic bound becomes $O(n^3)$.

Note that, as we observed before, the order of the three nested loops doesn't particularly matter; we could rearrange the loops into any one of the six possible combinations, and the overall performance of the algorithm won't be impacted.

2.3 Block Matrices

Our matrix-matrix multiplication algorithm is somewhat naïve, in the sense that it's iterative: we must iterate through each i, j, and k, one by one, until we calculate each entry b_{ij} in order. While this might have been the ideal approach with older, simpler computers, nowadays we have computers that have multiple CPU cores, gigabytes of memory, and all sorts of other performance improvements.

Thus, instead of making our algorithm iterate one entry at a time, let's take a different perspective and restructure our matrices into *blocks* or smaller submatrices. Again, our goal is to calculate $B = AX$. This time around, let's fix values r and s, and partition our matrix A into r blocks row-wise and s blocks column-wise:

$$A = \left[\begin{array}{c|c|c} A_{11} & \cdots & A_{1s} \\ \hline \vdots & \ddots & \vdots \\ \hline A_{r1} & \cdots & A_{rs} \end{array}\right].$$

Let's also partition our matrix X into s blocks row-wise and t blocks column-wise:

$$X = \left[\begin{array}{c|c|c} X_{11} & \cdots & X_{1t} \\ \hline \vdots & \ddots & \vdots \\ \hline X_{s1} & \cdots & X_{st} \end{array}\right].$$

Note that we must partition X *conformably*; that is, the structure of the blocks in rows of X must match the structure of blocks in columns of A. Finally, we consider the appropriate partitioning of the matrix B:

$$B = \left[\begin{array}{c|c|c} B_{11} & \cdots & B_{1t} \\ \hline \vdots & \ddots & \vdots \\ \hline B_{r1} & \cdots & B_{rt} \end{array}\right].$$

You might be able to observe that, with these partitions of A, X, and B into blocks, we can multiply blocks of A and X together to obtain a block appearing in B. Indeed, our two approaches—the naïve way and the block matrix way—are equivalent.

Theorem 1. *Let matrices A, X, and B be partitioned into blocks as described. Then $B = AX$ if and only if*

$$B_{ij} = \sum_{k=1}^{s} A_{ik}X_{kj}$$

where $1 \leq i \leq r$ and $1 \leq j \leq t$.

Proof. Follows from the definition of matrix multiplication. □

Since X must be partitioned conformably according to the partitioning of A, we would be able to perform multiplication on matrices partitioned like

$$\left[\begin{array}{c|cc} A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{array}\right] \left[\begin{array}{c|cc} X_{11} & X_{12} & X_{13} \\ \hline X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{array}\right],$$

but not on matrices partitioned like

$$\left[\begin{array}{c|cc} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right] \left[\begin{array}{c|cc} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ \hline X_{31} & X_{32} & X_{33} \end{array}\right],$$

since we wouldn't be able to multiply together, for example, the blocks $\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}$ and $\begin{bmatrix} X_{11} \\ X_{21} \end{bmatrix}$.

Altogether, our block matrix technique gives us the following pseudocode remarkably similar to that of Algorithm 2.

Algorithm 3: Block matrix multiplication

```
B ← 0
for i from 1 to r do
    for j from 1 to t do
        for k from 1 to s do
            B_ij ← B_ij + A_ik X_kj
return B
```

Example 2. Consider the matrices

$$A = \left[\begin{array}{cc|cc} 2 & -1 & 3 & 1 \\ 1 & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \text{ and } X = \left[\begin{array}{cc|c} 1 & 2 & 0 \\ -1 & 0 & 0 \\ \hline 0 & 5 & 1 \\ 1 & -1 & 0 \end{array}\right].$$

It is straightforward to show that the multiplication of these two matrices produces the matrix

$$B = \left[\begin{array}{cc|c} 4 & 18 & 3 \\ 3 & 5 & 1 \\ \hline 0 & 5 & 1 \\ 1 & -1 & 0 \end{array}\right].$$

As an example of an intermediate step, we can calculate

$$\begin{aligned} B_{11} &= A_{11}X_{11} + A_{12}X_{21} \\ &= \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 0 & 5 \\ 1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 14 \\ 2 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 4 & 18 \\ 3 & 5 \end{bmatrix}. \end{aligned}$$

So, why go through all of this effort of partitioning and creating block matrices when we get the same result as performing plain old matrix-matrix multiplication? This process ultimately doesn't affect our algorithm's performance, since we need to perform the same number of flops to multiply and add all of our intermediate

values. However, recall we briefly mentioned computer hardware in the opening paragraph of this section: that is the key motivation for partitioning our matrices.

If we're operating on particularly large matrices, then we likely have no choice but to store that data in slow memory, such as RAM or disk memory. Each time we perform a multiplication, we need to retrieve the relevant data from slow memory and move it into fast memory, such as cache memory. This is expensive!

On the other hand, if we partition our large matrices into smaller blocks, then we could retrieve entire blocks of data at a time to move into fast memory. Thus, instead of going back and forth by moving individual values into fast memory, we take full advantage of the fast memory to compute entire blocks that we can then move back into slow memory for storage.

As an additional benefit, the block matrix approach lends itself quite well to computers that are capable of parallelizing their computations. Instead of performing a completely-sequential multiplication, we can use multiple CPU cores to multiply many blocks in parallel.

2.4 The State of the Art

By now, we've established that the matrix-matrix multiplication algorithms we've seen in this lecture can't perform better than $O(n^3)$ time for square matrices. This *schoolbook algorithm*, so named because it's the simplest technique and often the one taught to young students in schools, is simply too naïve to improve upon this cubic upper bound.

That doesn't mean computer scientists have completely given up, though. A big question in complexity research asks how low we can make the exponent in the matrix multiplication bound, $O(n^\omega)$. We know that $2 \leq \omega \leq 3$, where 2 comes from the fact that any $n \times n$ matrix contains n^2 entries and 3 comes from our schoolbook algorithm.

In 1969, Volker Strassen published the first matrix multiplication algorithm that improved on the performance of the schoolbook algorithm. Strassen applied a divide-and-conquer heuristic to the schoolbook algorithm to partition the original $n \times n$ input matrix into a number of smaller $n/2 \times n/2$ matrices. Then, making the crucial observation that multiplying a pair of 2×2 matrices requires only 7 multiplications instead of 8, Strassen proved that his algorithm runs in $O(n^{\log_2(7)}) = O(n^{2.807})$ time.

Strassen's exponent of $\omega = 2.807$ held for almost a decade, until a number of improvements and refinements pulled the exponent down even more. In 1981, Don Coppersmith and Shmuel Winograd published a matrix multiplication algorithm with an exponent of $\omega = 2.496$, and further work by Strassen and Coppersmith & Winograd brought the exponent down further to $\omega = 2.3755$ by 1990.

From here, the bound on ω stood firm for twenty years, until a flurry of new results in the 2010s chipped away a few more decimal places. As of 2020, the best-known matrix multiplication algorithm runs in $O(n^{2.3728596})$ time, and this algorithm was published by Josh Alman and Virginia Vassilevska Williams. Can this exponent be brought down even further? Could it ever reach $\omega = 2$? Only time will tell.

All of these refined algorithms, however, lay bare one of the drawbacks of using Big-O notation. While the exponent is being ever-improved with each new paper and algorithm, the constant factors hidden by the Big-O notation are so huge that we would only really see practical performance improvements if we were multiplying very large matrices. That being said, these results do have significant theoretical impact, since matrix multiplication is frequently used as an intermediate step in various calculations. Thus, any improvement on the runtime of matrix multiplication is also an improvement for many other algorithms.

Year	**Exponent ω**	**Authors**
1969	2.8074	Strassen
1978	2.796	Pan
1979	2.780	Bini, Capovani, and Romani
1981	2.522	Schönhage
1981	2.517	Romani
1981	2.496	Coppersmith and Winograd
1986	2.479	Strassen
1990	2.3755	Coppersmith and Winograd
2010	2.3737	Stothers
2013	2.3729	Williams
2014	2.3728639	Le Gall
2020	2.3728596	Alman and Williams

Table 1: Matrix multiplication exponents throughout history