Project 3

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.

Background Reading:

Chapter 4.12

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

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

C is where the result is stored and we are doing all the calculations from just one matrix, A. You are **required to do all the calculations** and no optimization is allowed on this front to make benchmarking easier. Zip contains the following files :

Makefile: to make and benchmark
benchmark.c: do not modify. It check results and produce performance numbers
dgemm-naive.c: naïve implementation as shown above
dgemm-optimize.c: your optimization

Choose at most 3 of the following common optimizations (1 per function). The project worths 100 points, any extra points will go into your final exam as an extra credit.

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

You should not use any libraries for parallel computing, such as openMP. You may assume multiple cores. 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.

Note that you should not optimize just for your computer or one particular matrix size. Use your knowledge of computer architecture with all the modern features that tries to accelerate execution. Caches will play a big role but it is not safe to assume a particular architecture. But in general, optimizing memory accesses will lead to big gains. Matrix size and corner cases will also matter, as the same optimization will not work across the board. Have fun with this project.