**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»**

Институт компьютерных наук и кибербезопасности

ИССЛЕДОВАТЕЛЬСКАЯ РАБОТА

на тему:

Скорость вычисления языков программирования при перемножении матриц с вещественными числами

Автор работы:
Студент группы 5130203/40001
Очной формы обучения
Чеботарев Сергей Александрович

Консультант работы:
Кандидат технических наук,
Доцент — Высшая школа механики и процессов управления
Суханов Александр Алексеевич

Санкт-Петербург
2025г.

Содержание

Введение

В современной разработке каждый язык программирования занимает свою нишу, обладая уникальными особенностями. Специализация языков — это их сила: так, Python стал лидером в Data Scienc и машинном обучении, MathLab доминирует в научных и инженерных кругах, предлагая мощные инструменты для моделирования, симуляции и анализа динамических систем, а C и C++ остаются непревзойденными в задачах, требующих высокой производительности и низкоуровневого контроля. Эта расслоение создает широкий выбор для инженеров, теоретически позволяя всегда подобрать идеальный инструмент для конкретной задачи.

Однако на практике наблюдается, что разработчики зачастую не спешат осваивать и применять новые инструменты для решения специфических задач. Программист, глубоко изучивший один или два языка, стремится применять их повсеместно, даже сталкиваясь с проблемами, для решения которых эти языки изначально не предназначались. Ограничиваясь одним привычным языком, команды и отдельные специалисты могут упускать возможность добиться с меньшими затратами вычислительных ресурсов, времени и человеческих усилий более эффективного результата.

Чтобы подобные практики не получали дальнейшего распространения, просто необходимы именно наглядные сравнения. Конкретные цифры и графики — это одни из самых действенных аргументов, которые могут заставить человека задуматься и, возможно, действовать иначе.

Актуальность исследования обусловлена стремительным ростом объемов данных и сложности вычислительных задач, включая перемножение матриц, предъявляют высокие требования к производительности. В условиях, когда эффективность становится важным фактором, необходимость в объективных данных для выбора инструмента становится первостепенной. Данное исследование отвечает на этот вызов, предлагая наглядное и измеримое сравнение, которое может служить практическим руководством.

Объект исследования – процесс выполнения операции перемножения матриц с вещественными числами.

Предмет исследования – скорость выполнения (время вычисления) и эффективность использования ресурсов (памяти, процессора) при реализации алгоритма перемножения матриц на различных языках программирования.

Цель исследования – провести сравнительный анализ скорости вычислений и выявить наиболее эффективные языки и подходы для реализации операции перемножения матриц с вещественными числами.

Задачи работы:

- Реализовать алгоритм классического перемножения матриц (сложность O(n*)) на следующих языках программирования: Fortran, C, C++, Java, Pascal, Python и в среде GNU Octave (бесплатный и opensource аналог MATLAB);
- Провести серию экспериментов по замеру времени выполнения алгоритмов на матрицах различных размеров с определённым количеством повторений;
- Систематизировать результаты и представить их в виде диаграмм.

1 Исследовательская часть

В этой части работы мы разберём понятия матриц, их перемножения и рассмотрим особенности хранения данных в компьютере, которые критически влияют на производительность вычислений.

1.1 Понятие матриц и их перемножения

Матрица – это фундаментальный математический объект, представляющий собой прямоугольную таблицу чисел, символов или выражений, расположенных в строках и столбцах. В контексте вычислительных задач и программирования мы преимущественно имеем дело с числовыми матрицами. Формально матрица A размера $m \times n$ записывается как:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix},$$

где a_{ij} обозначает элемент, находящийся на пересечении i-й строки и j-го столбца.

Операция перемножения матриц является одной из ключевых операций линейной алгебры. Важно помнить, что в отличие от сложения или умножении матрицы на скаляр, умножение матриц не является поэлементным. Умножение определено только для согласованных матриц: количество столбцов первой матрицы (A, размер m×k) должно равняться количеству строк второй матрицы (B, размер k×n). Результатом умножения будет новая матрица C размера m×n.

Каждый элемент c_{ij} итоговой матрицы C вычисляется как скалярное произведение i-й строки первой матрицы (A) на j-й столбец второй матрицы (B):

$$c_{ij} = \sum_{l=1}^{k} a_{il} * b_{lj} .$$

Пример:
Умножим матрицу A (2×3) на матрицу B (3×2):

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \qquad B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$$

И получим матрицу C (3×3):

$$C = \begin{pmatrix} 1*7+2*9+3*11 & 1*8+2*10+3*12 \\ 4*7+5*9+6*11 & 4*8+5*10+6*12 \end{pmatrix} = \begin{pmatrix} 58 & 64 \\ 139 & 154 \end{pmatrix}$$

1.2 Особенности хранения данных в памяти и их влияние на производительность

Производительность операции перемножения матриц зависит не только от вычислительной сложности алгоритма, но и в огромной степени от того, как данные организованы в памяти компьютера, и как процессор имеет к ним доступ.

1.2.1 Кэш-память

Современные компьютерные архитектуры основаны на иерархии памяти: от быстрых, но маленьких регистров и кэшей процессора до медленной, но объёмной оперативной памяти (RAM). Скорость доступа к кэшу может быть в десятки раз выше, чем к RAM.

Что такое кэш-память (далее - кэш)? Кеш — промежуточный буфер с быстрым доступом к нему, содержащий информацию, которая может быть запрошена с наибольшей

вероятностью. И чтобы понять какие данные уйдут в кэш введём понятие “локальность обращения”, суть которого можно представить в двух пунктах:

1. Если к данным обратились однажды, к ним, вероятно, обратятся снова в ближайшем будущем;
2. Процессор, обращаясь к одному элементу данных, с высокой вероятностью в ближайшем будущем обратится к соседним элементам. Память подгружается в кэш блоками (кэш-линиями).

1.2.2 Способы хранения матриц в памяти

Матрицы, как двумерные структуры, должны быть отображены на линейное адресное пространство памяти. Существует три основных способа.

Первый способ – хранение по строкам (Row-Major order): элементы матрицы располагаются в памяти последовательно, строка за строкой. Этот способ используется в языках C, C++, Java, Pascal и многих других.

$$\text{Матрица: } \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$
$$\text{Память: } [1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6]$$

Второй способ – хранение по столбцам (Column-Major order): элементы матрицы располагаются в памяти последовательно, столбец за столбцом. Этот способ используется в языках Fortran, MATLAB:

$$\text{Матрица: } \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$
$$\text{Память: } [1 \quad 4 \quad 2 \quad 5 \quad 3 \quad 6]$$

Третий способ – хранение в абстракции массивов (Neither Row-Major Nor Column-Major): система не обязана хранить данные в едином, непрерывном блоке памяти, строго упорядоченном по строкам или столбцам, поэтому современные высокоуровневые языки (такие как Python) и библиотеки используют общую концепцию – страйд (stride).

Страйд – это количество байтов, которое необходимо пропустить в линейной памяти, чтобы переместиться к следующему элементу вдоль определенной оси массива.
Для массива 2D A[m][n]:

- **Страйд по строкам (stride[0]):** Сколько байтов нужно пропустить, чтобы перейти от строки i к строке i+1 (при фиксированном столбце).
- **Страйд по столбцам (stride[1]):** Сколько байтов нужно пропустить, чтобы перейти от столбца j к столбцу j+1 (при фиксированной строке).

Разберёмся подробнее:

Допустим, у нас есть матрица 2×3 32-битных целых чисел (каждое занимает 4 байта).

- В классическом **Row-Major**: $\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \rightarrow [a_{11} \quad a_{12} \quad a_{13} \quad a_{21} \quad a_{22} \quad a_{23}]$:

  Страйд по строкам: чтобы попасть из a_{11} в a_{21}, нужно пропустить 3 элемента (всю первую строку). 3 элемента * 4 байта = 12 байт.

  Страйд по столбцам: чтобы попасть из a_{11} в a_{21}, нужно пропустить 1 элемент. 1 элемент * 4 байта = 4 байта.

  Итого, страйд: [12, 4]

- В классическом **Column-Major**: $\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \rightarrow [a_{11} \quad a_{21} \quad a_{12} \quad a_{22} \quad a_{13} \quad a_{23}]$:

  Страйд по строкам: чтобы попасть из a_{11} в a_{21}, нужно пропустить 1 элемент. 1 элемент * 4 байта = 4 байта.

  Страйд по столбцам: чтобы попасть a_{11} в a_{21}, нужно пропустить 2 элемента. 2 элемента * 4 байт = 8 байт.

  Итого, страйд: [4, 8]

Страйды хороши в том, чтобы описать структуры, которые не являются ни чисто строчными, ни чисто столбцовыми.

Замечание: во время экспериментальной части было выявлено, что даже с абстракцией массивов языки, использующие эту технологию, по умолчанию прибегают к использованию хранению по строкам.

Приведу примеры для лучшего понимания абстракции массивов и как они могут ускорить работу с массивами и более эффективно работать с памятью:

Пример 1: Транспонированная матрица без копирования

Пусть у нас есть большая матрица $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$, хранящаяся в памяти как Row-Major. Нам нужна транспонированная матрица A^T. Вместо того чтобы создавать новую матрицу и копировать все элементы, мы можем просто создать новый объект с измененными метаданными:

Исходная матрица A (2×3, Row-Major): $[a_{11} \quad a_{12} \quad a_{13} \quad a_{21} \quad a_{22} \quad a_{23}]$:

- Размер: [2, 3];
- Страйды: [12, 4] (пропускаем 3 элемента к следующей строке, 1 элемент к следующему столбцу).

Транспонированная матрица A^T (3×2): мы меняем размер и страйды местами:

- Размер: [3, 2];
- Страйды: [4, 12] (пропускаем 1 элемент к следующей строке, 3 элемента к следующему столбцу).

В итоге, новый объект A^T ссылается на ту же самую область памяти, что и A, но его схема доступа к элементам не является ни Row-Major, ни Column-Major. Это гибрид, определяемый страйдами [4, 12]. Элемент $A^T[1][0]$ будет соответствовать элементу A[0][1].

Пример 2: Срезы и подматрицы

Возьмем матрицу B размера 100×100 (Row-Major). Мы хотим работать только с каждым 5-м столбцом.

Исходная матрица B:

- Размер: [100, 100];
- Страйды: [400, 4] (предположим, 4 байта на элемент).

Подматрица B_sub = B[:, ::5] (все строки, каждый 5-й столбец):

- Размер: [100, 20];
- Страйды**:** [400, 20] (страйд по столбцам теперь 5 элементов * 4 байта = 20 байт).

Получившаяся подматрица не является непрерывным блоком памяти в стандартном понимании Row-Major. Ее столбцы разбросаны в памяти с шагом в 20 байт.

Пример 3: Массивы с дополнением

Иногда данные в памяти специально выравнивают для оптимизации под конкретную архитектуру (например, под размер кэш-линии). Матрица 3×3 может храниться в памяти как блок 4×4, где последний столбец и строка — "пустые" (P – padding – “набивка”).

- Фактическая память:

$[a_{11} \quad a_{12} \quad a_{13} \quad P \quad a_{21} \quad a_{22} \quad a_{23} \quad P \quad a_{31} \quad a_{32} \quad a_{33} \quad P \quad P \quad P \quad P \quad P]$;

- Логический размер: [3, 3];
- Страйды: [16, 4] (переход к следующей строке = пропуск 4 элементов по 4 байта).

Эта схема хранения также не подпадает ни под чисто строчное, ни под чисто столбцовое представление.

1.2.3 Влияние способа хранения на производительность перемножения

Рассмотрим фрагмент кода для перемножения матриц (выполнен на языке C/C++):

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Доступ к матрице A: A[i][k]. При хранении по строкам доступ к элементам k в строке i является последовательным. Это идеальная “локальность обращения”, предсказуемая для предварительной подгрузки в кэш.

Доступ к матрице B: B[k][j]. При хранении по строкам мы перебираем индекс k, что означает прыжки по разным строкам для доступа к одному и тому же столбцу j. На каждой итерации внутреннего цикла мы обращаемся к элементу, отстоящему далеко от предыдущего. Это приводит к промахам кэша (после первого обращения кэш не запомнил нужные нам следующие данные), когда процессу постоянно приходится обращаться к медленной оперативной памяти.

1.2.4 Оптимизация через изменение порядка циклов

Простая перестановка циклов может кардинально изменить производительность. Рассмотрим вариант:

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        for (int j = 0; j < n; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Теперь оба внутренних цикла (k и j) работают с последовательными участками памяти:

- A[i][k] — фиксированная строка i, последовательный доступ по k.
- B[k][j] — фиксированная строка k, последовательный доступ по j. Элементы строки B[k][j] лежат в памяти друг за другом.

Такой подход снижает количество промахов кэша и может ускорить выполнение операции в несколько раз, при этом такой порядок становится всё эффективней по мере увеличение массивов по сравнению с другими переборами. Это наглядный пример того, как понимание архитектуры и организации данных позволяет писать высокопроизводительный код, и почему один и тот же алгоритм, реализованный на одном языке, но с разной оптимизацией доступа к памяти, может показывать разные результаты. Эти особенности будут проявляться по-разному в зависимости от того, как язык программирования и его компилятор или интерпретатор управляют памятью.

2 Экспериментальная часть

В этой части мы напишем программы для перемножения матриц на языках Fortran, C, C++, Java, Pascal, Python и рассмотрим код Python при использовании NumPy. Далее мы сравним скорость их работы при разных размерах (в файлах по умолчанию - 500×500) квадратных матриц при повторении операции по 100 раз (для наглядного увеличения времени работы программ).

2.1 Кода программ

Чтобы “гонка” языков была честной, будем реализовывать все программы на основе файла “SPEED2.FOR”, меняя в функции перемножения матриц перестановки при тройном цикле, следуя **1.2.2 Способы хранения матриц в памяти**. Программы будут работать по алгоритму: выделяем память для матриц, заходим в цикл повторений, заполняем эти матрицы и вызываем функцию их перемножений. Для удобства установки компиляторов будем работать в виртуальной машине с оперативной системой Linux Ubuntu. Запуск всех программ можно посмотреть в файле “speedtest.sh”.

2.1.1 Fortran

Вопреки тому, что у нас уже имеется файл “SPEED2.FOR”, но при попытке компиляции терминал, скорее всего выдаст ошибку, так как в файле используются структура, которая считается неактуальной, и чтобы всё равно проверить работу компилируем файл насильно (в пункте 2.1 указано, где можно посмотреть).

Чтобы сравнение языков было более честным перепишем код по современному подходу – файл “NEWSPEED.f90”.

2.1.2 C

Двойные массивы в C, C++ нельзя создать в статической памяти, поэтому для них мы выделяем динамическую память. Это также значит, что после перемножений нам нужно эту память освободить. Файл - “speedtest_c.c”.

2.1.3 C++

В C++ инициализация данных в динамической памяти выполняет команда new, а освобождения – delete (конечно, можно использовать команды C, но мы же разделяем языки для проверки скорости программ). Файл - “speedtest_cpp.cpp”.

2.1.4 Java

Java мне известна только поверхностна, поэтому, опираясь на предыдущие программы, сгенерировал код через искусственный интеллект и проверил на логические недочёты тем, что мне известно про этот язык. Также выделяем динамическую память, но теперь нам не надо её освобождать вручную – это сделает сборщик мусора (Garbage Collector) автоматически, когда объекты станут недостижимы. Файл - “speedtest_java.java”.

2.1.5 Pascal

Так же, как и с Java, опираясь на предыдущие программы, сгенерировал код через искусственный интеллект и проверил на логические недочёты. Файл - “speedtest_pascal.pas”.

2.1.6 Python

Программа на Python написан уже мной. Код на нём интуитивна понятен, единственное, что может смутить начинающего – задание двойных списков. Файл - “speedtest_python.py”.

2.1.7 NumPy

Хоть цель исследования выявить наиболее эффективные языки по скорости перемножении матриц, а в введении писалось, что из-за ограниченности программист упускает возможность добиться с меньшими затратами различных ресурсов более

эффективный результат, но не могу не заметить, что иногда язык имеет возможность зайти на “территорию” другого и, возможно, получить конечный результат лучше, чем изучение и использование других средств разработки. Эта возможность – библиотеки языков.

NumPy – библиотека (пакет) языка Python, специализирующийся на научных вычислениях. Он быстрый, потому что перекладывает тяжелую работу на проверенные десятилетиями низкоуровневые библиотеки (BLAS/LAPACK), написанные на C и Fortran и оптимизированы для работы с памятью, ускоренные для вычислений чуть ли не до возможного максимума.

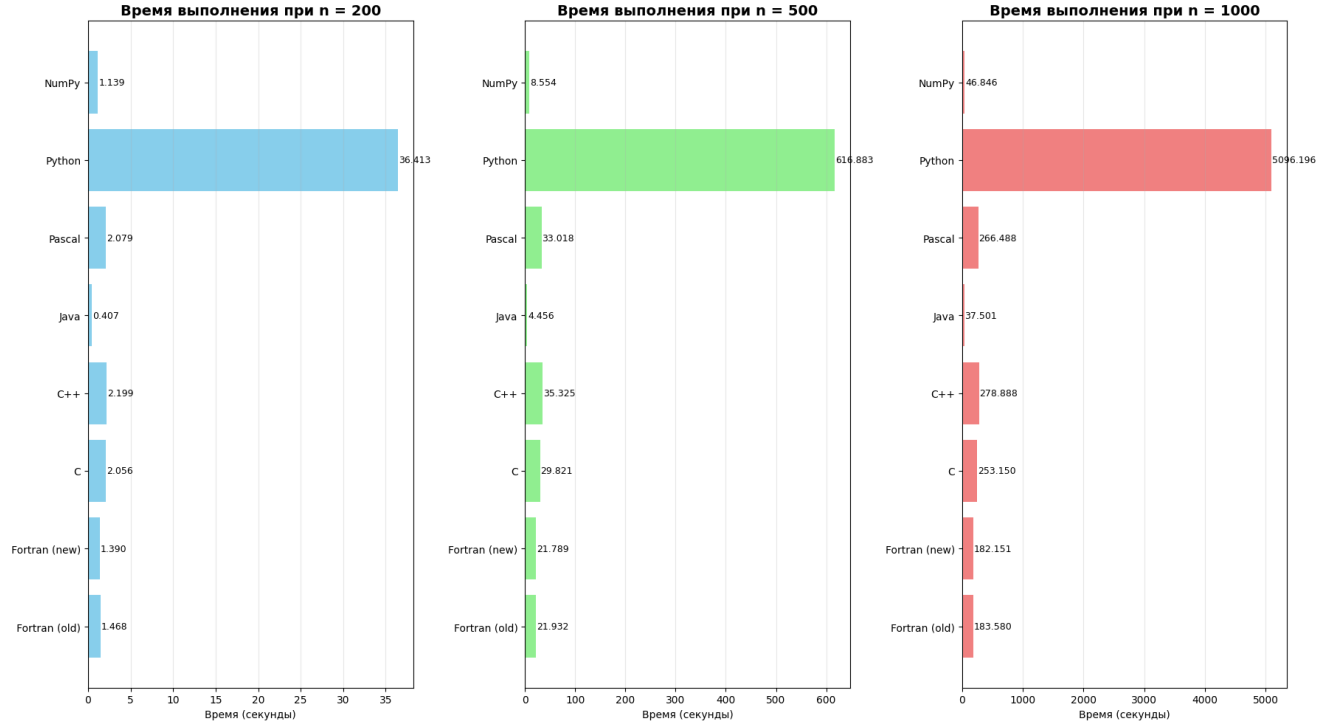
Файл - “speedtest_numpy.py”.

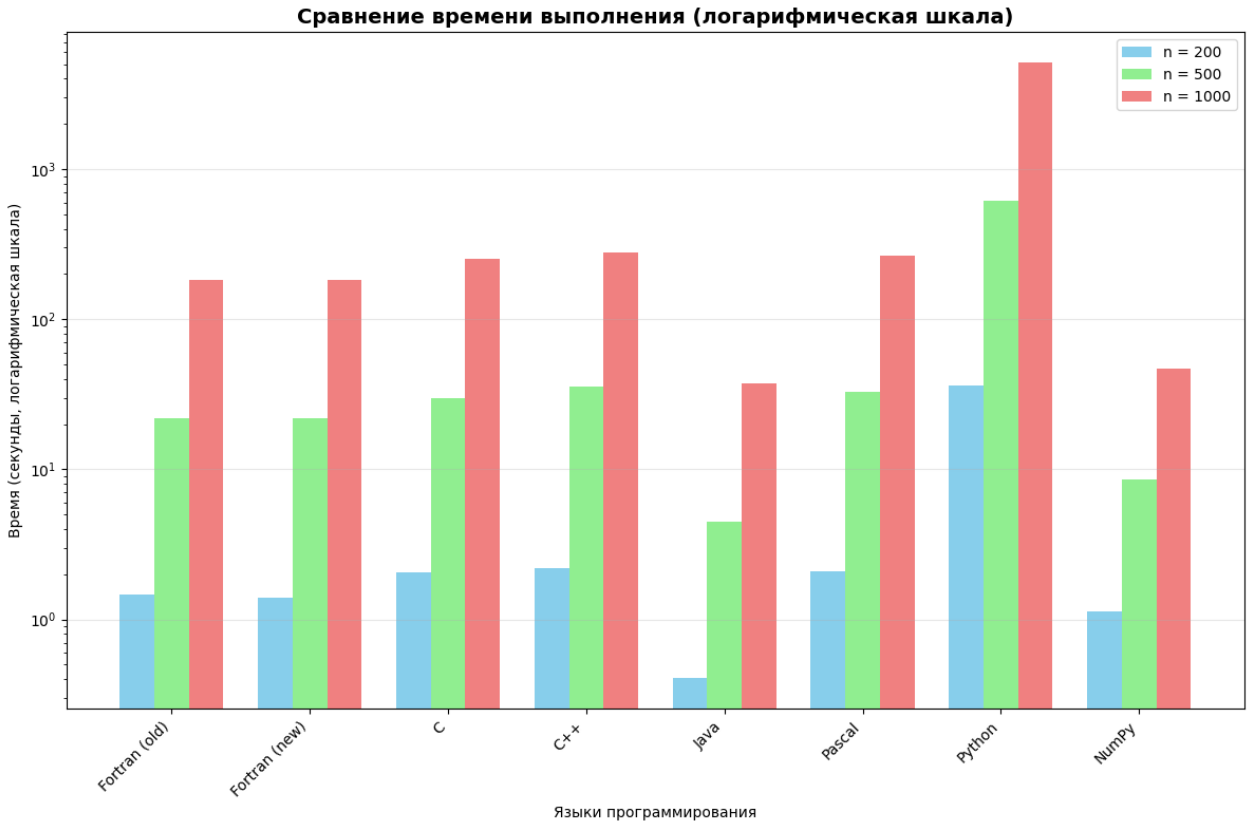
2.2 Сравнение работы программ

Сравним скорость работы программ в секундах при разных размерностях квадратных матриц при помощи таблицы:

Размерность матрицы (n×n)	Fortran (old)	Fortran (new)	C	C++	Java	Pascal	Python	NumPy
100	0.206	0.203	0.287	0.297	0.094	0.269	4.798	0.482
200	1.468	1.390	2.056	2.199	0.407	2.079	36.413	1.139
300	5.068	5.098	6.533	7.404	0.978	6.939	136.171	2.982
400	12.055	11.645	15.733	17.265	2.228	17.017	315.512	5.490
500	21.932	21.789	29.821	35.325	4.456	33.018	616.883	8.554
750	75.951	76.823	105.589	120.786	14.762	111.735	2159.524	21.992
1000	183.580	182.151	253.150	278.888	37.501	266.488	5096.196	46.846

Визуализируем значения через диаграммы строк при n = 200, 500, 1000:





Важно понимать, что данные измерения времени непостоянны и будут иметь небольшую погрешность: при каждом запуске время работы будут отличаться на маленькую величину.

2.3 Анализ работы программ

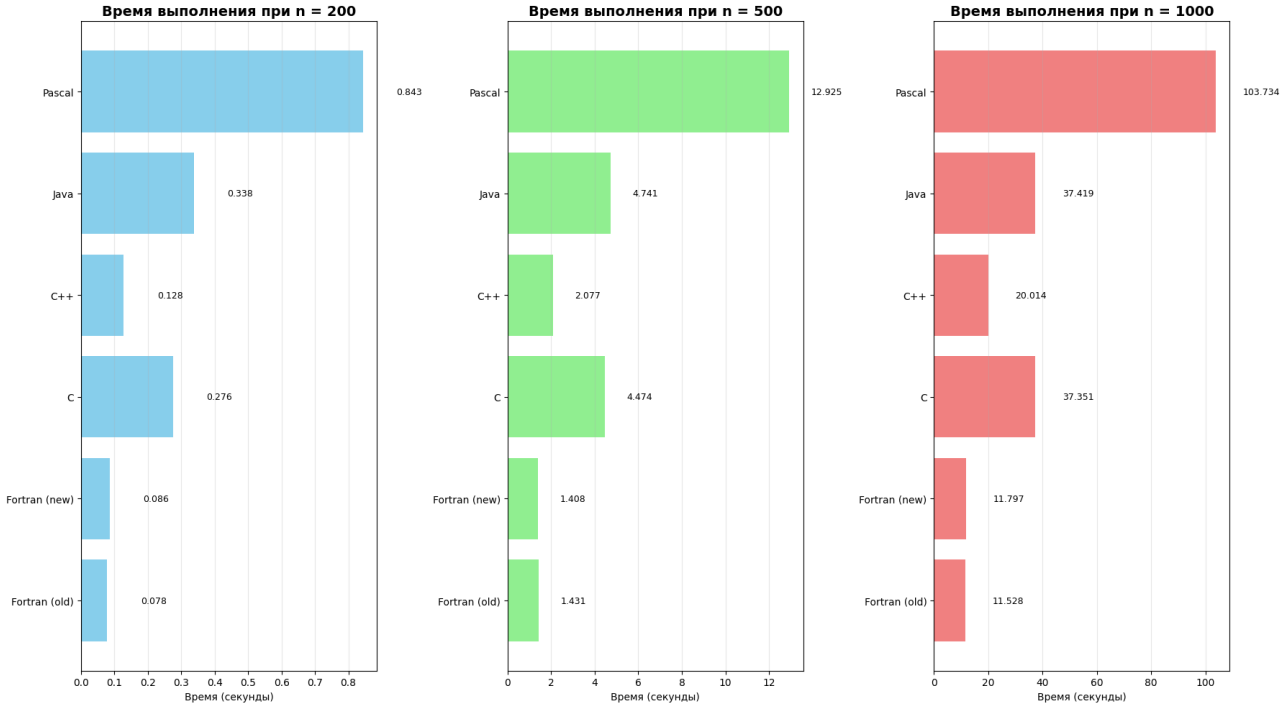
Сразу можем заметить, что почему-то Java всех обходит, хотя широко известно, что в этой гонке есть языки, которые намного быстрее в вычислении. Почему так произошло? Секрет в том, что Java использует JIT-компиляцию. Изначально код выполняется в интерпретаторе, но "горячие" участки кода (такие как внутренний цикл в dmul2, который выполняется миллионы раз) компилируются в высоко-оптимизированный машинный код непосредственно во время работы программы.

Да и сам по себе HotSpot JVM (компилятор Java) – это результат многолетней разработки и тончайшей настройки, спонсируемой такими крупными компаниями, как Oracle и Red Hat. Она оптимизирована для серверных приложений, где производительность циклов критична.

Давайте проведём повторный запуск программ с этими параметрами, но попросим компиляторы провести “агрессивную оптимизацию” с помощью флага “-O3 -march=native” (Для Pascal - “-O3”). И так как мы оптимизируем компиляторные языки, то Python и NumPy оставим те же.

Размерность матрицы (n×n)	Fortran (old)	Fortran (new)	C	C++	Java	Pascal	Python	NumPy
100	0.014	0.013	0.036	0.018	0.086	0.107	4.798	0.482
200	0.078	0.086	0.276	0.128	0.338	0.843	36.413	1.139
300	0.271	0.285	0.978	0.444	1.013	2.782	136.171	2.982
400	0.718	0.714	2.272	1.085	2.295	6.686	315.512	5.490
500	1.431	1.408	4.474	2.077	4.741	12.925	616.883	8.554
750	4.909	4.863	15.286	7.123	15.002	44.301	2159.524	21.992
1000	11.528	11.797	37.351	20.014	37.419	103.734	5096.196	46.846

Чтобы чётче показать разницу между скоростями, на диаграмме оставим только то, что ускоряли и Java:



Скорость компилируемых языков, кроме Java (мы её и не “ускоряли”) примерно выросла в 2-17 раз.

2.4 Вывод

На основе пунктов 2.2 и 2.3 можно сделать несколько умозаключений.

Во-первых, лидером этой “гонки” становится Fortran, показывающий наилучшие результаты после агрессивной оптимизации, особенно на больших размерностях матриц. Ему немного уступает C++. Это ожидаемо, так как эти языки исторически используются для высокопроизводительных научных вычислений и дают компилятору большую свободу для низкоуровневых оптимизаций.

Во-вторых, тот факт, что Java, показывает сравнимую с оптимизированным C результат, является впечатляющим. Это демонстрирует эффективность современного JIT-компилятора, на котором и работает Java и который способен проводить агрессивную оптимизацию "на лету" во время выполнения программы. Вероятно, в начальном тесте (без -O3 для других языков) JIT-компиляция дала Java значительное преимущество.

В-третьих, интерпретируемые языки (Python), как и ожидалось, сильно отстают из-за интерпретируемой природы и динамической типизации. Однако использование библиотеки NumPy, которая под капотом использует оптимизированные библиотеки высокопроизводительных научных языков, позволяет ускорить вычисления на порядки. NumPy демонстрирует время, сравнимое с компилируемыми языками на малых размерностях и приемлемое на больших.

И, наконец, при увеличении размерности задачи в 10 раз (со 100 до 1000) время выполнения растёт, что соответствует сложности алгоритма умножения матриц $O(n^3)$. Это подтверждает, что тест является вычислительно-насыщенным и корректно измеряет производительность.

3 Заключение

Проведённое исследование позволило наглядно сравнить производительность различных языков программирования при выполнении одной из операций линейной алгебры — перемножении матриц. Результаты эксперимента подтвердили, что выбор языка и способа оптимизации кода оказывают существенное влияние на скорость вычислений, особенно при работе с большими объёмами данных.

Наибольшую эффективность продемонстрировали компилируемые языки, такие как Fortran и C++, особенно при использовании агрессивной оптимизации компилятора. Их преимущество обусловлено низкоуровневым доступом к памяти и возможностью тонкой настройки вычислительных процессов. Хорошо себя показали и C, Java и Pascal.

Неожиданно высокие результаты Java без дополнительных оптимизаций подчёркивают важность современных технологий JIT-компиляции, которые позволяют достигать производительности, сопоставимой с традиционными компилируемыми языками.

С другой стороны, интерпретируемые языки, в частности Python, показали значительное отставание в скорости выполнения алгоритма. Однако использование специализированных библиотек, таких как NumPy, позволило сократить это отставание на огромный порядок, что подчёркивает важность применения профильных инструментов для решения специфических задач.

Хотелось бы закончить на том, что оптимизировать алгоритмы можно бесконечно, и для каждого языка найдутся свои “фокусы”, обусловленные его особенностями. Однако в условиях нехватки времени или высокой сложности разработки часто эффективнее не переключаться на изучение нового языка, а глубже освоить инструменты уже знакомого, подходящие для конкретной задачи. Программист и инженер должны уметь трезво оценивать свои силы, возможности и принимать рациональные решения!