1 Analysing x86 assembly generated by various compilers

Benchmark compiled using -m32 flag using various compilers using optimization level -O3.

1.1 GCC

No significant optimizations. Same as the source code.

1.2 Clang

Program order doesn't change except that the it uses vector instructions to perform multiple computations at a time both in the step where `C[i][j]` is multiplied by `beta` and the inner multiplication loop. Consider the following four operations:

```
C[i][0] += alpha * A[i][k] * B[k][0];
C[i][1] += alpha * A[i][k] * B[k][1];
C[i][2] += alpha * A[i][k] * B[k][2];
C[i][3] += alpha * A[i][k] * B[k][3];
```

Clang loads the 4 integers from `B[k][0] to B[k][3]` to a vector and uses vector multiplication to compute all the products at the same time. Similarly does all four additions using vector instruction. Which gives significant speedup.

1.3 ICC

Uses larger registers to multiply multiple `C[i][j]` with `beta` at a single time. But fails to vectorize main multiplication step.

1.4 Clang with polly

Performs a check if the matrices overlap with each other. If they don't then perform tiling optimization. Code is as follows

```
for(int i = 0; i < ___; i += 32){
    for(int j = 0; j < ___; j += 32){
        vector instructions for multiplying C by alpha
        for(int k = 0; k < ___; k += 512){
            vector instructions to multiply A,B
        }
    }
}
```

1.5 Running times

Compiler	Time(seconds)
Clang	4.033
ICC	7.512
GCC	7.646
Clang-Polly	3.635

We can see that vectorizing the main matrix multiplication part gives significant improvement (7.512s(ICC) $\rightarrow$ 4.033(Clang)) whereas vectorizing multiplication of `C[i][j]` with `beta` has very little effect (7.646s(GCC) $\rightarrow$ 7.512s(ICC)). We can also see that tiling optimization improves the running time further (4.033s(Clang) $\rightarrow$ 3.635s(Clang-Polly)).

Speedup provided by using vector instructions is substantial compared to speedup provided by tiling optimization. L1 cache of the machine I tested on is almost twice compared to cache line size multiplied by length of a column of a matrix which is the reason why non-tiled version is also almost as fast as tiled version.

1.6 Writing a faster implementation

Fastest time obtained using "-msse4.1" flag is for clang with polly. The time is 2.264s. This has tiling and uses vector instructions too. None of the changes I tried to make reduced the run-time. I hand-wrote tiled matrix multiplication and tried various tile sizes and hand wrote the parts of the computation that can be vectorized. The implementation was still slower than the code generated by polly.

2 What Polly does in the case of gemm

Clearly, Polly can figure out the commutativity of the following statement when the matrices A,B,C donot overlap in memory.

```
C[i][j] += alpha * A[i][k] * B[k][j]
```

Using this information, Polly does a tiling transformation which is more cache efficient i.e., does a lot more computation using an array entry before it is evicted from the cache.

3 Limitations of current Polly Framework

Polly frame work doesn't seem to check if further optimizations are possible over a transformed program resulting in loss of performance. I will explain this problem using following example:

```
for(int i = 0; i < 10000; i++){
    for(int j = 0; j < 1000000; j++){
        A[i] += 1;
    }
}

for(int j = 0; j < 1000000; j++){
    for(int i = 0; i < 10000; i++){
        A[i] += 1;
    }
}
```

In the first loop, the entire inner-loop can be removed by replacing it with the statement `A[i] += 1000000`. This is done by all the compilers when compiled using `-O3` flag.

The second loop is equivalent to the first loop and polyhedral frameworks like polly do loop-reversal. But in this case, Polly doesn't do loop reversal as it cannot identify that the reversed loop can be further optimized by removing a loop. Instead, it does tiling which is more cache efficient. Code generated by polly in 'C' style syntax is as follows (It vectorizes the inner-loop execution.)

```
for(int j = 0; j < 31250; j++){
    for(int i = 0; i < 313; i++){
        for(int jj = 0; jj < 32; jj++){
            for(int ii = i*32; ii < min(i*32+32,10000); ii++){
                A[ii] += 1;
            }
        }
    }
}
```

Though this is better than the naive incrementation, it is significantly worse than eliminating the loop.

This can be overcome by using beam-search like procedure when using polly. Here, we keep 't'('t' can be 2) best transformations for the program that polly can generate. Then optimize both this transformations and keep best 't' programs again and so on. This will increase the run-time of the optimizaiton phase by t^2 (4, when t = 2) times but remedies the problem of missing further optimizations.