Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

02.03.02 Фундаментальная информатика   
и информационные технологии

**ОТЧЕТ**

по учебной практике

по направлению 02.03.02 «Фундаментальная информатика и информационные технологии» направленность (профиль) – «Системное программное обеспечение»,

квалификация – бакалавр, программа прикладного бакалавриата, форма обучения – очная,

год начала подготовки (по учебному плану) – 2016

|  |  |  |
| --- | --- | --- |
| Выполнил:  студент гр. ИC-641  «30» мая 2018 г. | \_\_\_\_\_\_\_\_\_\_\_\_ | /Дьячков Д.А./ |
| Оценка «\_\_\_\_\_\_\_\_\_\_\_\_\_» |  |  |
| Руководитель практики от университета  д.т.н., доцент Кафедры ВС  «30» июня 2018 г. | \_\_\_\_\_\_\_\_\_\_\_\_ | /Пудов С.Г. / |

Новосибирск 2018

**Содержание**

Содержание задания3

Применение разреженных матриц4

Форматы хранения разреженных матриц6

Экспериментальный анализ10

Выводы15

Список источников16

Листинг17

**Содержание задания**

Изучить методы хранения разреженных матриц. Реализовать изученные методы. Провести экспериментальный анализ используемой памяти и эффективности хранения матриц определенным алгоритмом. Сделать выводы о рациональности использования форматов хранения.

**Применение разреженных матриц**

В ходе постановки и решения огромного спектра различных инженерно-технических и научных задач, например, при численном решении дифференциальных уравнений в частных производных, в теории электрических сетей, для организации цифровой связи и систем распределения энергии, в теории графов или в численном решении оптимизационных задач и возникают разреженные матрицы.

Разрежённая матрица — это матрица с преимущественно нулевыми элементами.

Среди специалистов нет единства в определении того, какое именно количество ненулевых элементов делает матрицу разрежённой. Для матрицы порядка n число ненулевых элементов:

* есть O(n). Такое определение подходит разве что для теоретического анализа асимптотических свойств матричных алгоритмов;
* ограничено, где < 1;
* таково, что для данного алгоритма и вычислительной системы имеет смысл извлекать выгоду из наличия в ней нулей.

Существует много задач, требующих обработки разреженных матриц. Например, всем известная – электронная таблица. Матрица в средней электронной таблице очень большая, но по факту занимает минимально необходимую память, так как используется лишь небольшая часть элементов массива.

Также разреженные матрицы используются для решения таких практических задач, как расчет пространственных конечно-элементных моделей зданий и сооружений на статику, динамику и устойчивость. Такие расчеты связаны с проблемой решения систем линейных алгебраических уравнений с симметричной разреженной матрицей коэффициентов. Учитывая тенденцию постоянного роста размерности, а также тот факт, что при поиске приемлемого конструкторского решения приходится многократно вносить изменения в расчетную модель и, следовательно, каждый раз выполнять анализ заново, возникает потребность разработки и внедрения в расчетные программные комплексы высокоэффективных методов решения систем уравнений, учитывающих разреженную структуру матрицы жесткости.

**Форматы хранения разреженных матриц**

**Координатный формат хранения (CS)**

Данный формат хранения является наиболее очевидным и простым способом хранения разреженной матрицы. Для ненулевых элементов матрицы и их координат создаются следующие одномерные массивы:

* массив ненулевых элементов (values);
* массив номеров строк, принадлежащих элементам массива values (rows);
* массив номеров столбцов, принадлежащих элементам массива values (cols);

Приведём пример разреженной матрицы А (рисунок 1).

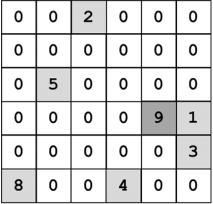


Рисунок 1 – Пример разреженной матрицы

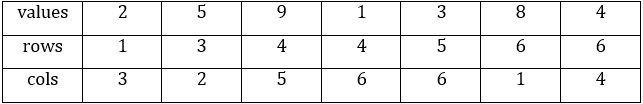
Координатный формат хранения матрицы А представлен на рисунке 2. 

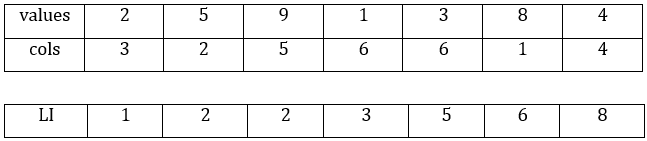
Рисунок 2 – Координатный формат хранения

Также такой метод хранения называют полным, т. к. представлены все элементы матрицы. Данный формат обеспечивает медленный доступ к элементам матрицы, и является затратным по используемой памяти. В рассмотренном выше примере избыточность по памяти образом проявляется в массиве rows, в котором строчные координаты хранятся неоптимальным образом.

**Разреженный строчный формат (CRS / CSR)**

CRS (Compressed Row Storage) или CSR (Compressed Sparse Rows) — разреженный строчной формат, один из наиболее популярных и часто используемых форматов. Данная схема предъявляет минимальные требования к памяти и является удобным представлением для ряда операций над разреженными матрицами. В двух массивах (values, cols) хранятся значения ненулевых элементов и индексы их столбца, в дополнение к ним добавляется еще один массив (LI), содержащий местоположение первого ненулевого элемента в каждой строке. Последний элемент массива LI ссылается на первую свободную позицию в values и cols и определяет количество ненулевых элементов + 1. Если строка i является полностью нулевой, то LI[i] = LI[i+1].

Матрица А представлена в CRS формате на рисунке 3.

Рисунок 3 – CRS формат хранения

**Диагональный формат хранения (DS)**

Диагональный формат хранения матриц используется, когда все ненулевые элементы матрицы расположены на различных, не плотно расположенных- диагоналях. Для хранение таких матриц используется массив размера n × m, где n - размерность исходной матрицы, m - количество ненулевых диагоналей.

Побочные диагонали доопределяются до общей размерности нулями. При этом дополнительно хранится массив целых чисел Index размера m, в котором указывается сдвиг каждой диагонали от главной.

Приведём пример разреженной матрицы B, элементы которой расположены на диагоналях (рисунок 4).

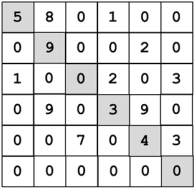


Рисунок 4 – Пример разрежённой матрицы B

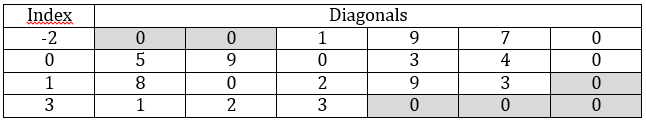
Диагональный формат хранения матрицы B представлен на рисунке 5. 

Рисунок 5 – Диагональный формат хранения

**Коротко - диагональный формат хранения (VSR)**

Коротко - диагональный формат хранения матриц аналогичен диагональному формату хранения, за тем исключением, что диагональ фиксированного размера равного v. Различают коротко-диагональные форматы с фиксированным и плавающим началом.

Для хранения в коротко-диагональные формате с фиксированным началом исходная матрица разбивается на матрицы размера v × n, где v – размер короткой диагонали, n – размерность исходной матрицы. После подобно диагональному формату для выделенных матриц составляется массив размером v × m, где m – количество ненулевых диагоналей в данной подматрице.

Коротко - диагональный формат хранения с длиной v = 2 матрицы B представлен на рисунке 6.

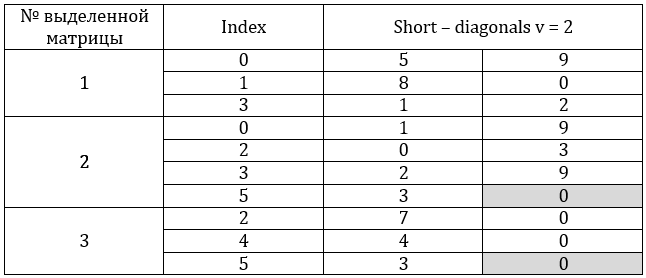


Рисунок 6 – Коротко-диагональный формат

Можно обратить внимание, что в коротко-диагональном формате хранения количество избыточных элементов меньше, чем в диагональном формате. Следовательно данный формат эффективней использует память.

**Экспериментальный анализ**

Для экспериментов в качестве реальных задач был выбран набор матриц, загруженных с коллекции матриц FLORIDA[5].

Первый экспериметн проводился на симметричной диагональной матрице – JGD\_Trefethen/Trefethen\_20 (рисунок 7).

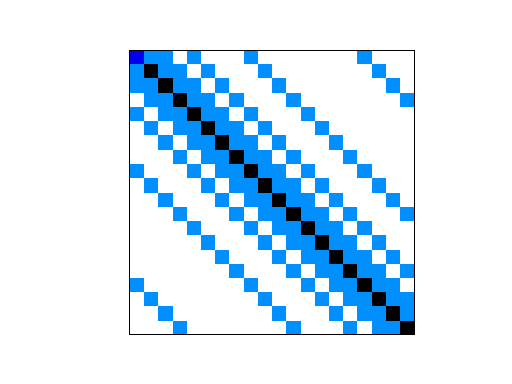


Рисунок 7 – Матрица JGD\_Trefethen/Trefethen\_20

Наиболее эффективные результаты использования памяти показали диагональный формат хранения и коротко-диагональный формат с длиной диагонали равной 32 байта (4 элемента). Коротко-диагональный формат с длиной диагонали равной 64 байта (8 элементов) показал результаты немного хуже, это связано с тем что размерность матрицы не кратна длине короткой-диагонали и в каждой диагонали последнего блока происходило до определение размера.

Второй эксперимент проводился на матрице направленного взвешенного графа сети – Pajek/Ragusa18 (рисунок 8).

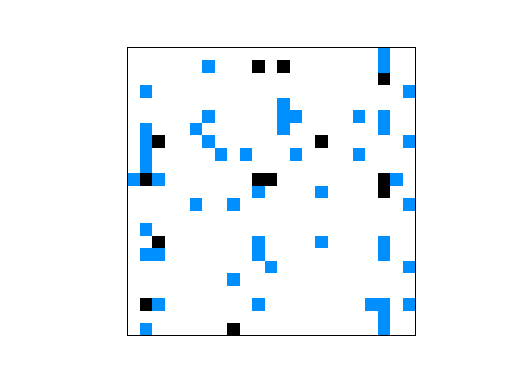


Рисунок 8 – Матрица Pajek/Ragusa18

Как ожидалось лучшая эффективность у координатного формата хранения. Это связанно с тем что элементы разбросаны неравномерно, из-за чего в диагональных форматах хранения много избыточных данных. Этот пример отлично демонстрирует недостатки диагональных форматов хранения и подчёркивает, что их следует использовать где расположение элементов преобладает на диагоналях.

Третий эксперимент проводился на матрице отображающей проблему моделирования схемы – Sandia/oscil\_dcop\_06 (рисунок 9).

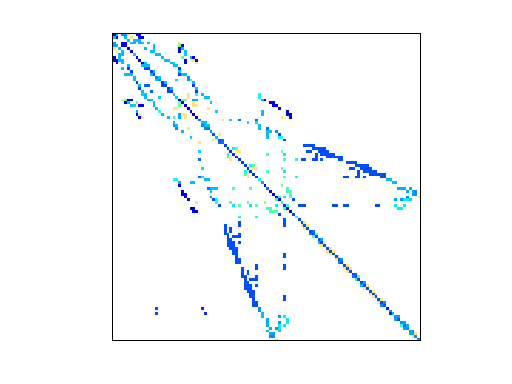


Рисунок 9 – Матрица Sandia/oscil\_dcop\_06

Наихудшая эффективность оказалась у диагонального формата хранения матриц. Это связанно с тем, что матрица большой размерности и данные на диагоналях расположены не равномерно, в следствии чего появляется большая избыточность. Коротко-диагональный формат с длиной диагонали v = 4 показал эффективность лучше, чем диагональный формат и коротко-диагональный с длиной v = 8, из-за того что в нём меньше хранится диагоналей с преобладанием нулевых или избыточных элементов.

Четвёртый эксперимент проводился на матрице метода ближних точек с 2D/3D проблемой – CPM/cz628 (рисунок 10).

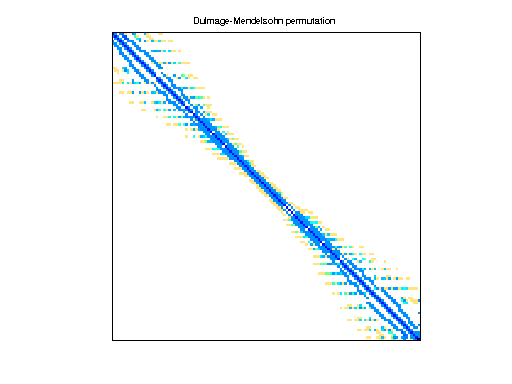


Рисунок 10 – Матрица CPM/cz628

При данном эксперименте диагональные форматы хранения матриц показали эффективность лучше, чем при предыдущем эксперименте. Связанно данное улучшение с тем, что элементы исходной матрицы расположены плотнее, но всё же количество диагоналей где преобладают ненулевые элементы не так много.

Проведя все эксперименты для сравнения эффективности использования памяти для хранения элементов матрицы относительно используемой памяти всей для каждого формата хранения построим гистограмму (рисунок 11).

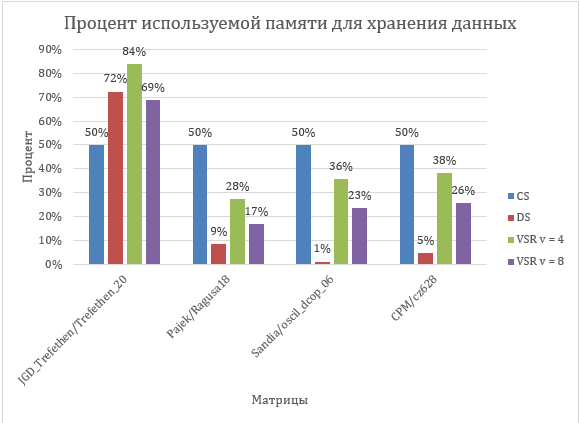


Рисунок 11 – Сравнение эффективности использования памяти

**Выводы**

После проведения экспериментов и построения гистограммы можно сделать вывод, что все форматы хранения подтвердили ожидаемые результаты. Диагональные форматы следует использовать где расположение ненулевых элементов преобладает на диагоналях. Так же стоит заметить, что вместо диагонального формата хранения можно и в большинстве случаев лучше использовать коротко-диагональный форматы, что позволит сократить хранение избыточной информации. В качестве длины для короткой-диагонали лучше использовать длину строки кэша (cache line).

**Список источников**

1. Тьюарсон. Р. Разреженные матрицы. М.: Мир, 1977. – 192 с
2. Писсанецки С. Технология разреженных матриц : пер. с англ. / С. Писсанецки . – М. : Мир, 1988 . – 411 с.
3. Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений. – М.: Мир, 1984.
4. Bell N., Garland M. Efficient Sparse Matrix-Vector Multiplication on CUDA 2008. C. 1–32.
5. Florida matrix collections. [Электронный ресурс]. URL: https://sparse.tamu.edu/

**Листинг**

**packet\_matrix\_base.hpp**

#ifndef SPARSEMATRIX\_PACKET\_MATRIX\_BASE\_HPP

#define SPARSEMATRIX\_PACKET\_MATRIX\_BASE\_HPP

#include <cinttypes>

#include <vector>

class IPacketMatrixBase {

public:

virtual size\_t getNZ() const = 0;

virtual size\_t getN() const = 0;

virtual uint64\_t getMatrixSize() const = 0;

};

#endif //SPARSEMATRIX\_PACKET\_MATRIX\_BASE\_HPP

**coord\_format\_matrix.hpp**

#ifndef SPARSEMATRIX\_COORD\_FORMAT\_MATRIX\_HPP

#define SPARSEMATRIX\_COORD\_FORMAT\_MATRIX\_HPP

#include "packet\_matrix\_base.hpp"

#include <string>

class CCoordMatrix : public IPacketMatrixBase {

struct stMatrixElement {

double value;

size\_t rowId;

size\_t colId;

stMatrixElement(double value, size\_t rowId, size\_t colId) :

value(value), rowId(rowId), colId(colId) {};

};

size\_t N;

std::vector<stMatrixElement> matrix;

public:

size\_t getNZ() const;

size\_t getN() const;

uint64\_t getMatrixSize() const;

size\_t generate(size\_t N, size\_t minInRow, size\_t maxInRow, double powerPrecent);

void loadFromMatrixMarketFile(const std::string& filename);

template <size\_t N>

void init(const double (&initMatrix)[N][N]) {

this->N = N;

matrix.clear();

for (size\_t i = 0; i < N; ++i)

for (size\_t j = 0; j < N; ++j)

if (initMatrix[i][j] != 0)

matrix.emplace\_back(initMatrix[i][j], i, j);

matrix.shrink\_to\_fit();

}

friend class CDiagonalMatrix;

template <size\_t> friend class CShortDiagonalMatrix;

friend class CEvaluation;

};

using mcoord\_t = CCoordMatrix;

#endif //SPARSEMATRIX\_COORD\_FORMAT\_MATRIX\_HPP

**coord\_format\_matrix.сpp**

#include "coord\_format\_matrix.hpp"

#include <fstream>

using namespace std;

size\_t CCoordMatrix::getNZ() const {

return matrix.size();

}

size\_t CCoordMatrix::getN() const {

return N;

}

uint64\_t CCoordMatrix::getMatrixSize() const {

return uint64\_t(N) \* uint64\_t(N);

}

size\_t CCoordMatrix::generate(size\_t N, size\_t minInRow, size\_t maxInRow, double powerPrecent) {

size\_t NZ(0);

this->N = N;

matrix.clear();

//generate matrix elements

matrix.shrink\_to\_fit();

return NZ;

}

void CCoordMatrix::loadFromMatrixMarketFile(const string& filename) {

ifstream in(filename);

string buffer;

while (getline(in, buffer) &&

buffer[0] == '%')//ignore comments

;

size\_t elementCount(0), m, n;

if (sscanf(buffer.c\_str(), "%u %u %u", &m, &n, &elementCount) && m == n) {

N = m;

matrix.clear();

size\_t i, j;

double value;

while (elementCount && getline(in, buffer)) {

sscanf(buffer.c\_str(), "%u %u %lf", &i, &j, &value);

matrix.emplace\_back(value, i - 1, j - 1);

--elementCount;

}

matrix.shrink\_to\_fit();

}

in.close();

}

**diagonal\_format\_matrix.hpp**

#ifndef SPARSEMATRIX\_DIAGONAL\_FORMAT\_MATRIX\_HPP

#define SPARSEMATRIX\_DIAGONAL\_FORMAT\_MATRIX\_HPP

#include "packet\_matrix\_base.hpp"

#include "coord\_format\_matrix.hpp"

class CCoordMatrix;

class CDiagonalMatrix : public IPacketMatrixBase {

struct stDiagonal {

std::vector<double> data;

int32\_t id;

stDiagonal& operator=(const stDiagonal& right);

void set(const CCoordMatrix::stMatrixElement& data);

stDiagonal(int32\_t id, const std::vector<double>& data) :

data(data), id(id) {};

stDiagonal(int32\_t id, const CCoordMatrix::stMatrixElement& data, size\_t length)

: data(length), id(id) {

set(data);

};

};

size\_t N;

std::vector<stDiagonal> matrix;

public:

CDiagonalMatrix(const CCoordMatrix& src) {

loadFromCoord(src);

}

size\_t getM() const;

size\_t getNZ() const;

size\_t getN() const;

uint64\_t getMatrixSize() const;

void loadFromCoord(const CCoordMatrix& src);

friend class CEvaluation;

};

using mdiag\_t = CDiagonalMatrix;

#endif //SPARSEMATRIX\_DIAGONAL\_FORMAT\_MATRIX\_HPP

**diagonal\_format\_matrix.сpp**

#include "diagonal\_format\_matrix.hpp"

#include <algorithm>

using namespace std;

CDiagonalMatrix::stDiagonal& CDiagonalMatrix::stDiagonal::operator=(const stDiagonal& right) {

data = move(right.data);

id = right.id;

}

void CDiagonalMatrix::stDiagonal::set(const CCoordMatrix::stMatrixElement& data) {

this->data[data.colId - id] = data.value;

}

size\_t CDiagonalMatrix::getM() const {

return matrix.size();

}

size\_t CDiagonalMatrix::getNZ() const {

size\_t count(0);

for (const auto &diagonal : matrix) {

size\_t begin((diagonal.id < 0) \* -diagonal.id);

size\_t end(N - (diagonal.id > 0) \* diagonal.id);

for (; begin < end; ++begin)

count += diagonal.data[begin] != 0;

}

return count;

}

size\_t CDiagonalMatrix::getN() const {

return N;

}

uint64\_t CDiagonalMatrix::getMatrixSize() const {

return uint64\_t(N) \* uint64\_t(N);

}

void CDiagonalMatrix::loadFromCoord(const CCoordMatrix& src) {

N = src.N;

matrix.clear();

for (const auto &elem : src.matrix) {

int32\_t id = int32\_t(elem.colId) - int32\_t(elem.rowId);

auto diagonal = find\_if(matrix.begin(), matrix.end(), [&id](const stDiagonal &l)

{ return l.id == id; });

if (diagonal == matrix.end())

matrix.emplace\_back(id, elem, N);

else

diagonal->set(elem);

}

matrix.shrink\_to\_fit();

sort(matrix.begin(), matrix.end(),

[](stDiagonal &l, stDiagonal &r)

{ return l.id < r.id; });

}

**short\_diagonal\_format.hpp**

#ifndef SPARSEMATRIX\_SHORT\_DIAGONAL\_FORMAT\_HPP

#define SPARSEMATRIX\_SHORT\_DIAGONAL\_FORMAT\_HPP

#include <algorithm>

#include "packet\_matrix\_base.hpp"

#include "coord\_format\_matrix.hpp"

template <size\_t diagonalLength>

class CShortDiagonalMatrix : public IPacketMatrixBase {

struct stDiagonal {

std::vector<double> data;

int32\_t id;

stDiagonal &operator=(const stDiagonal &right) {

data = move(right.data);

id = right.id;

}

void set(const CCoordMatrix::stMatrixElement &data) {

this->data[data.colId - id] = data.value;

}

stDiagonal(int32\_t id, const std::vector<double> &data) :

data(data), id(id) {};

stDiagonal(int32\_t id, const CCoordMatrix::stMatrixElement &data) :

data(diagonalLength), id(id) {

set(data);

};

};

struct stDiagonalLines {

std::vector<stDiagonal> diagonals;

size\_t lineId;

void insert(int32\_t id, const CCoordMatrix::stMatrixElement &data) {

diagonals.emplace\_back(id, data);

}

stDiagonalLines(size\_t lineId, int32\_t id, const CCoordMatrix::stMatrixElement &data) :

lineId(lineId) {

this->insert(id, data);

}

};

size\_t N;

std::vector<stDiagonalLines> matrix;

public:

CShortDiagonalMatrix(const CCoordMatrix &src) {

loadFromCoord(src);

}

size\_t getNZ() const {

size\_t count(0);

int32\_t threshold = N - diagonalLength;

for (const auto &line : matrix) {

for (const auto &diagonal : line.diagonals) {

size\_t begin((diagonal.id < 0) \* -diagonal.id);

size\_t end(diagonalLength - (diagonal.id > threshold ? diagonal.id - threshold : 0));

for (; begin < end; ++begin)

count += diagonal.data[begin] != 0;

}

}

return count;

}

size\_t getDiagonalLength() const {

return diagonalLength;

}

size\_t getM() const {

size\_t count(0);

for (const auto &line : matrix)

count += line.diagonals.size();

return count;

}

size\_t getN() const {

return N;

}

uint64\_t getMatrixSize() const {

return uint64\_t(N) \* uint64\_t(N);

}

void loadFromCoord(const CCoordMatrix &src) {

N = src.N;

matrix.clear();

for (const auto &elem : src.matrix) {

size\_t lineId = elem.rowId / diagonalLength;

int32\_t id = int32\_t(elem.colId) - int32\_t(elem.rowId % diagonalLength);

auto line = std::find\_if(matrix.begin(), matrix.end(), [&lineId](const stDiagonalLines& l) { return l.lineId == lineId; });

if (line != matrix.end()) {

auto diagonal = std::find\_if(line->diagonals.begin(), line->diagonals.end(), [&id](const stDiagonal& l) { return l.id == id; });

if (diagonal == line->diagonals.end())

line->insert(id, elem);

else

diagonal->set(elem);

} else

matrix.emplace\_back(lineId, id, elem);

}

matrix.shrink\_to\_fit();

for (auto& line : matrix)

std::sort(line.diagonals.begin(), line.diagonals.end(),

[](stDiagonal& l, stDiagonal& r)

{ return l.id < r.id; });

std::sort(matrix.begin(), matrix.end(),

[](stDiagonalLines &l, stDiagonalLines &r)

{ return l.lineId < r.lineId; });

}

friend class CEvaluation;

};

using msdiag\_t32 = CShortDiagonalMatrix<4>;//cache line: 32 bytes (256 bits) = 4 double variable

using msdiag\_t64 = CShortDiagonalMatrix<8>;//cache line: 64 bytes (512 bits) = 8 double variable

#endif //SPARSEMATRIX\_SHORT\_DIAGONAL\_FORMAT\_HPP

**evalutaion.hpp**

#ifndef SPARSEMATRIX\_EVALUTAION\_HPP

#define SPARSEMATRIX\_EVALUTAION\_HPP

#include "coord\_format\_matrix.hpp"

#include "diagonal\_format\_matrix.hpp"

#include "short\_diagonal\_format.hpp"

#include <iostream>

#include <iomanip>

class CEvaluation {

static void outBaseInformation(const IPacketMatrixBase& matrix, const std::string& type);

static void coordMatrix(const CCoordMatrix& matrix);

static void diagonalMatrix(const CDiagonalMatrix& matrix);

public:

static void storageEfficiency(const IPacketMatrixBase& matrix);

template<size\_t size>

static void storageEfficiency(const CShortDiagonalMatrix<size>& matrix) {

outBaseInformation(matrix, "Short-diagonal");

std::cout << "Length of short diagonal: " << size << " elements" << std::endl;

std::cout << "Number of nonzero short diagonals: " << matrix.getM()

<< " [" << matrix.getM() \* 100.0 / ((matrix.getN() + size - 1) \* matrix.getN() / size) << "%]" << std::endl;

size\_t sizeOfValue = sizeof(matrix.matrix[0].diagonals[0].data[0]);

size\_t sizeIndex = sizeof(CShortDiagonalMatrix<size>::stDiagonal::id);

size\_t sizeLine = sizeof(CShortDiagonalMatrix<size>::stDiagonalLines::lineId);

size\_t sizeOfLines = sizeLine \* matrix.matrix.size();

size\_t sizeDiagIndex = sizeIndex \* matrix.getM();

size\_t sizeOfElements = sizeOfValue \* matrix.getNZ();

uint64\_t totalSizeUsed = sizeOfValue \* size \* matrix.getM() + sizeDiagIndex + sizeOfLines;

size\_t sizeOfGarbage(0);

int32\_t threshold = matrix.getN() - size;

for (const auto &line : matrix.matrix) {

size\_t addition = ((line.lineId + 1) \* size > matrix.getN() ? (line.lineId + 1) \* size - matrix.getN() : 0);

for (const auto &diagonal : line.diagonals) {

size\_t count = (diagonal.id < 0 ? addition - diagonal.id : (diagonal.id > threshold ? std::max(size\_t(diagonal.id - threshold), addition) : addition));

sizeOfGarbage += count \* sizeOfValue;

}

}

uint64\_t sizeOfZero = totalSizeUsed - sizeOfElements - sizeDiagIndex - sizeOfGarbage - sizeOfLines;

std::cout << std::setprecision(3)

<< ">Information about the memory used:" << '\n'

<< " Summary memory used for storage: " << totalSizeUsed << " bytes\n"

<< " The memory used to store nonzero elements: " << sizeOfElements << " bytes [" << sizeOfElements \* 100.0 / totalSizeUsed << "%]" << '\n'

<< " The memory used to store id diagonals: " << sizeDiagIndex << " bytes [" << sizeDiagIndex \* 100.0 / totalSizeUsed << "%]" << '\n'

<< " Memory used to store zero items: " << sizeOfZero << " bytes [" << sizeOfZero \* 100.0 / totalSizeUsed << "%]" << '\n'

<< " Memory used for garbage storage: " << sizeOfGarbage << " bytes [" << sizeOfGarbage \* 100.0 / totalSizeUsed << "%]" << '\n'

<< " The memory used to store id lines: " << sizeOfLines << " bytes [" << sizeOfLines \* 100.0 / totalSizeUsed << "%]" << std::endl;

}

};

#endif //SPARSEMATRIX\_EVALUTAION\_HPP

**evalutaion.сpp**

#include "evalutaion.hpp"

using namespace std;

void CEvaluation::outBaseInformation(const IPacketMatrixBase& matrix, const string& type) {

cout << setprecision(3)

<< "======> " << type << " matrix storage format <======" << '\n'

<< "Matrix size: " << matrix.getN() << 'x' << matrix.getN()

<< " (" << matrix.getMatrixSize() << " elements)" << '\n'

<< "Number of nonzero elements: " << matrix.getNZ()

<< " [" << matrix.getNZ() \* 100.0 / matrix.getMatrixSize() << "%]" << endl;

}

void CEvaluation::coordMatrix(const CCoordMatrix& matrix) {

outBaseInformation(matrix, "Coordinate");

size\_t sizeOfValue = sizeof(CCoordMatrix::stMatrixElement::value);

size\_t sizeIndex = sizeof(CCoordMatrix::stMatrixElement::rowId);

size\_t sizeOfElements = sizeOfValue \* matrix.getNZ();

size\_t sizePosition = 2 \* sizeIndex \* matrix.getNZ();

size\_t totalSizeUsed = sizeOfElements + sizePosition;

cout << setprecision(3)

<< ">Information about the memory used:" << '\n'

<< " Summary memory used for storage: " << totalSizeUsed << " bytes\n"

<< " The memory used to store nonzero elements: " << sizeOfElements

<< " bytes [" << sizeOfElements \* 100.0 / totalSizeUsed << "%]" << '\n'

<< " The memory used to store index elements: " << sizePosition

<< " bytes [" << sizePosition \* 100.0 / totalSizeUsed << "%]" << endl;

}

void CEvaluation::diagonalMatrix(const CDiagonalMatrix& matrix) {

outBaseInformation(matrix, "Diagonal");

cout << "Number of nonzero diagonals: " << matrix.getM()

<< " [" << matrix.getM() \* 100.0 / (matrix.getN() \* 2 - 1) << "%]" << endl;

size\_t sizeOfValue = sizeof(matrix.matrix[0].data[0]);

size\_t sizeIndex = sizeof(CDiagonalMatrix::stDiagonal::id);

size\_t sizeOfElements = sizeOfValue \* matrix.getNZ();

size\_t sizeDiagIndex = sizeIndex \* matrix.getM();

uint64\_t totalSizeUsed = sizeOfValue \* matrix.getN() \* matrix.getM() + sizeDiagIndex;

size\_t sizeOfGarbage(0);

for (const auto& diagonal : matrix.matrix)

sizeOfGarbage += abs(diagonal.id) \* sizeOfValue;

uint64\_t sizeOfZero = totalSizeUsed - sizeOfElements - sizeDiagIndex - sizeOfGarbage;

cout << setprecision(3)

<< ">Information about the memory used:" << '\n'

<< " Summary memory used for storage: " << totalSizeUsed << " bytes\n"

<< " The memory used to store nonzero elements: " << sizeOfElements << " bytes [" << sizeOfElements \* 100.0 / totalSizeUsed << "%]" << '\n'

<< " The memory used to store id diagonals: " << sizeDiagIndex << " bytes [" << sizeDiagIndex \* 100.0 / totalSizeUsed << "%]" << '\n'

<< " Memory used to store zero items: " << sizeOfZero << " bytes [" << sizeOfZero \* 100.0 / totalSizeUsed << "%]" << '\n'

<< " Memory used for garbage storage: " << sizeOfGarbage << " bytes [" << sizeOfGarbage \* 100.0 / totalSizeUsed << "%]" << endl;

}

void CEvaluation::storageEfficiency(const IPacketMatrixBase& matrix) {

if (dynamic\_cast<const CCoordMatrix\*>(&matrix))

coordMatrix(dynamic\_cast<const CCoordMatrix&>(matrix));

else if (dynamic\_cast<const CDiagonalMatrix\*>(&matrix))

diagonalMatrix(dynamic\_cast<const CDiagonalMatrix&>(matrix));

}

**main.сpp**

#include "evalutaion.hpp"

using namespace std;

int main() {

mcoord\_t matrix;

matrix.loadFromMatrixMarketFile("test data\\matrix\_430\_430.mtx");

CEvaluation::storageEfficiency(matrix);

CEvaluation::storageEfficiency(mdiag\_t(matrix));

CEvaluation::storageEfficiency(msdiag\_t32(matrix));

CEvaluation::storageEfficiency(msdiag\_t64(matrix));

return 0;

}