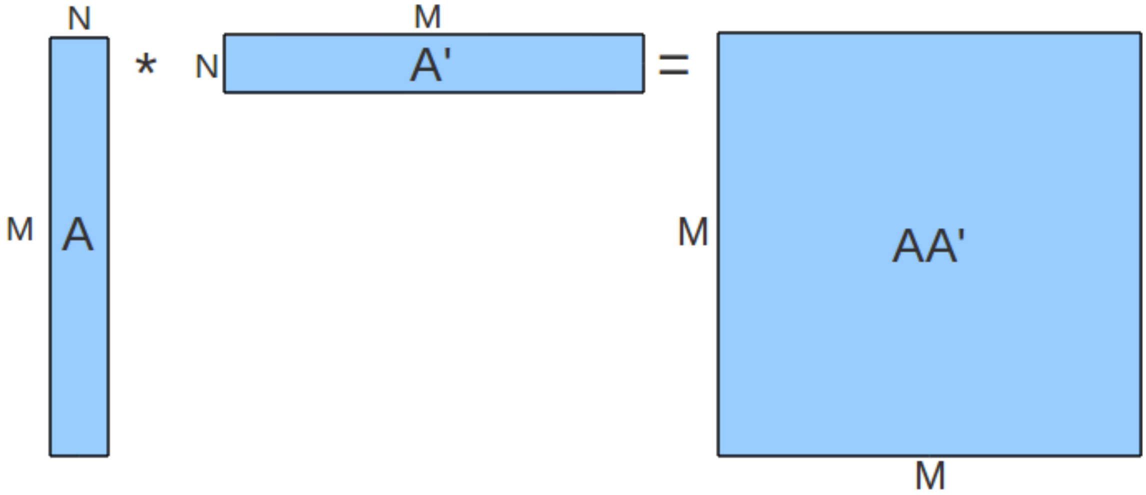
Matrix Multiply

You (plus optional teammate) are tasked with the job of making the fastest matrix multiplication program as possible for all machines. That means you cannot specifically target a machine. But you are free to research and find all usual architectures specification for personal and server machines. You may assume that everything is Intel architecture (x86_64) to make life easier.

The matrix is column major. Naïve implementation is given in sgemm-naive.c and you can run the bench-naive to see the output.

```
void sgemm( int m, int n, int d, float *A, float *C )
{
   for( int i = 0; i < n; i++ )
      for( int k = 0; k < m; k++ )
         for( int j = 0; j < n; j++ )
            C[i+j*n] += A[i+k*(n)] * A[j*(n+1)+k*(n)];
}
```

Where m is the size of the strip you are multiplying by, n is the height of the matrix, A is the original matrix, and C is the matrix resulting matrix. This is a special form of Matrix Multiplication where the original Matrix A is multiplied by its transposed A' to create C like the following picture shows.



This admittedly results in a symmetric matrix. However, you are **required to do all the calculations** and no optimization is allowed on this front to make benchmarking easier. Proj2.tgz contains the following files :

Makefile: generates all the files as required
benchmark.c: runs different size matrices to get performance numbers
sgemm-naive.c: contains 4 lines of code as shown in reference
sgemm-optimize.c: where your code goes that emulates functionality of sgemm-naive but albeit much faster

Your code must be able to handle small matrices of size and big sizes and optimization maybe different for them. One reason is that for small sizes, the matrices might not have any capacity misses in the cache and stay resident for the duration of the calculations. Some of the possible optimizations for this project are as follows:

- Reordering of instructions (compiler peep-hole optimization)
- Register blocking (reusing the same registers for multiple calculations)
- Cache optimizations
    - Blocking (trying to keep the data in the cache for large matrices)
    - Copying small matrix blocks into contiguous chunks of memory
    - Pre-compute transpose for spatial locality
- Loop optimizations
    - SSE instructions
    - Reordering
    - Unrolling
- Padding Matrices (odd sizes can hurt pipeline performance)

Anything else you can find or can think of is fine to increase performance. Just remember to calculate all the results (copying from one part of the resulting matrix to another is not allowed).

Your solution will be ran across different machines and the results aggregated