Our first step to optimizing the program was to jump directly into SSE by using the example script already written in Bitbucket. The hope was that the SSE code was a fast enough kernel that we can build a good multiplier around it; indeed, it an immediate speed-up was noticed! It approximately doubled our naive multiplication. See figure 1.

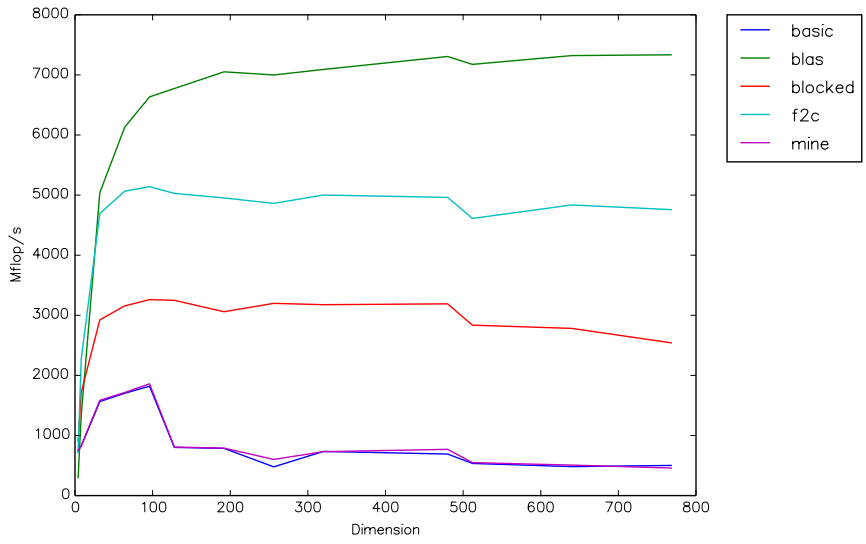

Figure 1: Our initial set-up using the SSE code provided. Note that due to laziness, the "blocked" line is actually our matrix multiply.

What the code does is fairly basic:

1. Intialize arrays At, Bt.
2. Takes in matrix A, B and copies them into At, Bt such that we have memory layout of 2 by P blocks in A and P by 2 blocks in B.
3. Loop over the blocks in A and B and multiply, adding the 2 by 2 matrix from the output into the C matrix.

We originally compiled this SSE code via the Intel compiler, but weirdly we found that compiling it with the GCC compiler results in a roughly 33% speedup over the Intel compiler! We're actually not sure why, but various flags of the ICC doesn't seem to coax the same performance out of the code.
This was all tested on even sizes intially, as there was no need to pad the matrices with anything. The odd cases are handled in the natural way, which was to pad a row and column and take the slight hit to the performance. When implemented, our performance was still pretty decent. See figure 2.
The next step to optimizing this was to play with the flags and use tiny code tricks.

Tricks

One trick that we played with was to do some loop reordering. In our main block, we have the following

```
for (bi = 0; bi < n_blocks; ++bi) {
  const int i = bi * BLOCK_SIZE;
  for (bj = 0; bj < n_blocks; ++bj) {
    const int j = bj * BLOCK_SIZE;
    // trans is our ``transitional'' matrix which contains both At and Bt
    kdgemm2P2(n, tempC, &trans[bi*n*2], &trans[temp + bj*n*2]);
```

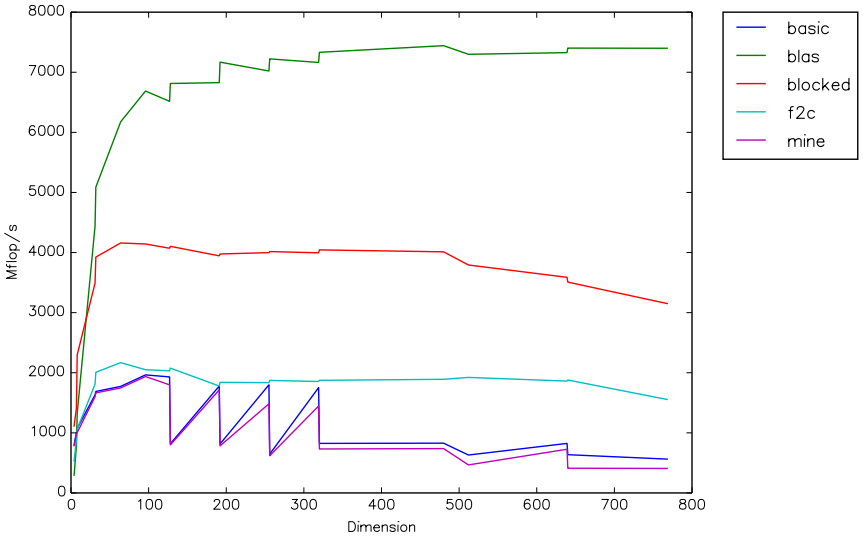

Figure 2: After adding the odd cases in (note the "blocked" line is ours).

Size	MFlops (flipped)	MFlops (as shown)
767	2660.82	3516.27
768	2348.95	3586.11
769	2229.79	3503.37

Table 1: Table of loop ordering

```
    }
  }
```

We noted that flipping bi and bj's loop order hurts the performance a lot. For example, at the far end of the spectrum of the sizes the different is quite noticable. See table 1, so we determined that

Compiler Flags