УДК 004.052.42

На пути к дедуктивной верификации реализаций умножения матриц с направленными на повышение эффективности использования кэш-памяти оптимизациями*

Агибалов М.С. (Новосибирский государственный университет)

Кондратьев Д.А. (Институт систем информатики им. А.П. Ершова СО РАН)

Задача повышения производительности умножения матриц является актуальной задачей современного программирования. При решении данной задачи в программном коде умножения матриц реализуют оптимизации, что затрудняет анализ полученных реализаций и может приводить к появлению в них ошибок. Гарантировать корректность программного кода относительно спецификаций, описывающих результат работы программы в зависимости от ее входных данных, может только дедуктивная верификация. Итого, задача дедуктивной верификации реализаций умножения матриц с направленными на повышение производительности оптимизациями является актуальной. В данной статье представлен первоначальный этап исследования, нацеленный на дедуктивную верификацию реализаций классического умножения матриц с направленными на повышение эффективности использования кэш-памяти оптимизациями.

Ключевые слова: дедуктивная верификация, умножение матриц, Frama-C, WP, Coq, Rocq, кэш-память

1. Введение

Задача повышения производительности умножения матриц является актуальной задачей современного программирования. Решение данной задачи особенно важно для работы с широко распространенными в настоящее время сверточными нейронными сетями, основанными на умножении тензоров. При решении задачи быстрого умножения матриц полезно определить особую реализацию, с которой можно сравнивать другие подходы и которая при этом относительно эффективна. Такой применяемой при работе со сверточны-

*Проект поддержан Лабораторией YADRO НГУ

ми нейронными сетями и во многом ориентированной на сравнение реализацией является minigemm [11, 86], разработанная в компании YADRO.

При задачи повышения производительности умножения матриц в программном коде умножения матриц реализуют оптимизации, что затрудняет анализ полученных реализаций и может приводить к появлению в них ошибок Обычно корректность программ проверяется тестированием, и тестирование имеет цель – найти ошибки в программах, однако оно не может гарантировать отсутствие ошибок. Гарантировать корректность программ может только формальная верификация [3, 4, 6, 13, 31]. Для проверки корректности исходного кода программ относительно спецификаций, описывающих результат работы программы в зависимости от ее входных данных, применяется такой вид формальной верификации, как дедуктивная верификация [10, 14, 16, 19, 20, 43, 46, 49, 51].

В случае умножений матриц с оптимизациями важно верифицировать такое свойство, что результат такого умножения совпадает с результатом классического умножения матриц. В таком случае в формальных спецификациях верифицируемой реализации оптимизированного умножения матриц будет описано, что результат оптимизированного умножения будет равен результату классического умножения матриц. Таким образом дедуктивная верификация позволяет проверить функциональную эквивалентность неэффективной, но эталонной классической реализации умножения матриц и эффективной, но из-за сложности требующей тщательного анализа реализации оптимизированного умножения матриц.

Итого, задача проверки с помощью дедуктивной верификации функциональной эквивалентности реализации minigemm и классического умножения матриц является актуальной. Поэтому, в Лаборатории YADRO НГУ был запущен проект "Методы и средства дедуктивной верификации реализаций умножения матриц с направленными на повышение производительности оптимизациями", целью которого является дедуктивная верификация реализации minigemm. Данный проект выполняют авторы данной статьи, а именно студент НГУ Агибалов Матвей Сергеевич под руководством к.ф.-м.н., научного сотрудника ИСИ СО РАН и старшего преподавателя НГУ Кондратьева Дмитрия Александровича.

Однако дедуктивная верификация реализации minigemm осложнена из-за наличия в minigemm разнообразных модификаций относительно классического умножения матриц, вещественной арифметики, традиционно представляющей сложности для дедуктивной верификации, и ассемблерных вставок, не позволяющими верифицировать всю реализацию

только как код на языке C. Чтобы преодолеть данные сложности, было принято решение выполнять данный проект в итеративном порядке так, чтобы на каждой итерации проекта накапливался опыт и создавался задел для последующих итераций.

Первой итераций данного проекта стала работа на Зимнем Системном Буткемпе лаборатории YADRO НГУ. Первоочередной задачей данного проекта было обучение участника проекта, поэтому в ходе данного проекта была начата дедуктивная верификация слабосвязанной с minigemm, но зато относительно простой для понимания блочного умножения матриц реализации алгоритма Штрассена [83]. Умножения матриц с оптимизациями обычно в целях производительности реализуют на основанном на указателях языке программирования C, поэтому на данном этапе проекта применялась ориентированная на поддержу работы с указателями с помощью сепарационной логики [52, 72, 73, 77, 78] система дедуктивной верификации VST [17, 18, 32]. Система VST основана на системе доказательства Coq [25, 76], недавно переименованной в Rocq. Негативным опытом по итогам первой итерации проекта стало понимание, что проводить дедуктивную верификацию в системе VST сложно из-за того, что в данной системе нет возможности при работе с вещественной арифметикой использовать ее идеальное (математическое) представление, а есть только работа с машинной вещественной арифметикой по стандарту IEEE 754, что приводит к чрезмерному усложнению начального этапа проекта. Также негативным опытом стало понимание, что работать в системе доказательства VST относительно сложно из-за применяемого в системе VST основанного на прямом прослеживании программы исчисления сильнейшего постусловия [41, 48], приводящего к сложностям при прямом прослеживании программы задавать инвариант цикла путем модификации постусловия [47]. При этом по итогам первой итерации проекта были получен важный и пригодившийся потом опыт работы в системе Coq, а именно удалось задать в Coq-представлении спецификации реализации алгоритма Штрассена и доказать в системе Coq свойство, что если размер матрицы не подходящий для алгоритма Штрассена (не кратен степени двойки), то вместо него применяется классическое умножение матриц. Награждение по итогам Зимнего Системного Буткемпа лаборатории YADRO НГУ приведено на рисунке 1.

Итого, по итогам первого этапа проекта было принято решение перейти от использования системы VST к использованию системы Frama-C [21] с плагином WP [35, 36], где есть поддержка идеальной машинной арифметики и основанного на обратном прослеживании программы исчисления слабейшего предусловия [23, 40, 45, 62], упрощающего применение

Рис. 1. Награждение по итогам Зимнего Системного Буткемпа лаборатории YADRO НГУ

при задании инвариантов циклов метода модификации постусловия.

Также по итогам первого этапа проекта с целью упрощения итераций было принято решение сначала проводить дедуктивную верификацию в идеальной (математической) вещественной арифметике, и только потом переходить к верификации в машинной арифметике [50].

Кроме того, по итогам первого этапа проекта было принято решение отказаться от продолжения дедуктивной верификации алгоритма Штрассена и перейти к дедуктивной верификации последовательности реализаций умножений матриц из статьи [5], где описывается эффективная реализация умножения матриц шаг за шагом. Данная последовательность реализаций была выбрана потому, что она, стартуя с классического умножения матриц, приводит к реализации, приближающейся по структуре к minigemm. Началом такой последовательности служат реализации, где применяются направленные на повышение эффективности использования кэш-памяти оптимизации, а именно изменения поряд-

ка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы [12, 84]. Поэтому, целью работы на текущем этапе проекта является разработка комплексного подхода к дедуктивной верификации реализаций умножения матриц над математическими вещественными числами с направленными на повышение эффективности использования кэш-памяти оптимизациями. Для достижения данной цели поставлены следующие задачи:

1. Задание спецификаций для реализаций классического умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы.
2. Задание теории предметной области для верификации реализации классического умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы.
3. Разработка стратегий доказательства условий корректности при дедуктивной верификации реализации классического умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы.
4. Проведение с помощью разработанного комплексного подхода экспериментов по дедуктивной верификации реализации классического умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы.

В данной статье мы презентуем текущие результаты нашей работы, общедоступные в репозитории проекта [1].

Обзор родственных работ В качестве основной родственной работы отметим статью [85] про реализацию в компиляторе CompCert оптимизации в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы. Отличительной особенностью компилятора CompCert [28, 29, 63, 64] является формальная верификация его кода, и рассматриваемая оптимизация также была формально верифицирована. В ходе формальной верификации данной оптимизации

было проверено свойство, что ее применение не нарушает корректность оптимизируемой программы. Отметим, что такая формальная верификация довольна схожа с задачами текущего этапа нашего проекта. Также верификация осуществлялась в системе доказательства Coq, как в многом происходит и в нашем проекте. Однако, в случае верифицируемого компилятора при верификации осуществлялась работа с промежуточными представлениями C-программы, тогда как в нашем проекте верифицируется непосредственно исходный код. Кроме того, отличаются глобальные цели проектов: в проекте CompCert глобальной целью является расширение компилятора как можно большим числом верифицируемых оптимизаций, тогда как в нашем проекте глобальной целью является создание подходов к дедуктивной верификации реализаций умножения матриц с оптимизациями. Стремление расширить компилятор CompCert различными оптимизациями демонстрируется исследованием [55], где предлагаются формально верифицированные оптимизации для циклов. Однако, предложенные в данном исследовании оптимизации мало похожи на встречаемые в нашем проекте.

Отдельным классом родственных работ являются исследования [33, 39, 74] по дедуктивной верификации алгоритма Штрассена. Данные исследования привели к разработке полезных библиотек [39, 74] с теоремами о матрицах для систем доказательства Coq и ACL2 [69]. Однако, в данных исследованиях доказывались свойства программ на функциональных языках программирования WhyML (промежуточный язык верификации в системе Frama-C) [30, 44, 60], Gallina (входной язык системы доказательства Coq) [25, 76] и Applicative Common Lisp (входной язык системы доказательства ACL2) [54], что, в отличие от нашего проекта, упростило верификацию из-за отсутствия необходимости работать с моделью памяти языка C. Также алгоритм Штрассена удавалось синтезировать по спецификации [79], что близко к методам дедуктивной верификации, однако далеко от проблем дедуктивной верификации программ в модели памяти языка C.

Также в качестве родственного исследования рассмотрим дедуктивную верификации частей операционной системы в системе дедуктивной верификации AstraVer [7, 42, 65, 67, 87]. Но используемые в системе AstraVer для доказательства условий корректности такие автоматические системы, как SMT-решатели, могут не справляться с моделированием работы с памятью в C-программах [66].

Кроме того, в качестве родственного исследования рассмотрим исследование по разработке методов автоматизации дедуктивной верификации программ на подмножестве

языка программирования C [70, 71] и реализации таких методов в системе C-lightVer [58, 59, 68]. Данная система позволяет упростить доказательство с помощью генерации вспомогательных лемм по определенным шаблонам [56, 57]. Отметим, что некоторые стратегии системы C-lightVer, а именно стратегия для финитных итераций над изменяемыми массивами [58], стратегия для программ с финитными итерациями над массивами [58] и стратегия для программ, спецификации которых содержат функции со свойством конкатенации [58], близки к полученным в ходе нашего проекта стратегиям. Однако триггерами для срабатывания шаблонов в системе C-lightVer служат фрагменты программного кода, а не фрагменты дерева доказательства, как в нашем проекте.

Родственным нашему проекту мероприятием является серия российских соревнований по формальной верификации программ VeHa. В рамках данной серии на текущий момент состоялись три соревнования, VeHa-2023 [15], VeHa-2024 [9] и VeHa-2025 [80]. На данных соревнованиях команды исследователей также решают сложные для формальной верификации задачи. Но на нашем проекте, в отличие от соревнований серии VeHa, временные рамки намного более мягкие (проект продолжается уже семестр вместо трех дней). Другим родственным мероприятием является проект Большой Математической Мастерской 2025 года по формальной верификации хэш-функции «Стрибог» [8]. Однако, на нашем проекте, в отличие от проекта на Большой Математической Мастерской 2025 года, поставлены другие цели.

Структура статьи Данная статья имеет следующую структуру: в разделе 2 описаны основы дедуктивной верификации программ в системе Frama-C/WP, в разделе 3 описана проведенная нами дедуктивная верификация реализаций классического умножения матриц и умножения матриц с направленными на повышение эффективности использования кэш-памяти оптимизациями, в заключении приведен список наших результатов и описаны наши планы на будущее, в Приложении A приведен пример одного из доказанных условий корректности вместе с доказательством.

Благодарности Авторы статьи выражают благодарность компании YADRO и Лаборатории YADRO НГУ, а также лично Власову Александру Александровичу и Латкину Евгению Ивановичу за организацию и анализ проекта.

2. Основы дедуктивной верификации в системе Frama-C/WP

Процесс дедуктивной верификации программ в системе Frama-C/WP начинается с задания формальных спецификаций верифицируемой программы на языке ACSL, а именно предусловия (ограничение на входные данные), постусловия (связь входных и выходных данных) и инвариантов циклов (истинны на входе в цикл, истинны на итерациях цикла и обеспечивают выход из цикла). Спецификации задаются внутри C-комментариев, где можно формировать теорию предметной области. Для верификации функциональной эквивалентности полезно разместить в теории предметной области предикаты, описывающие на логическом уровне рекурсивные определения, с которыми будет соотноситься верифицируемая программа. Для задания таких предикатов удобно использовать механизм индуктивных предикатов и их свойств. Отметим, что формально индуктивные предикаты не задают вычислимое определение, однако в их свойствах можно и полезно фактически задать такое определение. Такие свойства могут зависеть от меток программы, которые позволяют с помощью конструкции `at` указывать, в каком состоянии программы брать значения тех или иных переменных. Также еще одной важной конструкцией для задания спецификаций является конструкция `separated`, которая позволяет указать на расположение в разных областях памяти. Кроме того, еще одной важной конструкцией для задания спецификаций является конструкция `assert`, накладывающая ограничение в определенной точке программы, чтобы дальше по ходу исполнения программы можно было полагаться на такое ограничение. Введение конструкций `assert` позволяет упростить дедуктивную верификацию программ за счет доказательства таких промежуточных утверждений [23, 45, 62].

Логической основой плагина WP для системы Frama-C является исчисление слабейшего предусловия, которое позволяет для программы и ее спецификаций генерировать условия корректности, истинность которых означает корректность программы относительно ее спецификаций. Так как верифицируются программы на языке C, то неизбежно встает вопрос о работе на данном этапе с указателями. Вместо широко применяемой для таких целей сепарационной логики, в системе Frama-C/WP используется модель памяти, поддерживающая отображения из адресов в значения. При работе с такой моделью памяти применяется конструкция `havoc`, который позволяет сравнивать, изменилась ли область памяти после операций над памятью. Поэтому, доказательства условий корректности се-

рьезно зависят от работы с предикатом `havoc`.

Доказывать условия корректности в системе Frama-C/WP можно и автоматически с помощью SMT-решателей и систем автоматического доказательства теорем, а также интерактивно с помощью системы Coq. Такие SMT-решатели и системы автоматического доказательства теорем, как Z3 [26, 38], CVC4 [24], CVC5 [22], Vampire [61], Alt-Ergo [34] и E Prover [81] позволяют доказывать многие относительно простые условия корректности в автоматическом режиме. Таким образом, пользователю для интерактивного доказательства в системе Coq в основном остаются самые сложные условия корректности.

Рассмотрим вопрос автоматизации дедуктивной верификации в системе Frama-C/WP. На разных стадиях дедуктивной верификации для такой автоматизации можно использовать следующие подходы:

1. Пытаться переиспользовать в разных проектах по верификации одни и те же индуктивные предикаты из теории предметной области.
2. Если постусловие задано как параметр индуктивного предиката, то можно использовать метод модификации постусловия, чтобы задавать инварианты цикла как индуктивные предикаты из постусловия, но с параметром от счетчика цикла.
3. Вводить промежуточные утверждения с помощью `assert`.
4. Применять SMT-решатели для автоматического доказательства условий корректности.
5. При доказательстве условий корректности в системе Coq использовать плагин QuickChick [75] для поиска возможных контрпримеров к условиям корректности.
6. При доказательстве условий корректности в системе Coq использовать плагин CoqHammer [37], который позволяет с помощью тактики `hammer` использовать SMT-решатели для доказательства условий корректности.
7. При доказательстве условий корректности в системе Coq использовать для генерации доказательств методы машинного обучения с помощью плагина Tactician [27].
8. При доказательстве условий корректности в системе Coq использовать для генерации доказательства Большие Языковые Модели (LLM).
9. При доказательстве условий корректности в системе Coq задавать и применять стратегии доказательства в виде шаблонов, описывающих ситуацию для применения стратегии и схему доказательства, которое нужно генерировать в данных ситуациях.

В нашем проекте мы активно использовали из данного списка подходы 1, 2, 3, 4, 6, 8 и

9.

3. Дедуктивная верификация реализаций классического умножения матриц и умножения матриц с направленными на повышение эффективности использования кэш-памяти оптимизациями

В данном разделе мы представляем как наши практические результаты в виде успешной верификации реализаций умножения матриц с оптимизациями, так и теоретические результаты в виде методов автоматизации доказательства условий корректности для таких матриц на основе введенных нами стратегий.

3.1. Задание спецификаций классического алгоритма умножения матриц

Схема классического алгоритма умножения матриц [2, 82] приведена приведено на рисунке 2.

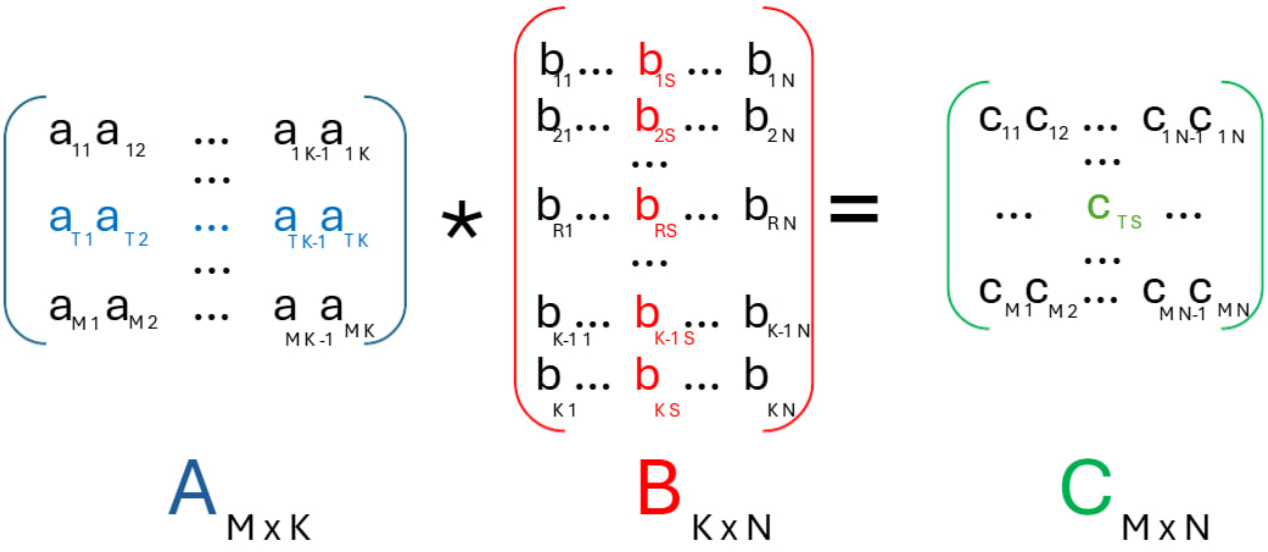

Рис. 2. Схема классического алгоритма умножения матриц

Данной схеме соответствует реализация в виде трех циклов, с которой начинается статья [5]:

```
void gemm_v0(int M, int N, int K, const float * A, const float * B, float *C)
{
    for (int i = 0; i < M; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            C[i*N + j] = 0.0;
            float sum = 0.0;

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

Для задания спецификаций такой реализации были заданы индуктивные предикаты. Индуктивный предикат `DotProduction` определяет, что его последний аргумент равен скалярному произведению строки на столбец:

```
    inductive DotProduction{L1}(float* A, float* B, integer to,
                                integer i, integer K, integer j, integer N,
                                real res)
 {
    case dotproduction_empty_range{L1}:
      \forall float* A, float* B, integer to, integer i, integer K, integer j,
      integer N;
      0 >= to ==> DotProduction{L1}(A, B, to, i, K, j, N, 0.0);

    case dotproduction_positive_range{L1}:
      \forall float* A, float* B, integer to, integer i,
      integer K, integer j, integer N, real res;
```

```
    (0 < to) &&
    DotProduction{L1}(A, B, to-1, i, K, j, N, res) ==>
    DotProduction{L1}(A, B, to, i, K, j, N,
    res + \at(A[i*K + (to-1)], L1) * \at(B[(to-1)*N + j], L1));
}
```

Данный предикат соответствует внутреннему циклу реализации.

Индуктивный предикат `RowResult` определяет, что все элементы строки результирующей матрицы заполнены скалярными произведениями строк на столбцы:

```
inductive RowResult{L1, L2}(float* A, float* B, float* C, integer to,
                            integer i, integer K, integer N)
{
  case rowresult_empty{L1, L2}:
    \forall float* A, float* B, float* C, integer to, integer i,
    integer K, integer N;
    0 >= to ==> RowResult{L1, L2}(A, B, C, to, i, K, N);

  case rowresult_step{L1, L2}:
    \forall float* A, float* B, float* C, integer to, integer i,
    integer K, integer N;
    0 < to &&
    RowResult{L1, L2}(A, B, C, to-1, i, K, N) &&
    DotProduction{L1}(A, B, K, i, K, (to-1), N, \at(C[i*N + (to-1)], L2))
    ==> RowResult{L1, L2}(A, B, C, to, i, K, N);
}
```

Таким образом предикат `RowResult` определен с помощью `DotProduction`. Полученный предикат соответствует среднему по вложенности циклу реализации.

Индуктивный предикат `MatrixResult` определяет, что все строки результирующей матрицы заполнены скалярными произведениями строк на столбцы:

```
inductive MatrixResult{L1, L2}(float* A, float* B, float* C, integer to,
                               integer K, integer N, integer M)
{
  case matrixresult_empty{L1, L2}:
```

```
        \forall float* A, float* B, float* C, integer to, integer K, integer N
        integer M;
        0 >= to ==> MatrixResult{L1, L2}(A, B, C, to, K, N, M);

      case matrixresult_step{L1, L2}:
        \forall float* A, float* B, float* C, integer to, integer K,
        integer N, integer M;
        0 < to &&
        MatrixResult{L1, L2}(A, B, C, to-1, K, N, M) &&
        RowResult{L1, L2}(A, B, C, N, to-1, K, N)
        ==> MatrixResult{L1, L2}(A, B, C, to, K, N, M);
    }
  }
```

Таким образом предикат `MatrixResult` определен с помощью `RowResult`. Полученный предикат соответствует внешнему циклу реализации.

Предусловие задано следующим образом:

```
requires \valid_read(A + (0 .. M*K-1));
requires \valid_read(B + (0 .. K*N-1));
requires \valid(C + (0 .. M*N-1));
requires \separated(A + (0 .. M*K-1), B + (0 .. K*N-1), C + (0 .. M*N-1));
requires M > 0 && N > 0 && K > 0;
```

Предикат `valid_read` из предусловия определяет, что из такой области памяти можно читать. Предикат `valid` из предусловия определяет, что можно и писать в такую область памяти и читать из такой области памяти.

Постусловие задано следующим образом:

```
ensures MatrixResult{Pre, Post}(A, B, C, M, K, N, M);
```

Фактически такое постусловие означает, что результаты реализации описываются применением `MatrixResult`.

Инвариант внутреннего цикла задан следующим образом:

```
  loop invariant 0 <= i < M;
              loop invariant 0 <= k <= K;
              loop invariant 0 <= j < N;
```

```
            loop invariant DotProduction{Here}(A, B, k, i, K, j, N, sum);


  loop invariant \separated(A + (0 .. M*K-1), B + (0 .. K*N-1),
            C + (0 .. M*N-1));
            loop assigns sum, k;
            loop variant K-k;
```

Такой инвариант описывает, что результаты внутреннего цикла после `k` итераций описываются применением `DotProduction` от параметра `k`.

Инвариант среднего по вложенности цикла задан следующим образом:

```
 loop invariant 0 <= i < M;
         loop invariant 0 <= j <= N;
         loop invariant RowResult{Pre, Here}(A, B, C, j, i, K, N);


         loop invariant \separated(A + (0 .. M*K-1), B + (0 .. K*N-1),
         C + (0 .. M*N-1));
         loop assigns C[i*N .. i*N+N-1], j;
         loop variant N-j;
```

Такой инвариант описывает, что результаты среднего по вложенности цикла после `j` итераций описываются применением `RowResult` от параметра `j`.

Инвариант внешнего цикла задан следующим образом:

```
      loop invariant 0 <= i <= M;
      loop invariant MatrixResult{Pre, Here}(A, B, C, i, K, N, M);


      loop invariant \separated(A + (0 .. M*K-1), B + (0 .. K*N-1),
      C + (0 .. M*N-1));
      loop assigns C[0 .. M*N-1], i;
      loop variant M - i;
```

Данный инвариант получен методом модификации постусловия в виде замены параметра `MatrixResult` из постусловия на параметр цикла `i`. Получилось, что такой инвариант описывает, что результаты внешнего цикла после `i` итераций описываются применением `MatrixResult` от параметра `i`.

Для упрощения верификации во внутреннем цикле была задана следующая конструк-

ция assert:

```
assert step: DotProduction{Here}(A, B, k+1, i, K, j, N,
                                 sum + A[i*K+k] * B[k*N+j]);
```

Введение такого промежуточного утверждения позволило упростить доказательство сохранения на итерациях инварианта внутреннего цикла.

Для упрощения верификации в среднем по вложенности цикле была задана следующая конструкция assert:

```
assert row_step: RowResult{Pre, Here}(A, B, C, j+1, i, K, N);
```

Введение такого промежуточного утверждения позволило упростить доказательство сохранения на итерациях инварианта среднего по вложенности цикла.

Для упрощения верификации во внешнем цикле была задана следующая конструкция assert:

```
assert mat_step: MatrixResult{Pre, Here}(A, B, C, i+1, K, N, M);
```

Введение такого промежуточного утверждения позволило упростить доказательство сохранения на итерациях инварианта внешнего цикла.

3.2. Задание спецификаций умножения матриц с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы

Реализация умножения матриц с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы была задана на основе статьи [5]:

```
void gemm_v1(int M, int N, int K, const float * A, const float * B, float * C)
{
    for (int i = 0; i < M; ++i)
    {
        float * c = C + i * N;

        for (int j = 0; j < N; ++j)
        {
```

```
            c[j] = 0;
        }

        for (int k = 0; k < K; ++k)
        {
            const float * b = B + k * N;
            float a = A[i*K + k];

            for (int t = 0; t < N; ++t)
            {
                c[t] += a * b[t];
            }
        }
    }
}
```

Отметим, что в такой реализации появился внутренний цикл для заполнения строки результирующей матрицы нулями.

Для задания спецификаций данной реализации была переиспользована теория предметной области реализации классического умножения матриц. Это позволило задать для рассматриваемой реализации то же самое предусловие, то же самое постусловие и тот же самый инвариант внешнего цикла, а также использовать предикат `DotProduction` для задания инвариантов внутренних циклов. Таким образом, фактически с помощью дедуктивной верификации производится проверка эквивалентности заданной в теории предметной области классического умножения матриц и рассматриваемой реализации.

Для задания инварианта внутреннего цикла теория предметной области была расширена индуктивным предикатом `zeroed`:

```
inductive zeroed{L}(float* a, integer b, integer e){
case zeroed_empty{L}:
    \forall float* a, integer b, e; b >= e ==> zeroed{L}(a, b, e);
case zeroed_range{L}:
    \forall float* a, integer b, e; b < e ==>
    zeroed{L}(a, b, e-1) && a[e-1] == 0 ==> zeroed{L}(a,b,e);
```

Данный предикат описывает заполненность области массива нулями.

Рассмотрим новые относительно реализации классического умножения матриц спецификации. Инвариант цикла для заполнения строки результирующей матрицы нулями задан следующим образом:

```
loop invariant 0 <= j <= N;
loop invariant 0 <= i < M;
loop invariant \separated(A + (0 .. M*K-1), B + (0 .. K*N-1),
C + (0 .. M*N-1));
loop invariant zeroed{Here}(C, i*N, i*N + j);

loop assigns j, C[i*N .. i*N + N - 1];
loop variant N-j;
```

Такой инвариант описывает, что результаты цикла для заполнения строки результирующей матрицы нулями после `j` итераций описываются применением `zeroed` от параметра `j`.

Инвариант среднего по вложенности цикла с параметром `k` задан следующим образом:

```
loop invariant 0 <= k <= K;
loop invariant 0 <= i < M;
loop invariant \separated(A + (0 .. M*K-1), B + (0 .. K*N-1),
C + (0 .. M*N-1));
loop invariant \forall integer l; 0<=l<N ==>
DotProduction(A, B, k, i, K, l, N, c[l]);
loop assigns k, C[i*N .. i*N + N - 1];
loop variant K-k;
```

Отметим, что результаты среднего по вложенности цикла с параметром `k` можно описать с помощью `DotProduction`.

Инвариант внутреннего цикла с параметром `t` задан следующим образом:

```
loop invariant 0 <= t <= N;
loop invariant 0 <= k < K;
loop invariant 0 <= i < M;
loop invariant \separated(A + (0 .. M*K-1), B + (0 .. K*N-1),
C + (0 .. M*N-1));
```

```
loop invariant \forall integer q; 0<=q<t ==>
DotProduction(A, B, k+1, i, K, q, N, c[q]);
loop invariant \forall integer q; t <= q < N ==>
DotProduction(A, B, k, i, K, q, N, c[q]);
loop assigns t, C[i*N .. i*N + N - 1];
loop variant N-t;
```

Отметим, что результаты внутреннего цикла с параметром `t` можно описать с помощью `DotProduction`.

Для упрощения верификации во внутреннем цикле с параметром `t` была задана следующая конструкция `assert`:

```
assert step: DotProduction(A, B, k+1, i, K, t, N, c[t]+a*b[t]);
```

Введение такого промежуточного утверждения позволило упростить доказательство сохранения на итерациях инварианта внутреннего цикла.

Для упрощения верификации в среднем по вложенности цикле с параметром `k` была задана следующая конструкция `assert`:

```
assert dot_done:  \forall integer l; 0<=l<N ==>
            DotProduction(A, B, k+1, i, K, l, N, c[l]);
```

Введение такого промежуточного утверждения позволило упростить доказательство сохранения на итерациях инварианта среднего по вложенности цикла с параметром `k`.

Для упрощения верификации в цикле для заполнения строки результирующей матрицы нулями была задана следующая конструкция `assert`:

```
assert zeroed{Here}(C, i*N, i*N + j + 1);
```

Введение такого промежуточного утверждения позволило упростить доказательство сохранения на итерациях инварианта цикла для заполнения строки результирующей матрицы нулями.

Для упрощения верификации между циклом для заполнения строки результирующей матрицы нулями и средним по вложенности циклом с параметром `k` была задана следующая конструкция `assert`:

```
assert zeroed{Here}(C, i*N, i*N + N);
```

Введение такого промежуточного утверждения позволило упростить доказательство условия выполнения инварианта на входе в средний по вложенности цикл с параметром `k`.

Для упрощения верификации после среднего по вложенности цикла с параметром `k` была задана следующая конструкция `assert`:

```
assert row_done: RowResult{Here}(A, B, C, N, i, K, N);
```

Введение такого промежуточного утверждения позволило доказать, что результат среднего по вложенности цикла с параметром `k` описывается применением `RowResult`.

Для упрощения верификации во внешнем цикле была задана следующая конструкция `assert`:

```
assert mat_step: MatrixResult{Here}(A, B, C, i+1, K, N, M);
```

Введение такого промежуточного утверждения позволило упростить доказательство сохранения на итерациях инварианта внешнего цикла.

3.3. Доказательство условий корретности

И в случае верификации реализации обеих реализаций абсолютное большинство условий корректности были доказаны SMT-решателями. При этом особый интерес представляют случаи, когда условия корректности пришлось доказывать в интерактивной системе доказательства Coq. Для автоматизации такого доказательства были активно использованы возможности SMT-решателей для доказательства подцелей с помощью плагина CoqHammer, а также генерация доказательств с помощью большой языковой модели Gemini [53]. Отметим, что нами для автоматизации доказательства были выделены и применены стратегии доказательства в виде шаблонов, описывающих ситуацию для применения стратегии и схему доказательства, которое нужно генерировать в данных ситуациях. Рассмотрим такие стратегии подробнее. Пример условия корректности и его доказательства, где применяются все введенные нами стратегии доказательства, можно посмотреть в приложении А.

3.3.1. Стратегия доказательства того, что разделены области памяти, в которые осуществляется запись и из которых осуществляется чтение

Данная стратегия применяется в таких случаях, когда нужно доказать, что разделены области памяти в которые осуществляется запись и из которых осуществляется чтение. Отметим, что такие случаи встречаются в условиях корректности обеих реализаций, что указывает на востребованность введения такой стратегии. Особенно отметим, что данная стратегия применяется внутри определенной далее стратегии доказательства того, что

при записи в одну область памяти значения в другой области памяти не изменяются.

Так как разделение областей памяти в системе Frama-C/WP определяется с помощью предиката `separated`, то приведем определение такого предиката:

```
Definition separated (p:addr) (a:Numbers.BinNums.Z) (q:addr)
    (b:Numbers.BinNums.Z) : Prop :=
  (a <= 0%Z)%Z \/
  (b <= 0%Z)%Z \/
  ~ ((base p) = (base q)) \/
  (((offset q) + b)%Z <= (offset p))%Z \/
  (((offset p) + a)%Z <= (offset q))%Z.
```

Данная стратегия применяется тогда, когда есть гипотеза `Hxx` в которой говорится о том, что память разделена.

Отметим, что данная стратегия разбивается на два случая, отличающихся тем, когда именно возникает необходимость в применении данной стратегии.

Случай записи в одну область памяти и чтения из другой области памяти В данном случае стратегия зависит от следующих метапеременных:

- `R` – массив из которого читаем;
- `W` – массив в который пишем;
- `base_R`, `base_W` – адреса массивов;
- `offset_R`, `offset_W` – смещения внутри массивов;
- `size_R`, `size_W` – размеры массивов;
- `idx` – индекс ячейки которую читаем из R;
- `idx_write` – индекс ячейки в которую пишем в W.

Рассмотрим схему доказательства, порождаемую стратегией в данном случае:

```
(* Перед началом делаем:
   destruct R, W.
   unfold base, offset, shift, separated in *.
   injection Heq as HeqBase HeqOff.
*)
H_SEP: separated R size_R W size_W
```

```
destruct H_SEP as [Hb1 | [Hb2 | [Hb3 | [Hb4 | Hb5]]]].

(* ================================================================ *)
(* ВЕТКА 1: Размер массива ЧТЕНИЯ <= 0 *)
(* ================================================================ *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   H_dim1 : (0 < M)%Z  (Высота матрицы R > 0)
   H_dim2 : (0 < N)%Z  (Ширина матрицы R > 0)
   H_size1 : size_R = (M * N)%Z
   Hb1    : (size_R <= 0)%Z
*)
- nia.

(* ================================================================ *)
(* ВЕТКА 2: Размер массива ЗАПИСИ <= 0 *)
(* ================================================================ *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   H_dim1 : (0 < M2)%Z  (Высота матрицы W > 0)
   H_dim2 : (0 < N2)%Z  (Ширина матрицы W > 0)
   H_size1 : size_W = (M2 * N2)%Z
   Hb2    : (size_W <= 0)%Z
*)
- nia.

(* ================================================================ *)
(* ВЕТКА 3: адреса разные *)
(* ================================================================ *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   HeqBase : base_R = base_W
```

```
   Hb3     : base_R <> base_W
*)
- right. right. left. exact Hb3.
  (* ИЛИ просто ‘congruence.‘ *)

(* ================================================================ *)
(* ВЕТКА 4: Массив ЗАПИСИ (W) лежит целиком ДО массива ЧТЕНИЯ (R) *)
(* (Запись <= Конец_W <= Начало_R <= Чтение) *)
(* ================================================================ *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   Hb4          : (offset_W + size_W <= offset_R)%Z
   HeqOff       : (offset_W + idx_write = offset_R + idx)%Z
   H_read_min   : (0 <= idx)%Z
   H_write_outer: (i < M)%Z (Текущая строка записи < Высоты матрицы W)
   H_write_inner: (j < N)%Z (Текущий столбец записи < Ширины матрицы W)
   H_idx_w  : (idx_write < size_W)%Z
*)
- right. right. right. left.
  (* Подсказка 1: Адрес чтения не уходит левее начала массива R *)
  assert (HintR: (offset_R <= offset_R + idx)%Z) by nia.

  (*offset_W + size_w <= offset_W + idx_write)

  (* Подсказка 2: Адрес записи не превышает конец массива W. *)
  assert (HintW:
  (offset_W + idx_write + 1 <= offset_W + size_W)%Z) by nia.
  (* Противоречие *)
  nia.

(* ================================================================ *)
(* ВЕТКА 5: Массив ЧТЕНИЯ (R) лежит целиком ДО массива ЗАПИСИ (W) *)
```

```
(* (Чтение < Конец_R <= Начало_W <= Запись) *)
(* ============================================================== *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   Hb5          : (offset_R + size_R <= offset_W)%Z
   HeqOff       : (offset_R + idx = offset_W + idx_write)%Z
   H_read_max   : (idx < size_R)%Z
   H_write_outer_min : (0 <= i)%Z (Текущая строка записи >= 0)
   H_write_inner_min : (0 <= j)%Z (Текущий столбец записи >= 0)
   H_idx_wr : (idx_write >= 0)%Z
*)
- right. right. right. right.
  (* Подсказка 1: Адрес чтения строго меньше конца массива R *)
  assert (HintR: (offset_R + idx + 1 <= offset_R + size_R)%Z) by nia.

  (*offset_W + idx_write + 1<= offset_R + size_R <= offset W*)

  (* Подсказка 2: Адрес записи не уходит левее начала массива W. *)
  assert (HintW: (offset_W <= offset_W + idx_write)%Z) by nia.
  (* Противоречие *)
  nia.
```

Отметим, что такая схема доказательства разбивается на 5 ветвей, так как определение `separated` разбивается на 5 логических "или".

Случай применения `havoc` к блоку памяти В данном случае стратегия зависит от следующих метапеременных:

- `R` – массив из которого мы читаем;
- `W` – массив, в который мы писали и где применен `havoc`;
- `base_R`, `base_W` – адреса массивов;
- `off_R`, `off_W` – смещения внутри массивов;
- `size_R`, `size_W` – размеры массивов;
- `idx` – индекс ячейки, которую мы читаем из `R`;

- `idx_write_start` – индекс начала блока, который мы меняем в `W`.
- `size_W_havoc` – размер блока в `W`.

Рассмотрим схему доказательства, порождаемую стратегий в данном случае:

```
(* Перед началом делаем:
   destruct R as [base_R off_R].
   destruct W as [base_W off_W].
   unfold separated, shift, base, offset,
   <ПЕРЕМЕННЫЕ_АДРЕСОВ_ТИПА_x1_x2_a4_a5> in *.
   simpl in *.
*)

H_SEP: separated R size_R W size_W

destruct H_SEP as [Hb1 | [Hb2 | [Hb3 | [Hb4 | Hb5]]]].

(* ================================================================ *)
(* ВЕТКА 1: Размер массива ЧТЕНИЯ <= 0 *)
(* ================================================================ *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   H_dim1 : (0 < M)%Z  (Высота матрицы R > 0)
   H_dim2 : (0 < N)%Z  (Ширина матрицы R > 0)
   H_size1 : size_R = (M * N)%Z
   Hb1    : (size_R <= 0)%Z
*)
- nia.

(* ================================================================ *)
(* ВЕТКА 2: Размер массива ЗАПИСИ <= 0 *)
(* ================================================================ *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
```

```
   H_dim1 : (0 < M2)%Z  (Высота матрицы W > 0)
   H_dim2 : (0 < N2)%Z  (Ширина матрицы W > 0)
   H_size1 : size_W = (M2 * N2)%Z
   Hb2    : (size_W <= 0)%Z
*)
- nia.

(* ================================================================ *)
(* ВЕТКА 3: адреса разные *)
(* ================================================================ *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   Hb3    : base_R <> base_W
   Hxx    : base_R = base_W
*)
- (* Выбираем 3-е условие в нашей текущей цели separated *)
  right. right. left.
  exact Hb3.

(* ================================================================ *)
(* ВЕТКА 4: Массив записи (W) лежит целиком до массива чтения (R) *)
(* (Блок записи <= Конец_W <= Начало_R <= Индекс чтения) *)
(* ================================================================ *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   Hb4          : (off_W + size_W <= off_R)%Z
   Hidx_min     : (0 <= idx)%Z (Читаемый индекс >= 0)
   H_write_dims : Гипотезы о границах циклов,
                  чтобы доказать, что блок havoc не вылез за массив W.
*)
- (* Выбираем 4-е условие в нашей текущей цели separated *)
  right. right. right. left.
```

```
  (* Подсказка 1: Изменяемый блок не выходит за конец всего массива W *)
  assert (HintW: (idx_write_start + size_W_havoc <= size_W)%Z) by nia.

  (* Подсказка 2 (не всегда обязательна, но надежна): Адрес чтения
  не уходит в минус *)
  assert (HintR: (off_R <= off_R + idx)%Z) by nia.

  (* nia выстраивает цепочку: Конец_Блока_W <=
  Конец_W (HintW) <= Начало_R (Hb4) <= Индекс_Чтения_R (HintR) *)
  nia.

(* ================================================================= *)
(* ВЕТКА 5: Массив чтения (R) лежит целиком до массива записи (W) *)
(* (Индекс чтения < Конец_R <= Начало_W <= Блок записи) *)
(* ================================================================= *)
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   Hb5          : (off_R + size_R <= off_W)%Z
   Hidx_max     : (idx < size_R)%Z (Читаемый индекс строго внутри
   массива R)
   H_write_dims : Гипотезы о том, что блок записи не уходит в минус
   (idx_write_start >= 0)
*)
- (* Выбираем 5-е условие в нашей текущей цели separated *)
  right. right. right. right.

  (* Подсказка 1: Читаемая ячейка (idx) строго меньше конца всего
  массива R *)
  (* (Мы добавляем +1, так как размер ячейки равен 1) *)
  assert (HintR:
  (off_R + idx + 1 <= off_R + size_R)%Z) by nia.
```

```
(* Подсказка 2: Блок записи не уходит левее начала массива W *)
(* (Обычно это очевидно, так как индексы циклов i, j >= 0) *)
assert (HintW:
(off_W <= off_W + idx_write_start)%Z) by nia.

(* nia выстраивает цепочку: Конец_Ячейки_R (HintR) <= Конец_R
<= Начало_W (Hb5) <= Начало_Блока_W (HintW) *)
nia.
```

Отметим, что такая схема доказательства тоже разбивается на 5 ветвей, так как определение `separated` разбивается на 5 логических "или".

3.3.2. Стратегия доказательства того, что при записи в одну область памяти значения в другой области памяти не изменяются

Данная стратегия применяется в таких случаях, когда нужно доказать, что для разделенных с помощью `separated` областей памяти при записи в одну область памяти значения в другой области памяти не изменяются. Необходимость в данной стратегии возникает из-за следующей особенности Frama-C/WP: у Frama-C/WP единая память и после того, как происходит запись в массив `C`, гипотезы о том, что матрицы `A` и `B` не меняются, верны для старой памяти, но у нас уже появляется новая память, и в ней приходится это доказывать. Данная стратегия нужна чтобы доказать что матрицы `A` и `B` в новой и старой памяти совпадают. Отметим, что такие случаи встречаются в условиях корректности обеих реализаций, что указывает на востребованность введения такой стратегии.

Так как результат изменения памяти в системе Frama-C/WP определяется с помощью конструкции `havoc`, то приведем описывающую данную конструкцию аксиому:

```
Axiom havoc_access :
  forall {a:Type} {a_WT:WhyType a},
  forall (m0:addr -> a) (m1:addr -> a), forall (q:addr) (p:addr),
  forall (a1:Numbers.BinNums.Z),
  (separated q 1%Z p a1 -> ((havoc m0 m1 p a1 q) = (m1 q))) /\
  (~ separated q 1%Z p a1 -> ((havoc m0 m1 p a1 q) = (m0 q))).
```

Отметим, что данная стратегия разбивается на два случая, отличающихся тем, когда

именно возникает необходимость в применении данной стратегии.

Случай доказательства неизменности значений при применении `havoc` В данном случае стратегия применяется, когда гипотеза в доказательстве удовлетворяет следующему шаблону:

```
(*
   [ОБЩИЕ НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   <ГИПОТЕЗА_SEP> : separated ... (Глобальное разделение
   МАССИВА_ЧТЕНИЯ и МАССИВА_ЗАПИСИ)
*)
```

а цель доказательства удовлетворяет следующему шаблону:

```
assert (Eq_Memory: forall idx, (0 <= idx)%Z -> (idx <
<РАЗМЕР_МАССИВА_ЧТЕНИЯ>)%Z ->
    <НОВАЯ_ПАМЯТЬ> (shift <МАССИВ_ЧТЕНИЯ> idx) = <СТАРАЯ_ПАМЯТЬ>
    (shift <МАССИВ_ЧТЕНИЯ> idx)).
```

Рассмотрим схему доказательства, порождаемую стратегией для такой цели:

```
assert (Eq_Memory: forall idx, (0 <= idx)%Z -> (idx <
<РАЗМЕР_МАССИВА_ЧТЕНИЯ>)%Z ->
    <НОВАЯ_ПАМЯТЬ> (shift <МАССИВ_ЧТЕНИЯ> idx) = <СТАРАЯ_ПАМЯТЬ>
    (shift <МАССИВ_ЧТЕНИЯ> idx)).
{
  intros idx Hidx_min Hidx_max.

  (* Раскрываем переменные памяти(a6, a8, x2, x3 и т.д.) *)
  unfold <НОВАЯ_ПАМЯТЬ>, <shift_X1_X2>.

  (* Используем аксиому доступа для нашего конкретного читаемого адреса.
     АРГУМЕНТЫ ПО ПОРЯДКУ:
     1. tX (отображение addr->real)
     2. СТАРАЯ_ПАМЯТЬ (например Mf32, t1)
     3. АДРЕС_НАЧАЛА_ЗАПИСИ (shift <МАССИВ_ЗАПИСИ> <СМЕЩЕНИЕ_ЗАПИСИ>)
```

```
     4. РАЗМЕР_МАССИВА
     5. ЧИТАЕМЫЙ_АДРЕС (shift <МАССИВ_ЧТЕНИЯ> idx) *)

  destruct (havoc_access <tX> <СТАРАЯ_ПАМЯТЬ>
                         (shift <МАССИВ_ЗАПИСИ> <СМЕЩЕНИЕ_ЗАПИСИ>)
                         <РАЗМЕР_МАССИВА>
                         (shift <МАССИВ_ЧТЕНИЯ> idx)) as [Hacc _].

  (* Применяем левую часть аксиомы (условие, что адреса разделены) *)
  rewrite Hacc; [reflexivity |].

  (* Разбиваем абстракции Coq до чистых чисел *)
  (* Адрес в массиве - это пара начало массива и сдвиг *)
  destruct <МАССИВ_ЧТЕНИЯ> as [base_R off_R].
  destruct <МАССИВ_ЗАПИСИ> as [base_W off_W].

  (* Раскрываем определения для получения неравенств над
  числами, удобных для тактики nia. *)
  unfold separated, shift, base, offset,
  <ПЕРЕМЕННЫЕ_АДРЕСОВ_a3_a4_a5> in *.
  simpl in *. (* Избавляемся от конструкций match *)

  (* Применяем стратегию доказательства того, что разделены
 области памяти, в которые осуществляется запись и из которых
 осуществляется чтение *)
  destruct <ГИПОТЕЗА_SEP> as [Hb1 | [Hb2 | [Hb3 | [Hb4 | Hb5]]]].
  ...
}
```

Отметим, что рассматриваемая стратегия применяет рассмотренную ранее стратегию доказательства того, что разделены области памяти, в которые осуществляется запись и из которых осуществляется чтение.

Случай доказательства неизменности значений при применении `MAP.SET` В данном случае стратегия применяется, когда гипотеза в доказательстве удовлетворяет следующему шаблону:

```
(*
   [ОБЩИЕ НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   <ГИПОТЕЗА_SEP> : separated ... (Глобальное разделение
   МАССИВА_ЧТЕНИЯ и МАССИВА_ЗАПИСИ)
*)
```

а цель доказательства удовлетворяет следующему шаблону:

```
assert (Eq_Memory: forall idx, (0 <= idx)%Z -> (idx <
<РАЗМЕР_МАССИВА_ЧТЕНИЯ>)%Z ->
    <НОВАЯ_ПАМЯТЬ> (shift <МАССИВ_ЧТЕНИЯ> idx) = <СТАРАЯ_ПАМЯТЬ>
    (shift <МАССИВ_ЧТЕНИЯ> idx)).
```

Рассмотрим схему доказательства, порождаемую стратегией для такой цели:

```
assert (Eq_Memory: forall idx, (0 <= idx)%Z -> (idx <
<РАЗМЕР_МАССИВА_ЧТЕНИЯ>)%Z ->
    <НОВАЯ_ПАМЯТЬ> (shift <МАССИВ_ЧТЕНИЯ> idx) = <СТАРАЯ_ПАМЯТЬ>
    (shift <МАССИВ_ЧТЕНИЯ> idx)).
{
  intros idx Hidx_min Hidx_max.

  (* Раскрываем новую память и адрес записи *)
  unfold <НОВАЯ_ПАМЯТЬ>, <МАКРОС_АДРЕСА_ЗАПИСИ_A8>, Map.set.

  (* Разбиваем цель на 2 ветки: адреса совпали (Heq) или
  разные (Hneq) *)
  destruct (why_decidable_eq (shift <МАССИВ_ЗАПИСИ> <ИНДЕКС_ЗАПИСИ>)
  (shift <МАССИВ_ЧТЕНИЯ> idx)) as [Heq | Hneq].

  - (* АДРЕСА СОВПАЛИ. Доказываем, что это невозможно *)
    (* Адрес в массиве - это пара начало массива и сдвиг *)
```

```
    destruct <МАССИВ_ЧТЕНИЯ> as [base_R off_R].
    destruct <МАССИВ_ЗАПИСИ> as [base_W off_W].

    (* Раскрываем определения для получения неравенств над числами
       чтобы потом применить injection*)
    unfold separated, shift, base, offset,
    <ПЕРЕМЕННЫЕ_АДРЕСОВ_a3_a4_a5> in *.
    simpl in *.

    (* Извлекаем уравнение баз и смещений из факта совпадения адресов *)
    injection Heq as HeqBase HeqOff.

    (* Применяем стратегию доказательства того, что разделены
    области памяти, в которые
    осуществляется запись и из которых осуществляется чтение *)
    destruct <ГИПОТЕЗА_SEP> as [Hb1 | [Hb2 | [Hb3 | [Hb4 | Hb5]]]].
    ...
    - (* АДРЕСА РАЗНЫЕ *)
      reflexivity.
}
```

Отметим, что и в данном случае рассматриваемая стратегия применяет рассмотренную ранее стратегию доказательства того, что разделены области памяти, в которые осуществляется запись и из которых осуществляется чтение.

3.3.3. Стратегия доказательства того, что при записи в одну область памяти значения индуктивного предиката на другой области памяти остаются прежними

Данная стратегия применяется в таких случаях, когда нужно доказать, что при записи в одну область памяти значения индуктивного предиката на другой области памяти остаются прежними. Отметим, что такие случаи встречаются в условиях корректности обеих реализаций, что указывает на востребованность введения такой стратегии.

Так как данная стратегии зависит от индуктивного предиката, то приведем применяе-

мое в данной стратегии шаблонное определение индуктивного предиката:

```
(* Шаблон определения индуктивного приката *)
Inductive <ИМЯ_ПРЕДИКАТА>:
  (addr -> <ТИП_ЗНАЧЕНИЯ>) -> (* 1. Состояние памяти (Mem) *)
  addr -> ... ->              (* 2. Адреса читаемых
  массивов (Ptr1, Ptr2...) *)
  Numbers.BinNums.Z ->        (* 3. Параметры,ограничивающие
  обрабатываемую часть массива (from / to) *)
  Numbers.BinNums.Z -> ... -> (* 4. Константные параметры
  (Размеры, индексы строк/столбцов) *)
  <ТИП_ЗНАЧЕНИЯ> -> Prop :=   (* 5. НАКОПЛЕННЫЙ РЕЗУЛЬТАТ (res) *)

  | <КОНСТРУКТОР_БАЗОВОГО_СЛУЧАЯ> :
      forall (Mem: addr -> <ТИП_ЗНАЧЕНИЯ>)
             (Ptr1...: addr)
             (to: Numbers.BinNums.Z)
             (Ctx1...: Numbers.BinNums.Z),

      (* Условие: Длина <= 0 *)
      (to <= 0%Z)%Z ->

      <ИМЯ_ПРЕДИКАТА> Mem Ptr1... to Ctx1...
      <Начальное_значение_результата(res)>

  | <КОНСТРУКТОР_ШАГА_ИНДУКЦИИ> :
      forall (Mem: addr -> <ТИП_ЗНАЧЕНИЯ>)
             (Ptr1...: addr)
             (to: Numbers.BinNums.Z)
             (Ctx1...: Numbers.BinNums.Z)
             (OldRes: <ТИП_ЗНАЧЕНИЯ>),

      (* Создание переменной для предыдущего шага *)
```

```
      let prev_to := ((-1%Z)%Z + to)%Z in

      (* Условие продолжения и рекурсивный вызов для
      старого результата *)
      (0%Z < to)%Z ->
      <ИМЯ_ПРЕДИКАТА> Mem Ptr1... prev_to Ctx1... OldRes ->
      (<ФУНКЦИЯ_ОБРАБОТКИ_prev_to> Mem shift (Ptr1 ...) ...
      prev_to Ctx1) ->
      (* Заключение: Предикат для текущей длины равен комбинации
         старого результата и чтения из памяти *)
      <ИМЯ_ПРЕДИКАТА> Mem Ptr1... to Ctx1...
```

Отметим, что данному шаблону соответствуют заданные нами в теории предметной области индуктивные предикаты `DotProduction`, `RowResult` и `MatrixResult`, где также выделены отдельные свойства для базы индукции и шага индукции.

Стратегия применяется, когда гипотеза в доказательстве удовлетворяет следующему шаблону:

```
(*
   [НЕОБХОДИМЫЕ ГИПОТЕЗЫ В КОНТЕКСТЕ]:
   EqA : forall idx, ... -> <НОВАЯ_ПАМЯТЬ> (...) = <СТАРАЯ_ПАМЯТЬ> (...)
   EqB : forall idx, ... -> <НОВАЯ_ПАМЯТЬ> (...) = <СТАРАЯ_ПАМЯТЬ> (...)
*)
```

а цель доказательства удовлетворяет следующему шаблону:

```
assert (Hframe_pred: forall <ПАРАМЕТРЫ_ИНДУКЦИИ> <РЕЗУЛЬТАТ>,
    <ГРАНИЦЫ_ДЛЯ_ПАРАМЕТРОВ> ->
    <ПРЕДИКАТ> <СТАРАЯ_ПАМЯТЬ> <МАТРИЦЫ> <ИНДЕКСЫ> <РЕЗУЛЬТАТ> ->
    <ПРЕДИКАТ> <НОВАЯ_ПАМЯТЬ> <МАТРИЦЫ> <ИНДЕКСЫ> <РЕЗУЛЬТАТ>).
```

Рассмотрим схему доказательства, порождаемую стратегией для такой цели:

```
assert (Hframe_pred: forall <ПАРАМЕТРЫ_ИНДУКЦИИ> <РЕЗУЛЬТАТ>,
    <ГРАНИЦЫ_ДЛЯ_ПАРАМЕТРОВ> ->
    <ПРЕДИКАТ> <СТАРАЯ_ПАМЯТЬ> <МАТРИЦЫ> <ИНДЕКСЫ> <РЕЗУЛЬТАТ> ->
    <ПРЕДИКАТ> <НОВАЯ_ПАМЯТЬ> <МАТРИЦЫ> <ИНДЕКСЫ> <РЕЗУЛЬТАТ>).
{
```

```
(* ШАГ 1: Вводим переменные и гипотезу предиката *)
intros <ПАРАМЕТРЫ_ИНДУКЦИИ> <РЕЗУЛЬТАТ> H_bounds H_pred_old.

(* ШАГ 2: СОХРАНЕНИЕ ПРЕЖНИХ ПЕРЕМЕННЫХ *)
(* Если в <ИНДЕКСЫ> есть сложные математические формулы
(например, -1 + to), при индукции в Coq они могут быть потеряны.
Запишем их в не участвующие в индукции переменные: *)
remember <СЛОЖНАЯ_ФОРМУЛА_1> as idx_1 eqn:Heq_idx1.
remember <СЛОЖНАЯ_ФОРМУЛА_2> as idx_2 eqn:Heq_idx2.
(* Также полезно запомнить старую память, чтобы не возникла
путаница с новой *)
remember <СТАРАЯ_ПАМЯТЬ> as mem_old.

(* ШАГ 3: Запускаем индукцию по предикату в случае старой памяти*)
induction H_pred_old.

- (* ВЕТКА 1: Базовый случай (например, длина = 0) *)
  (* Возвращаем запомненные переменные из 'remember' обратно,
  если нужно *)
  subst.
  apply <КОНСТРУКТОР_EMPTY_RANGE>.
  lia.

- (* ВЕТКА 2: Шаг индукции (добавление нового элемента) *)
  (* Возвращаем запомненные переменные из 'remember' обратно,
  если нужно *)
  subst.

  (* ШАГ 4: Переписываем память в текущей цели (с новой на старую).
     Используем наши леммы EqA/EqB, доказанные ранее.
     Квадратные скобки автоматически доказывают индексы для EqA/EqB. *)
  rewrite <- EqA; [ | lia | nia ].
```

```
    rewrite <- EqB; [ | lia | nia ].
    (* rewrite EqC. -- (если есть равенство для матрицы C) *)

    (* Теперь память в формулах суммы/результата совпадает
    со СТАРОЙ памятью *)

    (* ШАГ 5: Применяем конструктор индуктивного шага *)
    apply <КОНСТРУКТОР_POSITIVE_RANGE>.

    + (* Подцель 5.1: Доказать, что длина > 0 *)
      lia.

    + (* Подцель 5.2: Доказать индукционный переход *)
      (* Применяем гипотезу индукции, которую сгенерировал Coq
      (IHH_pred_old) и доказываем её *)
      apply IHH_pred_old; try assumption; try nia; try lia; try reflexivity.
}
```

Отметим, что необходимость в данной стратегии возникает, например, при таких распространенных при работе с массивами случаях, когда нужно доказать, что при записи в ячейку массива значение индуктивного предиката на предыдущих ячейках массива не изменяется.

4. Заключение

В данной статье получены следующие результаты:

1. Заданы спецификации для реализаций классического умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы.
2. Задана теория предметной области для дедуктивной верификации реализации классического умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы.

3. Разработаны стратегии доказательства условий корректности при дедуктивной верификации реализации классического умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы. Разработаны следующие стратегии:
   (a) Стратегия доказательства того, что разделены области памяти, в которые осуществляется запись и из которых осуществляется чтение.
   (b) Стратегия доказательства того, что при записи в одну область памяти значения в другой области памяти не изменяются.
   (c) Стратегия доказательства того, что при записи в одну область памяти значения индуктивного предиката на другой области памяти остаются прежними.
4. Проведены эксперименты по дедуктивной верификации реализации классического умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы..

Отметим, что данные результаты могут служить прототипом комплексного подхода к дедуктивной верификации реализаций умножения матриц над математическими вещественными числами с направленными на повышение эффективности использования кэш-памяти оптимизациями.

Мы планируем дальше следовать планам нашего проекта по дедуктивной верификации все более и более оптимизированных реализаций умножения матриц из статьи [5], чтобы достить цели в виде дедуктивной верификации реализации minigemm.

Список литературы

1. Агибалов М.С. Верификация алгоритмов умножения матриц. URL: `https://github.com/ylab-nsu/wb26-trust-minigemm/tree/GemmV1-d` (дата обращения: 20.05.2026)
2. Алексеев Е.Р., Демин П.А., Болтачева Н.Ю. Новые технологии разработки высокоэффективных и параллельных приложений на современном Фортране // Прикладная информатика. 2018. Т. 13, № 1. С. 103–120.
3. Васенин В.А., Кривчиков М.А. Формальные модели программ и языков программирования. Часть 1. Библиографический обзор 1930–1989 гг. // Программная инженерия. 2015. № 5. С. 10–19.
4. Васенин В.А., Кривчиков М.А. Формальные модели программ и языков программирования.

Часть 2. Современное состояние исследований // Программная инженерия. 2015. № 6. С. 24–33.

5. Ермолаев И. Умножение матриц: эффективная реализация шаг за шагом. 2019. [Электронный ресурс]. URL: https://habr.com/ru/articles/359272/ (дата обращения: 20.05.2026)
6. Камкин А.С. Введение в формальные методы верификации программ. М.: ДМК Пресс, 2024. – 304 с.
7. Кокорин А.О., Тиевский С.Д., Девянин П.Н. Приемы дедуктивной верификации программного кода с использованием AstraVer Toolset // Прикладная дискретная математика. Приложение. 2022. № 15. С. 80–90.
8. Кондратьев Д.А., Бояндин Л.К., Гончар Г.Е., Марченко В.В., Обухова А.А., Разбитнова Ю.Ю., Хованская А.С., Янбулатов Д.Р. Формальная верификация реализации хэш-функции «Стрибог» с «Группой Астра» // Системная информатика. 2025. № 28. С. 25–52.
9. Кондратьев Д.А., Старолетов С.М., Шошмина И.В., Красненкова А.В., Зиборов К.В., Шилов Н.В., Гаранина Н.О., Черганов Т.Ю. Соревнования по формальной верификации VeHa-2024: накопленный в течение двух лет опыт и перспективы // Труды Института системного программирования РАН. 2025. Т. 37. № 1. С. 159–184.
10. Лейно К.Р.М. Доказательство корректности программ. М.: ДМК Пресс, 2024. – 522 с.
11. Лылова С.С., Власов А.А., Латкин Е.И., Знаменский И.И., Волокитин В.Д. Исследование производительности умножения плотных матриц в библиотеке OpenBLAS на архитектуре RISC - V с использованием векторных инструкций // Вычислительные технологии. 2026. Т. 31. № 2. С. 104–119.
12. Маркова В.П., Киреев С.Е., Остапкевич М.Б., Перепелкин В.А. Эффективное программирование современных микропроцессоров: учебное пособие. Новосибирск: Издательство НГТУ, 2014. – 148 с.
13. Миронов А.М. Методы верификации программ. М.: ДМК Пресс. 2023 – 332 с.
14. Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. М.: Радио и связь, 1988. – 256 с.
15. Старолетов С.М., Кондратьев Д.А., Гаранина Н.О., Шошмина И.В. Соревнования по формальной верификации VeHa-2023: опыт проведения // Труды Института системного программирования РАН. 2024. Т. 36. № 2. С. 141–168.
16. Шилов Н.В. Основы синтаксиса, семантики, трансляции и верификации программ: учебное пособие. Новосибирск: Издательство Новосибирского государственного университета, 2011. – 292 с.
17. Appel A.W. Verified Software Toolchain // Lecture Notes in Computer Science. 2011. Volume 6602. pp. 1–17.
18. Appel A.W., Beringer L., Cao Q., Dodds J. Verifiable C: Applying the Verified Software Toolchain to C programs. 2023. URL: https://github.com/PrincetonUniversity/VST/raw/master/doc/VC.pdf (Accessed 25 Jul 2025)

19. Apt K.R., Olderog E.-R. Assessing the Success and Impact of Hoare's Logic // Theories of Programming: The Life and Works of Tony Hoare. New York: ACM, 2021. pp. 41–76.
20. Apt K.R., Olderog E.-R. Fifty years of Hoare's logic // Formal Aspects of Computing. 2019. Volume 31. Issue 6. pp. 751–807.
21. Baudin P., Bobot F., Bühler D., Correnson L., Kirchner F., Kosmatov N., Maroneze A., Perrelle V., Prevosto V., Signoles J., Williams N. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform // Communications of the ACM. 2021. Volume 64. Issue 8. pp. 56–68.
22. Barbosa H., Barrett C., Brain M., Kremer G., Lachnitt H., Mann M., Mohamed A., Mohamed M., Niemetz A., Nötzli A., Ozdemir A., Preiner M., Reynolds A., Sheng Y., Tinelli C., Zohar Y. cvc5: A Versatile and Industrial-Strength SMT Solver // Lecture Notes in Computer Science. Volume 13243. pp. 415–442.
23. Barnett M., Leino K.R.M. Weakest-precondition of unstructured programs // Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. Lisbon, Portugal, September 5–6, 2005. New York: Association for Computing Machinery, 2005. pp. 82–87.
24. Barrett C., Conway C.L., Deters M., Hadarean L., Jovanović D., King T., Reynolds A., Tinelli C. CVC4 // Lecture Notes in Computer Science. 2011. Volume 6806. pp. 171–177.
25. Bertot Y. A Short Presentation of Coq // Lecture Notes in Computer Science. 2008. Volume 5170. pp. 12–16.
26. Bjørner N., Nachmanson L. Navigating the Universe of Z3 Theory Solvers // Lecture Notes in Computer Science. 2020. Volume 12475. pp. 8–24.
27. Blaauwbroek L., Urban J., Geuvers H. The Tactician // Lecture Notes in Computer Science. 2020. Volume 12236. pp. 271–277.
28. Blazy S., Dargaye Z., Leroy X. Formal Verification of a C Compiler Front-End // Lecture Notes in Computer Science. 2006. Volume 4085. pp. 460–475.
29. Blazy S., Leroy X. Mechanized Semantics for the Clight Subset of the C Language // Journal of Automated Reasoning. 2009. Volume 43. Issue 3. pp. 263–288.
30. Bobot F., Filliâtre J.-C., Marché C., Paskevich A. Let's verify this with Why3 // International Journal on Software Tools for Technology Transfer. 2015. Volume 17. Issue 6. pp. 709–727.
31. Brain M., Polgreen E. A Pyramid Of (Formal) Software Verification // Lecture Notes in Computer Science. 2025. Volume 14934. pp. 393–419.
32. Cao Q., Beringer L., Gruetter S., Dodds J., Appel A.W. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs // Journal of Automated Reasoning. 2018. Volume 61. Issue 1. pp. 367–422.
33. Clochard M., Gondelman L., Pereira M. The Matrix Reproved (Verification Pearl) // Journal of Automated Reasoning. 2018. Volume 60. Issue 3. pp. 365–383.
34. Conchon S., Iguernelala M., Mebsout A. A collaborative framework for non-linear integer arithmetic reasoning in Alt-Ergo // Proceedings of the 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. Timișoara, Romania, September 23–26, 2013. IEEE,

2013. pp. 161–168.

35. Correnson L. Qed. Computing What Remains to Be Proved // Lecture Notes in Computer Science. 2014. Volume 8430. pp. 215–229.
36. Cuoq P., Monate B., Pacalet A., Prevosto V. Functional dependencies of C functions via weakest pre-conditions // International Journal on Software Tools for Technology Transfer. 2011. Volume 13. Issue 5. pp. 405–417.
37. Czajka Ł., Kaliszyk C. Hammer for Coq: Automation for Dependent Type Theory // Journal of Automated Reasoning. 2018. Volume 61. Issue 1. pp. 423–453.
38. de Moura L., Bjørner N. Z3: An Efficient SMT Solver // Lecture Notes in Computer Science. 2008. Volume 4963. pp. 337–340.
39. Dénès M., Mörtberg A., Siles V. A Refinement-Based Approach to Computational Algebra in Coq // Lecture Notes in Computer Science. 2012. Volume 7406. pp. 83–98.
40. Dijkstra E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs // Communications of the ACM. 1975. Volume 18. Issue 8. pp. 453–457.
41. Dijkstra E.W., Schölten C.S. The strongest postcondition // Predicate Calculus and Program Semantics. New York: Springer, 1990. pp 209–215.
42. Efremov D., Mandrykin M., Khoroshilov A. Deductive verification of unmodified Linux kernel library functions // Lecture Notes in Computer Science. 2018. Volume 11245. pp. 216–234.
43. Filliâtre J.C. Deductive software verification // International Journal on Software Tools for Technology Transfer. 2011. Volume 13. Issue 5. Article ID: 397.
44. Filliâtre J.-C., Paskevich A. Why3 — Where Programs Meet Provers // Lecture Notes in Computer Science. Volume 7792. pp. 125–128.
45. Flanagan C., Saxe J.B. Avoiding exponential explosion: Generating compact verification conditions // Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '01). London, United Kingdom, January 17–19, 2001. New York: Association for Computing Machinery, 2001. pp. 193–205.
46. Floyd R.W. Assigning meanings to programs // Proc. Symposia in Applied Mathematics. Providence, 1967. Volume 19. pp. 19–32.
47. Furia C.A., Meyer B. Inferring Loop Invariants Using Postconditions // Lecture Notes in Computer Science. 2010. Volume 6300. pp. 277–300.
48. Grigore R., Charles J., Fairmichael F., Kiniry J. Strongest postcondition of unstructured programs // Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs (FTfJP '09). Genova, Italy, July 6, 2009. New York: Association for Computing Machinery, 2009. pp. 6:1–6:7.
49. Hähnle R., Huisman M. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools // Lecture Notes in Computer Science. 2019. Volume 10000. pp. 345–373.
50. Harrison J. Theorem Proving with the Real Numbers. London: Springer, 1998. – 186 p.
51. Hoare C.A.R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Volume 12. Issue 10. pp. 576–580.

52. Ishtiaq S.S., O'Hearn P.W. BI as an assertion language for mutable data structures // Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '01). London, United Kingdom, January 17–19, 2001. New York: Association for Computing Machinery, 2001. pp. 14–26.
53. Islam R., Ahmed I. Gemini-the most powerful LLM: Myth or Truth // Proceedings of the 2024 5th Information Communication Technologies Conference (ICTC). Nanjing, China, May 10–12, 2024. IEEE, 2024. pp. 303–308.
54. Kaufmann M., Moore J.S. An industrial strength theorem prover for a logic based on Common Lisp // IEEE Transactions on Software Engineering. 1997. Volume 23. Issue 4. pp. 203–213.
55. Knothe D., Bringmann O. Combining Small-Step and Big-Step Semantics to Verify Loop Optimizations // arXiv preprint arXiv:2602.19868. 2026. Access mode: `https://arxiv.org/abs/2602.19868`.
56. Kondratyev D.A., Maryasov I.V., Nepomniaschy V.A. The Automation of C Program Verification by the Symbolic Method of Loop Invariant Elimination // Automatic Control and Computer Sciences. 2019. Volume 53. Issue 7. pp. 653–662.
57. Kondratyev D., Maryasov I., Nepomniaschy V. Towards Automatic Deductive Verification of C Programs over Linear Arrays // Lecture Notes in Computer Science. 2019. Volume 11964. pp. 232–242.
58. Kondratyev D.A., Nepomniaschy V.A. Automation of C Program Deductive Verification without Using Loop Invariants // Programming and Computer Software. 2022. Volume 48. Issue 5. pp. 331–346.
59. Kondratyev D.A., Promsky A.V. Developing a self-applicable verification system. Theory and practice // Automatic Control and Computer Sciences. 2015. Volume 49. Issue 7. pp. 445–452.
60. Kosmatov N., Marché C., Moy Y., Signoles J. Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014 // Lecture Notes in Computer Science. 2016. Volume 9952. pp. 461–478.
61. Kovács L., Voronkov A. // Lecture Notes in Computer Science. 2013. Volume 8044. pp. 1–35.
62. Leino K.R.M. Efficient weakest preconditions // Information Processing Letters. 2005. Volume 93. Issue 6. pp. 281–288.
63. Leroy X. A formally verified compiler back-end // Journal of Automated Reasoning. 2009. Volume 43. Issue 4. pp. 363–446.
64. Leroy X. Formal verification of a realistic compiler // Communications of the ACM. 2009. Volume 52. Issue 7. pp. 107–115.
65. Mandrykin M.U., Khoroshilov A.V. High-level memory model with low-level pointer cast support for Jessie intermediate language // Programming and Computer Software. 2015. Volume 41. Issue 4. pp. 197–207.
66. Mandrykin M.U., Khoroshilov A.V. Region analysis for deductive verification of C programs // Programming and Computer Software. 2016. Volume 42. Issue 5. pp. 257–278.
67. Mandrykin M.U., Khoroshilov A.V. Towards deductive verification of C programs with shared data // Programming and Computer Software. 2016. Volume 42. Issue 5. pp. 324–332.

68. Maryasov I.V., Nepomniaschy V.A., Promsky A.V., Kondratyev D.A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. 2014. Volume 48. Issue 7. pp. 407–414.
69. Moore J.S. Milestones from the Pure Lisp theorem prover to ACL2 // Formal Aspects of Computing. 2019. Volume 31. Issue 6. pp. 699–732.
70. Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promsky A.V. Towards verification of C programs. C-light language and its formal semantics // Programming and Computer Software. 2002. Volume 28. Issue 6. pp. 314–323.
71. Nepomniaschy V.A., Anureev I.S., Promskii A.V. Towards Verification of C Programs: Axiomatic Semantics of the C-kernel Languages // Programming and Computer Software. 2003. Volume 29. Issue 6. pp. 338–350.
72. O'Hearn P. Separation logic // Communications of the ACM. 2019. Volume 62. Issue 2. pp. 86–95.
73. O'Hearn P. Separation Logic Tutorial // Lecture Notes in Computer Science. 2008. Volume 5366. pp. 15–21.
74. Palomo-Lozano F., Medina-Bulo I., Alonso-Jiménez J. Certification of matrix multiplication algorithms. Strassen's algorithm in ACL2 // Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics. Edinburgh, United Kingdom, September 3–6, 2001. pp. 283–298.
75. Paraskevopoulou Z., HriŢcu C., Dénès M., Lampropoulos L., Pierce B.C. Foundational Property-Based Testing // Lecture Notes in Computer Science. 2015. Volume 9236. pp. 325–343.
76. Paulin-Mohring C. Introduction to the Coq Proof-Assistant for Practical Software Verification // Lecture Notes in Computer Science. 2012. Volume 7682. pp. 45–95.
77. Reynolds J.C. An Overview of Separation Logic // Lecture Notes in Computer Science. 2008. Volume 4171. pp. 460–469.
78. Reynolds J.C. Separation logic: a logic for shared mutable data structures // Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. Copenhagen, Denmark, July 22–25, 2002. IEEE, 2002. pp. 55–74.
79. Srivastava S., Gulwani S., Foster J.S. From program verification to program synthesis // Proceedings of 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Madrid, Spain, January 17–23, 2010. New York: Association for Computing Machinery, 2010. pp. 313–326.
80. Staroletov S., Kondratyev D., Shelekhov V., Kogtenkov A., Shilov N.V., Garanina N., Shoshmina I., Cherganov T., Dyadov V. VeHa: A Hybrid National Verification Hackathon for Better Formal Methods Education // Lecture Notes in Computer Science. 2026. Volume 16566. pp. 127–146.
81. Schulz S. E – a brainiac theorem prover // AI Communications. 2002. Volume 15. Issue 2-3. pp. 111–126.
82. Smith T.M. Theory and Practice of Classical Matrix-Matrix Multiplication for Hierarchical Memory Architectures: thes... doct. phylosophy (computer sci.). – Austin, 2017. – 136 p.
83. Strassen V. Gaussian elimination is not optimal // Numerische Mathematik. Volume 13. Issue 4.

pp. 354–356.

84. Sulatycke P.D., Ghose K. Caching-Efficient Multithreaded Fast Multiplication of Sparse Matrices // Proceedings of the 12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98). Orlando, USA. March 30 – April 3, 1998. IEEE, 1998. pp.117–123.
85. Turner O., Chakraborty S. VLIM: Verified Loop Interchange for Optimised Matrix Multiplication // Proceedings of the 29th Design, Automation and Test in Europe Conference DATE 2026. Verona, Italy, April 20 – April 22, 2026.
86. Vlasov A.A., Lylova S.S., Latkin E.I., Veretennikov A.A. minigemm. URL: `https://gitflic.ru/project/nsu/minigemm` (Accessed 20 May 2025)
87. Volkov G., Mandrykin M. Efremov D. Lemma functions for Frama-C: C programs as proofs // Proceedings of the 2018 Ivannikov Ispras Open Conference (ISPRAS). Moscow, Russia, November 22–23, 2018. IEEE, 2018. pp. 31–38.

A. Условие корректности и его доказательство, где применяются все введенные нами стратегии доказательства

Рассмотрим одно из условий корректности реализации умножения матриц над математическими вещественными числами с оптимизациями в виде изменения порядка вложенности циклов для построчного обхода результирующей матрицы и умножаемой матрицы, а также доказательство такого условия корректности:

```
intros.
 assert (EqA: forall idx, (0 <= idx)%Z ->
 (idx < i * i1)%Z -> a13 (shift a idx) = a9 (shift a idx)).
  {
    intros idx Hidx_min Hidx_max. unfold a13, a11, Map.set.
    destruct (why_decidable_eq (shift a2 (i7 + x3)%Z)
    (shift a idx)) as [Heq | Hneq].
    - (* Раскрываем базу И СМЕЩЕНИЕ (HeqOff)! *)
      unfold shift, base, offset, a4, a5, x1, x2 in *.
      injection Heq as HeqBase HeqOff.
      (* Разбираем гипотезу H25 (A и C разделены) *)
      destruct H25 as[Hb1|[Hb2|[Hb3|[Hb4|Hb5]]]].
      * nia.
```

```
      * nia.
      * unfold base, shift in Hb3.
        unfold base in Hb3.
        congruence.
      * (* Подсказка: конец изменяемой строки C
      не выходит за пределы матрицы C *)
        assert (Step: (i7 + i2 * i3 < i1 * i2)%Z) by nia.
         destruct a2 as[b_a2 o_a2]. destruct a as [b_a o_a].
        unfold shift, offset, base in *. simpl in *.
      nia.

      *  destruct a2 as [b_a2 o_a2]. destruct a as [b_a o_a].
        unfold shift, offset, base in *. simpl in *.
        nia.
    - reflexivity.
 }
assert (EqB: forall idx, (0 <= idx)%Z ->
(idx < i * i2)%Z -> a13 (shift a1 idx) = a9 (shift a1 idx)).
 {
   intros idx Hidx_min Hidx_max.
   unfold a13, a11, Map.set.
   destruct (why_decidable_eq (shift a2 (i7 + x3)%Z)
   (shift a1 idx)) as[Heq | Hneq].
   -
     unfold shift, base, offset, a3, a5, x, x2, x3 in *.
     injection Heq as HeqBase HeqOff.

     (* Разбираем гипотезу H24 (B и C разделены) *)
     destruct H24 as [Hb1|[Hb2|[Hb3|[Hb4|Hb5]]]].
     * nia.
     * nia.
           * assert (Step: (i7 + i2 * i3 < i1 * i2)%Z) by nia.
```

```
        destruct a2 as[b_a2 o_a2]. destruct a1 as [b_a o_a].
        unfold shift, offset, base, x3 in *. simpl in *. nia.
    * (* Ветка 4: C ДО B *)
      assert (Step: (i7 + i2 * i3 < i1 * i2)%Z) by nia.
      destruct a2 as [b_a2 o_a2]. destruct a1 as [b_a1 o_a1].
      simpl in *.
      (* Теперь nia видит Hidx_min (0 <= idx), HeqOff и Step.
      Это тоже успешное доказательство! *)
      nia.
    * (* Ветка 5: B ДО C *)
      destruct a2 as [b_a2 o_a2]. destruct a1 as [b_a1 o_a1].
      simpl in *.
      (* Теперь nia видит Hidx_max (idx < i * i2) и HeqOff.
      Это тоже успешное доказательство! *)
      nia.
  - reflexivity.
}
assert (Hframe_dot: forall k_dot col res_,
    (k_dot <= i)%Z -> (col < i2)%Z -> (0 <= col)%Z ->
    P_DotProduction a9 a a1 k_dot i3 i col i2 res_ ->
    P_DotProduction a13 a a1 k_dot i3 i col i2 res_).
{
  intros k_dot col res_ Hk_max Hcol_min Hcol_max Hdot_k.

  (* Делаем индукцию. Переменная 'to' будет означать текущую длину *)
  induction Hdot_k as [to_empty Hempty | to_pos Hpos Hdot_prev IH].
  - apply Q_dotproduction_empty_range. lia.
  - (* Заменяем новую память на старую.
       nia видит Hcol_max (col < i2) и Hk_max (to_pos <= i) и
       легко докажет границы! *)
    rewrite <- EqA; [ | lia | nia ].
    rewrite <- EqB.
```

```
      apply Q_dotproduction_positive_range.
      + lia.
      + apply IHHdot_k; try assumption. nia.
      + nia. +nia.
  }

  assert (H_cases: (i4 < i7 \/ i4 = i7)%Z) by lia.
  destruct H_cases as [H_lt | H_eq].
-
  assert (EqC: a13 (shift a2 (i4 + x3)%Z) = a9 (shift a2 (i4 + x3)%Z)).
    {
      unfold a13, a11, Map.set.
      destruct (why_decidable_eq (shift a2 (i7 + x3)%Z)
      (shift a2 (i4 + x3)%Z)) as[Heq | Hneq].
      * unfold shift, offset in Heq. injection Heq.
        lia. (* nia понимает, что i7 не может быть равно i4 *)
      * reflexivity.
    }
rewrite EqC.
apply Hframe_dot.
lia. nia. nia.
apply H36.
nia. nia.
-rewrite H_eq.
 assert (EqC_new: a13 (shift a2 (i7 + x3)%Z) = a12).
    {
      unfold a13, a11, Map.set.
      destruct (why_decidable_eq (shift a2 (i7 + x3)%Z)
      (shift a2 (i7 + x3)%Z)) as[Heq | Hneq].
      * reflexivity.
      * congruence. (* Абсурд, адрес всегда равен самому себе *)
```

```
    }
rewrite EqC_new.
replace i7 with i4 by H_eq.
apply Hframe_dot.
lia.
rewrite H_eq. nia. nia.
replace i4 with i7 by H_eq.
exact H32.
```

В данном доказательстве применяются все заданные нами стратегии доказательства условий корректности.