Различные реализации параллельных алгоритмов решения СЛАУ методом Гаусса

1. Общие сведения

1.1.Метод Гаусса

Метод Гаусса — это один из методов решения системы линейных алгебраических уравнений, метод последовательного исключения переменных, основанный на привидении системы уравнений к равносильной системе треугольного вида путем элементарных преобразований (домножение уравнения на ненулевой действительный коэффициент и прибавление к нему другого уравнения системы, домноженного на действительный коэффициент; перемена местами двух уравнений системы), из которой последовательно находятся все остальные переменные.

Пусть исходная система выглядит следующим образом:

$$\begin{cases} a_{11}x_1 + \ \dots \ + a_{1n}x_n = b_1 \\ \dots \\ a_{m1}x_1 + \ \dots \ + a_{mn}x_n = b_m \end{cases}$$

$$A = \begin{pmatrix} a_{11} \dots a_{1n} \\ \dots \\ a_{m1} \dots a_{mn} \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ \dots \\ b_m \end{pmatrix} \quad \bar{A} = \begin{pmatrix} a_{11} \dots a_{1n}\ b_1 \\ \dots \\ a_{m1} \dots a_{mn}\ b_m \end{pmatrix}$$

Прямой ход:

Прямой ход заключается в приведении системы к ступенчатой или треугольной форме путем элементарных преобразований.

$$a(i,j) = a(i,j) - \frac{a(i,k)}{a(k,k)}\, a(k,j), \qquad \forall i,j = \overline{1,n}, \forall k = \overline{1,\iota - 1}$$

В результате прямого хода получают систему вида:

$$\begin{cases} x_1 + e_{12}x_2 + e_{13}x_3 + \ \dots \ + e_{1k}x_k + e_{1,k+1}x_{k+1} + \ \dots \ + e_{1n}x_n = f_1 \\ x_2 + e_{23}x_3 + \ \dots \ + e_{2k}x_k + e_{2,k+1}x_{k+1} + \ \dots \ + e_{2n}x_n = f_2 \\ \dots \\ x_k + e_{k,k+1}x_{k+1} + \ \dots \ + e_{kn}x_n = f_k \\ 0 = f_{k+1} \\ \dots \\ 0 = f_m \end{cases}$$

1.2.OpenMP

OpenMP - это открытый стандарт директив компилятора, библиотечных процедур и переменных окружения, которые предназначены разработки высокоуровневых многопоточных приложений на языках C/C++ и Fortran.

Для разработки многопоточных приложений необходимо выделять блоки кода, которые могут быть распределены между процессорами, на что уходит огромное количество времени. В большинстве программ, код, выполняемый на одном процессоре, зависит от результатов другого. Для упрощения разработки, программисты в 1980х создали специальные нотации, чтобы стандартизировать как будет проходить разбиение работы между отдельными процессорами и чтобы обеспечить правильный порядок обращений к данным. Нотация имеет вид специальных инструкций и директив, которые компилятор использует, чтобы создать конкретный код для выполнения на каждом процессоре.

OpenMP имеет ряд преимуществ, благодаря которым стоит обратить на него внимание:

- Он является наиболее широко-используемым стандартом для симметричных многопроцессорных систем
- Поддерживает три разных языка: Fortran, C, C++
- Довольно небольшая и простая спецификация
- Довольно много исследований проводится над ней, что позволяет держать ее в тренде с последними достижениями техники

2. Реализация параллельных алгоритмов метода Гаусса на OpenMP

Полный код приложения, осуществляющего прямой ход метода Гаусса рассмотренными ниже способами приведен в приложении. Программа, к сожалению, не выполняет проверку на вырожденность матрицы, но гарантирует, что если решение СЛАУ существует, то прямой ход будет выполнен без ошибок.

Для решения берется случайно сгенерированная матрица $A_{n,n}$ и вектор ответа x, вектор $b = Ax$, сравнение результатов происходит с точностью $e = 10^{-3}$, которая может появиться в виду операций над вещественными числами.

2.1.Тайлинг: Линейный подход

Основным подходом при реализации многопоточных алгоритмов является тайлинг, рассмотрим линейную реализацию алгоритма прямого хода метода Гаусса:

```
for (int k = 0; k < N - 1; k++) {
    for (int i = k + 1; i < N; i++) {
        for (int j = k + 1; j < N + 1; j++) {
            a[i][j] = a[i][j] - a[i][k] * a[k][j] / a[k][k];
        }
    }
}
```

Основа алгоритма: проходимся по каждой строке, и отнимаем ее с соответствующим коэффициентом от всех последующих. Преобразуем алгоритм таким образом, чтобы указанные итерации осуществлялись последовательно по прямоугольным участкам, и обработку таких участков вынесем в отдельную функцию. Пусть r_1, r_2 – размеры этих участков, а Q_1, Q_2 – количество их в столбце и ряду соответственно.

```
for (int i_gl = 0; i_gl < Q1; ++i_gl) {
    for (int j_gl = 0; j_gl < Q2; ++j_gl) {
        // Начало тайла
        for (int k = 0; k < N - 1; ++k) {
            int start_i = max(k+1, 1+i_gl * r1);
            int finish_i = min(1+r1*(1 + i_gl), N);
            int start_j = max(k+1, 1 + j_gl * r2);
            int finish_j = min(1+r2 * (1 + j_gl), N+1);

            for (int i = start_i; i < finish_i; ++i) {
                double coef = matrix[i][k] / matrix[k][k];
                for (int j = start_j; j < finish_j; ++j) {
                    matrix[i][j] = matrix[i][j] - matrix[k][j] * coef;
                }
            }
        }
        // Конец тайла
    }
}
```

Таким образом, тайл можно представить следующим методом:

```
void tile(int i_gl, int j_gl, int r1, int r2, double** matrix) {
    for (int k = 0; k < N - 1; ++k) {
        int start_i = max(k+1, 1+i_gl * r1);
        int finish_i = min(1+r1*(1 + i_gl), N);
        int start_j = max(k+1, 1 + j_gl * r2);
        int finish_j = min(1+r2 * (1 + j_gl), N+1);

        for (int i = start_i; i < finish_i; ++i) {
            double coef = matrix[i][k] / matrix[k][k];
            for (int j = start_j; j < finish_j; ++j) {
                matrix[i][j] = matrix[i][j] - matrix[k][j] * coef;
            }
        }
    }
}
```

2.2. Параллелизация внешнего и внутреннего цикла

При помощи директивы parallel можем распределить выполнение тайлов между процессорами двумя способами: распараллелив внешний цикл или внутренний.

Внешний:

```
#pragma omp parallel for
    for (int i_gl = 0; i_gl < Q1; ++i_gl) {
        for (int j_gl = 0; j_gl < Q2; ++j_gl) {
                tile(k, i_gl, j_gl, r1, r2, a);
        }
    }
```

Внутренний:

```
for (int i_gl = 0; i_gl < Q1; ++i_gl) {
    #pragma omp parallel for
        for (int j_gl = 0; j_gl < Q2; ++j_gl) {
            tile(i_gl, j_gl, r1, r2, a, N);
        }
}
```

2.3.Параллелизация обоих циклов

Начиная с версии 2.3 OpenMP предлашает возможность распараллелить более чем один цикл, используя ключевую функцию collapse(n), где n – количество циклов, которые необходимо

распараллелить. Аналогичного результата можно достигнуть методом введения единого параметра при использовании более ранних версий.

```
for (int k = 0; k < N - 1; ++k) {
    #pragma omp parallel for collapse(2)
    for (int i_gl = 0; i_gl < Q1; ++i_gl) {
        for (int j_gl = 0; j_gl < Q2; ++j_gl) {
              tile(k, i_gl, j_gl, r1, r2, a, N);
       }
    }
}
```

2.4.Скошенный параллелизм

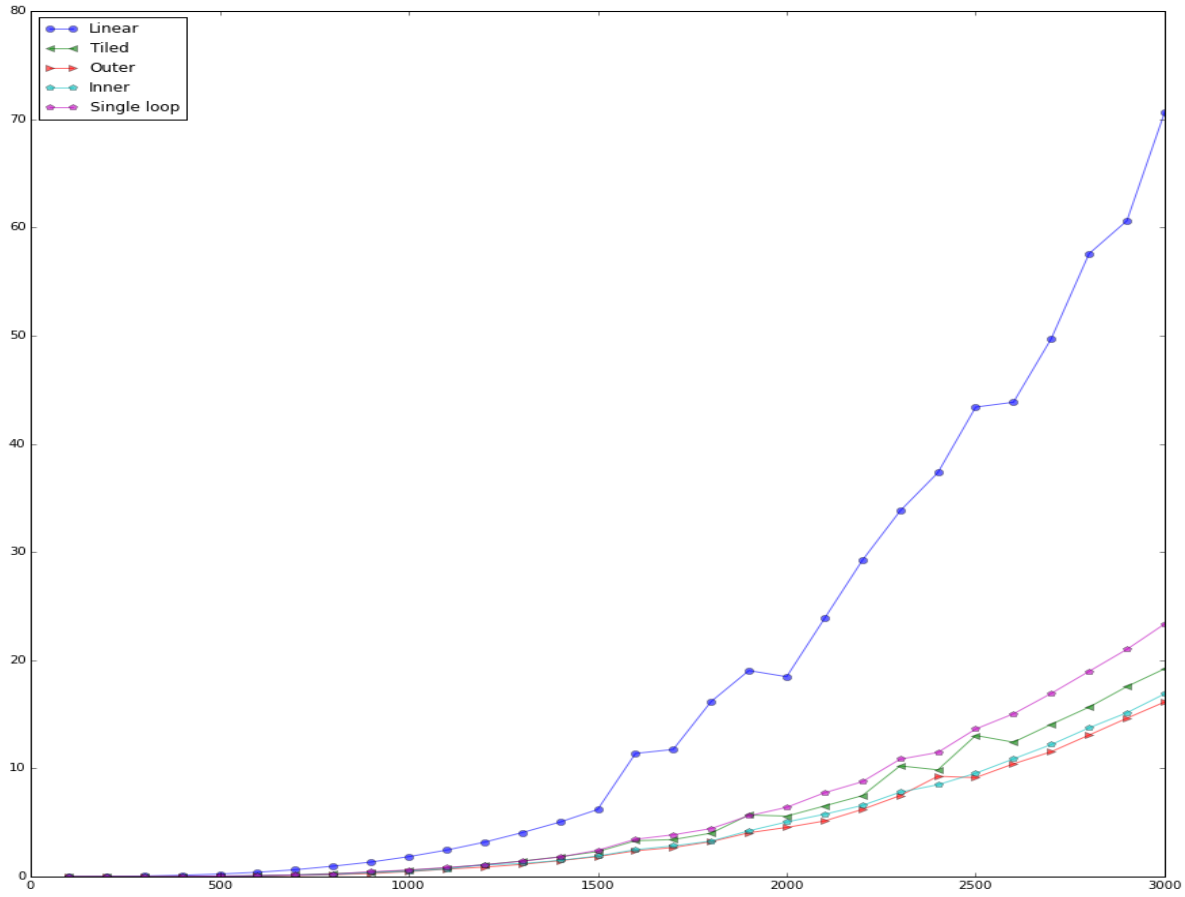
Применив аффинное преобразование $t = i_{gl} + j_{gl}, j_{gl}' = j_{gl}$ можно получить следующий алгоритм:

```
for (int k = 0; k < N - 1; ++k) {
       for (int t = 0; t < Q1 + Q2 - 1; ++t) {
#pragma omp parallel for
              for (int j_gl = 0; j_gl < Q2; ++j_gl) {
                     int i_gl = t - j_gl;
                     tile(k, i_gl, j_gl, r1, r2, a, N);
              }
       }
}
```

3. Анализ быстродействия

3.1.Преимущества параллельных подходов

Значительное ускорение методу Гаусса дает даже просто тайлирование, без распараллеливания на несколько процессоров. Обусловлено это, в первую очередь, повышением КПД кэша, так как при этом подходе в полной мере проявляет себя принцип локальности по данным.



На примере тайлирующего алгоритма мы можем убедиться в том, что нельзя не учитывать роль кэша при анализе быстродействия и проектировании параллельных алгоритмов.

Алгоритм запускался для квадратных матриц размером от 100x100 до 3000x3000, с шагом в 100. Тестовая машина оснащена процессором Intel i5 3317U. Характеристики:

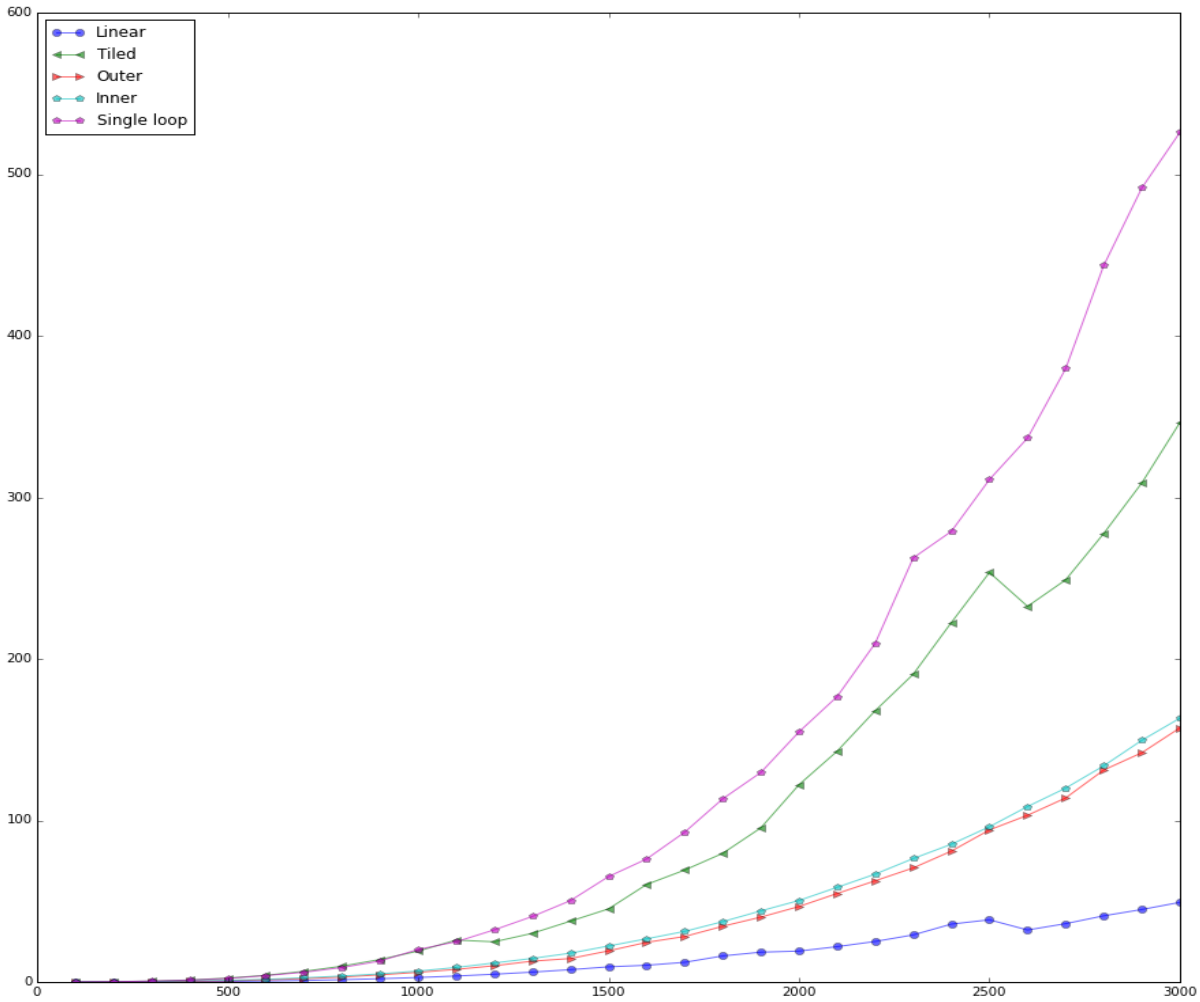
- 2 ядра, 4 потока
- 3 Мб интеллектуальной кэш-памяти Intel (архитектура, которая позволяет всем ядрам динамически совместно использовать доступ к кэшу последнего уровня)

Также на тестовой машине был установлен дополнительный твердый носитель объемом в 24 Гб, синхронизированный с системой по технологии Samsung Express Cache и выполняющий исключительно функции кэша.

3.2.Изучение вырожденных случаев

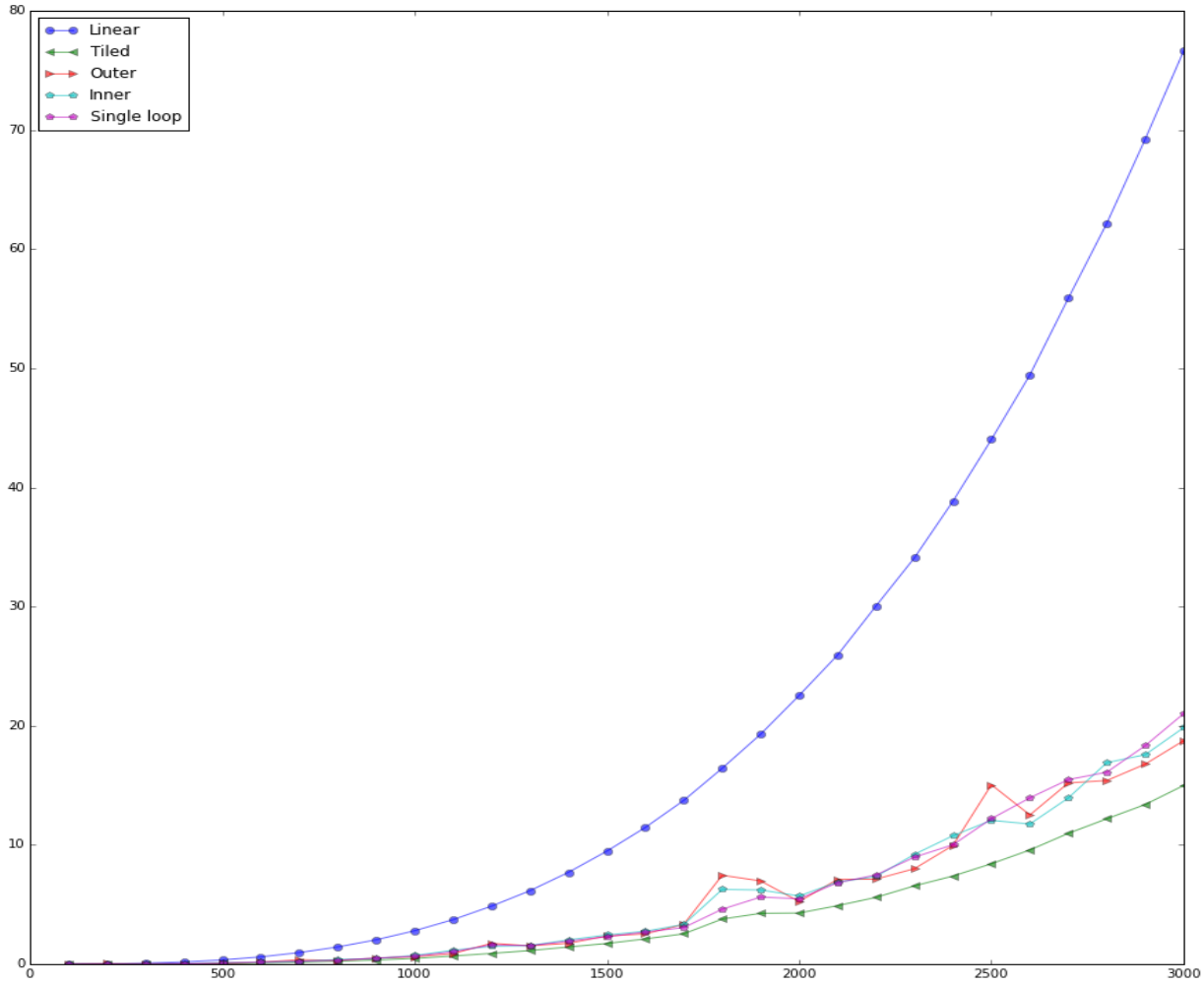
Основными параметрами параллельного алгоритма для метода Гаусса являются размеры тайла. Каким образом значение $r1$ и $r2$ вляет на производительность различных алгоритмов?

3.2.1. R = 1 (тайлы размером 1x1)



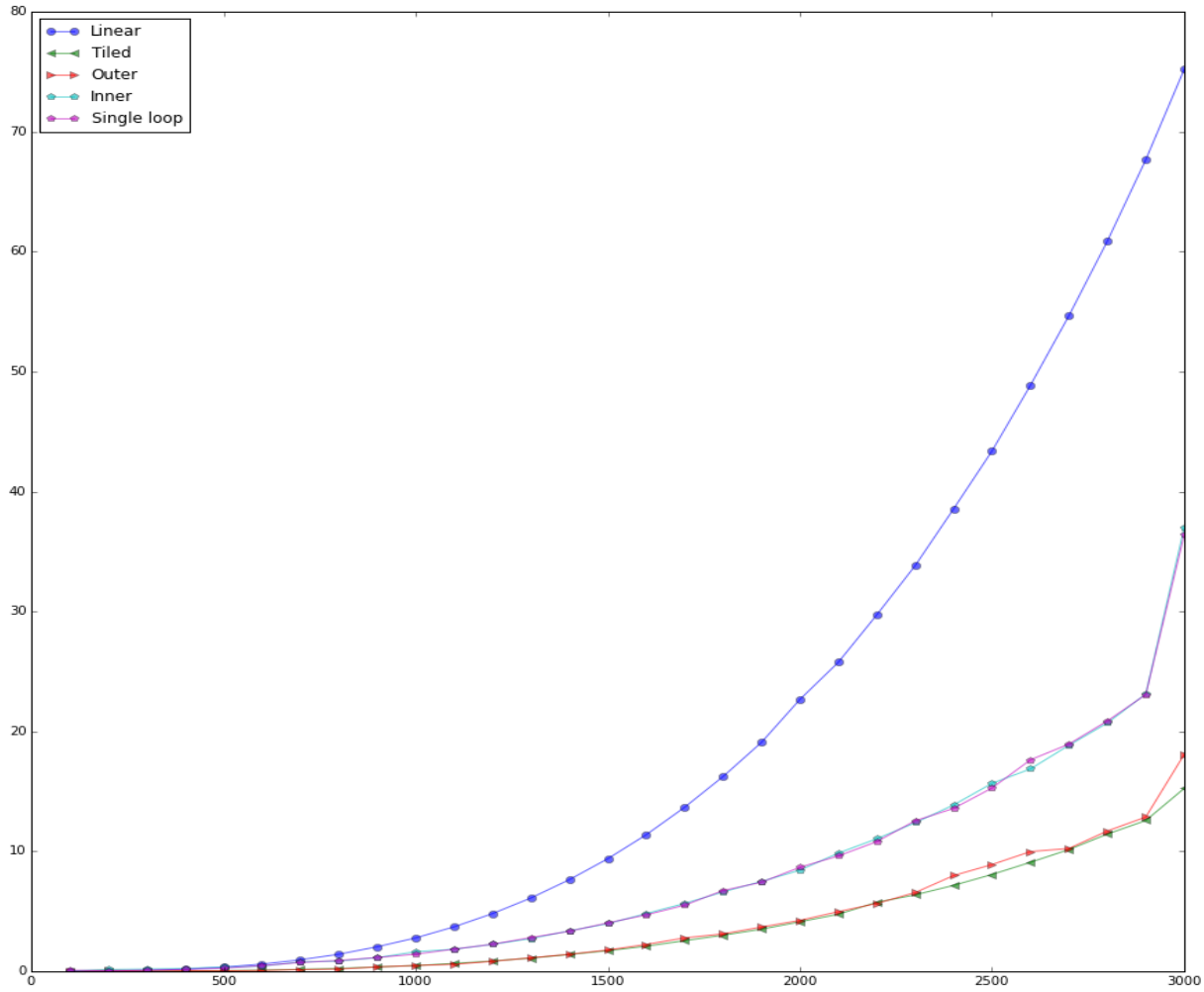
Нетрудно заметить, что искусственное уменьшение размера тайла до 1x1 создает большое количество лишних операций, которое особенно при необходимости распределения алгоритма меду несколькими процессами, лишь ухудшает время выполнения вплоть до того, что линейный алгоритм дает лучшие результаты.

3.2.2. R = N (один тайл)



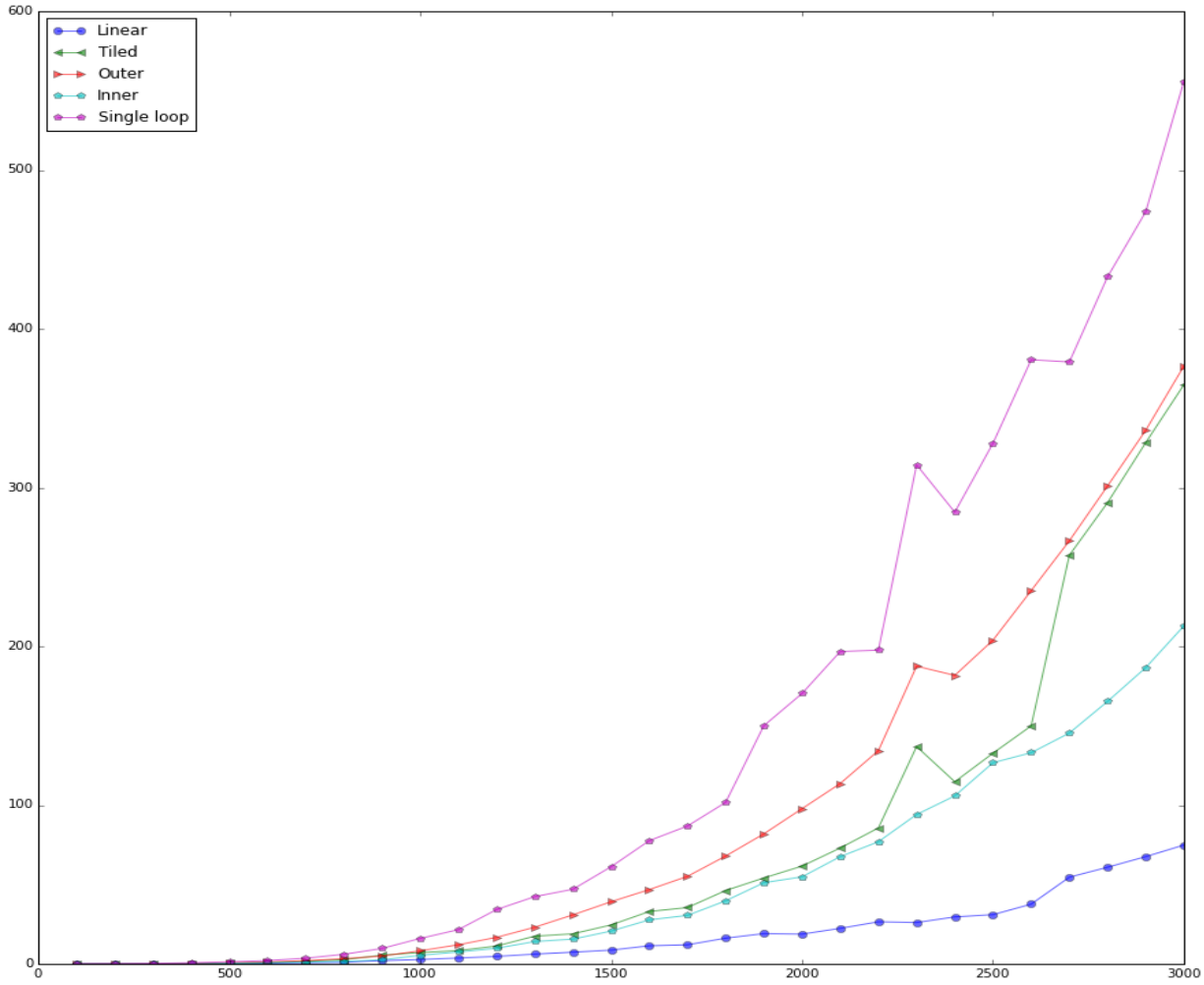
Ввиду того, что тайл всего один и обработку его не распределить между процессорами, все алгоритмы использующие тайлинг дают примерно одинаковый результат. Что характерно, линейный алгоритм в этом случае дает худший результат, так как тайловый алгоритм использует аппаратные ресурсы более оптимально.

3.2.3. R1 =1, R2 = N



При задании размера тайла таким образом, что он занимает полную строку матрицы, мы в полной мере пользуемся принципом локальности по данным, а значит высокую роль в ускорении играет кэш. Так как на тестовой машине установлен кэш больших размеров, то постройный тайлинг дает наилучшие результаты. Однако на машине без такой модификации его быстродействие будет чуть выше среднестатистического.

3.2.4. R1 = N, R2 = 1



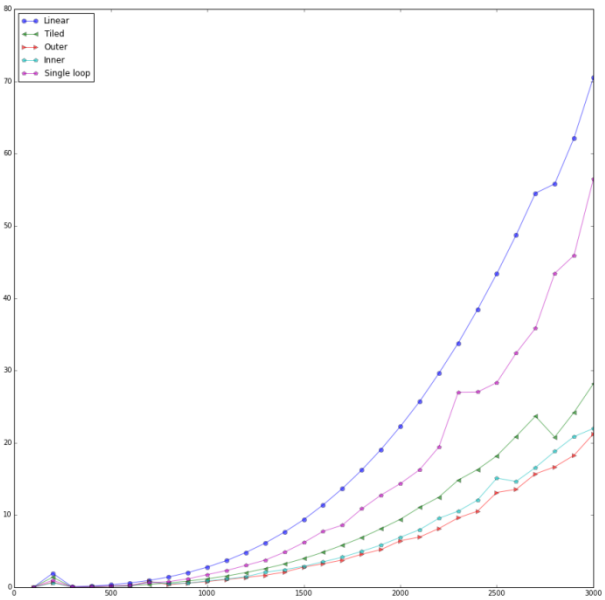
При использовании тайла, занимающего весь столбец матрицы мы наоборот, отказываемся от ускорения предлагаемого кэшом, и, так как один из размеров тайла равен 1, создаем дополнительные операции для распараллеливания. Как результат – время работы алгоритма лишь слегка лучше полученного на тайлах 1x1.

3.3.Изучение влияния размера тайла на быстродействие

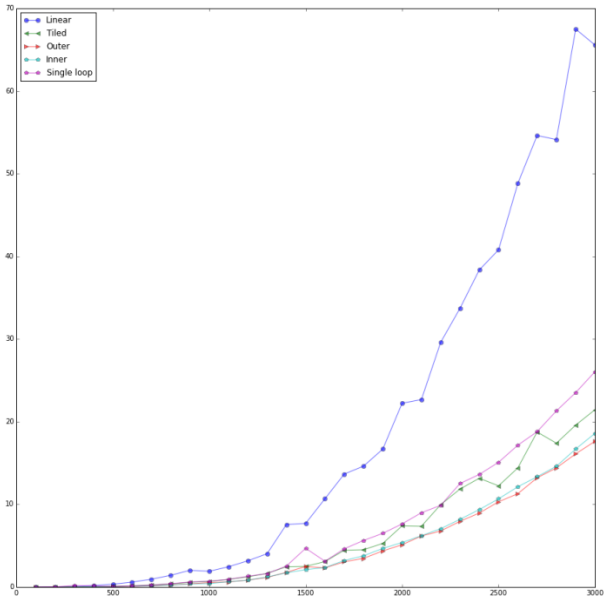
В общем случае размер тайла должен быть

1) достаточно велик чтобы ускорение, получаемое при помощи распараллеливания алгоритма на несколько процессоров, перекрывало время затраченное на организацию параллельности.
2) Достаточно мало, чтобы тайлинг имел смысл и в полной мере пользовался аппаратными ресурсами, такими как, например, принцип локальности по данным.
3) Симметричным, то есть тайл должен представлять собой квадрат, чтобы удачно и легко распределять аппаратные ресурсы.

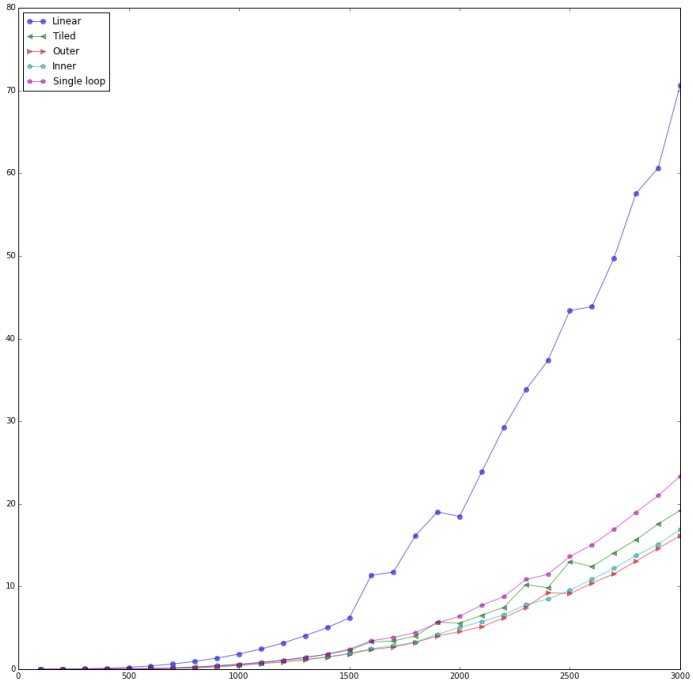
Ниже приведены графики быстродействия алгоритма Гаусса при размерах тайла $10x10, 50x50, 100x100$



Тайл размером $10x10$



Тайл размером 50x50



Размер тайла 100x100

Из графиков видно, что при размере тайла 100x100 достигается наилучший результат, хотя отличия и незначительны.

4. Заключение

В процессе работы были изучены и реализованы следующие алгоритмы метода Гаусса: линейный, тайлированый алгоритм, параллелизация внешнего и внутреннего цикла, скошенный параллелизм. Во время проведения тестирования были получены следующие результаты:

- Практически на всех вариантах размеров и формы тайлов наилучшие результаты показал алгоритм с параллелизацией внешнего цикла. Исключение составил лишь столбчатый тайлинг и эксперимент с единственным тайлом, где параллелизация внутреннего цикла оказалась быстрее.
- Хуже всего проявил себя скошенный параллелизм
- Вырожденные тайлы, кроме построчного, дают худший результат, чем тайлы правильных размеров. Построчный тайлинг, ввиду активного использования принципа локальности по данным, дает результат сравнимый с правильным тайлингом
- Тайлы правильных размеров дают гарантированное ускорение в 4 раза, при запуске на 4 процессах