[Opt Kodu] Zadanie 2

Autor rozwiązania: Maciej Sikora

Wersja podstawowa

Kod przekopoiowany z https://en.wikipedia.org/wiki/LU_decomposition i zgodnie z informacją od prowadzącego został pozostawiony sam fragment odpowiedzialny za wyliczanie LU faktoryzacji.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
// #include <x86intrin.h>
// #include <immintrin.h>

#define IDX(i, j, n) (((j)+ (i)*(n)))
int SIZE = 1000;
static double gtod_ref_time_sec = 0.0;



int LUPDecompose(double *A, int N) {

    int i, j, k;

    for (i = 0; i < N; i++) {
        for (j = i + 1; j < N; j++) {
            A[IDX(j,i,SIZE)] /= A[IDX(i,i,SIZE)];

            for (k = i + 1; k < N; k++)
                A[IDX(j,k,SIZE)] -= A[IDX(j,i,SIZE)] * A[IDX(i,k,SIZE)];
        }
    }

    return 0;  //decomposition done
}

double dclock()
{
    double the_time, norm_sec;
    struct timeval tv;
    gettimeofday(&tv, NULL);
    if (gtod_ref_time_sec == 0.0)
        gtod_ref_time_sec = (double)tv.tv_sec;
    norm_sec = (double)tv.tv_sec - gtod_ref_time_sec;
    the_time = norm_sec + tv.tv_usec * 1.0e-6;
```

```
    return the_time;
}

int main(){
    int i,j, iret;
    double dtime;
    double* matrix;

    matrix = malloc(SIZE*SIZE*sizeof(double));

    srand(1);

    for (i = 0; i < SIZE; i++)
    {
        for (j = 0; j < SIZE; j++)
        {
            matrix[IDX(i, j, SIZE)] = rand();
        }
    }

    printf("call LU-decomposition\n");
    dtime = dclock();
    iret = LUPDecompose(matrix, SIZE);
    dtime = dclock() - dtime;
    printf("Time: %le \n", dtime);

        double check = 0.0;
    for (i = 0; i < SIZE; i++)
    {
        for (j = 0; j < SIZE; j++)
        {
            check = check + matrix[IDX(i,j,SIZE)];
        }
    }
    printf("Check-> rv: %d, sum:%le \n", iret, check);
    fflush(stdout);

}
```

W wersji podstawowej została wprowadzona reprezentacja macierzy w przestrzeni ciągłej, a nie jako tablica pointerów do poszczególnych wierszy.

Optymalizacja 1

Wprowadzenie rejestrów na często używane wartości.

```
int LUPDecompose(double *A, int N) {
    register int i, j, k;
    register double divider;
    for (i = 0; i < N; i++) {
        divider = A[IDX(i,i,SIZE)];
        for (j = i + 1; j < N; j++) {
```

```
            A[IDX(j,i,SIZE)] /= divider;
            for (k = i + 1; k < N; k++)
                A[IDX(j,k,SIZE)] -= A[IDX(j,i,SIZE)] * A[IDX(i,k,SIZE)];
        }
    }
    return 0;
}
```

Optymalizacja 2

Ręczne wykonywanie obliczeń na 16 elementach.

```
int LUPDecompose(double *A, int N) {

    register int i, j, k;
    register double divider, tmp;

    for (i = 0; i < N; i++) {

        divider = A[IDX(i,i,SIZE)];

        for (j = i + 1; j < N; j++) {

            A[IDX(j,i,SIZE)] /= divider;

            for (k = i + 1; k < N; ){
                if(k+15 < N){
                    tmp = A[IDX(j,i,SIZE)];

                    A[IDX(j,k,SIZE)] -= tmp * A[IDX(i,k,SIZE)];
                    A[IDX(j,k+1,SIZE)] -= tmp * A[IDX(i,k+1,SIZE)];
                    A[IDX(j,k+2,SIZE)] -= tmp * A[IDX(i,k+2,SIZE)];
                    A[IDX(j,k+3,SIZE)] -= tmp * A[IDX(i,k+3,SIZE)];
                    A[IDX(j,k+4,SIZE)] -= tmp * A[IDX(i,k+4,SIZE)];
                    A[IDX(j,k+5,SIZE)] -= tmp * A[IDX(i,k+5,SIZE)];
                    A[IDX(j,k+6,SIZE)] -= tmp * A[IDX(i,k+6,SIZE)];
                    A[IDX(j,k+7,SIZE)] -= tmp * A[IDX(i,k+7,SIZE)];
                    A[IDX(j,k+8,SIZE)] -= tmp * A[IDX(i,k+8,SIZE)];
                    A[IDX(j,k+9,SIZE)] -= tmp * A[IDX(i,k+9,SIZE)];
                    A[IDX(j,k+10,SIZE)] -= tmp * A[IDX(i,k+10,SIZE)];
                    A[IDX(j,k+11,SIZE)] -= tmp * A[IDX(i,k+11,SIZE)];
                    A[IDX(j,k+12,SIZE)] -= tmp * A[IDX(i,k+12,SIZE)];
                    A[IDX(j,k+13,SIZE)] -= tmp * A[IDX(i,k+13,SIZE)];
                    A[IDX(j,k+14,SIZE)] -= tmp * A[IDX(i,k+14,SIZE)];
                    A[IDX(j,k+15,SIZE)] -= tmp * A[IDX(i,k+15,SIZE)];

                    k+=16;
                }else{
                    A[IDX(j,k,SIZE)] -= A[IDX(j,i,SIZE)] * A[IDX(i,k,SIZE)];
                    k++;
                }
            }
        }
```

```
    }
    return 0;
}
```

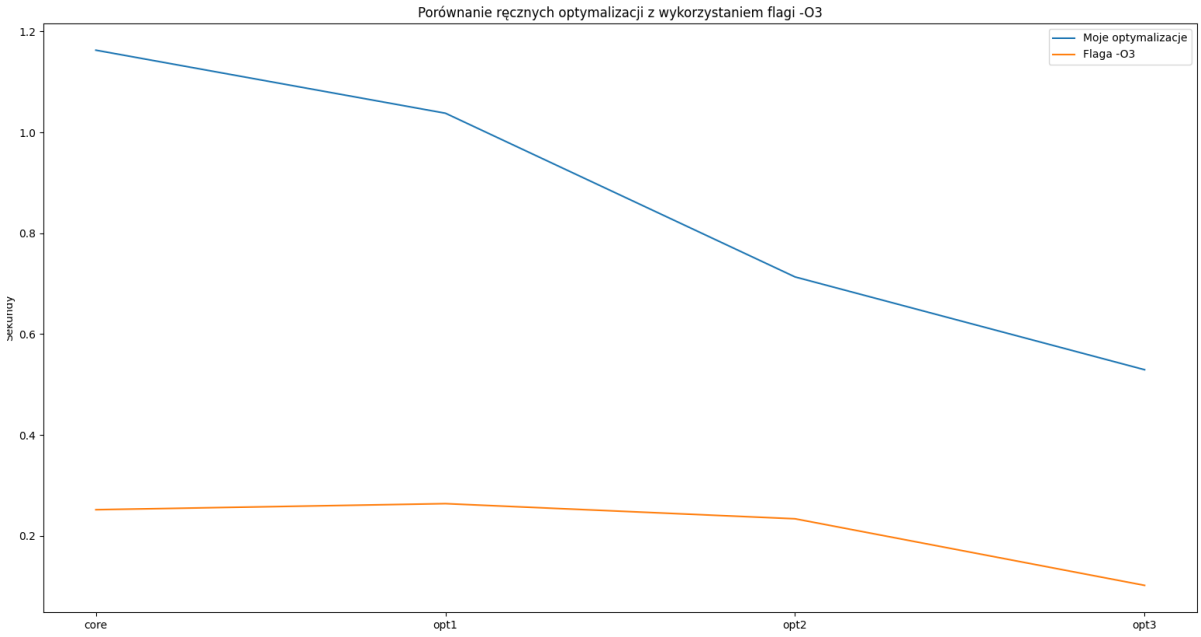
Optymalizacja 3

Wprowadzenie operacji wektorowych na 256 bitowych zmiennych wektorowych

```
int LUPDecompose(double *A, int N) {

    register int i, j, k;
    register double divider, tmp;
    register __m256d tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
    register __m256d mm_multiplier;

    for (i = 0; i < N; i++) {

        divider = A[IDX(i,i,SIZE)];

        for (j = i + 1; j < N; j++) {

            A[IDX(j,i,SIZE)] /= divider;
            tmp = A[IDX(j,i,SIZE)];
            mm_multiplier[0] = tmp;
            mm_multiplier[1] = tmp;
            mm_multiplier[2] = tmp;
            mm_multiplier[3] = tmp;

            for (k = i + 1; k < N; ){
                if(k+15 < N){
                    tmp0 = _mm256_loadu_pd(A+IDX(i,k,SIZE));
                    tmp1 = _mm256_loadu_pd(A+IDX(i,k+4,SIZE));
                    tmp2 = _mm256_loadu_pd(A+IDX(i,k+8,SIZE));
                    tmp3 = _mm256_loadu_pd(A+IDX(i,k+12,SIZE));

                    tmp4 = _mm256_loadu_pd(A+IDX(j,k,SIZE));
                    tmp5 = _mm256_loadu_pd(A+IDX(j,k+4,SIZE));
                    tmp6 = _mm256_loadu_pd(A+IDX(j,k+8,SIZE));
                    tmp7 = _mm256_loadu_pd(A+IDX(j,k+12,SIZE));

                    tmp0 = _mm256_mul_pd(tmp0, tmp4);
                    tmp1 = _mm256_mul_pd(tmp1, tmp5);
                    tmp2 = _mm256_mul_pd(tmp2, tmp6);
                    tmp3 = _mm256_mul_pd(tmp3, tmp7);

                    tmp0 = _mm256_sub_pd(tmp4, tmp0);
                    tmp1 = _mm256_sub_pd(tmp5, tmp1);
                    tmp2 = _mm256_sub_pd(tmp6, tmp2);
                    tmp3 = _mm256_sub_pd(tmp7, tmp3);

                    A[IDX(j,k,SIZE)]    = tmp0[0];
                    A[IDX(j,k+1,SIZE)]  = tmp0[1];
                    A[IDX(j,k+2,SIZE)]  = tmp0[2];
```

```
                    A[IDX(j,k+3,SIZE)]  = tmp0[3];
                    A[IDX(j,k+4,SIZE)]  = tmp1[0];
                    A[IDX(j,k+5,SIZE)]  = tmp1[1];
                    A[IDX(j,k+6,SIZE)]  = tmp1[2];
                    A[IDX(j,k+7,SIZE)]  = tmp1[3];
                    A[IDX(j,k+8,SIZE)]  = tmp2[0];
                    A[IDX(j,k+9,SIZE)]  = tmp2[1];
                    A[IDX(j,k+10,SIZE)] = tmp2[2];
                    A[IDX(j,k+11,SIZE)] = tmp2[3];
                    A[IDX(j,k+12,SIZE)] = tmp3[0];
                    A[IDX(j,k+13,SIZE)] = tmp3[1];
                    A[IDX(j,k+14,SIZE)] = tmp3[2];
                    A[IDX(j,k+15,SIZE)] = tmp3[3];
                    k+=16;
                }else{
                    A[IDX(j,k,SIZE)] -= tmp * A[IDX(i,k,SIZE)];
                    k++;
                }
            }
        }
    }
    return 0;
}
```

Porównanie z flagą -O3



Wnioski

Każda z wybranych przezemnie znaczących optymalizacji przyniosła pozytywny efekt na czas wykonania funkcji. Optymalizację 2 można dodatkowo ulepszyć

wprowadzając przypadki dla fragmentów 8 elementowych, 4 elementowych i 2 elementowych aby maksymalnie wykorzystać możliwości wektorów. Oczywiście przy dwóch elementach można skorzystać z mniejszych zmiennych wektorowych, ale nie jest to konieczne.

Najważniejszym punktem który można wynieść z tego ćwiczenia, jest to, że zawsze warto kożystać z flagi -O3 ponieważ daje ona bardzo dobre efekty, których mi się udało się uzyskać korzystając z ręcznych optymalizacji.