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.

The BLAS Library is a

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

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

This split uses 8 multiplications of size $\frac{n}{2}$, which produces the typical runtime of $O(n^3)$.
However, if we instead create 7 temporary matrices, $M_0, \dots, M_6$, with

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

and add and subtract ($O(n^2)$) these block matrices to create the parts of C as follows

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

This works because (ignoring the α and C_{ij} since they obviously work out simply), substitution of these equations yields

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

What this is effectively doing is using the linear dependence of the original block matrix equations to turn a multiplication into several extra additions. Note that this can be done recursively to give $O(n) = 7O(\frac{n}{2}) + c * n^2$ for some constant c. By the master equation, this is $O(n) = n^{\log_2 7} \approx n^{2.8}$.

Implimentation

Our code implements 3 main methods for multiplication: naiveMult, which is a fairly well optimized n^3 multiplication algorithm. For small matrices, this is within a 1.5x time of the highly optimized C library, BLAS. For larger matrices, however this is less optimized with respect to cache usage.
Our second is a 1 level strassen multiplication which can use either Julia's default multiplication (blas), or naiveMult. When using naiveMult this doesn't catch up to BLAS, but if julia's mult is called for the small multiplications, it is able in many cases ($n = m > 200$).
The third is a recursive Strassen implementation that splits until one of the dimmensions of the arrays gets smaller than a constant. This strategy is very slow due to excessive allocation, but once the matrices get large enough ($m = n > 2000$), it starts to out-perform both base and the 1 level algorithm.

Apendix A: 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

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)
    m = size(A,1)
    n = size(B,2)
    @boundscheck size(A,2) == size(B,1)
    TS = promote_op(matprod, eltype(A), eltype(B))
    mr = 8
    nr = 4

    C = zeros(TS, m, n)
    tempC = Matrix{TS}(undef, mr, nr)
    tempA = Matrix{TS}(undef, mr, size(A,1))
    tempB = Matrix{TS}(undef, nr, size(B,2))
    #tempC = SMatrix{mr, nr, TS}
    #tempA = SMatrix{mr, m, TS}
    #tempB = SMatrix{nr, n, TS}
    return blockedMult!(C, A, B, tempA, tempB, tempC, m,n,Val{mr}(), Val{nr}())
end


function blockedMult!(C, A, B, tempA, tempB, tempC,m,n, ::Val{mr}, ::Val{nr}) where {mr, nr}
    @inbounds for i in 1:mr:size(A,1)                # iterates over cols of A (rows of B)
        tempA .= (@view A[i:i+mr-1,:])
        for j in 1:nr:size(B,2)                      # iterates over rows of begin
            tempB .= (@view B[:,j:j+nr-1])'
            #tempC = SizedMatrix{mr,nr}(C[i:i+mr-1,j:j+nr-1]) #Load C into registers
            tempC .= @view C[i:i+mr-1,j:j+nr-1] #Load C into registers
            microkernel(tempA,tempB,tempC,Val{mr}(), Val{nr}())
            C[i:i+mr-1,j:j+nr-1] .= tempC
        end
    end
    return C
end

function microkernel(tempA,tempB,tempC,::Val{mr}, ::Val{nr}) where {mr, nr}
    @inbounds for p in 1:size(tempA,2)               # Rank 1 updates
        #poff = (p-1)*mr
        #for jj in 1:nr
        #    b = tempB[jj,p]
        #    joff = (jj-1)*mr
        #    @simd for ii in 1:mr
        #        tempC[ii+joff] += tempA[ii+poff] * b
        #    end
        #end
    end
end

function pad!(A,B,m,n,p)
    if p%2 == 1
        if n%2 ==1
            Anew=zeros(eltype(A), n+1,p+1)
            Anew[1:end-1,1:end-1] .= A
            A = Anew
        else
            Anew=zeros(eltype(A), n,p+1)
            Anew[1:end,1:end-1] .= A
            A = Anew
        end
        if m%2==1
            Bnew=zeros(eltype(B), p+1, m+1)
            Bnew[1:end-1,1:end-1] .= B
            B = Bnew
        else
            Bnew=zeros(eltype(B), p+1, m)
            Anew[1:end,1:end-1] .= B
            B = Bnew
        end
    else
        if n%2 ==1
            Anew=zeros(eltype(A), n+1, p)
            Anew[1:end-1,1:end] .= A
            A = Anew
        end
        if m%2==1
            Bnew=zeros(eltype(B), p, m+1)
            Bnew[1:end,1:end-1] .= B
            B = Bnew
        end
    end
    return A,B
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)

    a1,a2 = (div(n,2),div(p,2))
    b1,b2 = (div(p,2),div(m,2))

    A, B = pad!(A,B,m,n,p)
    TS = promote_op(matprod, eltype(A), eltype(B))
    C = zeros(TS, size(A,1), size(B,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,2),div(p,2))
    b1,b2 = (div(p,2),div(m,2))
    @inbounds @fastmath @views begin
        A00 = A[1:a1,1:a2]
        A10 = A[a1+1:end,1:a2]
        A01 = A[1:a1,a2+1:end]
        A11 = A[a1+1:end,a2+1:end]

        B00 =B[1:b1,1:b2]
        B10 =B[b1+1:end,1:b2]
        B01 =B[1:b1,b2+1:end]
        B11=B[b1+1:end,b2+1:end]

        M0=mult(A00.+A11, B00.+B11)
        M1=mult(A10.+A11, B00)
        M2=mult(A00,      B01.-B11)
        M3=mult(A11,      B10.-B00)
        M4=mult(A00.+A01, B11)
        M5=mult(A10.-A00, B00.+B01)
        M6=mult(A01.-A11, B10.+B11)

        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 = 501
    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 testMults(N::Int64,T)
    slowValid=0
    stupidValid=0
    strassenValid=0
    A = rand(T,N,N)
    B = rand(T,N,N)

    #precompile
    x = rand(T,16,16)
    mult(x,x)
    naiveMult(x,x)
    println("b")
    blockedMult(x,x)
    #@code_warntype blockedMult!(x,x,x)
    strassenNoRecurse(x,x)
    strassenRecurse(x,x)

    # when btime ing use $A etc)
    #C2=@time strassenNoRecurse(A,B)
    C  = @time mult(A,B)
    C3 = @time blockedMult(A,B)
    C1 = @time naiveMult(A,B)
    C2 = @time strassenNoRecurse(A,B)
    #C3=@btime strassenRecurse($A,$B)
    for i in 1:N
        for j in 1:N
            slowValid = max(slowValid, abs(C[i,j]-C1[i,j]))
            stupidValid = max(stupidValid, abs(C[i,j]-C2[i,j]))
            strassenValid = max(strassenValid, abs(C[i,j]-C3[i,j]))
        end
    end

    println(slowValid)
    println(stupidValid)
    println(strassenValid)
end

testMults(400, Int)
```