Lab 1: Matrix Multiplication

姜海天 19307110022

This report is to record the performance optimization of an matrix multiplication problem.

In this problem, we are going to calculate $\alpha \cdot AB + \beta \cdot C$, where $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}, C \in \mathbb{R}^{m\times n}$, and store the answer in C.

0.1 Testing Environment

All the experiments are conducted on a server equipped with dual 64-core, 128-thread `AMD EPYC 7742 64-Core` CPU. The peak double-precision performance for a single CPU is 3.48 Tflops. L1d cache: 4MiB, L1i cache: 4MiB, L2 cache: 64MiB, L3 cache: 512MiB

The compiler used for the experiments is `clang++ 14.0.6`.

All the test uses `./executable 4096 4096 4096` as the test command.

0.2 Computation Amount

If use the algorithm stated in the naive implementation, there will be $mn + 3mnk$ floating-point operations. If $m = n = k = 4096$ in my test case, there is 206.2G floating-point operations. So the theoretical minimum computing time is $206.2G/(2 * 3.48T/s) = 30ms$.

0.3 Main Results

Optimization	Running Time(s)	Relative Speedup	Absolute Speedup	Percent of Peak
Naive	2246.3	1	1	0.0013%
+ interchange loops	224.76	10	10	0.013%
+ optimization flags	38.85	5.78	57.8	0.077%
+ reduced computation	32.36	/	69.4	0.062%
Parallel loops	1.40	4.2	1609	2.15%
+tiling	0.34	27.8	6691	8.94%
+compiler vectorization	0.32	1.04	6941	9.27%

1 Naive Implementation

We use the naive implementation without any compiler optimization as a baseline.

```cpp
void student_gemm(int m, int n, int k,
                  const double * A, const double * B, double * C,
                  double alpha, double beta,
                  int lda, int ldb, int ldc) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            C[i + j * ldc] *= beta;
            for (int p = 0; p < k; p++) {
                C[i + j * ldc] += alpha * A[i + p * lda] * B[p + j * ldb];
            }
        }
    }
}
```

The computing time for the naive method is **37min26s (2246252.1556ms)**.

Speedup: **1x**, percent of peak: **0.0013%**.

```
$ ~/opencilk/bin/clang++ gemm.cpp -o 1-naive
$ ./1-naive-1 4096 4096 4096
input: 4096 x 4096 x 4096
minimal time spent: 2246252.1556 ms
result: correct (err = 0.000000e+00)
```

The cache miss rate for the naive method: **4.7%** by the `valgrind` command.

2 Loop Order

Change the loop order from `i,j,p` in the naive implementation to `j,p,i` so that we have the locality for columns, since the matrices are stored in column major.

```cpp
void student_gemm(int m, int n, int k,
                  const double * A, const double * B, double * C,
                  double alpha, double beta,
                  int lda, int ldb, int ldc) {
    // compute β·C first
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            C[i + j * ldc] *= beta;
        }
    }
    for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p++) {
            // put the whole column for C and A in cache
            for (int i = 0; i < m; i++) {
                C[i + j * ldc] += alpha * A[i + p * lda] * B[p + j * ldb];
            }
        }
    }
}
```

The computing time for the better loop order is **3min44s (224764.6316ms)**.

Speedup: **10x**, percent of peak: **0.013%**.

```
$ ~/opencilk/bin/clang++ gemm.cpp -o 2-loop-order
$ ./2-loop-order 4096 4096 4096
input: 4096 x 4096 x 4096
minimal time spent: 224764.6316 ms
result: correct (err = 0.000000e+00)
```

The cache miss rate for the naive method: **1.1%** by the `valgrind` command.

3 Compiler Optimization

The most aggressive optimization `-O3` got the best performance, and the running time is only **38.8s (38847.4110ms)** now.

Speedup: **57.8x**, percent of peak: **0.077%**.

```
$ ~/opencilk/bin/clang++ gemm.cpp -o 3-O1 -O1
$ ./3-O1 4096 4096 4096
input: 4096 x 4096 x 4096
minimal time spent: 44184.6329 ms
result: correct (err = 0.000000e+00)

$ ~/opencilk/bin/clang++ gemm.cpp -o 3-O2 -O2
$ ./3-O2 4096 4096 4096
input: 4096 x 4096 x 4096
minimal time spent: 39349.4487 ms
result: correct (err = 0.000000e+00)

$ ~/opencilk/bin/clang++ gemm.cpp -o 3-O3 -O3
$ ./3-O3 4096 4096 4096
input: 4096 x 4096 x 4096
minimal time spent: 38847.4110 ms
result: correct (err = 0.000000e+00)
```

4 Reduce Floating-point Operations

The current implementation calculates

$$c_{i,j} = \beta \cdot c_{i,j} + \sum_p \alpha \cdot a_{i,p} \cdot b_{p,j}$$

There are lots of unnecessary floating-pointoperations. The calculation with reduced floating-point operation is

$$c_{i,j} = \alpha \cdot \left(\frac{\beta}{\alpha} \cdot c_{i,j} + \sum_p a_{i,p} \cdot b_{p,j} \right)$$

Now the amount of floating-point operations have been reduced from $mn + 3mnk$ to $2mn + 2mnk$. However, it requires more read and write of memory. So there is trade-off.

This method will not be suitable for the further optimization aiming at reducing memory operations because it brings more memory operations natually.

Implementation:

```cpp
void student_gemm(int m, int n, int k,
                  const double * A, const double * B, double * C,
                  double alpha, double beta,
                  int lda, int ldb, int ldc) {
    double ratio = beta / alpha;
    // C ← (β/α)C
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            C[i + j * ldc] *= ratio;
        }
    }
    // C ← C + AB
    for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p++) {
            for (int i = 0; i < m; i++) {
                C[i + j * ldc] += A[i + p * lda] * B[p + j * ldb];
            }
        }
    }
    // C ← αC
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            C[i + j * ldc] *= alpha;
        }
    }
}
```

Experiment: The running time is **32.36s**.

Speedup: **69.4x**, percent of peak: **0.062%**. The percent of peak dropped because the amount of floating-point operations is reduced and more memory access is required.

```
$ ~/opencilk/bin/clang++ gemm.cpp -o 4-coeff-O3 -O3
$ ./4-coeff-O3 4096 4096 4096
input: 4096 x 4096 x 4096
minimal time spent: 32360.3723 ms
result: correct (err = 2.913225e-13)
```

5 Multi-Core Parallel Computation

Add parallelable for-loop for the reordered loop version. The version with excessive memory access doesn't perform better because there are too much write to memory, which is much slower than floating-point operations.

```
void student_gemm(int m, int n, int k,
                  const double * A, const double * B, double * C,
                  double alpha, double beta,
                  int lda, int ldb, int ldc) {
    cilk_for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            C[i + j * ldc] *= beta;
        }
    }
    cilk_for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p++) {
            for (int i = 0; i < m; i++) {
                C[i + j * ldc] += alpha * A[i + p * lda] * B[p + j * ldb];
            }
        }
    }
}
```

Experiment: The running time is **1.4s**.

Speedup: **1609x**, percent of peak: **2.15%**.

```
$ ~/opencilk/bin/clang++ gemm.cpp -fopencilk -O3 -o 5-cilk-lo
$ ./5-cilk-lo 4096 4096 4096
input: 4096 x 4096 x 4096
minimal time spent: 1395.9282 ms
result: correct (err = 0.000000e+00)
```

6 Tiling

I added the tiling size `s` as an optional command line parameter. The default one is 128, which is the emperical best size.

Implementation:

```c
// ...
int s = 128; // global default value for s
void student_gemm(int m, int n, int k,
                  const double * A, const double * B, double * C,
                  double alpha, double beta,
                  int lda, int ldb, int ldc) {
    cilk_for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            C[i + j * ldc] *= beta;
        }
    }
    cilk_for (int jh = 0; jh < n; jh += s) {
        int J = MIN(n-jh, s);  // in case n % s != 0
        cilk_for (int ih = 0; ih < m; ih += s) {
            int I = MIN(m-ih, s);  // in case m % s != 0
            for (int ph = 0; ph < k; ph += s) {
                int P = MIN(k-ph, s);  // in case p % s != 0
                for (int jl = 0; jl < J; ++jl) {
                    int j = jh + jl;
                    for (int pl = 0; pl < P; ++pl) {
                        int p = ph + pl;
                        for (int il = 0; il < I; ++il) {
                            int i = ih + il;
                            C[i + j * ldc] += alpha * A[i + p * lda] * B[p + j * ldb];
                        }
                    }
                }
            }
        }
    }
}
int main(int argc, const char * argv[]) {
    if (argc != 4 && argc != 5) {
        printf("Test usage: ./test m n k [s]\n");
        exit(-1);
    }
    int m = atoi(argv[1]);
```

```cpp
    int n = atoi(argv[2]);
    int k = atoi(argv[3]);
    if (argc == 5) {
        s = atoi(argv[4]);
    }
    // ...
}
```

Experiment: The best tiling step size is 128, and the running time is **0.3s(335.7088ms)**

Speedup: **6691x**, percent of peak: **8.94%**.

```
$ ~/opencilk/bin/clang++ gemm.cpp -fopencilk -O3 -o 6-tiling

$ ./6-tiling 4096 4096 4096 32
input: 4096 x 4096 x 4096
minimal time spent: 1455.8920 ms
result: correct (err = 0.000000e+00)

$ ./6-tiling 4096 4096 4096 64
input: 4096 x 4096 x 4096
minimal time spent: 708.2708 ms
result: correct (err = 0.000000e+00)

$ ./6-tiling 4096 4096 4096 128
input: 4096 x 4096 x 4096
minimal time spent: 335.7088 ms
result: correct (err = 0.000000e+00)

$ ./6-tiling 4096 4096 4096 256
input: 4096 x 4096 x 4096
minimal time spent: 554.6172 ms
result: correct (err = 0.000000e+00)

$ ./6-tiling 4096 4096 4096 512
input: 4096 x 4096 x 4096
minimal time spent: 1507.3377 ms
result: correct (err = 0.000000e+00)
```

7 Divide and Conquer

We only optimize for the case where the dimension is a power of 2 by recursively divide and conquer.

```c
#define ISPOWER(x) ((x) & (-(x))) == (x)
void mm_base(int m, int n, int k, double alpha,
            const double *A, const double *B, double *C,
            int lda, int ldb, int ldc) {
    for (int j = 0; j < n; j++) {
        for (int p = 0; p < k; p++) {
            for (int i = 0; i < m; i++) {
                C[i + j * ldc] += alpha * A[i + p * lda] * B[p + j * ldb];
            }
        }
    }
}
void mm_dac(int m, int n, int k, double a,
            const double *A, const double *B, double *C,
            int ldA, int ldB, int ldC) {
    assert(ISPOWER(m) && ISPOWER(n) && ISPOWER(k));
    if (MIN(m, n) <= DAC_THOLD || k <= DAC_THOLD) {
        mm_base(m, n, k, alpha, A, B, C, ldA, ldB, ldC);
    } else {
        int& R_A = m; int& R_B = k; int& R_C = m;
        int& C_A = k; int& C_B = n; int& C_C = n;
        #define X(M, r, c) (M + r * (R_ ## M / 2) + c * (ld ## M) * (C_ ## M / 2))
        cilk_spawn mm_dac(m/2, n/2, k/2, a, X(A,0,0), X(B,0,0), X(C,0,0), ldA, ldB, ldC);
        cilk_spawn mm_dac(m/2, n/2, k/2, a, X(A,0,0), X(B,0,1), X(C,0,1), ldA, ldB, ldC);
        cilk_spawn mm_dac(m/2, n/2, k/2, a, X(A,1,0), X(B,0,0), X(C,1,0), ldA, ldB, ldC);
                   mm_dac(m/2, n/2, k/2, a, X(A,1,0), X(B,0,1), X(C,1,1), ldA, ldB, ldC);
        cilk_sync;
        cilk_spawn mm_dac(m/2, n/2, k/2, a, X(A,0,1), X(B,1,0), X(C,0,0), ldA, ldB, ldC);
        cilk_spawn mm_dac(m/2, n/2, k/2, a, X(A,0,1), X(B,1,1), X(C,0,1), ldA, ldB, ldC);
        cilk_spawn mm_dac(m/2, n/2, k/2, a, X(A,1,1), X(B,1,0), X(C,1,0), ldA, ldB, ldC);
                   mm_dac(m/2, n/2, k/2, a, X(A,1,1), X(B,1,1), X(C,1,1), ldA, ldB, ldC);
        cilk_sync;
    }
}
```

However, the result is that the program cannot suck all CPU power due to the synchronization, and the best result is **521.0503ms** for `DAC_THOLD=128`.

8 Compiler Flags

```
$ ~/opencilk/bin/clang++ gemm.cpp -fopencilk -O3 -o 8-avx -march=native -ffast-math
$ ./8-avx 4096 4096 4096
input: 4096 x 4096 x 4096
minimal time spent: 323.6070 ms
result: correct (err = 4.973799e-14)
```

Fast math doesn't guarantee the `err` to be 0, but it is faster.

The running time is **0.32s**.

Speedup: **6941x**, percent of peak: **9.27%**.