МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«Векторизация вычислений»

студента 2 курса, 19201 группы

Рудометова Андрея Сергеевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А. Ю. Власенко

Новосибирск 2020

Содержание

ЦЕЛЬ

Изучить SIMD-расширения архитектуры x86/x86-64 и способы их использования в программах на Си, получить навыки использования.

ЗАДАНИЕ

Реализовать три программы, вычисляющих обратную данной матрицу: без оптимизаций, с векторизацией вычислений и с использованием BLAS-библиотеки.

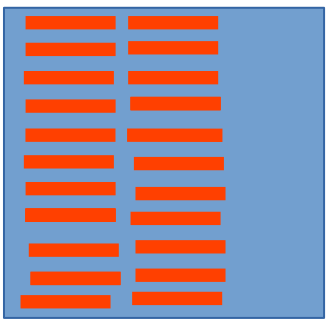
ОПИСАНИЕ РАБОТЫ

Три программы были реализованы в качестве трех функций trivial_invert, vect_invert, blas_invert (программа в Приложении 1). Функции суммирования и умножения были вынесены в отдельную единицу трансляции, mult.cpp с заголовочным файлом mult.h (см. Приложение 2).

Программа без оптимизаций: по сравнению с наивным умножением матриц по формуле для каждой клетки матрицы-результата был изменен общий порядок обхода так, чтобы для A*B=C C и B обходились последовательно, а в A был минимум выгрузок из памяти ячеек матрицы, находящихся на расстоянии N — стороне матрицы (C заполняется в N итераций, j-ю итерацию j-я клетка строки умножается на j-й элемент столбца, которые в свою очередб образуют строку в B).

Векторизация: было принято решение пользоваться встроенными SIMD-intrinsics для расширения AVX2. При наивной замене вычислений в предыдущем варианте операциями над 256-битными векторами на каждую операцию будет приходиться 2 загрузки вектора и одна запись (с учетом возможностей avx2 можем выполнять операции c = a*b + c, b и c в нашем случае векторы, а a - float). Выясним пропускную способность самого быстрого кэша, L1: ноутбук, под запуск на котором оптимизовывалась программа оснащен процессором i5-4300m, по данным en.wikichip.org соответствующая микроархитектура Haswell имеет пропускную способность кэша 64B/такт (см. Приложение 4), с учетом того что у двух ядер свой L1-кэш и он делится на кэш комманд и кэш данных, то большую часть времени программа будет простаивать, ожидая пока вектора загрузятся — выгрузятся.

С целью оптимизации количества операций чтения-записи было написано микроядро размером 4*16 ячеек, состоящее в проекции на матрицу из 4 строк по 2 256-битных регистра в каждой. Всего в распоряжении этого процессора 16 256-битных регистра (ymm0-ymm15), но оптимизация использует только 10 из них, поскольку ядро большего размера было бы не кратно размерам матрицы 2048*2048. Теперь вектора, соответствующие матрице C читаются и записываются только по одному разу, пока два вектора, соответствующие матрице B пробегают полностью 16 столбцов сверху вниз (единомоментно два вектора содержат всю i-ю строку, см. рис)



И, наконец, т.к каждую итерацию эта пара векторов сдвигается по матрице на N=2048 вправо (т. к. матрица хранится в памяти подряд) было произведено предварительное переупорядочивание в последовательность данных векторов для работы предвыборки во время вычисления.

Оптимизация суммирования ограничена представлением матриц в качестве длинных векторов, и последовательным векторизованным сложением 256-векторов.

Третья программа - вычисление обратной матрицы
реализовано с помощью BLAS-функций в реализации библиотеки cblas, cblas_gemm для умножения и cblas_saxpy для сложения.

Полученные программы были скомпилированы без оптимизаций компилятора, проверены на матрице 2*I размером 16*16 — чтобы полностью покрывалось микроядрами, цикл микроядер проходит сначала по столбцам, а уже потом по строкам для работы переупорядочивания матрицы (см. Приложение 3). Было измерено время работы функций на матрице размером N=2048 и с M=10 итерациями.

Заранее в файл input_l была сгенерирована функцией random_matrix() матрица размером 2048*2048; В целях удобства измерения в программа принимает на вход флаг, описывающий тип используемого обращения - -t, -v и -b для trivial, vector и blas соответственно; команда для компиляции и запуска программы приведена ниже:

```
[andy: Code/2020_seminars/evm/lab7]$ ls
input_l.txt  input.txt  lab7  lab7.cpp  lab7_Rudometov_19201.doc  mult.cpp  mult.h  output.txt
[andy: Code/2020_seminars/evm/lab7]$ g++ mult.cpp lab7.cpp -lcblas -march=haswell -o lab7
[andy: Code/2020_seminars/evm/lab7]$ ./lab7 -t
optimized  : 232.247
[andy: Code/2020_seminars/evm/lab7]$ ./lab7 -v
corereorder: 32.3573
[andy: Code/2020_seminars/evm/lab7]$ ./lab7 -b
blas       : 55.0541
[andy: Code/2020_seminars/evm/lab7]$ g++ -O3 mult.cpp lab7.cpp -lcblas -march=haswell -o lab73
[andy: Code/2020_seminars/evm/lab7]$ ./lab73 -t
optimized  : 23.9431
[andy: Code/2020_seminars/evm/lab7]$
```

Видно, что векторизация с микроядром и переупорядочиванием оказалась эффективнее blas-реализации, но все еще немного проигрывает оптимизирующему компилятору, вероятно поскольку тот использует помимо avx2 векторизации дополнительные оптимизации для ускорения загрузки с помощью кэша.

ЗАКЛЮЧЕНИЕ

Для ознакомления с SIMD-расширениями и векторизацией вычислений была реализована программа, обращающая квадратную матрицу с оптимизациями в виде векторизации вычислений и использования предвыборки в кэш; время работы вручную векторизованной функции было доведено до меньшего, чем у BLAS-реализации и ненамного превысило время работы оптимизованной O3 уровнем компилятора gcc.

Приложение 1. Общая программа для трех функций обращения матрицы

```
#include <cblas.h>
#include <time.h>
#include <cmath>
#include <cstring>
#include "mult.h"
using namespace std;

void time_delta(
        const char* head,
        struct timespec * start,
        struct timespec * end)
{
        std::cout << head << end->tv_sec - start->tv_sec
        + 1e-9*(end->tv_nsec - start->tv_nsec) << endl;
}

void random_matrix(Matrix *A){
        srand(time(NULL));
        for(int i = 0; i < A->n*A->n; ++i){
                A->m[i] = rand();
        }
}

void trivial_b(Matrix* B, Matrix* A){
        int size = A->n;
        float* A1s = (float*)calloc(size, sizeof(float));
        float* Ainfs = (float*)calloc(size, sizeof(float));
        for(int i = 0; i< size; ++i){
                float* a = A->m + i*size;
                for(int j = 0; j< size; ++j){
                        float cell = fabs(a[j]);
                        A1s[j] += cell;
                        Ainfs[i] += cell;
                }
        }
        float A1 = 0, Ainf = 0;
        for(int i = 0; i < size; ++i){
                if(A1s[i] > A1)
                        A1 = A1s[i];
        }
        for(int i = 0; i < size; ++i){
                if(Ainfs[i] > Ainf)
                        Ainf = Ainfs[i];
        }
        float Adiv = A1*Ainf;
        //unoptimal in any case
        for(int i = 0; i < size; ++i){
                float* b = B->m + i*size;
                for(int j = 0; j< size; ++j){
                        b[j] = A->m[j*size + i]/Adiv;
                }
        }
        free(A1s); free(Ainfs);
}

void ptr_swap(Matrix** a, Matrix** b){
        Matrix* c = *a;
        *a = *b;
        *b = c;
```

```
}

void trivial_invert(int N, int M,Matrix* A, Matrix* A1){
        Matrix R(N,true), B(N,true),
        C(N,true), R1(N), R2(N);
        trivial_b(&B, A);
        trivial_mult_opt(A1, &B, A); //reusing as BA
        trivial_sub(&R, A1);
        Matrix* rcur = &R1;
        Matrix* rnext = &R2;
        trivial_sum(rcur, &R);
        trivial_sum(&C, rcur);
        for(int i = 2; i < M; ++i){
                trivial_mult_opt(rnext, rcur, &R);
                trivial_sum(&C, rnext);
                ptr_swap(&rnext, &rcur);
                memset(rnext->m, 0, sizeof(float)*N*N);
        }
        memset(A1->m, 0, sizeof(float)*N*N);
        trivial_mult_opt(A1, &C, &B);
}

void vect_invert(int N, int M,Matrix* A, Matrix* A1){
        Matrix R(N,true), B(N,true),
        C(N,true), R1(N), R2(N);
        trivial_b(&B, A);
        vector_mult_opt(&B, A, A1); //reusing as BA
        vector_sub(&R, A1);
        Matrix* rcur = &R1;
        Matrix* rnext = &R2;
        vector_sum(rcur, &R);
        vector_sum(&C, rcur);
        for(int i = 2; i < M; ++i){
                vector_mult_opt(rcur, &R,rnext);
                vector_sum(&C, rnext);
                ptr_swap(&rnext, &rcur);
                memset(rnext->m, 0, sizeof(float)*N*N);
        }
        memset(A1->m, 0, sizeof(float)*N*N);
        vector_mult_opt(&C, &B,A1);
}

void blas_invert(int N, int M,Matrix* A, Matrix* A1){
        Matrix R(N,true), B(N,true),
        C(N,true), R1(N), R2(N);
        int matsize = N*N;
        trivial_b(&B, A);
        cblas_sgemm(CblasRowMajor,CblasNoTrans, CblasNoTrans,
                        N,N,N,1.0,B.m,N,A->m,N, 0.0,A1->m,N);
        cblas_saxpy(matsize,-1.0, A1->m ,1.0, R.m, 1.0); //ba for now
        Matrix* rcur = &R1;
        Matrix* rnext = &R2;
        cblas_saxpy(matsize,1.0, R.m ,1.0, rcur->m, 1.0);
        cblas_saxpy(matsize,1.0, rcur->m, 1.0, C.m ,1.0);
        for(int i = 2; i < M; ++i){
                cblas_sgemm(CblasRowMajor,CblasNoTrans, CblasNoTrans,
                        N,N,N,1.0,rcur->m,N,R.m,N, 0.0,rnext->m,N);
                cblas_saxpy(matsize,1.0, rnext->m, 1.0, C.m ,1.0);
                ptr_swap(&rnext, &rcur);
                memset(rnext->m, 0, sizeof(float)*N*N);
        }
        memset(A1->m, 0, sizeof(float)*N*N);
        cblas_sgemm(CblasRowMajor,CblasNoTrans, CblasNoTrans,
                        N,N,N,1.0,C.m,N,B.m,N, 0.0,A1->m,N);
```

```
}

int main(int argc, char** argv){
        int N=1,M;
        if(argc < 2){return EXIT_FAILURE;}
        struct timespec start,lap1,lap2,lap3,end;
        ifstream fin("input.txt");
        ofstream fout("output.txt");fout.close();
        fin >> N >> M;
        // (N,true) == I
        Matrix A(N), C(N);
        for(int i = 0; i < N*N; ++i){
                fin >> A.m[i];
        }
        fin.close();

        switch(argv[1][1]){
                case 't':
                        clock_gettime(CLOCK_MONOTONIC_RAW, &start);
                        trivial_invert(N,M,&A, &C);
                        clock_gettime(CLOCK_MONOTONIC_RAW, &end);
                        time_delta("optimized  : ",&start, &end);
                        break;
                case 'v':
                        clock_gettime(CLOCK_MONOTONIC_RAW, &start);
                        vect_invert(N,M,&A, &C);
                        clock_gettime(CLOCK_MONOTONIC_RAW, &end);
                        time_delta("corereorder: ",&start, &end);
                        break;
                case 'b':
                        clock_gettime(CLOCK_MONOTONIC_RAW, &start);
                        blas_invert(N,M,&A, &C);
                        clock_gettime(CLOCK_MONOTONIC_RAW, &end);
                        time_delta("blas       : ",&start, &end);
        }
        C.out();
        return EXIT_SUCCESS;
}
```

Приложение 2. Реализация сложения и умножения

mult.h

```
#pragma once
#include <iostream>
#include <fstream>
#include <immintrin.h>

using namespace std;

struct Matrix{
        int n;
        float* m; //matrix
        Matrix(int n_): n(n_){
                m = new float[n*n]();
        }
        Matrix(int n_, bool p): Matrix(n_){
                // I
                for(int i = 0; i < n; ++i){
                        m[i*n + i] = 1.0;
                }
        }

        ~Matrix(){delete [] m;}

        void out(){
                ofstream fout("output.txt", std::ios::app);
                for(int i = 0; i< n; ++i){
                        for(int j = 0; j < n; ++j){
                                fout << m[i*n + j] << ' ';
                        }
                        fout << endl;
                }
                fout.close();
        }
};

struct buf16{
        float* d;
        int n;
        buf16(int size):
                n(size),
                d((float*)_mm_malloc(16 * size * 4, 64))
                {}
        ~buf16(){
                _mm_free(d);
        }
};

void trivial_sum(Matrix* A, Matrix* B);
void trivial_sub(Matrix* A, Matrix* B);
void trivial_mult_opt(Matrix* A, Matrix* B, Matrix* C);

void vector_sum(Matrix* A, Matrix* B);
void vector_sub(Matrix* A, Matrix* B);
void vector_mult(Matrix* A, Matrix* B, Matrix* C);
void vector_mult_opt(Matrix* A, Matrix* B, Matrix* C);
```

mult.cpp

```
#include "mult.h"

using namespace std;

void trivial_sum(Matrix* A, Matrix* B){
    int I_MAX =  A->n*A->n;
    for(int i = 0; i < I_MAX; ++i){
        A->m[i] += B->m[i];
    }
}

void trivial_sub(Matrix* A, Matrix* B){
    int I_MAX =  A->n*A->n;
    for(int i = 0; i < I_MAX; ++i){
        A->m[i] -= B->m[i];
    }
}

void vector_sum(Matrix* A, Matrix* B){
    int I_MAX =  A->n*A->n;
    float* a = A->m;
    float* b = B->m;
    __m256 a0 = _mm256_setzero_ps();
    for(int i = 0; i < I_MAX; i+=8){
        a0 = _mm256_loadu_ps(a + i);
        _mm256_storeu_ps(a + i, _mm256_add_ps(a0, _mm256_loadu_ps(b +
i)));
    }
}

void vector_sub(Matrix* A, Matrix* B){
    int I_MAX =  A->n*A->n;
    float* a = A->m;
    float* b = B->m;
    __m256 a0 = _mm256_setzero_ps();
    for(int i = 0; i < I_MAX; i+=8){
        a0 = _mm256_loadu_ps(a + i);
        _mm256_storeu_ps(a + i, _mm256_sub_ps(a0, _mm256_loadu_ps(b +
i)));
    }
}

void trivial_mult(Matrix* A, Matrix* B, Matrix* C){
    int I_MAX = A->n;
    for(int i = 0; i < I_MAX; ++i){
        for(int k = 0; k < I_MAX; ++k){
            for(int j = 0; j < I_MAX; ++j){
                A->m[i*I_MAX + k] += B->m[i*I_MAX + j]*C->m[j*I_MAX + k];
            }
        }
    }
}

void trivial_mult_opt(Matrix* A, Matrix* B, Matrix* C){
    int size = A->n;
    //Aik = Bij Cjk
    //sequently caching only b and c row
    for(int i = 0; i < size; ++i){
        float* a = A->m + i*size;
        for(int j = 0; j < size; ++j){
            float b = B->m[i*size + j]; //value not pointer
```

```
                float* c = C->m + j*size;
                for(int k = 0; k < size; ++k){
                    a[k] += b*c[k];
                }
            }
        }
    }

    /*
    * lscpu:
    * i5-4300M
    * L1d cache: 64 KiB
    * L1i cache: 64 KiB
    * L2 cache:  512 KiB
    * L3 cache:  3 MiB
    * avx avx2
    */

    void microcore_4x16(
        int size, float * A, float * B, int ldb, float * C)
    {
        //not 6x16 `cause 2048 size
        //A*B = C
        __m256 c00 = _mm256_setzero_ps();
        __m256 c10 = _mm256_setzero_ps();
        __m256 c20 = _mm256_setzero_ps();
        __m256 c30 = _mm256_setzero_ps();
        __m256 c01 = _mm256_setzero_ps();
        __m256 c11 = _mm256_setzero_ps();
        __m256 c21 = _mm256_setzero_ps();
        __m256 c31 = _mm256_setzero_ps();
        __m256 b0, b1, a0, a1;
        for(int i = 0; i < size; ++i){
            b0 = _mm256_loadu_ps(B);
            b1 = _mm256_loadu_ps(B + 8);
            a0 = _mm256_set1_ps(A[0*size]);
            a1 = _mm256_set1_ps(A[1*size]);
            c00 = _mm256_fmadd_ps(a0,b0,c00);
            c01 = _mm256_fmadd_ps(a0,b1,c01);
            c10 = _mm256_fmadd_ps(a1,b0,c10);
            c11 = _mm256_fmadd_ps(a1,b1,c11);
            a0 = _mm256_set1_ps(A[2*size]);
            a1 = _mm256_set1_ps(A[3*size]);
            c20 = _mm256_fmadd_ps(a0, b0, c20);
            c21 = _mm256_fmadd_ps(a0, b1, c21);
            c30 = _mm256_fmadd_ps(a1, b0, c30);
            c31 = _mm256_fmadd_ps(a1, b1, c31);
            B += ldb;
            A += 1;
        }
        _mm256_storeu_ps(C + 0, _mm256_add_ps(c00, _mm256_loadu_ps(C + 0)));
        _mm256_storeu_ps(C + 8, _mm256_add_ps(c01, _mm256_loadu_ps(C + 8)));
        C += size;
        _mm256_storeu_ps(C + 0, _mm256_add_ps(c10, _mm256_loadu_ps(C + 0)));
        _mm256_storeu_ps(C + 8, _mm256_add_ps(c11, _mm256_loadu_ps(C + 8)));
        C += size;
        _mm256_storeu_ps(C + 0, _mm256_add_ps(c20, _mm256_loadu_ps(C + 0)));
        _mm256_storeu_ps(C + 8, _mm256_add_ps(c21, _mm256_loadu_ps(C + 8)));
        C += size;
        _mm256_storeu_ps(C + 0, _mm256_add_ps(c30, _mm256_loadu_ps(C + 0)));
        _mm256_storeu_ps(C + 8, _mm256_add_ps(c31, _mm256_loadu_ps(C + 8)));
    }

    void init_res_core(int N, float* C, int size){
```

```
        for(int i = 0; i < N; ++i, C+=size){
            _mm256_storeu_ps(C + 0, _mm256_setzero_ps());
            _mm256_storeu_ps(C + 8, _mm256_setzero_ps());
        }
    }

    void vector_mult(Matrix* A, Matrix* B, Matrix* C){
        int size = A->n;
        if(size % 16 != 0){
            std::cerr << "wrong matrix size, aborting\n";
            return;
        }
        for(int i = 0; i < size; i+=4){
            for(int j = 0; j < size; j+=16){
                init_res_core(4,C->m + i*size + j, size);
                microcore_4x16(size,
                    A->m + i*size, B->m + j, size,
                    C->m + i*size + j);
            }
        }
    }

    void b_reorder_to_16(buf16* buf, float* B, int size){
        float* bufB = buf->d;
        for(int i = 0; i<size; ++i, bufB +=16, B+=size){
            _mm256_storeu_ps(bufB    , _mm256_loadu_ps(B    ));
            _mm256_storeu_ps(bufB + 8, _mm256_loadu_ps(B + 8));
        }
    }

    void vector_mult_opt(Matrix* A, Matrix* B, Matrix* C){
        int size = A->n;
        if(size % 16 != 0){
            std::cerr << "wrong matrix size, aborting\n";
            return;
        }
        // reordering B for using cache &
        // prioritizing columns over rows
        for(int j = 0; j < size; j+=16){
            //submatrix size_c*16 to sequence for caching
            buf16 B_buf(size);
            b_reorder_to_16(&B_buf, B->m + j, size);
            for(int i = 0; i < size; i+=4){
                init_res_core(4,C->m + i*size + j, size);
                microcore_4x16(size,
                    A->m + i*size, B_buf.d, 16,
                    C->m + i*size + j);
            }
        }
    }
```

Приложение 3. Вывод на тестовой матрице