Evolution of performance

CPU Benchmark (Floops)

Approach 1

A naive approach would be to have a for loop with random floating point calculations and record the time it takes to finish the computation. The code for this would look something like this

```
for (int i = 0; i < loopcount; i++) {
    x = a*x + c;
}
```

This is clearly inefficient since it does not take advantage of either parallelism with SIMD or with multi threading. As expected the resulting GFLOPS are 1.69 GFLOPS, much lesser than what can be achieved.

Approach 2

We can optimize the previous approach by asking telling the compiler that there are no dependencies and it can be vectorized. While this produces incorrect output, we will achieve out goal of using our CPU to its maximum potential. Adding a small change to our code we get an $8\times$ improvement.

```
#pragma omp simd
for (int i = 0; i < loopcount; i++) {
    x = a*x + c;
}
```

We achieve a total of 13.4 GFLOPS using vector instructions. Running this code with Intel Advisor gives us the suggestion to compile with -xCORE-AVX2 option. I'm assuming this is since it defaults to using avx instructions set instead of avx2 supported by my CPU. Using the option doesn't work, I'm guessing its because its using instructions specific to the Intel architecture. Using `gcc` we get no improvement at all, in fact it does not vectorize either.

```
icc flags = -g -O3 -mfma -mavx2 -qopenmp
gcc flags = -g -O3 -mfma -mavx2 -fopenmp -ftree-vectorize
```

Approach 3

Since we `gcc` cannot vectorize the code, I came up with something that can be vectorized.

```
int vec_size = 8;
for (int i =0; i < loopcount; i++)
{
    for (int i = 0; i < vec_size; i++)
    {
        #pragma code_align 32
        Y[i] = a * X[i] + Y[i];
    }
}
```

With the same optimizations as above with the `icc` compiler we achieve a total of 44.7 GFLOPS, which is a $4\times$ improvement over the last iteration. With `gcc` however the results are more interesting, with the same flags as above we get 22.5 GFLOPS, which is around $16\times$ improvement over the last iteration. Analyzing the assembly we can see that the lower performance of `gcc` compiled version is due to the absence of `fma` instructions. After doing a bit of research, turns out we need to functions such as `_mm_set1_ps`, `_mm_load_ps`, etc for `gcc` to compile them into `fma` instructions. 1

Approach 4

Parallelization!

```
#pragma omp parallel for
for (long i = 0; i < loopcount; i++)
{
    #pragma code_align 32
    for (int j = 0; j < vec_size; j++)
    {
        Y[j] = a * X[j] + Y[j];
    }
}
```

Using `icc` we achieve around 350 GFLOPS, which is around $8\times$ improvement over the last iteration. This is speedup matches the expectations from using 8 cores on my CPU instead of just 1. However this isn't the max GFLOPS capable since `icc` doesn't use the `AVX2` supported on AMD processors.

Approach 5

Due to some unknown compiler optimizations, the above code wasn't running as expected with `gcc` so I run the benchmark sixteen times in parallel. So the loop now looks like

```
#pragma omp parallel for
for (int i = 0; i < omp_get_max_threads(); i++)
{
    for (long i = 0; i < loopcount; i++)
    {
        #pragma code_align 32
        for (int j = 0; j < vec_size; j++)
        {
            Y[j] = a * X[j] + Y[j];
        }
    }
}
```

This gives around 400 GFLOPS with both `gcc` and `icc` compiler. Using `FMA` however brings this down to 120 GFLOPS.

Memory Benchmark (Geeps)

The idea for memory benchmark is pretty straight forward. Read from as memory as much as possible, but also make sure that everything read is being used so compiler doesn't optimize it out.

```
 for (int i = 0; i < num_words; i++)
{
    e = b[i] - a[i];
}
```

I accumulated the difference `b[i]-a[i]` in e so that compiler doesn't remove the loop and also the time taken by the operation should be negligible. However this does not draw enough bandwidth since its on a single thread. Adding `#pragma omp parallel for` makes the code multi threaded successfully drawing the maximum bandwidth of 33.5 GB/s.

BLAS Problems

xSCAL

For this problem a single loop is sufficient.

```
for (int i = 0; i < N; i++)
{
    X[i * incX] *= alpha;
}
```

This code is vectorized automatically for stride 1 with the -O3 flag. To improve performance we could try parallelizing this loop. Interestingly adding OpenMP pragma's has no effect on performance, since it only uses one thread.

xDOT

Like the other BLAS Level 1 problems, its a for loop that only benefits from O3 and Vectorization. OpenMP has no effect on this function.

```
float dot = 0.0;
for (int i = 0; i < N / (incX > incY ? incX : incY); i++)
{
    dot += X[i * incX] * Y[i * incY];
}
return dot;
```

xAXPY

```
for (int i = 0; i < N / (incX > incY ? incX : incY); i++)
{
    Y[i * incY] += alpha * X[i * incX];
}
```

Like the other BLAS Level 1 problems, its a for loop that only benefits from `-O3` and Vectorization. `OpenMP` has no effect on this function.

xGEMV

We have to deal with different cases in this problem.

1. Matrix is row major and not Trans or Matrix is column major and is Trans
   In this case our loop looks like

   ```
   for (int i = 0; i < lenY; i += incY)
   {
       float temp = 0.0;
       for (j = 0; j < lenX; j += incX)
       {
           temp += X[j] * A[lda * i + j];
       }
       Y[i] += alpha * temp;
   }
   ```

   This code is auto vectorized with the `-O3` flag. Vectorization in this case provides a speedup of up to $4-5\times$. However this can be further optimized using `OpenMP` pragmas. The outer loop can be parallelized by adding `#pragma omp parallel for`.

   ```
   #pragma omp parallel for
   for (int i = 0; i < lenY; i += incY)
   {
       float temp = 0.0;
       for (j = 0; j < lenX; j += incX)
       {
           temp += X[j] * A[lda * i + j];
       }
       Y[i] += alpha * temp;
   }
   ```

   This provides a speedup of $7-8\times$.

2. Matrix is row major and is Trans or Matrix is column major and is not Trans
   In this case our loop looks like

   ```
   for (j = 0; j < lenX; j += incX)
   {
       const float temp = alpha * X[j];
       for (i = 0; i < lenY; i += incY)
       {
           Y[i] += temp * A[lda * j + i];
       }
   }
   ```

   With `-O3` flag, our code is auto vectorized an provides a speed $8-10\times$. Due to loop dependency we cannot parallelize the outer loop but we can try parallelizing the inner loop, but this however slows down our process by $4\times$. However limiting the number of threads to 2 gives us a performance boost of $1.5\times$ our previous best. Our final code looks like

```
for (j = 0; j < lenX; j += incX)
{
    const float temp = alpha * X[j];
    #pragma omp parallel for num_threads(2)
    for (i = 0; i < lenY; i += incY)
    {
        Y[i] += temp * A[lda * j + i];
    }
}
```

xGEMM

The optimizations in all cases pretty much the same.

In case of all matrices being not transposed, we have this for loop for our computation.

```
for (int k = 0; k < K; k++)
{
    for (int i = 0; i < t1; i++)
    {
        const float temp = alpha * X[ldx * i + k];
        for (int j = 0; j < t2; j++)
        {
            C[ldc * i + j] += temp * Y[ldy * k + j];
        }
    }
}
```

This code is auto vectorized when using `-O3` flag giving a speedup of $13.5\times$. However this code also benefit from multi threading, by using OpenMP pragma

```
for (int k = 0; k < K; k++)
{
#pragma omp parallel for
    for (int i = 0; i < t1; i++)
    {
        const float temp = alpha * X[ldx * i + k];
        for (int j = 0; j < t2; j++)
        {
            C[ldc * i + j] += temp * Y[ldy * k + j];
        }
    }
}
```

This gives a further speedup of $5.25\times$

2D Stencil

The code for stencil is pretty straightforward.

```
for (i = 0; i < dimY; i++)
{
    for (j = 0; j < dimX; j++)
    {
        Y[i * dimX + j] = 0.0;
```

```
            for (kx = 0; kx < k; kx++)
            {
                if (i + kx >= dimY)
                    break;
                for (ky = 0; ky < k; ky++)
                {
                    if (j + ky >= dimX)
                        break;
                    Y[i * dimX + j] += X[(i + kx) * dimX + (j + ky)] * S[kx * k +
ky];
                }
            }
        }
    }
```

This is auto vectorized with `-O3` flag and this gives a performance of boost of $2-3\times$. This code does benefit from parallelization but not as much as the other problems here due to loop dependencies in it. Only the outer loop can be parallelized which gives a performance boost of $4\times$.