Fast Practical Matrix Multiplication

Oscar Smith and David White

November 25, 2019

Introduction

Multiplication of n by n matrices is a bottleneck for a variety of important applications, especially since the speed of many other linear algebra problems including finding inverses, many matrix factorizations, and eigenvalues.
Since these operations are so fundamental and are often performance constraints in solving real problems, in the 1970's, the BLAS library was created as a way of focusing all the low level optimizations in one common set of routines. As a result, almost all high level programming languages will simply transform linear algebra operations into calls to BLAS. However some newer languages such as Julia have a focus on allowing mixed precision arithmetic and other operations that BLAS has not been optimized for, having high performance generic methods to fall back on is important. Currently, for an example in pure Julia the difference between the time to multiply 2 moderately sized Float64 matrices and multiplying an Int matrix by Float64 matrix is a factor of 30. Some of this slowdown is inherent due to the cost of converting elements to Float64, but most of it is due to going from a BLAS method to a naive fallback. In this paper, we present an implementation that could be used to speed up the default case to near BLAS speeds.

Big O and Θ notation

In computer science the typical method for discussing the run-time of an algorithm is via Big O notation. Formally, $f(x) = O(g(x))$ if $\exists x_0, c > 0 : \forall x > x_o, f(x) \leq cg(x)$.
Intuitively, this means "The highest order term of the run-time complexity of f is not greater than g, up to a constant factor". The reason for using this notation is that smaller order terms don't matter for large enough inputs, and constant factors are rarely significant from an algorithmic perspective, since every computer will have different constants attached to the elementary operations anyways. The major benefit of using this type of notation is that it focuses on the parts of the complexity that make a difference and are easy to track.
A slightly stronger version of Big O is Big Θ. Formally,
$f(x) = \Theta(g(x))$ if $\exists x_0, c_0, c_1 > 0 : \forall x > x_o, c_0 g(x) \leq f(x) \leq c_1 g(x)$.
The difference here is that big Θ notation guarantees that the run-time of f is asymptotically equal (instead of less than or equal) to g.
These notations are also used in error analysis with identical meanings. For matrix operations, algorithmic run times are given in terms of $\Theta(n)$, even though the actual matrices A and B are size $n \times n$. For example, matrix addition requires a one operation addition on each of the n^2 elements of the $n \times n$ matrix and is thus considered $\Theta(n^2)$

Blocked Methods

Naive Blocking

If we represent the matrices $A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$, and likewise for B and C, then the fundamental operation $C = \alpha AB + C$ can be decomposed into the 4 block equations

$$\begin{aligned} C_{00} &= A_{00}B_{00} + A_{01}B_{10} \\ C_{01} &= A_{00}B_{01} + A_{10}B_{11} \\ C_{10} &= A_{10}B_{00} + A_{11}B_{10} \\ C_{11} &= (A_{10}B_{01} + A_{11}B_{11} \end{aligned} \tag{1}$$

This split uses 8 multiplications of size $\frac{n}{2}$, which produces the typical run-time of $\Theta(n^3)$. As such, there is no asymptotic improvement in using this over the traditional method. Despite this, in practice BLAS implementations will use more complicated but fundamentally similar blocking techniques to keep memory accesses near to each other which is essential for performance on modern processors.

Strassen's Algorithm

We will now use a different thing with our blocks that, while being completely un-intuitive, can be fairly easily shown to compute the same result of matrix multiplication. We will then show how to use this to implement a faster matrix multiplication algorithm than the standard one.
To start, create 7 temporary matrices, $M_0, \dots, M_6$, with

$$\begin{aligned}
M_0 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\
M_1 &= (A_{10} + A_{11})B_{00} \\
M_2 &= A_{00}(B_{01} - B_{11}) \\
M_3 &= A_{11}(B_{10} - B_{00}) \\
M_4 &= (A_{00} + A_{01})B_{11} \\
M_5 &= (A_{10} - A_{00})(B_{00} + B_{01}) \\
M_6 &= (A_{01} - A_{11})(B_{10} + B_{11})
\end{aligned} \tag{2}$$

Note that each M_i uses only 1 matrix multiplication, for a total of 7, compared to 8 for the naive blocking.
Now add and subtract these block matrices to create the parts of C as follows

$$\begin{aligned}
C_{00} &= M_0 + M_3 + M_6 - M_4 \\
C_{01} &= M_2 + M_4 \\
C_{10} &= M_1 + M_3 \\
C_{11} &= M_0 + M_2 + M_5 - M_1
\end{aligned} \tag{3}$$

This produces a valid multiplication because substitution of these equations yields

$$\begin{aligned}
C_{00} &= (A_{00} + A_{11})(B_{00} + B_{11}) + A_{11}(B_{10} - B_{00}) + (A_{01} - A_{11})(B_{10} + B_{11}) - (A_{00} + A_{01})B_{11} \\
&= A_{00}(B_{00} + B_{11} - B_{11}) \\
&\quad + A_{01}(B_{10} + B_{11} - B_{11}) \\
&\quad + A_{11}(B_{00} + B_{01} + B_{10} - B_{00} - B_{10} - B_{11}) \\
&= A_{00}B_{00} + A_{01}B_{10} \\
C_{01} &= A_{00}(B_{01} - B_{11}) + (A_{00} + A_{01})B_{11} \\
&= A_{00}(B_{01} - B_{11} + B_{11}) \\
&\quad + A_{01}B_{11} \\
&= A_{00}B_{01} + A_{01}B_{11} \\
C_{10} &= (A_{10} + A_{11})B_{00} + A_{11}(B_{10} - B_{00}) \\
&= A_{10}B_{00} \\
&\quad + A_{11}(B_{10} + B_{00} - B_{00}) \\
&= A_{10}B_{00} + A_{11}B_{10} \\
C_{11} &= (A_{00} + A_{11})(B_{00} + B_{11}) + A_{00}(B_{01} - B_{11}) + (A_{10} - A_{00})(B_{00} + B_{01}) - (A_{10} + A_{11})B_{00} \\
&= A_{00}(B_{00} + B_{11} + B_{01} - B_{11} - B_{00} - B_{01}) \\
&\quad + A_{10}(B_{00} + B_{01} - B_{00}) \\
&\quad + A_{11}(B_{00} + B_{11} - B_{00}) \\
&= A_{10}B_{01} + A_{11}B_{11}
\end{aligned} \tag{4}$$

In essence, this split is using the linear dependence of the original block matrix equations to turn one of the multiplications into several extra additions. As such, if we denote the time of matrix multiplication $T(n)$, this method has run-time $T(n) = 7T(\frac{n}{2}) + cn^2$, since it splits a matrix multiplication into 7 multiplications of half the size. If we recursively use this algorithm to do the small matrix multiplications, we get an asymptotic improvement in the run-time. This can be seen informally by noticing that for large n, the recursive step takes almost all the time, so ignoring the cn^2 term, we get the simpler equation $T(n) = 7T(\frac{n}{2})$. This is a geometric sequence with the ratio between terms of $log_2(7)$, so the total becomes $T(n) = n^{log_2(7)}$. This intuitive argument is formalized by the Master Theorem, which is a general method for solving recursive relations like this. The overall result is that with Strassen's algorithm, $T(n) = \Theta(n^{log_2(7)}$, a significant improvement over the naive n^3 for large matrices. As we will show in our implementation section, Strassen is only helpful when the matrices in question are bigger than about 500 elements per side. As such, the full version of the Strassen algorithm is to recursively use Strassen until the sub-matrices are smaller than this threshold, and use a direct blocked method for small sized multiplications.

Error

It is important to consider how numerically stable the Strassen method is. The naive method of multiplication has an error bound given by

$$|C - \hat{C}| \leq nu|A||B| + O(u^2)$$

, where n is the size of the $n \times n$ matrices, and u is the error in machine representation of the numbers involved (usually extremely small). The $O(u^2)$ thus represents error that is out of the control of the matrix multiplication algorithm. The Strassen method, by comparison, is less numerically stable, with

$$\left\|C - \hat{C}\right\| \leq (\frac{n}{n_0}^{\log_2 12}({n_0}^2 + 5n_0) - 5n)u \left\|A\right\| \left\|B\right\| + O(u^2)$$

(Note: n_0 is the size at which the algorithm stops recursing and switches to naive multiplication) Proving this is beyond the scope of this paper, however the general method is to establish a recursive expression for the error of the sub blocks, C_{11} for example, and use it to find the bounding expression.

In general, the types of matrices that will cause the most error under Strassen are those that have a large difference in the relative magnitude of non-zero terms. For example, let $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $B = \begin{pmatrix} 1 & \epsilon \\ \epsilon & \epsilon^2 \end{pmatrix}$

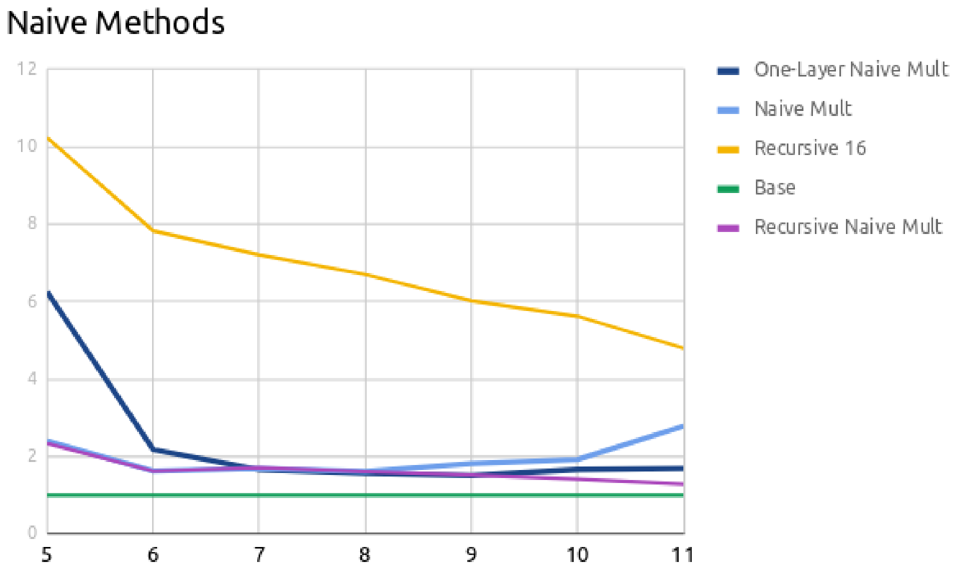
Using a naive method, AB_{22} is computed by $\epsilon * 0 + \epsilon^2 * 1$. This comes out be a number around ϵ^2. However, using Strassen, AB_{22} is computed by $2(1 + \epsilon^2) + (\epsilon - \epsilon)^2 - 1 - (1 + \epsilon)$. Note that $1 + \epsilon^2$ is going to be very close to 1, making the ϵ^2 relatively disappear and increasing error due to numerical imprecision. This error can then be magnfied by scaling A as much as desired. In general, Strassen will be accurate for matrices that are well-conditioned under the Frobenius norm, as all of the terms have to be of similar magnitude.

Strassen Implementation and Results

Our implementation of the Strassen algorithm has two different pieces. First, a one-layer Strassen multiplication. This method splits the matrix into blocks (adding an extra row of zeros as padding if necessary) according to the Strassen method, but uses a different multiplication operation to actually multiply the blocks. Thus, we have also implemented the naive method outlined initially, as well as Julia's built-in multiplication for use with this function.

Second, a recursive Strassen implementation. This method simply calls the one-layer Strassen method, but tells it to use the recursive Strassen method for the block multiplications. The method also has a base case, where matrices smaller than a certain size will be multiplied naively, instead of via Strassen.

We graphed these Strassen functions (using naive mult) against our naive method and the base Julia multiplication. For sizes, we used increasing powers of 2, so an increase in x by one represents doubling every dimension of the matrices involved. In addition, to make the graph more illustrative, results have been normalized against the base Julia multiplication. Thus the output of a given algorithm at that size is the ratio of times that algorithm takes at that size to the same sized base multiplication. Our raw data without normalization is included in Appendix B.



Methods.png

The results clearly show the importance of the cutoff. With a cutoff of 16, the setup time for Strassen outweighs the algorithmic advantage until n is moderately large. In fact, the extra recursive layer for Strassen doesn't become faster than the naive multiplication until around an n of 512 or 2^9. Thus, a cutoff of 500 is reasonable when using Strassen with our naive multiplication. A better optimized Strassen implementation would slightly lower this bound, but a more highly optimized base multiplication would counteract the effect.

Optimizing For Cache Size: Blocked Mult

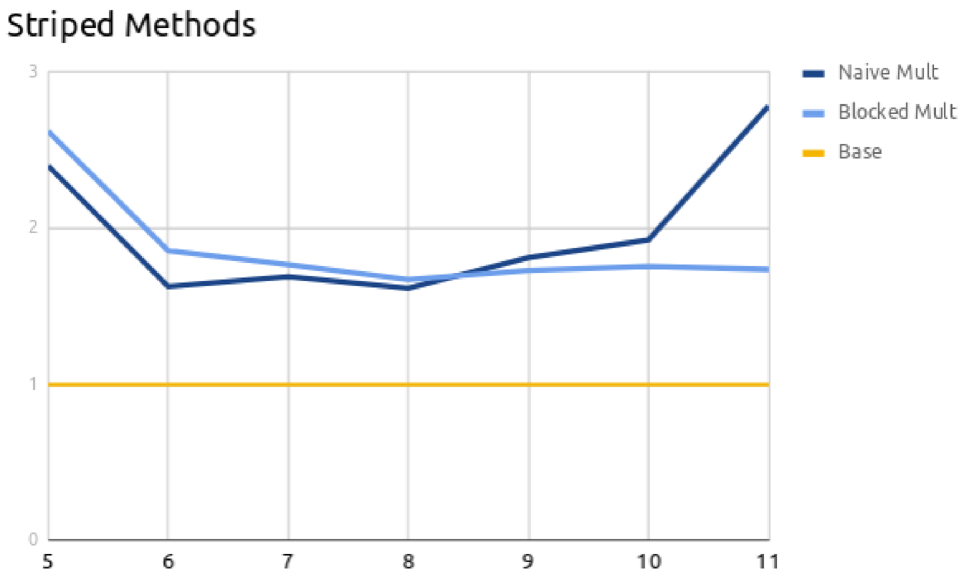
Modern computers have gotten continually faster processors, but the speeds of memory have not kept up. Much of this is from the pure physics reason that (physical) size of memory mandates a minimum distance from the processor, which in turn mandates a minimum latency via the speed of light. As such, modern computers use caches, which are smaller but designed to be accessed faster. A modern computer typically has 3 caches, L1, L2, and L3, each larger and slower than the previous. Thus, one key element of writing fast code is ensuring that the algorithm uses piece-wise contiguous memory (looping over 2 matrices in memory order is quick, but accessing random elements of a matrix is not). Unfortunately, naive matrix multiplication constantly requires reading elements out of memory order. Since, for any given programming language, matrices are either stored row by row or column by column, and naive matrix multiplication loops over the rows of one matrix and the columns of the other, one of these loops will be out of order. Thus the run-time of naive multiplication is constrained by the out of order loading required for one of the matrices.
The simplest improvement to this is to transpose one of the matrices beforehand. The two downsides of this are that this requires allocating a lot of memory, and that this operation has the same problem as multiplication with cache usage.
In practice, the solution is to instead break your matrices into kxn and nxk sections where k is some small number (we used 16). Then transposing one of these stripes is cheap because accesses in the "wrong" direction are only 16 apart, which still fits in the cache. Once this is done, all of the memory accesses are in a cache friendly order. Another benefit of this is it allows the processor to use specially designed vectorization instructions that can do multiple element-wise multiplications per clock cycle.

Implementation

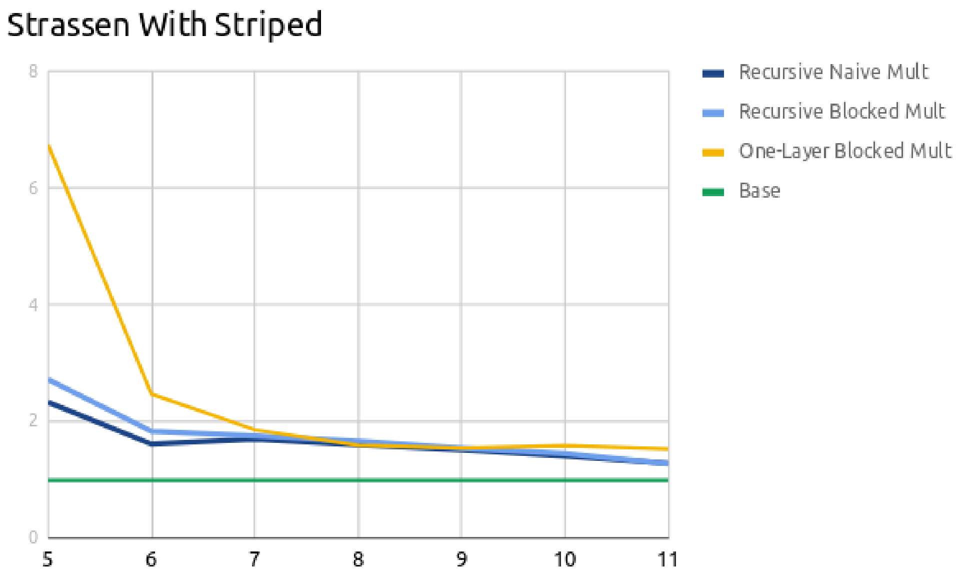
We added a BlockedMult function that breaks a matrix into size $16xn$ and $nx16$ pieces. If the matrix has a size that is not a multiple of 16, we calculate the effects of the extra rows and/or columns separately, and only block the pieces that are a multiple.

Results

BlockedMult is a significant improvement over NaiveMult at larger sizes, as shown in the chart. It is worth noting that the sizes used with both NaiveMult and BlockedMult are actually 1 larger than the power of 2. This is due to the behavior of the cache, as large powers of 2 are multiples of the cache size, so accesses along a column all attempt to access the same spot in the cache, rendering the cache effectively useless. This can result in the algorithm taking more than 8 times longer to run. Thus, by adding 1 to each dimension, the algorithm runs significantly faster. Similarly to the previous graph, all results here have a log_2 x scale, and are normalized relative to Julia's base multiplication.

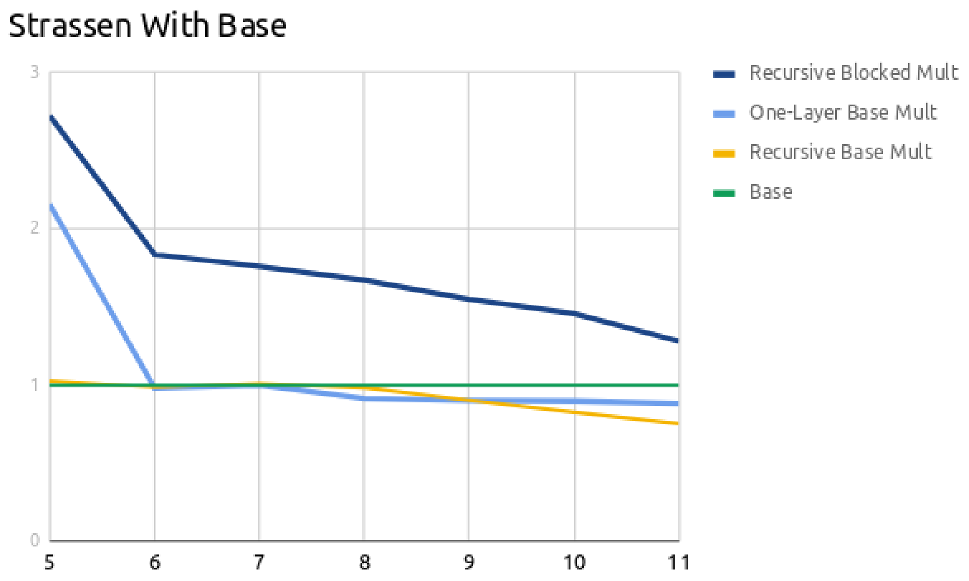


Strassen Stripe.png Using BlockedMult for Strassen is a significant improvement for the one-layer version. However, as BlockedMult is comparable to NaiveMult for small n, it is roughly identical for the recursive Strassen. Note that for both of these, the input size was again increased by 1, as this ensured neither BlockedMult nor NaiveMult had to handle a multiplication of a power of 2.



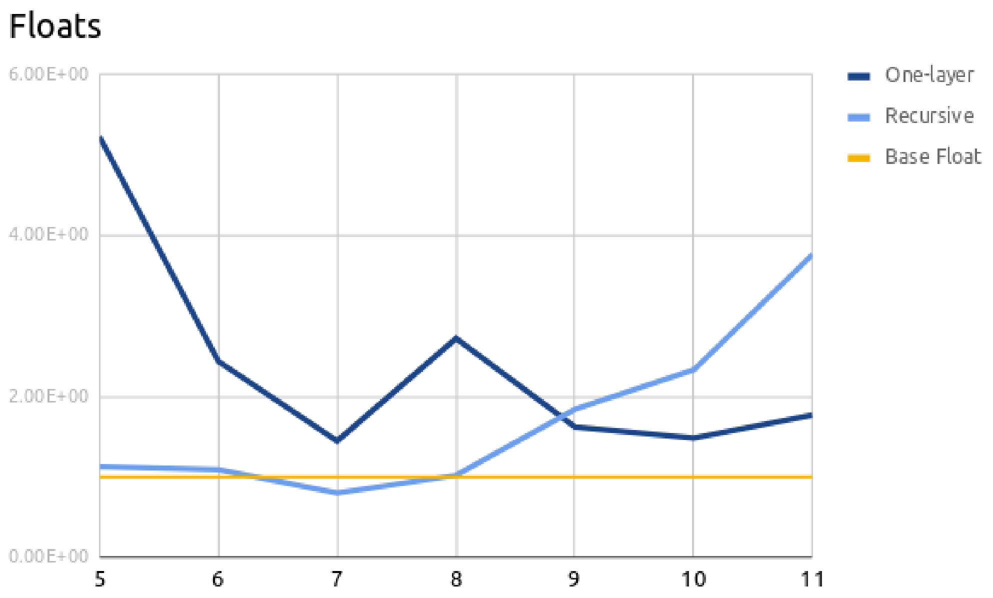
Strassen.png

The fastest version of our Strassen implementation is the the version that uses the built-in multiplication for submatrices. This version is actually able to outperform the built-in multiplication by around a size of 2^9, as can be seen by the one-layer line. The recursive version continues to improve as n becomes larger.

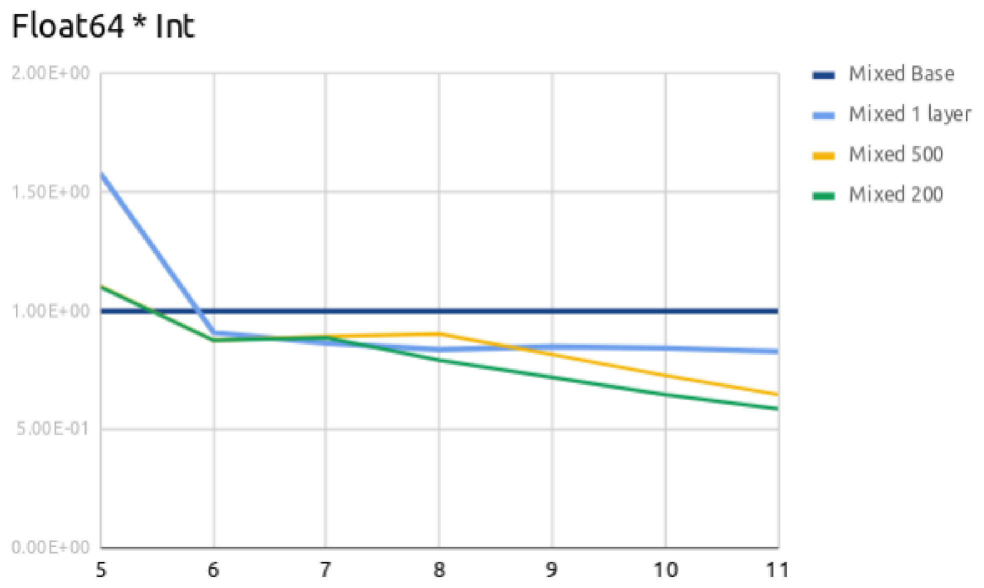


Strassen.png

On floats, however, our implementation cannot keep up with the built-in implementation. This is because BLAS multi-threaded for floating point matrix operations, so without a multi-threaded Strassen algorithm (theoretically easy in a future Julia version), we were unable to detect a size at which Strassen becomes more efficient. Furthermore, for floats, it is possible that OpenBLAS uses Strassen as well (we could not



Moving to mixed types, the results get even better. Specifically, we tested multiplication of Int matrices by Float64 matrices. This is representative of the vast majority of possible matrix representation methods (even if not most matrix multiplications used in practice). Since any individual combination of types will be exceedingly rare, there are no BLAS methods for these types, allowing even our relatively naive Strassen algorithm to get up to 40% faster than base by $n = 2048$. This effect is so pronounced, that for these methods, we see a benefit from lowering the recursion limit to 200 from the 500 that we used for the Int only version.



Type.png

Further Improvements

Non-Square Matrices

For the simplicity of this project, we only considered multiplication of square matrices. There were 2 reasons for this. The first is that it makes for a much simpler implementation, and the second is that the benefits of Strassen decrease the less square the matrices are. This happens because naive matrix multiplication of mxn times nxp matrices takes $\Theta(mnp)$, and the obvious lower bound for any matrix multiplication algorithm is $\Theta(max(mn, np, mp))$, ie the time to touch all elements in the biggest matrix of the input or output.
To make our Strassen work with non-square matrices, the simplest method would be to have a method that calls it on any large enough square sub-matrix multiplications of the result, and add on the rest with a $\Theta(mnp)$ algorithm.

Multi-threading

The biggest place where our methods could be significantly sped up is via use of multi-threading. Matrix multiplication is in the class of "embarrassingly parallel" algorithms (problems which gain full speedup from many cores). As such, for even relatively small problem, there are significant benefits from using all the cores in your computer. Julia does this for Float64 matrices, which is a large part of the reason we did all our benchmarks with Int matrices to have a more fair comparison. However, in less than 1 month, Julia 1.3.0 will be released, which has support for multi-threaded code (current versions have very limited support). Once this is released, it would be very interesting to rewrite our blocked multiplication to use these tools, as it should then be able to be significantly faster than all but the most optimized BLAS operations.

Temporary Matrix Reduction

The last significant improvement we could make would be to have a version of Strassen that handles the small matrix multiplications without ever materializing the temporary matrix operands. This can be done using striped multiplication methods similar to the one we made, but which instead of taking 2 matrices to multiply, take A_1, A_2, B_1, B_2, and computes $(A_1 + A_2)(B_1 + B_2)$ and only allocates the sum when it copies a stripe.
Doing this would save the allocation of 2 temporary matrices ($Atemp, Btemp$ in our code), and would lead to a significant number fewer passes through memory, which would give further speedups.

Works Cited

"Fast Matrix Multiplication." Accuracy and Stability of Numerical Algorithms, by Nicholas J. Higham, Society for Industrial and Applied Mathematics, 2002, pp. 446–463.

Huang, Jianyu. "Practical Fast Matrix Multiplication Algorithms." Texas ScholarWorks, The University of Texas at Austin, 2018, repositories.lib.utexas.edu/handle/2152/69013.

Appendix B: code

```
using LinearAlgebra
using BenchmarkTools
using Random
using InteractiveUtils
using Unrolled
using StaticArrays

# This shouldn't fix things, but does
promote_op = Base.promote_op

matprod(x, y) = x*y + x*y
const bs=16
function mult(A::AbstractMatrix, B::AbstractMatrix)
    @boundscheck size(A,2) == size(B,1)
    A*B
end

function naiveMult(A::AbstractMatrix,B::AbstractMatrix)
    @boundscheck size(A,2) == size(B,1)
    TS = promote_op(matprod, eltype(A), eltype(B))
    C = zeros(TS, size(A,1), size(B,2))
    return naiveMult!(C,A,B)
end

function naiveMult!(C,A,B)
    """ Fastest for ~nxn*nx16 for n in [50,75] """
    @inbounds for i in 1:size(A,1)
        for j in 1:size(B,2)
            s = C[i,j]
            @fastmath @simd for z in 1:size(B,1)
                s += A[i,z]*B[z,j]
            end
            C[i,j] = s
        end
    end
    return C
end

function blockedMult(A::AbstractMatrix,B::AbstractMatrix)

    @boundscheck size(A,2) == size(B,1)
    TS = promote_op(matprod, eltype(A), eltype(B))
    C = zeros(TS, size(A,1), size(B,2))
    Btemp = zeros(eltype(B),bs,bs)
    return blockedMult!(C, A, B, Btemp)
end


function blockedMult!(C, A, B, Btemp)
    n = size(A,1)
    left = n%bs
    @inbounds @fastmath for kk in 1:bs:n-left        # iterates over cols of A (rows of B)
        for jj in 1:bs:n-left                        # iterates over rows of B
            Btemp .= (@view B[kk:kk+bs-1,jj:jj+bs-1])'
            for i in 1:n-left                          # pick slice A[i,kk:kk+bs]
                for j in 1:bs            # Make dot product  A[i,kk:kk+bs] * B_block[:,j] for all j
                    s = C[i,j+jj-1]
                    @simd for k in 1:bs
                        s += A[i,k+kk-1] * Btemp[j,k]
                    end
                    C[i,j+jj-1] =s
                end
            end
        end
    end
    if left>0 #not a multiple of 16
        @inbounds @fastmath for j in 1:n-left
            for i in 1:n-left
                s = C[i,j]
                for k in n-left+1:n
                    s += A[i,k]*B[k,j]
                end
                C[i,j] = s
            end
        end
        @inbounds @fastmath for j in n-left+1:n
            for i in 1:n
                s = C[i,j]
                for k in 1:n
                    s += A[i,k]*B[k,j]
                end
                C[i,j] = s
            end
        end
        @inbounds @fastmath for j in 1:n-left
            for i in n-left+1:n
                s = C[i,j]
                for k in 1:n
                    s += A[i,k]*B[k,j]
                end
                C[i,j] = s
            end
        end
    end
    return C
end

function padAndSplit!(A,width,height)
    a1,a2=div(width+1,2),div(height+1,2)

    pad1=width%2
    pad2=height%2
    @views begin
        A00 = A[1:a1,1:a2]
        if pad1==0
            A10 = A[a1+1:end,1:a2]
        else
            A10=zeros(eltype(A),div(width+1,2),div(height+1,2))
            A10[1:end-pad1,1:end] .= A[a1+1:end,1:a2]
        end
        if pad2==0
            A01 = A[1:a1,a2+1:end]
        else
            A01=zeros(eltype(A),div(width+1,2),div(height+1,2))
            A01[1:end,1:end-pad2] .= A[1:a1,a2+1:end]
        end
        if pad1+pad2==0
            A11 = A[a1+1:end,a2+1:end]
        else
            A11=zeros(eltype(A),div(width+1,2),div(height+1,2))
            A11[1:end-pad1,1:end-pad2] .= A[a1+1:end,a2+1:end]
        end
        return A00,A10,A01,A11,zeros(eltype(A),div(width+1,2),div(height+1,2))
    end

end

function strassenBase(A::AbstractMatrix,B::AbstractMatrix, mult)
    @boundscheck size(A,2) == size(B,1)
    n, p, m = size(A,1), size(B,1), size(B,2)

    TS = promote_op(matprod, eltype(A), eltype(B))

    C = zeros(TS, n+n%2, m+m%2)
    return strassenBase!(C,A,B,mult)[1:n, 1:m]
end

function strassenBase!(C:: AbstractMatrix, A::AbstractMatrix,B::AbstractMatrix, mult)
    """ Takes in pre-padded arrays"""

    n, p, m = size(A,1), size(B,1), size(B,2)
    a1,a2 = (div(n+1,2),div(p+1,2))
    b1,b2 = (div(p+1,2),div(m+1,2))

    @inbounds @fastmath @views begin
        A00,A10,A01,A11,Atemp=padAndSplit!(A,n,p)
        B00,B10,B01,B11,Btemp=padAndSplit!(B,p,m)
        Atemp .= A00.+A11
        Btemp .= B00.+B11
        M0=mult(Atemp, Btemp)
        Atemp .= A10.+A11
        M1=mult(Atemp, B00)
        Btemp .= B01.-B11
        M2=mult(A00, Btemp)
        Btemp .=B10.-B00
        M3=mult(A11,Btemp)
        Atemp .= A00.+A01
        M4=mult(Atemp, B11)
        Atemp .= A10.-A00
        Btemp .= B00.+B01
        M5=mult(Atemp, Btemp)
        Atemp .= A01.-A11
        Btemp .= B10.+B11
        M6=mult(Atemp, Btemp)

        C[1:a1,1:a2] .= M0.+M3.+M6.-M4
        C[a1+1:end,1:b2] .= M1.+M3
        C[1:a1,b2+1:end] .= M2.+M4
        C[a1+1:end,b2+1:end] .= M0.+M2.+M5.-M1
        return C
    end
end

function strassenNoRecurse(A::AbstractMatrix,B::AbstractMatrix)
    return strassenBase(A, B, mult)
end

function strassenRecurse(A::AbstractMatrix,B::AbstractMatrix)
    minSize = 500
    if(size(A,1)<minSize||size(A,2)<minSize||size(B,1)<minSize||size(B,2)<minSize)
        return mult(A,B)
    else
        return strassenBase(A,B,strassenRecurse)
    end
end

function strassenStripedNoRecurse(A::AbstractMatrix,B::AbstractMatrix)
    return strassenBase(A, B, blockedMult)
end

function strassenStripedRecurse(A::AbstractMatrix,B::AbstractMatrix)
    minSize = 500
    if(size(A,1)<minSize||size(A,2)<minSize||size(B,1)<minSize||size(B,2)<minSize)
        return blockedMult(A,B)
    else
        return strassenBase(A,B,strassenStripedRecurse)
    end
end

function strassenNaiveNoRecurse(A::AbstractMatrix,B::AbstractMatrix)
    return strassenBase(A, B, naiveMult)
end

function strassenNaiveRecurse(A::AbstractMatrix,B::AbstractMatrix)
    minSize = 500
    if(size(A,1)<minSize||size(A,2)<minSize||size(B,1)<minSize||size(B,2)<minSize)
        return naiveMult(A,B)
    else
        return strassenBase(A,B,strassenNaiveRecurse)
    end
end

function strassenOptimalRecurse(A::AbstractMatrix,B::AbstractMatrix)
    minSize = 200
    if(size(A,1)<minSize||size(A,2)<minSize||size(B,1)<minSize||size(B,2)<minSize)
        return mult(A,B)
    else
        return strassenBase(A,B,strassenOptimalRecurse)
    end
end

function strassenRecurse16(A::AbstractMatrix,B::AbstractMatrix)
```

```
    minSize = 16
    if(size(A,1)<minSize||size(A,2)<minSize||size(B,1)<minSize||size(B,2)<minSize)
        return mult(A,B)
    else
        return strassenBase(A,B,strassenRecurse16)
    end
end

function strassenRecurse16Naive(A::AbstractMatrix,B::AbstractMatrix)
    minSize = 16
    if(size(A,1)<minSize||size(A,2)<minSize||size(B,1)<minSize||size(B,2)<minSize)
        return naiveMult(A,B)
    else
        return strassenBase(A,B,strassenRecurse16Naive)
    end
end
```

Appendix B:Data

All data was created by taking the minimum of 20 random $n \times n$ matrix multiplications. This is to find the example where outside CPU usage has the smallest effect. To find the normalized values used in the graphs, simply divide each entry by the proper entry for Base at the same size

Ints

Size	32	64	128	256	512	1024	2048
Base	1E-05	9.1E-05	0.000691	0.005612	0.044208	0.34949	2.861422
Recursive 16	0.000102331	0.000712348	0.004977969	0.037602414	0.266144363	1.963810172	13.71453539
Recursive Naive Mult	2.3384E-05	0.000147264	0.001178454	0.009021309	0.067146637	0.493571725	3.677466333
One-Layer Base Mult	2.1561E-05	8.9256E-05	0.000690817	0.005125127	0.039886945	0.313542969	2.526809222
One-Layer Naive Mult	6.2477E-05	0.000198359	0.001144202	0.008759832	0.066471679	0.580822485	4.833306747
Naive Mult	2.4E-05	0.000148	0.001167	0.009072	0.080077	0.67268	7.962451
Blocked Mult	2.6219E-05	0.000168856	0.001220168	0.009378975	0.076408877	0.613799129	4.968098872
One-Layer Blocked Mult	6.7506E-05	0.00022525	0.001287823	0.008963681	0.068479216	0.556513717	4.392551399
Recursive Blocked Mult	2.7231E-05	0.000167132	0.001215028	0.009380399	0.06849242	0.509115499	3.665025811
Recursive Base Mult	1.025E-05	8.9807E-05	0.000697299	0.005516729	0.039887895	0.28883987	2.152823764
Recursive 16 Naive Mult	0.00026314	0.001834953	0.012998918	0.094350584	0.663852429	4.820723668	33.38543509
Recursive 340 Base Mult	1.0399E-05	9.124E-05	0.000694497	0.005549739	0.040462356	0.292049804	2.209768401

Floats

base	3.03E-06	1.71E-05	0.000121687	0.000346627	0.002062359	0.014239755	0.10493753
One-layer	1.58E-05	4.17E-05	0.000176219	0.000943621	0.003343702	0.021136152	0.185625871
Recursive	3.42E-06	1.87E-05	9.74E-05	0.000353741	0.003800995	0.033190224	0.395230583