МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Кафедра: высокопроизводительных вычислений и системного программирования

Направление подготовки: «Прикладная математика и информатика»
Магистерская программа: «Вычислительные методы и суперкомпьютерные технологии»

ОТЧЕТ

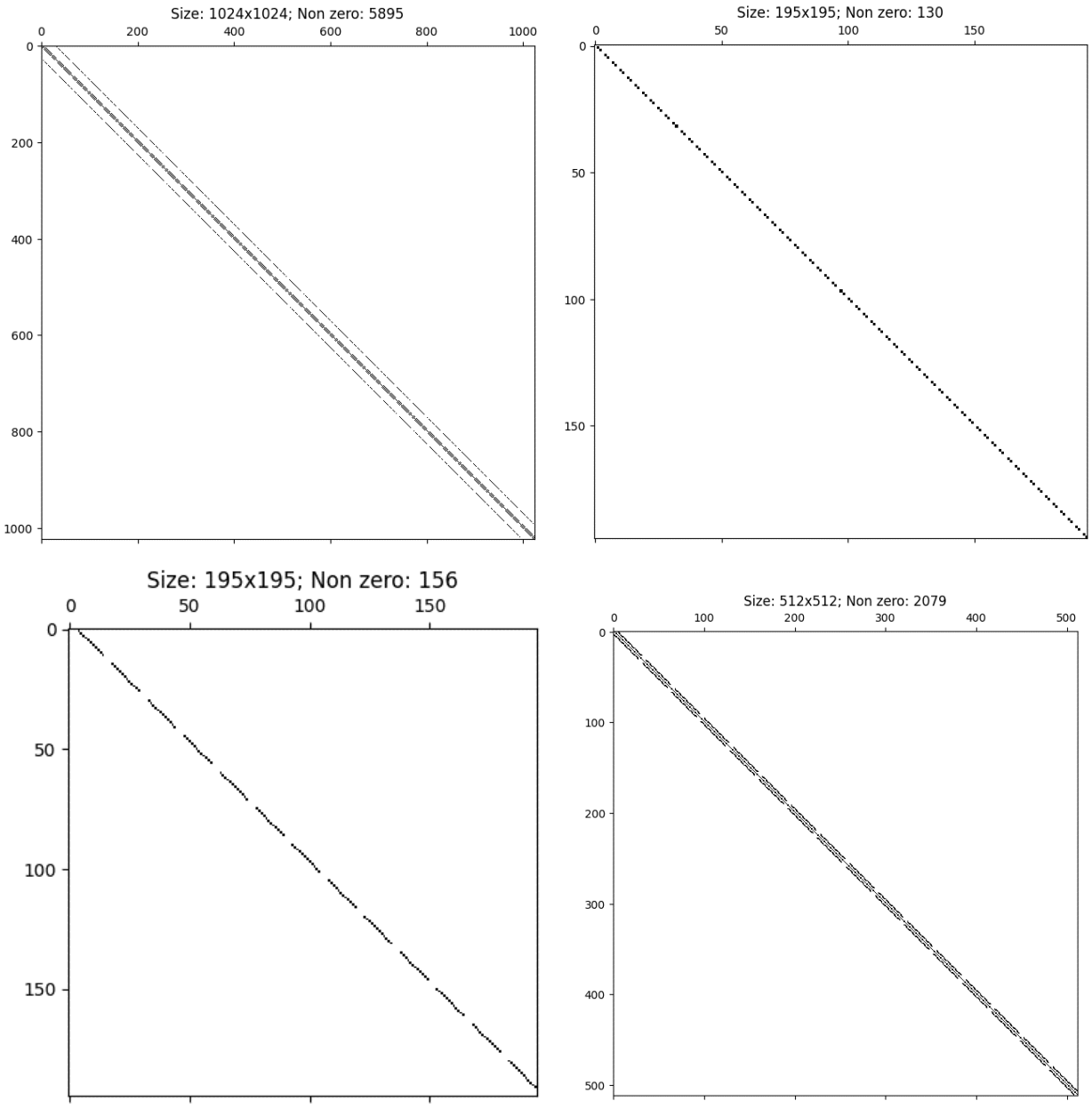
по “Анализ производительности и оптимизация программ”

Выполнил: студент
группы 3824М1ПМвм
Ивлев Александр Дмитриевич

Нижний Новгород
2024

1. Введение

В ходе решения некоторой задачи выяснилось, что подавляющее время работы программы тратиться на умножение квадратных матриц. При более детальном исследовании выяснилось, что в большинстве случаев умножается разреженная матрица на плотную или плотная на разреженную. Причем разреженные матрицы имеют характерные профили расположения ненулевых элементов, а их количество находится в диапазоне от 0.5n до 10n, где n – сторона квадратной матрицы. Ниже приведены примеры расположения ненулевых элементов.



Поэтому в данной работе будет рассмотрена реализация класса разреженной матрицы и алгоритмов умножения разреженной матрицы на плотную или плотной на разреженную. Будет проведено сравнение с базовым алгоритмом, где разреженная матрица храниться в виде плотной.

2. Программно-аппаратного окружение и методика проведения экспериментов

Для тестирование использовался ПК с характеристиками:

- Процессор: AMD Ryzen 5 Mobile 3550 H (4 ядра, 8 потоков)
- Оперативная память: 32 GBytes DDR4
- Операционная система: Windows 10

Программное окружение:

- Среда разработки: Microsoft Visual Studio 2022
- Конфигурация сборки: Release, x64, стандарт языка C++ 17, сборка модулей стандартной библиотеки C++ 23
- Дополнительные параметры сборки: /openmp:experimental
- Профилировщик производительности: встроенный профилировщик Visual Studio

Параметры тестирования:

- Размеры матриц: 1024x1024, 2048x2048, 4096x4096
- Плотные матрицы заполняются случайными комплексными числами a+bi, где a, b числа в диапазоне [-10;10]
- Разреженные матрицы генерируются по правилам исходной задачи с таким набором параметров, что количество ненулевых элементов около 6n. Все параметры, кроме размера матриц остаются неизменными
- Замер времени проводится с помощью omp_get_wtime()
- Для каждого размера эксперимент запускается трижды для разных случайных плотных матриц, после результат усредняется

Дополнительно задействуется nlohmann/json (https://github.com/nlohmann/json) для загрузки параметров из json файла.

3. Плотная матрица и базовый алгоритм

В моей реализации класса плотной матрицы, она хранится в виде комплексного одномерного вектора с использованием структуры `std::vector<std::complex<double>>`. Кроме непосредственно алгоритма умножения матриц были реализованы: сложение матриц, умножение на константу, произведение Кронекера, транспонирование, эрмитовое сопряжение, вывод в консоль или файл и некоторые вспомогательные методы.

В базовой реализации разреженная матрица хранится в виде плотной. Базовый алгоритм умножения плотных матриц:

```
    Matrix multiply(const Matrix& other) const {
        Matrix result(rows, other.cols, false);

        for (int i = 0; i < rows; ++i) {
            for (size_t j = 0; j < cols; ++j) {
                for (int k = 0; k < other.cols; ++k)
                    result.matrix[i * other.cols + k] += matrix[i * cols + j] *
other.matrix[j * other.cols + k];
            }
        }
        return result;
    }
```

Протестируем его и запишем результаты в таблицу, где указано время исполнения в секундах:

	1024x1024	2048x2048	4096x4096
Базовый алгоритм (разреженная на плотную)	3.43503	27.7743	205.388
Базовый алгоритм (плотная на разреженную)	3.65587	25.6139	199.156

Результаты показывают, что порядок умножения при одинаковом хранении почти не влияет на результат. Также подтвердилось ожидаемое увеличение времени работы примерно в 8 раз при увеличении n в 2 раза, так как сложность алгоритма $O(n^3)$.

Первая оптимизация распараллелить внешний цикл и добавить принудительную векторизацию внутреннего с помощью openMP.

```
    Matrix multiply(const Matrix& other) const {
        Matrix result(rows, other.cols, false);

        #pragma omp parallel for
        for (int i = 0; i < rows; ++i) {
            for (size_t j = 0; j < cols; ++j) {
                #pragma omp simd
                for (int k = 0; k < other.cols; ++k)
                    result.matrix[i * other.cols + k] += matrix[i * cols + j] *
other.matrix[j * other.cols + k];
            }
        }
        return result;
```

```
}
```

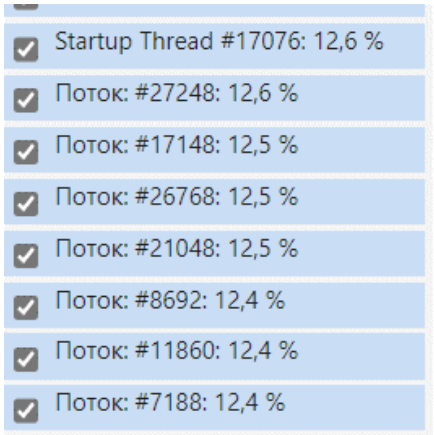
Но изначально при компиляции я столкнулся с сообщением: Цикл omp simd не векторизован по следующей причине: "1305" (Недостаточно сведений о типе). Оказалось, что данная команда может не работать с типами отличными от стандартных. При получении дополнительной информации о векторизации с помощью /Qvec-report:2 можно увидеть, что большинство циклов в данном класс не были векторизованы по причине "1305". Не меняя структуру хранения, добиться векторизации не удалось.

Тогда запишем результаты, только с добавлением параллелизма, где указано время исполнения в секундах, а в скобках ускорение относительно базовой реализации:

	1024x1024	2048x2048	4096x4096
Базовый алгоритм `parallel` (разреженная на плотную)	0.773958 (4,438)	5.95298 (4,666)	52.1028 (3,942)
Базовый алгоритм `parallel` (плотная на разреженную)	0.70456 (5,189)	5.80217 (4,415)	50,0518 (3,979)

Можно заметить, что было достигнуто ускорение около 4 при 8 потоках. С помощью профилировщика VisualStudio для размера 4096x4096 была собрана следующая информация:

Работа между потоками была распределена почти равномерно:



Процессор был загружен на 80-100%, но с просадками. Не полная загруженность, возможно связана с работающими на фоне прочими программами. Но особенно выделяется просадка в конце, когда главный поток собирает результаты:

В рамках одного потока загруженность непосредственно умножением составляет в среднем 98%, а общая для всех потоков 86%:

Имя функции	Общее время Ц...	Собственное вре.
▲ lvlevOpt (ИП: 22548)	350782 (100,00 %)	0 (0,00 %
[Внешний вызов] ntdll.dll!0x00007ffa9415cc91	350782 (100,00 %)	370 (0,11 %
Matrix::multiplyomp1	350311 (99,87 %)	344669 (98,26 %

Имя функции	Общее время Ц...	Собственное вре...
▲ lvlevOpt (идентификатор процесса: 22548)	350782 (100,00 %)	0 (0,00 %)
▲ [Внешний вызов] ntdll.dll!0x00007ffa9415...	350782 (100,00 %)	370 (0,11 %)
▶ Matrix::multiplyomp1	306310 (87,32 %)	301471 (85,94 %)

4. Разреженная матрица в формате Ellpack

Так как полученные разреженные матрицы имеют примерно равное количество ненулевых элементов в строках, а количество строк только с нулевыми элементами невелико, то мною был выбран формат хранения матрицы Ellpack.

Ввиду исходной задачи Ellpack реализован только для квадратных матриц поэтому:

- size_t rows – размер стороны квадратной матрицы
- size_t cols – максимальное количество ненулевых элементов в строке
- std::vector<std::complex<double>> value – одномерный вектор с значениями
- std::vector<size_t> col – одномерный вектор номеров столбцов

К кроме методов, связанных с умножением, также были реализованы некоторые вспомогательные методы.

В классе Ellpack реализован метод умножения на плотную матрицу Matrix:

```
Matrix Ellpack::multiply(const Matrix& other) const {
  Matrix result(rows, other.cols);

  for (int i = 0; i < rows; i++) {
      for (size_t j = 0; j < cols; j++) {
          for (size_t k = 0; k < other.cols; k++)
              result.matrix[i * other.cols + k] += value[i * cols + j] *
other.matrix[col[i * cols + j] * other.cols + k];
      }
  }
  return result;
}
```

В классе Matrix реализован метод умножения на разреженную матрицу Ellpack:

```
Matrix multiply(const Ellpack& other) const {
  Matrix result(rows, other.rows);

  for (int i = 0; i < rows; i++) {
      for (size_t j = 0; j < cols; j++) {
          for (size_t k = 0; k < other.cols; k++)
              result.matrix[i * other.rows + other.col[j * other.cols + k]] +=
other.value[j * other.cols + k] * matrix[i * cols + j];
      }
  }
  return result;
}
```

Так же, как и для базового алгоритма порядок циклов имеет значение, из-за более оптимального прохождения по памяти. Проведём тестирование реализованных алгоритмов, где указано время исполнения в секундах, а в скобках ускорение относительно базовой реализации:

	1024x1024	2048x2048	4096x4096
Ellpack алгоритм (разреженная на плотную)	0.0407601 (84,27)	0.171534 (161,92)	0.649657 (316,15)

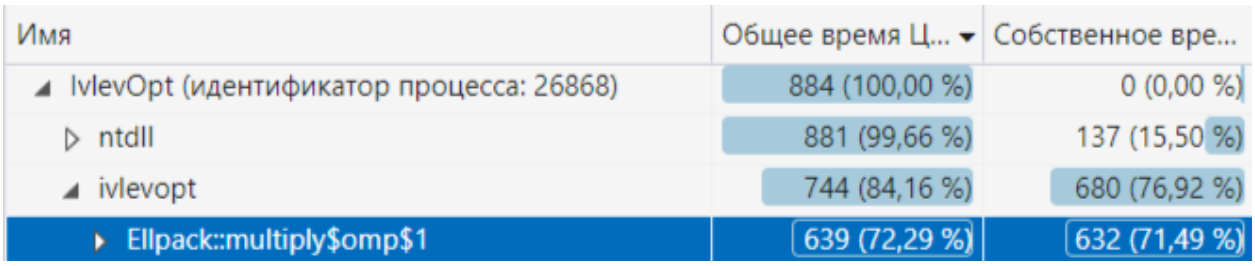
Ellpack алгоритм (плотная на разреженную)	0.038921 (93,93)	0.146376 (174,99)	0.538801 (369,63)

Время действительно заметно уменьшилось при использовании разреженного формата хранения Ellpack. Причем увеличение n в 2 раза увеличивает время выполнения уже в меньшее количество раз, чем 8. Чтобы достоверно оценить данное значение необходимо провести большее количество экспериментов, так как оно может зависеть от n. Также можно отметить, что ускорение относительно базового алгоритма при увеличении n в 2 раза тоже увеличивается примерно в 2 раза. Такой рост относительного ускорения связан с тем, что при увеличении размера матрицы процент ненулевых элементов уменьшается в 2 раза. Так на примере матриц, использующихся для тестирования процент ненулевых элементов при размерах 1024x1024, 2048x2048 и 4096x4096 примерно: 0.58%, 0.29% и 0.15%.

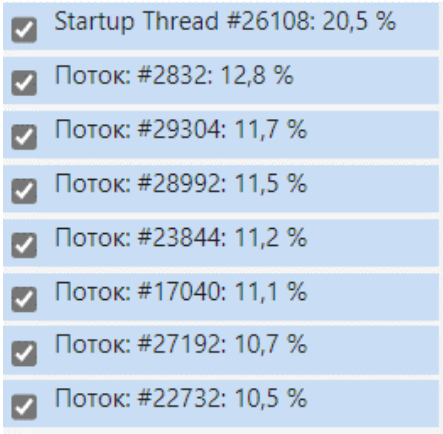
Аналогично базовому алгоритму распараллелим алгоритмы относительно внешнего цикла с помощью OpenMP. Проведём тестирование, где указано время исполнения в секундах, а в скобках ускорение относительно реализации без параллелизма:

	1024x1024	2048x2048	4096x4096
Ellpack алгоритм `parallel` (разреженная на плотную)	0.0115366 (3,5331)	0.0481232 (3,5645)	0.240756 (2,6984)
Ellpack алгоритм `parallel` (плотная на разреженную)	0.0103664 (3,7545)	0.0473919 (3,0886)	0.208107 (2,5891)

Можно заметить, что эффективность применения параллелизма уменьшилась, но он всё ещё позволяет получить выигрыш в производительности. Также наблюдается снижение эффективности с ростом размеров матриц, но для подтверждения необходимо провести большее количество экспериментов на больших размерах. Так как даже для размера 4096x4096 накладные расходы составляют около 28.5%:



Что также видно на примере загруженности потоков, где главный поток берёт почти в 2 раза больше нагрузки, чем остальные:



5. Результаты

По результатам работы были оптимизированы умножение разреженной матрицы на плотную и плотной на разреженной. При размере матриц 4096x4096 и условиях, описанных в 2 пункте, удалось достигнуть ускорения примерно в 900 раз за счет использования параллелизма и формата хранения разреженных матриц Ellpack. Анализ алгоритмов был проведён с помощью встроенного профилировщика Visual Studio.

Но перед непосредственного использованием в исходном проекте, необходимо дополнительно реализовать и оптимизировать ещё некоторые методы, например умножение и сложение разреженных матриц.

6. Приложение

Полный код можно найти по ссылке: https://github.com/Faert/Opt.