2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

doi 10.26089/NumMet.v27r104 УДК 519.6

Модификации матричного метода муравьиных колоний для параметрической оптимизации с применением GPU с технологией CUDA

Ю. П. Титов
Московский авиационный институт (национальный исследовательский университет),
Москва, Российская Федерация
ORCID: 0000-0002-9093-6755, e-mail: kalengul@mail.ru

Аннотация: Статья посвящена разработке и исследованию матричных модификаций метода муравьиных колоний (ACO) для решения параметрических задач оптимизации с использованием графических процессоров (GPU) на архитектуре CUDA. В отличие от большинства существующих исследований, фокусирующихся на задаче коммивояжера (TSP), предложенный метод специализирован для поиска оптимальных значений независимых параметров, что приводит к иной структуре графа и обуславливает необходимость разработки новых схем распараллеливания. Представлены оригинальные схемы параллелизма, включающие разделение алгоритма на три этапа и введение "параметрического" параллелизма по измерениям задачи. Разработаны и протестированы различные стратегии распараллеливания и управления памятью: от базовой модификации MatrixACO-Basic до оптимизированных версий MatrixACO-Optimization (с warp-редукцией и оптимизацией хэш-таблицы) и MatrixACO-Transported (с транспонированным хранением данных). Эксперименты проводились на графических ускорителях NVIDIA Tesla V100 16GB SXM2, платформе Jetson ORIN и различных персональных компьютерах с использованием многомерных тестовых функций. Результаты показывают, что для задачи с 128 параметрами оптимизированная версия MatrixACO-Optimization обеспечивает время обработки одного слоя графа, равное 0.13 мс, что соответствует ускорению более чем в 30 раз по сравнению с оптимизированной CPU-реализацией на основе OpenMP, а в случае задачи с 65536 параметрами способствует ускорению почти в 45 раз. Для задачи с 512 параметрами GPU-реализация демонстрирует на 76% более высокую энергоэффективность по сравнению с CPU-версией, а также почти восьмикратное (в 7.9 раза) снижение Energy-Delay Product. Исследование подтверждает необходимость создания специализированных GPU-реализаций для параметрических задач и выявляет отличную масштабируемость предложенных алгоритмов.

Ключевые слова: метод муравьиных колоний, параметрическая задача, оптимизация, CUDA, Tesla V100.

Благодарности: Автор выражает благодарность В. А. Судакову за полезные и плодотворные обсуждения, которые способствовали лучшему изложению материала.

Для цитирования: Титов Ю.П. Модификации матричного метода муравьиных колоний для параметрической оптимизации с применением GPU с технологией CUDA // Вычислительные методы и программирование. 2026. **27**, № 1. 46–66. doi 10.26089/NumMet.v27r104.

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

Modifications of the ant colony matrix method for parametric optimization using CUDA-based GPUs

Yurii P. Titov
Moscow Aviation Institute (National Research University),
Moscow, Russia
ORCID: 0000-0002-9093-6755, e-mail: kalengul@mail.ru

Abstract: This article focuses on the development and investigation of the matrix-based modifications of the Ant Colony Optimization (ACO) method for solving parametric optimization problems using CUDA-based Graphics Processing Units (GPUs). Unlike most existing studies focusing on the Traveling Salesman Problem (TSP), the proposed method is specialized for finding optimal values of independent parameters, which leads to a different graph structure and requires the development of novel parallelization schemes. Original parallelization strategies are proposed, including a three-stage decomposition of the algorithm and introduction of "parameter-level" parallelism in the problem dimensions. Various strategies for parallelization and memory management were developed and tested: from the basic MatrixACO-Basic modification to the optimized versions of MatrixACO-Optimization (featuring warp reduction and optimized hash-table handling) and MatrixACO-Transported (utilizing transposed data storage). Experiments were conducted on NVIDIA Tesla V100 16GB SXM2 GPU, the Jetson ORIN platform, and various personal computers using multi-dimensional test functions. Results demonstrate that for a problem with 128 parameters the optimized version of MatrixACO-Optimization provides an execution time of 0.13 ms per graph layer, which corresponds to a speed-up of more than 30 times compared to the optimized CPU implementation based on OpenMP, and in the case of a very large-scale problem with 65536 parameters it speeds up by almost 45 times. For a problem with 512 parameters, the GPU realization shows 76% higher energy efficiency compared to the CPU version, as well as an almost eightfold (7.9 times) reduction in Energy-Delay Product. The study confirms the necessity of developing specialized GPU implementations for parametric tasks and reveals the excellent scalability of the proposed algorithms.

Keywords: Ant Colony Optimization, parametric problem, optimization, CUDA, Tesla V100.

Acknowledgements: The author would like to thank V. A. Sudakov for useful and fruitful discussions, which contributed to a better presentation of the material.

For citation: Yu. P. Titov, "Modifications of the ant colony matrix method for parametric optimization using CUDA-based GPUs," Numerical Methods and Programming. **27** (1), 46–66 (2026). doi 10.26089/NumMet.v27r104.

1. Введение. Метод оптимизации, основанный на имитации роевого поведения муравьев, был впервые предложен в работе [1] под названием Ant Algorithm, а затем усовершенствован до Ant System (ACS) в работе [2]. В обобщенном методе муравьиных колоний (Ant Colony Optimization, ACO) используются программные агенты (или "муравьи"), каждый из которых пытается найти решение заданной задачи. Для иллюстрации работы данных алгоритмов традиционно рассматривается задача коммивояжера (TSP). Кроме нее, алгоритм ACO успешно применялся для решения ряда комбинаторных задач, включая маршрутизацию транспортных средств [3], задачу о квадратичном назначении [4], задачу о множественном рюкзаке [5] и другие. Для устранения проблемы стагнации и повышения скорости поиска оптимального решения были предложены следующие модификации: Elitist Ant System (EAS) [6], ANT-Q (ANTQ) [7], ранговая версия Ant System (RAS) [8], MAX–MIN Ant System (MMAS) [9], Best-Worst Ant System (BWAS) [10].

Модификации ACO могут быть легко распараллелены при помощи как крупнозернистых (основанных на параллелизме итераций или целых колоний), так и мелкозернистых (основанных на одновременном выполнении вычислений отдельными муравьями-агентами) схем. До 2005 г. основное внимание в исследованиях параллельных реализаций ACO уделялось крупнозернистым методам, превосходившим мелкозернистые по эффективности. Последние же, несмотря на активную разработку, по-прежнему про-

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

должали страдать от высоких накладных расходов на связь [11]. Действительно, в то время соответствующие методы реализовывались преимущественно на распределенных системах, которые представляли собой наиболее "жизнеспособную" платформу для распараллеливания. Примерами успешной реализации крупнозернистой методологии являются модификации P-ACO [12] и Parallel ACO [11].

Существенный скачок в развитии параллельных модификаций ACO был обусловлен появлением архитектуры CUDA в 2007 г. Одним из ранних примеров стал GPU-ориентированный вариант алгоритма MMAS, известный как GMMAS, реализованный с использованием фреймворка CUDA [13]. Центральным нововведением GMMAS стало использование матриц для представления состояний феромонов и путей. Вместо прямого обхода всей сети вычисляется лишь подмножество наиболее перспективных путей. Феромонные значения представляются в виде матрицы смежности, которая обновляется после каждой итерации путем матричных преобразований.

В дальнейшем появилось множество различных параллельных модификаций ACO. В PartialACO [14] предлагается вместо использования всех найденных путей проводить обновление только некоторых маршрутов, сохраняя важную информацию о предыдущем шаге и снижая таким образом временные затраты. Одной из главных инноваций в ACS-GPU-Alt [15] стала концепция чередующихся обновлений (Alternating Updates). Она предусматривает разделение всего пространства графа на две группы, называемые зонами. Во время одного цикла обработки обновляются феромоны только одной зоны, а вторая остается замороженной. Таким образом, работа происходит попеременно, что устраняет необходимость глобальной блокировки и позволяет более эффективно загружать ядра GPU. Эта техника решает проблему доступа к памяти, характерную для традиционных реализаций ACS на GPU, и предотвращает задержку вычислений из-за ожидания освобождения общих структур данных.

Реализация Multi-Colony Ant Colony Optimization (MCACO) [16] на CUDA предполагает одновременное функционирование нескольких колоний муравьев, работающих параллельно на разных ядрах GPU. Каждая колония ищет собственное решение, что в конечном счете позволяет рассматривать большее пространство возможных решений. Каждая колония действует независимо, перемещается по пространству графа и оставляет феромонные следы. Периодически проводится коллективная оценка полученных решений с последующим обменом наилучшей информацией между колониями. Важной частью реализации стало эффективное хранение и обработка матриц расстояний и феромонных следов, что обеспечивается специальной структурой данных, подходящей для массового параллелизма, и новой схемой хранения данных, называемой block-level data partitioning, позволяющей каждому блоку CUDA обрабатывать отдельный участок данных.

Разработка параллельных модификаций коснулась и наиболее распространенной одиночной модификации MMAS. Предложенная параллельная реализация продемонстрировала значительное ускорение (до 21 раза) по сравнению с ранее опубликованными методами, апробируемыми на схожих GPU архитектурах. Новая версия способна генерировать свыше миллиона кандидатных решений в секунду для некоторых задач TSP среднего размера [17].

Разработан высокоэффективный параллельный алгоритм ACS на GPU с использованием специальной стратегии разделения работы между потоками и статико-динамического сбалансированного набора кандидатов (Static–Dynamic Balanced Candidate Set Strategy, ID-CS). Применяется модель специализации потоков (warp specialization), при которой каждый поток делится на две части: приватную (private work) и публичную (public work). Приватная часть занимается построением индивидуального решения каждой отдельной особи-муравья, а публичная отвечает за обновление глобальных данных, совместно используемых всеми муравьями. Испытания подтвердили значительное сокращение числа глобальных поисков и существенное улучшение скорости выполнения по сравнению с традиционным GPU-решением ACS-GPU-Alt [18].

Однако представленные параллельные модификации, как и исходный алгоритм ACO, в основном разрабатывались и усовершенствовались для задач комбинаторной оптимизации, например TSP, где решение представляет собой гамильтонов цикл, а выбор следующей вершины зависит от уже пройденного пути (используется список запретов — tabu list). Параметрическая задача, рассматриваемая в данной работе, имеет принципиально иную природу: требуется найти оптимальный вектор значений для множества независимых параметров. Это условие преобразует задачу в поиск по параметрическому графу, в котором каждый слой соответствует одному параметру, а путь муравья-агента представляет собой выбор конкретного значения в каждом слое, не зависящий от выборов в других слоях. Данная особенность открывает возможность для принципиально иного, "поэлементного" параллелизма, когда потоки могут независи-

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

мо обрабатывать отдельные параметры (слои графа), что невозможно в TSP. Основная научная задача данной работы заключается в разработке и исследовании матричной модификации ACO и соответствующих схем распараллеливания на GPU, способных эффективно использовать специфику параметрических задач для преодоления ограничений, присущих прямым портированиям алгоритмов на архитектуру GPU.

2. Применение метода муравьиных колоний при решении параметрической задачи. Расширение применимости метода на параметрические задачи позволило решать такие задачи, в которых необходимо определить значения параметров, обеспечивающих оптимальное значение целевого критерия [19]. Определение значения критерия осуществляется по итогам работы аналитической или имитационной модели.

В формализованном виде решение параметрической задачи может быть определено следующим образом. Пусть имеется дискретное множество параметров $P = (p_1, p_2, \ldots, p_i, \ldots, p_n)$. У каждого i-го параметра существует множество допустимых значений $V_i = (v_{1,i}, v_{2,i}, \ldots, v_{j,i}, \ldots, v_{m_i,i})$, $i = \overline{1,n}$. Количество допустимых значений определяется величиной $m_i > 0$, зависящей от номера параметра. Так, для различных параметров может быть определено различное количество возможных значений. В результате работы алгоритма оптимизации определяется вектор значений параметров (решение) $X_k = (x_{1,k}, x_{2,k}, \ldots, x_{i,k}, \ldots, x_{n,k})$, где $\forall i\ \exists j:\ (x_{i,k} = v_{j,i}) \wedge (v_{j,i} \in V_i)$. Найденный вектор X_k отправляется на вычислитель (аналитическую или имитационную модель), который возвращает значение целевой функции $f(X_k)$. Решения рассматриваются в дискретном пространстве значений параметров. На функцию $f(X_k)$ не накладываются требования непрерывности и дифференцируемости. Нужно найти оптимальное решение X_i^*; в случае, если таких решений несколько, необходимо определить множество $Y = \{X_1^*, X_2^*, \ldots, X_z^*\}$, такое что $f(X_i^*) = Z$, $\forall i$, где Z — оптимальное значение целевой функции [20].

Ключевым отличием данной работы от классических подходов является применение ACO не к комбинаторным задачам, а к параметрическим. Расширение применимости метода к данному классу задач потребовало его существенной переформулировки. Если в задаче TSP решением является упорядоченная последовательность вершин, то в параметрической задаче решение представляет собой вектор X^*, где каждая его компонента x_i выбирается из своего множества допустимых значений $v_{j,i}$ независимо от других. Такой подход позволяет определить параметрический граф, в котором путь муравья-агента задает точное значение каждого параметра. Отсутствие зависимостей между параметрами (в противоположность TSP, где выбор зависит от истории пройденного пути) является фундаментальным свойством, лежащим в основе предлагаемых параллельных модификаций.

В случае, когда параметры задачи принимают непрерывные значения, выполняется их дискретизация с определенным шагом, в результате чего для любых параметров формируется дискретное множество допустимых значений V_i. Однако при мелком шаге дискретизации размер множества V_i становится чрезмерно большим, что негативно сказывается на эффективности ACO. Для этой решения проблемы предложена процедура декомпозиции набора значений параметров на слои, содержащие отдельные компоненты параметров, а итоговое значение параметра определяется путем линейной комбинации компонент. Подобная декомпозиция с разложением на степенные части, а затем на простые сомножители показала свою эффективность при решении параметрических задач [21].

Важной особенностью решения подобного сорта задач является использование сложной аналитической или имитационной модели, для которой время вычисления значений целевой функции может быть существенным. Для уменьшения данного времени предлагается использовать промежуточное хранилище — хэш-таблицу, в которой хранятся уже рассмотренные пути муравьев-агентов. Предложены модификации ACOCN, ACOCNI, ACOCCyN, ACOCCyI, ACOCT и ACOCTSort, показавшие высокую эффективность [22].

3. Матричная модификация метода муравьиных колоний. Эффективность применения различных модификаций ACO обычно оценивается по количеству итераций, которые требуются алгоритму для нахождения наилучшего решения, а также по точности найденного решения. Уменьшение количества итераций достигается за счет модификаций метода. Для сокращения времени вычислений в алгоритме ACO используются его различные параллельные модификации. Параллельные методы можно разделить на два класса: 1) методы, сохраняющие исходную логику алгоритма, но допускающие многопоточное выполнение отдельных его этапов с применением механизмов синхронизации и блокировки; 2) методы, преобразующие алгоритм к виду, пригодному для матричных и векторных вычислений, что позволяет эффективно использовать SIMD- и SIMT-архитектуры ускорителей. При этом переход к параметрической

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

задаче кардинально меняет структуру данных и возможные стратегии параллелизма. Для задачи TSP состояние феромона часто представляется матрицей смежности полного графа, а параллелизм осуществляется на уровне независимых муравьев или колоний. В нашем случае параметрический граф естественно кодируется матрицей V. Это представление не только компактно, но и прямо отображает возможность параллелизма по измерениям: поскольку выбор значения одного параметра не зависит от выбора значений других, вычисления для каждого параметра (столбца матрицы) могут выполняться в отдельных потоках. Таким образом, в дополнение к традиционному "муравьиному" параллелизму (по агентам), мы вводим "параметрический" параллелизм (по измерениям задачи), что является уникальным преимуществом параметрической формулировки.

Для решения параметрической задачи структура графа может быть представлена в виде матрицы V размерности $n \times m$, где n — число параметров, а m — максимальное количество допустимых значений (вершин) параметра. Для создания матрицы параметры, содержащие менее m значений, дополняются незначащими вершинами. Если в незначащих вершинах количество феромона будет равно нулю, то и вероятность их выбора также будет равна нулю. Как правило, в параметрических задачах количество допустимых значений параметров значительно превышает число самих параметров, т.е. $m \gg n$. В отличие от задачи TSP, в которой путь из вершины может зависеть от уже пройденных путей (tabu list), в параметрической задаче каждое значение параметра или слоя может быть определено независимо от других параметров, т.е. количество потоков может достигать размерности параметрического графа n. Кроме того, возможно разделение на потоки отдельных муравьев-агентов, что соответствует мелкозернистой параллелизации ACO.

Представление исходных данных параметрической задачи в виде матрицы позволяет также хранить и обрабатывать в виде матриц информацию о количестве феромона, количестве посещений вершин и т.д. При такой постановке формула из работы [19] для определения матрицы P вероятностей выбора значений параметра примет вид:

$$P_{i,j} = \frac{Z_{i,j}}{Z'_i}, \quad Z_{i,j} = \lambda_1 \left(T_{\text{norm}}\right)_{i,j} + \lambda_2 \frac{1}{\theta_{i,j}}, \quad Z'_i = \sum_{j=1}^{m} Z_{i,j},$$

где P — матрица вероятностей размером $n \times m$, Z — матрица $n \times m$ значений аддитивной (в оригинальном алгоритме мультипликативной) свертки, на основе которой определяются вероятности выбора вершин, Z' — вектор для нормировки результатов свертки, T_{norm} — нормированная матрица феромона, $\left(T_{\text{norm}}\right)_{i,j} = T_{i,j} \Big/ \sum\limits_{s=1}^{m} T_{i,s}$, T — матрица размера $n \times m$ для хранения абсолютных значений феромона, $T_{i,j}$ — ее значения, θ — матрица $n \times m$ количества посещений вершин, соответствующих конкретному значению параметра, $\dfrac{1}{\theta_{i,j}}$ — матрица обратных значений, λ_1, λ_2 — весовые коэффициенты аддитивной свертки. В приведенных исследованиях $\lambda_1 = \lambda_2 = 1$. В случае наличия эвристической информации, заданной в виде матрицы $\mathcal{H}$ размера $n \times m$, формула вычисления матрицы значений аддитивной свертки принимает вид $Z_{i,j} = \left(\lambda_1 \left(T_{\text{norm}}\right)_{i,j} + \lambda_2 \dfrac{1}{\theta_{i,j}}\right)^{\alpha} \mathcal{H}_{i,j}^{\beta}$, где α и β — весовые коэффициенты мультипликативной свертки, являющиеся целыми неотрицательными числами. Операции умножения и возведения в степень для матриц выполняются поэлементно.

Путь муравья-агента вычисляется с помощью значений матричной функции распределения F, где $F_{i,j} = \sum\limits_{k=1}^{j} P_{i,k}$, $i = \overline{1,n}$, $j = \overline{1,m}$. Выбор конкретной вершины осуществляется методом обратной функции распределения, для которого требуется значение равномерно распределенной случайной величины $r_{i,k}$, где i — номер параметра, k — номер муравья-агента. Для одного для муравья-агента эта величина представляется вектором R с n компонентами для каждого параметра. С учетом множества агентов на одной итерации — матрицей R размера $K \times n$, где K — параметр метода муравьиных колоний, определяющий количество муравьев-агентов (размер поколения) на итерации. При применении метода обратной функции необходимо найти такое значение s, чтобы выполнялись неравенства $F_{i,s-1} \leqslant r_{i,k} \leqslant F_{i,s}$ для всех i. Пути муравьев-агентов хранятся в матрице X размера $K \times n$, на основе которой вычисляется вектор значений целевой функции Y с K компонентами.

После вычисления всех значений целевой функции на итерации t происходит обновление феромона. Количество феромона, откладываемого на вершины пути муравья-агента, прямо пропорционально

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

качеству полученного решения: чем ближе решение к оптимальному, тем больше феромона добавляется. Значения матрицы феромонов на следующей итерации $(t+1)$ определяются по формуле: $T_{i,j}(t+1) = \rho T_{i,j}(t) + \dfrac{Q}{\sum_k Y_k}$ для всех i и j, где $\rho \in (0,1]$ — коэффициент испарения феромона, определяющий скорость "забывания" предыдущего опыта и предотвращающий неограниченное накопление феромонов на субоптимальных путях, Q — масштабирующий коэффициент, регулирующий интенсивность отложения нового феромона и влияющий на скорость сходимости алгоритма. Несоответствие размерностей $n \times m$ матрицы T и $K \times 1$ вектора Y разрешается путем добавления значения элемента вектора Y только для вершин, которые выбрал муравей-агент. Алгоритмическая запись данной процедуры может быть представлена в виде $T\big[X[k,j],j\big](t+1) := T\big[X[k,j],j\big](t+1) + Q/Y[k]$, где обновление выполняется для всех $j = \overline{1,n}$ и $k = \overline{1,K}$.

3.1. Особенности реализации матричной модификации метода муравьиных колоний на GPU с технологией CUDA. Остановка работы алгоритма происходит по достижении заданного числа итераций N, которое является гиперпараметром алгоритма. Общий алгоритм матричной модификации метода муравьиных колоний состоит из последовательных матричных и векторных преобразований. На каждой итерации последовательно вычисляется вектор $\sum\limits_s T_{i,s}$, матрицы T_{norm} и Z, вектор Z', матрицы P и F. Далее в цикле для каждого муравья из поколения размерностью K вычисляется матрица R, позиция s, матрица X и вектор Y, после чего происходит обновление матриц θ и T.

Для работы с технологией NVIDIA CUDA определяется количество потоков и блоков, которые в виде переменных (`threadIdx` и `blockIdx` на языке C++) могут быть использованы в алгоритме для совершения параллельного доступа к данным. Эффективность выполнения кода на CUDA в значительной степени зависит от минимизации расхождения внутри warp'а (warp divergence), которое возникает при выполнении условных операторов (например, `if (threadIdx.x == 0)`) или циклов с переменным числом итераций. В алгоритме ACO такие операции неизбежны на этапе выбора пути агентом (поиск позиции s). При divergence warp CUDA выполняет инструкции для разных ветвей кода последовательно, что снижает эффективность работы вычислительных ядер. Другим критическим фактором является организация доступа к памяти. Для глобальной памяти необходимо обеспечивать консолидированный (coalesced) доступ, когда соседние потоки обращаются к физически смежным адресам памяти, в противном случае пропускная способность резко падает. Использование разделяемой (shared) памяти позволяет снизить задержки при обращении к данным, общим для потоков одного блока, но требует тщательной синхронизации и сталкивается с ограничениями по объему (обычно 48–96 КБ на блок). Эти архитектурные особенности напрямую определяют стратегию распараллеливания матричной модификации ACO.

Между итерациями необходимо фиксировать состояние графа матрицами θ и T, которые непосредственно определяют вероятности выбора значений на следующей итерации. Параллелизм в рамках одной итерации может быть организован на уровне отдельных муравьев-агентов: каждый агент работает в своем потоке, независимо строя полное решение (вектор значений параметров). Такой подход соответствует крупнозернистым модификациям ACO. В рамках CUDA-реализации вся популяция из K агентов распределяется по блокам и потокам, а уникальный номер агента в популяции вычисляется следующим образом: `nom_ant = threadIdx.x + blockIdx.x * blockDim.x`. В этой схеме каждый поток выполняет полную последовательность шагов алгоритма для своего агента, включая вычисление нормированной матрицы феромонов T_{norm}, матрицы Z, вектора Z', матриц P и F, а также обновление матриц θ и T после перемещения муравья-агента. Поскольку все указанные матричные и векторные операции для разных агентов имеют идентичную структуру (одинаковое число итераций в циклах и отсутствие условных переходов), они могут выполняться потоками синхронно и однотипно, что соответствует принципу "одна инструкция — множество потоков" (Simple Instruction, Multiple Threads — SIMT) на уровне warp'а. Благодаря этому все агенты на данном этапе работают с одинаковыми промежуточными данными. Ситуация меняется на этапе выбора конкретных значений параметров, когда для каждого агента определяется позиция s с помощью метода обратной функции. Поскольку на этом этапе осуществляется проверка неравенства $F_{i,s-1} \leqslant r_{i,k} \leqslant F_{i,s}$, возникают условные переходы и необходимость использования переменного числа операций сравнения, что нарушает синхронность выполнения потоков внутри warp'а (warp divergence). Хотя повторное вычисление "общих" матриц в каждом потоке не является оптимальным с точки зрения использования ресурсов GPU, такая организация гарантирует корректность результатов и их полное соответствие результатам последовательной CPU-реализации алгоритма.

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

Особое внимание следует уделить оптимизации обращений к памяти при реализации на GPU. Архитектура CUDA предусматривает четыре типа памяти, отличающихся по скорости выделения и обращения (от самой медленной к самой быстрой): 1) глобальная память (Global Memory), которая может быть выделена из кода, выполняемого на CPU, общая для всех потоков и блоков; 2) разделяемая память (Shared Memory) является общей для всех потоков одного блока и выделяется в коде, выполняющемся на GPU; 3) локальная память потока (Local Memory) содержит уникальные для потока переменные; 4) константная память (Constant Memory), содержащая неизменяемые на стороне GPU переменные. В глобальную память размещаются данные, которые могут быть получены на CPU, а также результаты работы метода на GPU. Основными входными данными для алгоритма является матрица значений параметров V, которая требуется для вычисления целевой функции. Эта матрица обычно загружается из файла или генерируется в соответствии с ограничениями и точностью, установленными пользователем на значения параметров системы. Размещение данной матрицы в константной памяти было бы оптимальным решением, если бы не ограничения последней по объему. Так, например, для видеокарт, рассматриваемых в работе объем константной памяти составляет 65536 байт. Для оптимальной структуры графа, в которой каждый слой имеет 5 значений, матрица V имеет размер $n \times 5$, что соответствует $5n$ элементам типа double, которые занимают $40n$ байт памяти. Максимальное количество слоев параметров, которые можно записать в константную память, не превышает 1617 (77 параметров). Если значения параметров лежат в диапазоне $(-1, 1)$ и задаются с точностью 10^{-10}, то для каждого параметра требуется $2d + 1 = 21$ слой, где d — количество слоев с дискретными значениями в промежутке $[0, 10]$. В общем виде при применении оптимального разделения на слои графа ограничение на использование константной памяти может быть получено из условия $W/c = 5n(2d + 1)$, где W — объем константной памяти (обычно 65536 байт), c — размер одного элемента в байтах (для типа `double` $c = 8$ байт, для `float` или `int` $c = 4$ байта). Если матрица V не помещается в локальной памяти, то требуется ее размещение в глобальной памяти.

Матрицы θ и T могут размещаться в разделяемой памяти, если все потоки выполняются в одном блоке. Однако объем разделяемой памяти весьма небольшой (обычно 48–96 КБ на блок), что накладывает ограничения на размер матриц. Для двух матриц размером $n \times 5$ типа `double` ($80n$ байт) максимальное количество слоев n варьируется от 614 (при 48 КБ) до 1228 (при 96 КБ). Данные матрицы могут быть инициализированы внутри GPU. В результате необходимость в работе с матрицами на CPU отсутствует, и они могут быть размещены в разделяемой памяти при условии, что все муравьи-агенты выполняются в одном блоке. Обычно для GPU с технологией CUDA максимальное число потоков в одном блоке составляет 1024, что накладывает верхнее ограничение на количество агентов в одном поколении. При большем числе муравьев-агентов в одном поколении их выполнение распределяется по различным блокам, а матрицы θ и T должны размещаться в глобальной памяти.

Результатом работы модификации ACO является оптимальный вектор значений параметров X^*. Данный вектор объявляется в глобальной памяти, так как по завершении вычислений на GPU требуется его копирование в оперативную память CPU. Остальные матрицы T_{norm}, Z, P, F и вектор Z' могут быть определены в локальной памяти. Данные, уникальные для каждого агента, могут быть представлены скалярами и векторами в локальной памяти, например, векторами X и R, позициями s, скаляром Y, определяющим одно единственное значение целевой функции для конкретного муравья-агента. Матричную модификацию ACO, в которой все этапы работы выполняются в одном вызове GPU, а данные хранятся в константной и локальной памяти, будем называть MatrixACO-Local или префикс-Local для других версий алгоритмов. Вариант, требующий размещения матриц V, θ и T в глобальной памяти, назовем MatrixACO-Global (префикс-Global). Схема алгоритма приведена на рис. 1, 2.

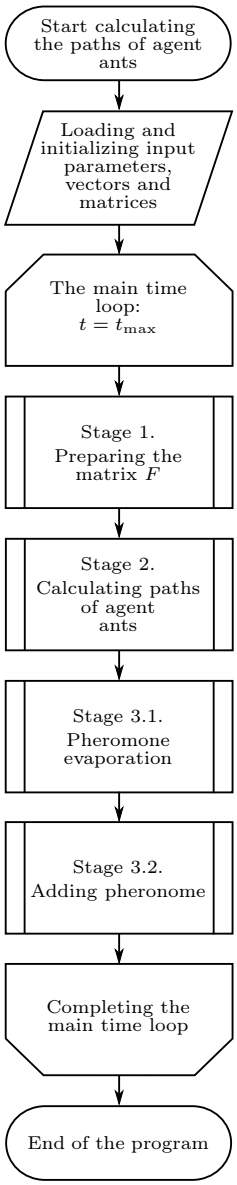

Рис. 1. Алгоритм матричной модификации ACO для выполнения на GPU

Fig. 1. ACO matrix modification for GPU execution

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

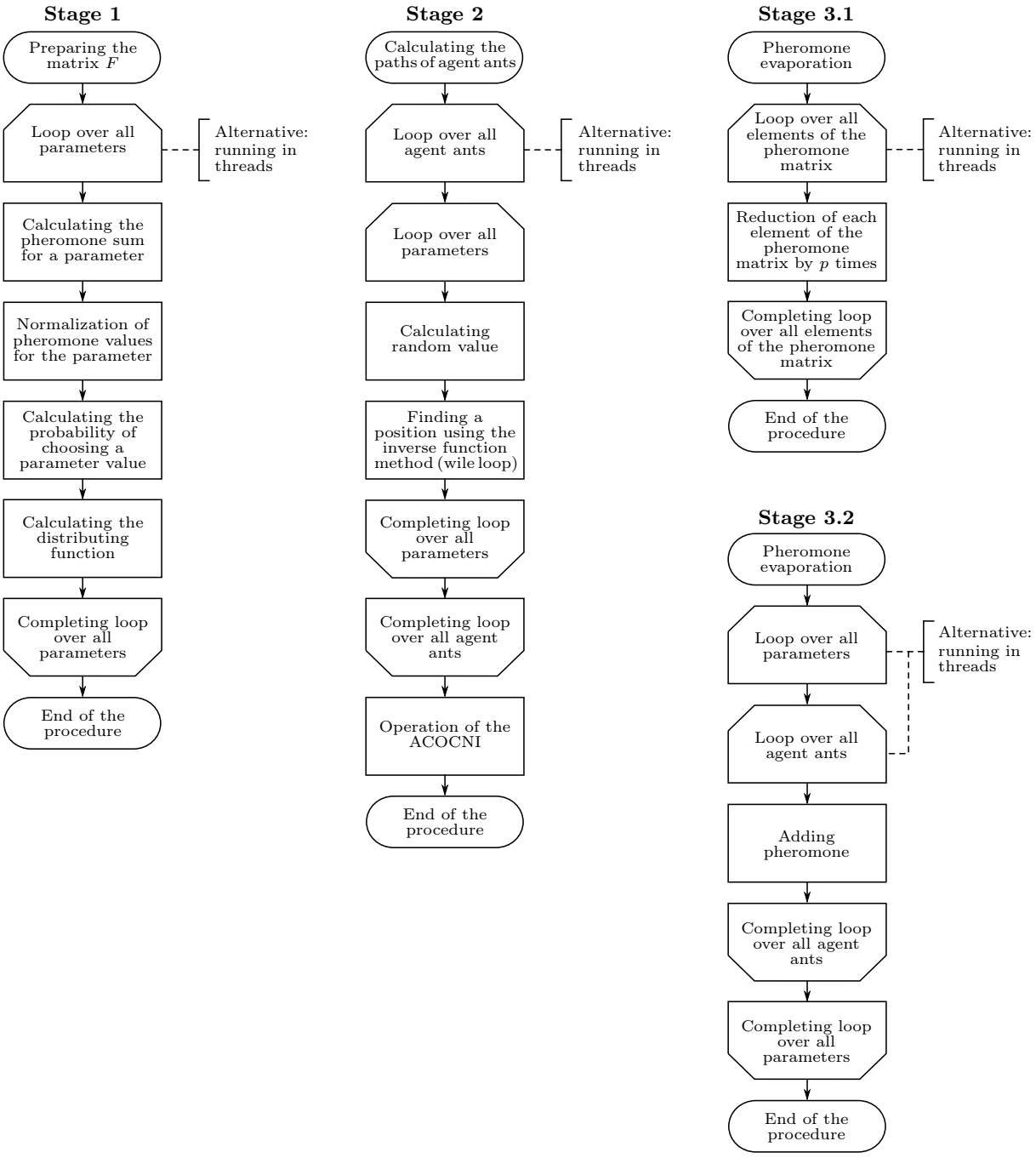

Рис. 2. Блок-схемы процедур Stage 1, Stage 2, Stage 3.1, Stage 3.2

Fig. 2. Flowcharts of procedures Stage 1, Stage 2, Stage 3.1, Stage 3.2

Дальнейшее развитие предложенного алгоритма связано с параллелизацией вычислений, выполняемых в циклах по отдельным слоям матриц T_{norm}, Z, Z', P, F, θ и T. В таком алгоритме в каждом потоке выполняются вычисления только для одного слоя, а номер блока определяет номер муравья-агента. Данный алгоритм (MatrixACO-Layer) имеет ограничение в 1024 слоя, обусловленное числом потоков в блоке.

Недостатком предложенных модификаций является необходимость выполнения одновременных многократных вычислений одинаковых значений матриц и векторов T_{norm}, Z, Z', P и F, а также обновление

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

матриц θ и T после нахождения вектора Y. Для оптимизации потребления вычислительных ресурсов целесообразно разделить алгоритм на 3 основные части:

1. Подготовка матрицы F, общей для всех муравьев-агентов из популяции.
2. Вычисление путей муравьев-агентов X и вектора Y.
3. Пересчет состояний матриц θ и T в зависимости от значений вектора Y и матрицы X.

Первый этап может быть выполнен параллельно, так как каждый параметр или слой параметра не зависит от других слоев, т.е. можно обеспечить параллелизм по столбцам матриц T_{norm}, Z, P и F. Если число столбцов меньше максимального количества потоков в блоке, то возможно определение номера столбца по переменной `threadIdx`. В противном случае множество столбцов распределяется по потокам и блокам, а номер столбца вычисляется как `nom_parametr = threadIdx.x + blockIdx.x * blockDim.x`. Входными данными являются матрицы θ и T, а выходной — матрица F. Эти матрицы передаются через глобальную память, а значения матриц T_{norm}, Z, P и нормировочных сумм хранятся в локальной памяти.

Второй этап может выполняться в двух версиях:

- Параллельно для каждого муравья-агента. В данном алгоритме осуществляется циклический поиск каждого значения параметра при помощи нахождения позиции s_i для каждого столбца матрицы F. После этого выполняется модификация ACOCCyN и ее взаимодействие с хэш-таблицей. По результатам вычислений каждый параллельный поток CUDA определяет одно значение вектора Y. Также в данной процедуре определяется оптимальный вектор X^* по результатам сравнения значений целевой функции. Данная операция требует атомарного исполнения и выполняется при помощи функции atomicCAS.
- Параллельно для каждого муравья-агента и для каждого параметра. В данном алгоритме каждый поток вычисляет только один параметр, индекс данного потока определяется переменной `threadIdx`, а переменная `blockIdx` определяет номер муравья-агента. Поскольку все матрицы передаются через общую или константную память, то применения разделяемой памяти не требуется. Вычисления организуются в рамках одного блока, где каждый поток обрабатывает один параметр. В данном алгоритме не требуется цикл по всем столбцам матрицы F, а каждый столбец обрабатывается в своем потоке. Так как поиск позиции s_i выполняется в каждом потоке параллельно и может требовать разного количества проверок условия, он не может полноценно называться SIMD вычислением, что может привести к гонкам и неопределенному поведению алгоритма. Поэтому по завершении поиска позиции s_i в потоке устанавливается барьерная блокировка (`__syncthreads`), которая приостанавливает выполнение программы до тех пор, пока все активные потоки в блоке не достигнут этой точки. Дальнейшие операции по выполнению действий модификации ACOCNI и взаимодействию с хэш-таблицей для каждого муравья-агента осуществляются параллельно всеми потоками. За счет симметричности алгоритма и данных и отсутствия атомарных функций (кроме вычисления оптимального решения), многократное повторение операций, таких как поиск значения в хэш-таблице, добавление значения в хэш-таблицу и вычисление значения целевой функции, не приводит к неопределенности поведения алгоритма и задержкам времени его выполнения.

На втором этапе входная матрица F, а также выходные данные (матрица выбранных агентами вершин X и вектор значений целевой функции Y) и указатель на хэш-таблицу передаются через глобальную память. Также алгоритму требуется матрица V, которая размещается либо в константной, либо в глобальной памяти. Для экономии памяти матрица R может быть представлена в виде скаляров, создаваемых при помощи генератора псевдослучайных чисел в моменты поиска позиции s_i. В локальной памяти потока также хранится вектор X', содержащий значения параметров (слоев), полученных из матрицы V. В случае нехватки локальной памяти векторы X' могут храниться в глобальной памяти в виде матрицы путей всех муравьев-агентов X' размером $K \times n$.

На третьем этапе на основе входных данных (матрицы X и вектора Y) выполняется обновление матриц θ и T посредством стандартных операций ACO — добавления и испарения феромона. Испарение феромона может выполняться одновременно во всех вершинах. Добавление феромона осуществляется при помощи матрицы X. Так как каждый муравей-агент обязательно должен выбрать ровно одну вершину из каждого слоя, то весь третий этап может выполняться параллельно, причем количество потоков определяется числом параметров. В данном случае, помимо счетчиков циклов, алгоритм не требует объявления дополнительных локальных переменных, вся работа может осуществляться с глобальными переменными. Аналогично первому этапу, на третьем этапе возможно выполнение всех операций в одном блоке, если

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

количество параметров меньше 1024; в противном случае вычисления распределяются по нескольким блокам.

Матричную модификацию ACO, разделенную на три этапа, будем называть базовой (MatrixACO-Basic). Для второго этапа стандартным (постфикс -Standart) является подход, в котором каждый блок выполняет вычисления для одного муравья-агента, а каждый поток в блоке — для отдельного параметра. В таком алгоритме нет необходимости выполнять циклы по всем параметрам. Альтернативный вариант второго этапа отличается от последовательного варианта тем, что алгоритм выполняется для каждого муравья-агента в отдельных потоках и блоках (алгоритмы в названии имеют постфикс -Agent). При этом возможны различные комбинации выполнения отдельных этапов алгоритмов.

Первый и третий этап в таком алгоритме выполняются параллельно для каждого параметра и могут быть объединены в одно обращение от CPU к GPU, реализованное в виде одной функции, исполняемой на GPU. В такой функции сначала выполняется третий этап с обновлением матриц θ и T , а затем следует первый этап, заканчивающийся вычислением матрицы F. Для работы такой функции необходимо задать начальные значения матрицы X и вектора Y. На этапе инициализации создаются “фиктивные” муравьи, которые не повлияют на начальные значения матриц θ и T. Алгоритм с объединением первого и третьего этапа называется MatrixACO-Optimal.

Для преодоления критичной проблемы неконсолидированного доступа к глобальной памяти, ведущей к катастрофическому падению пропускной способности, была реализована поддержка транспонированного представления ключевых матриц (постфикс -Transported). В классической для ACO форме хранения, где каждый параметр представлен последовательностью своих возможных значений, потоки, обрабатывающие разные параметры, обращаются к разрозненным участкам памяти. В представленном оптимальном графе с разделением на слои каждый поток обращается к структуре данных со смещением в 5 элементов. В новой схеме данные организуются так, что все потоки, работающие с одним и тем же параметром, обращаются к последовательным адресам. Это позволяет аппаратуре GPU объединять обращения в широкие транзакции, максимально используя пропускную способность шины памяти.

Кроме прямых, “нативных” алгоритмов, разработан алгоритм с учетом особенностей GPU CUDA. В предложенном алгоритме (MatrixACO-Optimization) процедура обновления феромонов, изначально являвшаяся узким местом из-за обилия атомарных операций, была полностью переработана с учетом warp-ориентированной архитектуры CUDA. Вместо того чтобы каждый поток атомарно добавлял феромон в глобальную память, была применена стратегия warp-редукции. Каждый warp из 32 потоков накапливает изменения феромона для всех значений своего параметра в регистрах, используя высокоскоростные инструкции `__shfl_down_sync()`. После завершения редукции только первый поток warp выполняет атомарное сложение итогового значения в глобальную память. Это сокращает количество конфликтующих атомарных операций на два порядка. Для случаев, когда количество параметров невелико, была разработана альтернативная версия добавления феромона, которая использует разделяемую память для редукции результатов внутри всего блока потоков, что еще больше уменьшает нагрузку на глобальную память. Взаимодействие с хэш-таблицей, предназначенной для кэширования уже вычисленных решений и избегания повторных дорогостоящих вызовов целевой функции, также было оптимизировано. Для уменьшения кластеризации вместо линейной стратегии разрешения коллизий использована квадратичная. Операции атомарного сравнения-обмена (atomicCAS) применяются теперь только на этапе первоначальной установки ключа в ячейку таблицы, что минимизирует конфликты между потоками, пытающимися записать различные решения. Сама проверка наличия значения в кэше выполняется неатомарно, что допустимо в рамках предложенной модели, где редкие повторные вычисления считаются приемлемыми, тогда как длительные блокировки полностью исключены. Ядро выбора пути муравьем-агентом было реструктурировано для максимального соответствия модели SIMT. Условные переходы и циклы с переменным числом итераций при поиске подходящего значения параметра с помощью метода обратной функции были минимизированы за счет развертывания циклов и использования предвычисленных бинарных деревьев решений. Это снижает эффект warp divergence, когда потоки внутри одного warp’а вынуждены выполнять разные инструкции, что ведет к их последовательной, а не параллельной работе. Для генерации псевдослучайных чисел, необходимых на этом этапе, каждый поток инициализирует собственный независимый генератор curandState, что исключает конфликты и обеспечивает высококачественную случайность.

4. Результаты исследований. Для проведения исследований разработано программное обеспечение на языке C++ [23], включающее следующие реализации матричной модификации ACO: версию для CPU (MatrixACO-CPU); варианты для выполнения на GPU с применением технологии CUDA и их опти-

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

мизированную версию с внутриблочной редукцией[1]. Для валидации предложенного метода, адаптированного для параметрических задач, проводилось тестирование на многомерных функциях (Растригина, Розенброка, Акли, сферических функциях, Гринванка, Захарова, Шейфеля, Леви, Михалевича–Викинского, Шаффера и др. [24]). Каждая переменная x_i такой функции трактовалась как независимый параметр с собственным диапазоном значений и точностью дискретизации, что позволило сформировать параметрический граф. Такой подход обеспечил возможность анализа масштабируемости метода в зависимости от количества параметров n, что представляет собой центральный вопрос при решении реальных параметрических задач. Оценка эффективности применения матричной модификации ACO для реализации на CUDA проводилась на основе измерений времени выполнения на различном оборудовании. Компиляция проводилась с помощью NVCC 12.6 и GCC 14.1.0 с оптимизацией `-O3`. Версия для CPU (MatrixACO-CPU) была скомпилирована под архитектуру x86-64.

Все измерения проводились на сервере с процессором Intel Xeon Gold 6130 (2.10 GHz, 16 cores, 2TB RAM) и GPU NVIDIA Tesla V100 16GB SXM2. Для исключения влияния фоновых процессов каждый эксперимент запускался на выделенном ядре CPU. Результаты представлены в виде оценки математического ожидания и доверительных интервалов (уровень доверия 0.99), рассчитанных по 200 запускам после выполнения пяти "холостых" итераций для разогрева кэша. В ходе исследования варьировались параметры алгоритма, влияющие на возможности параллельной обработки: количество входных параметров n для тестовой функции, размер поколения K, число итераций. Все результаты размещены в файлах в репозитории[2].

Время вычисления для различных тестовых функций отличается незначительно, а полученные результаты оказываются весьма близкими. Результаты приводятся для исследования тестовой функции Шаффера

$$f(x') = \frac{1}{2} - \frac{\sin^2(\sqrt{x'}) - 0.5}{1 + 0.001x'}, \quad x' = \sum_{i=1}^{n} x_i^2, \quad \forall\, i \;\; x_i \in [-10, 10],$$

для которой задана точность каждой переменной x_i на уровне 10^{-9} при различном количестве параметров n в условиях задачи минимизации. Параметры алгоритма были взяты следующими: $\lambda_1 = \lambda_2 = \lambda_3 = 1$, $Q = 1, \quad \rho = 0.999, \quad K = N = 500$. При декомпозиции одного параметра реализован 21 слой с максимальным числом значений этого параметра, равным $m = 5$. При этом рассматривались задачи с числом параметров, возрастающим по степеням двойки от $n = 2$ (классическая тестовая функция) до $n = 2^{16} = 65536$, а именно: $2, 4, 8, 16, 32, 64, 128, \ldots, 65536$. На практике двухкритериальные задачи малой размерности встречаются нечасто, и для их решения обычно применяются более простые алгоритмы. Благодаря развитию высокопроизводительных вычислительных систем и технологий параллельной обработки инструкций все больший интерес вызывают задачи высокой размерности. Например, в случае 16 параметров общее число слоев уже достигает 336. Оптимизация задач сверхбольшой размерности является актуальной и выполнимой, но требует дальнейшего исследования [25].

В нашей работе выбор верхней границы в виде степени двойки обусловлен не ограничениями алгоритма, а удобством организации вычислительных блоков и потоков на GPU, а также достаточностью для демонстрации как масштабируемости, так и влияния ограничений, накладываемых различными типами памяти. Алгоритм не имеет фундаментальных ограничений на количество параметров, однако при дальнейшем увеличении размерности необходимо учитывать исчерпание памяти GPU и возможный переход к многоузловым конфигурациям.

В табл. 1 приведены оценки математического ожидания и доверительные интервалы времени выполнения 500 итераций метода муравьиных колоний для различных матричных модификаций, реализованных с применением технологии CUDA. Модификация MatrixACO-Layer-Global, в которой матричный алгоритм работает на GPU без переключения на CPU, не эффективна не только по сравнению с другими GPU-реализациями, но и по сравнению с CPU-версией. Во многом это связано с взаимодействием с глобальной памятью, поскольку модификация MatrixACO-Layer-Local показывает существенное (более чем в 35 раз) ускорение. Но для графа с 2688 слоями объем локальной памяти, выделяемый одному потоку, оказывается недостаточным для хранения всех переменных. Модификация MatrixACO-Global, в которой каждый поток выполняет действия для одного параметра (слоя), а каждый блок — для одного

[1] https://github.com/kalengul/ACO_SIMD/tree/main/CUDA C++ Optimal

[2] https://github.com/kalengul/ACO_SIMD/tree/main/TRACE_ITOG_EXCEL

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

Таблица 1. Оценка математического ожидания и доверительный интервал времени выполнения (мс) на один слой для различных модификаций алгоритма на GPU NVIDIA Tesla V100 16GB SXM2 (200 прогонов, 500 итераций, 500 муравьев-агентов)

Table 1. Mathematical expectation estimation and confidence interval of execution time (ms) per one layer for different algorithm modifications on NVIDIA Tesla V100 16GB SXM2 GPU (200 runs, 500 iterations, 500 ant agents)

Количество параметров Number of parameters Количество слоев графа Number of graph layers		2 42	4 84	8 168	16 336	32 672	64 1344	128 2688
MatrixACO-CPU	ME	84.44	63.23	51.69	47.46	44.15	42.78	42.57
	CI	±13.03	±6.90	±3.99	±2.60	±1.70	±1.17	±0.82
MatrixACO-Global	ME	49.13	29.46	12.09	11.78			
	CI	±1.05	±0.47	±3.94	±0.83			
MatrixACO-Layer-Global	ME	83.66	86.12	87.07	88.02	89.76	91.66	92.95
	CI	±0.66	±1.99	±1.91	±1.20	±0.65	±6.54	±3.66
MatrixACO-Layer-Local	ME		2.14	2.16	2.49	2.57	2.60	
	CI		±0.01	±0.02	±0.01	±0.00	±0.00	
MatrixACO-Basic-Standart	ME	3.46	2.21	1.58	1.52			
	CI	±0.21	±0.12	±0.08	±0.02			
MatrixACO-Basic-Agent	ME	3.64	2.36	1.66	1.47	1.44	1.34	1.14
	CI	±0.22	±0.25	±0.16	±0.02	±0.16	±0.01	±0.12
MatrixACO-Optimal-Standart	ME	3.43	2.18	1.58	1.52			
	CI	±0.18	±0.14	±0.09	±0.04			
MatrixACO-Optimal-Agent	ME	3.60	2.31	1.66	1.46	1.44	1.36	1.14
	CI	±0.17	±0.20	±0.18	±0.04	±0.17	±0.10	±0.14
MatrixACO-Optimization	ME	2.05	1.12	0.58	0.31	0.18	0.16	0.13
	CI	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00
MatrixACO-Transported	ME	1.93	1.08	0.67	0.47	0.36	0.33	0.31
	CI	±0.01	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00

муравья-агента, работает быстрее, чем модификация MatrixACO-Layer-Global. Однако она все еще остается неэффективной и ограничена 336 слоями из-за существования предела на количество потоков в одном блоке CUDA на GPU.

Матричная модификация с разделением на этапы и возвратом управления на CPU после каждого из них показывает существенно превосходящую эффективность (с ускорением более 25 раз) по сравнению с реализацией, полностью выполняемой на GPU. Аналогичная эффективность достигается и для алгоритма, в котором первый и третий этапы объединены в один. Следует отметить, что основное время работы алгоритма занимает второй этап, включающий генерацию случайных чисел и циклический поиск позиции с неопределенным количеством итераций. Предложенные модификации демонстрируют значительно более высокую эффективность по сравнению с CPU-реализацией, обеспечивая ускорение более чем в 30 раз. Оптимизированный алгоритм показывает хорошую эффективность при достаточно большом количестве параметров. С ростом размерности задачи время работы, нормированное на один слой параметров, составляет 0.13 мс. При этом в оптимизированной под GPU реализации MatrixACO-Optimization объединение второго и третьего этапов в один дает прирост производительности не более чем на 10%, причем этот эффект сильно зависит от размерности параметрического графа.

Хотя транспонированная организация данных изначально рассматривалась как способ устранения проблемы warp divergence, на практике она показывает значительно более низкую производительность по сравнению с классическими модификациями. Существенное увеличение времени выполнения этапа добавления феромона обусловлено тем, что соседние потоки GPU обращаются к далеко отстоящим друг от друга и не расположенным подряд участкам памяти параметрического графа. При работе с транспонированными матрицами одновременное чтение пяти значений для каждого параметра приводит к промахам

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

в кэше L1/L2 для больших графов и конфликтам внутри банков в shared memory. Для такого алгоритма характерно время выполнения 0.3 мс на один слой параметров.

Для хранения матриц рассматривались варианты их размещения в константной, локальной и глобальной памяти (табл. 2). Хранение матрицы параметров в константной памяти ускоряет работу модификации до 10% по сравнению с хранением в локальной памяти и демонстрирует ускорение от 10% до 25% по сравнению с хранением в глобальной памяти. Хранение в локальной памяти потока CUDA информации муравья-агента позволяет ускорить алгоритм от 5% до 40% при увеличении размерности. При хранении данных только в глобальной памяти GPU алгоритм стабильно работает даже при существенном увеличении размерности.

Рекомендуется использование алгоритмов с автоматическим определением размерности и выбором типа памяти для его работы в зависимости от количества значений в графе. Для оптимальной модификации использование константной памяти задавалось автоматически по результатам проверки размерности матрицы V, а локальная и shared память блока применялась для редукции и эффективного распараллеливания, причем объемы последних задавались статически. В примерах исследовались задачи с числом параметров до 65536, при этом точность определения каждого параметра составляла 10^{-9}. Применение константной памяти позволяет получить ускорение работы модификации более чем в 45 раз по сравнению с выполнением на CPU. Полученные результаты демонстрируют применимость предложенных модификаций ACO для решения задач конденсации графов больших размерностей [25]. В табл. 3 приведены исследования для модификации MatrixACO-Optimal-Agent на различном оборудовании. Сервер DELL POWEREDGE C4140 и платформа NVIDIA Jetson Orin обладают относительно слабыми CPU и особой структурой взаимодействия компонентов, что обеспечивает высокую эффективность использования GPU

Таблица 2. Оценка математического ожидания и доверительный интервал времени выполнения (мс) на один слой на GPU NVIDIA Tesla V100 16GB SXM2 при различных типах памяти (200 прогонов, 500 итераций, 500 муравьев-агентов)

Table 2. Mathematical expectation estimation and confidence interval of execution time (ms) per one layer on NVIDIA Tesla V100 16GB SXM2 GPU at different memory types (200 runs, 500 iterations, 500 ant agents)

Количество параметров Number of parameters		2	4	8	16	32	64	128	256
Количество слоев графа Number of graph layers		42	84	168	336	672	1344	2688	5 376
MatrixACO-Optimal-Agent-Local	ME	3.34	2.12	1.51	1.30	1.34	1.24	Недостаточно памяти	
	CI	±0.11	±0.04	±0.05	±0.00	±0.03	±0.11	Not enough memory	
MatrixACO-Optimal-Agent	ME	3.60	2.31	1.66	1.46	1.44	1.36	1.14	1.09
	CI	±0.17	±0.20	±0.18	±0.04	±0.17	±0.10	±0.14	±0.02
MatrixACO-Optimal-Agent-Global	ME	3.72	2.52	1.91	1.75	1.69	1.55	1.47	1.44
	CI	±0.08	±0.02	±0.01	±0.02	±0.05	±0.01	±0.01	±0.11
MatrixACO-Optimization	ME	2.05	1.12	0.58	0.31	0.18	0.16	0.13	0.13
	CI	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00

Количество параметров Number of parameters		512	1 024	2 048	4 096	8 192	16 384	32 768	65 536
Количество слоев графа Number of graph layers		10 752	21 504	43 008	86 016	172 032	344 064	688 128	1 376 256
MatrixACO-Optimal-Agent-Local	ME	Недостаточно памяти							
	CI	Not enough memory							
MatrixACO-Optimal-Agent	ME	1.03	Недостаточно памяти						
	CI	±0.09	Not enough memory						
MatrixACO-Optimal-Agent-Global	ME	1.42	1.42	1.42	1.44	1.43	1.51	1.43	1.59
	CI	±0.03	±0.16	±0.16	±0.10	±0.03	±0.19	±0.07	±0.21
MatrixACO-Optimization	ME	0.11	0.10	0.10	0.12	0.12	0.12	0.12	0.12
	CI	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00	±0.00

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

Таблица 3. Оценка математического ожидания и доверительный интервал времени выполнения (мс) на один слой на различном оборудовании (200 прогонов, 500 итераций, 500 муравьев-агентов)

Table 3. Mathematical expectation estimation and confidence interval of execution time (ms) per one layer on different equipment (200 runs, 500 iterations, 500 ant agents)

Количество параметров Number of parameters		2	4	8	16	32	64	128
Количество слоев графа Number of graph layers		42	84	168	336	672	1344	2688
DELL PowerEdge C4140 Intel Xeon Gold 6130 2TB RAM+	ME	3.60	2.31	1.66	1.46	1.44	1.36	1.14
NVIDIA Tesla V100 16GB SXM2	CI	±0.17	±0.20	±0.18	±0.04	±0.17	±0.10	±0.14
NVIDIA Jetson Orin	ME	11.32	7.29	4.95	4.23	4.33	4.07	3.89
	CI	±0.04	±0.16	±0.03	±0.03	±0.60	±0.23	±0.24
AMD Ryzen 5-7500F 6 Core Processor 3.70GHz 8GB RAM+	ME	12.07	7.57	5.30	4.27	3.90	3.53	3.31
NVIDIA GeForce RTX 4060 Ti	CI	±0.35	±0.43	±0.52	±0.59	±0.42	±0.03	±0.12
13th Gen Intel Core i5-13600K 3.50 GHz 64GB RAM+	ME	17.54	11.20	7.90	6.55	5.96	5.32	5.07
NVIDIA GeForce RTX 3060	CI	±0.10	±0.13	±0.17	±0.21	±0.24	±0.24	±0.13
12th Gen Intel Core i5-12450H 2.00 GHz 16Gb RAM+	ME	16.14	10.60	7.61	6.14	5.62	5.35	5.07
NVIDIA GeForce RTX 3060 Laptop GPU	CI	±0.12	±0.23	±0.08	±0.15	±0.06	±0.08	±0.03
9th Gen Intel Core i5-9400F 2.90 GHz 16Gb RAM+	ME	28.30	18.63	15.64	13.62	13.46	12.78	11.97
NVIDIA GeForce GTX 1050 Ti	CI	±0.57	±0.86	±1.23	±1.14	±2.76	±2.45	±1.21
8th Gen Intel Core i5-8300H 2.30 GHz 8Gb RAM+	ME	26.26	18.24	14.15	12.30	11.40	10.80	10.54
NVIDIA GeForce GTX 1050 Ti	CI	±0.57	±0.21	±0.27	±0.09	±0.14	±0.09	±0.03

на этих платформах. Для персональных систем (ноутбуков и компьютеров на базе процессоров Intel и AMD) ускорение модификации при применении GPU варьируется в пределах от 2 до 6.

Согласно результатам исследований, предложенные модификации метода муравьиных колоний минимально зависят от типа CPU и в значительно большей степени — от количества ядер CUDA, на которых могут выполняться параллельные вычисления. Применение алгоритма оказывается особенно эффективным при большом количестве параметров задачи или слоев параметрического графа, так как с увеличением размерности время, затрачиваемое на обработку одного слоя, уменьшается.

В табл. 4 представлены результаты исследования при варьировании количества муравьев-агентов на одной итерации. Время, затрачиваемое на вычисление решения (500 итераций) на одного муравья-агента, уменьшается с увеличением числа муравьев-агентов на итерацию (столбец $\sum/K$ в табл. 4). Время выполнения первого этапа, на котором вычисляются значения матрицы F, не зависит от количества муравьев-агентов и в целом несущественно по сравнению со временем выполнения второго и третьего этапов. Время третьего этапа линейно зависит от количества муравьев-агентов из-за внутреннего цикла.

Второй этап выполняется параллельно в отдельных потоках CUDA для каждого муравья-агента. Ограничение на количество потоков в одном блоке установлено в объеме 512 потоков, поэтому при числе муравьев-агентов до 500 каждый агент обрабатывается отдельным потоком. Это вызывает рост времени выполнения второго этапа из-за необходимости синхронизации между потоками. При дальнейшем увеличении количества муравьев-агентов время второго этапа не увеличивается, поскольку новые агенты выполняются в дополнительных блоках. Отдельно следует отметить, что объединение первого и третьего этапов эффективнее, чем их раздельное выполнение, но из-за несущественности времени первого этапа по сравнению с общим временем работы алгоритма, выигрыш от такого объединения незначителен. Эффект от объединения первого и третьего этапов наблюдается также и в оптимальной модификации алгоритма. При увеличении количества итераций N зависимость остается линейной.

На третьем этапе выполняются две операции над параметрическим графом: испарение феромона (уменьшение значений феромона в фиксированное число раз) на всех вершинах параметрического графа и добавление феромона (в зависимости от значения целевой функции) на все посещенные муравьем-агентом вершины. В случае применения модификации MMAS также проверяется выход за максимальную и минимальную границу количества феромона в вершинах. При добавлении феромона возможно применение стратегии элитизма, когда исследуется только один самый лучший маршрут. Элитизм выполняется на

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

Таблица 4. Оценка математического ожидания и доверительный интервал времени выполнения (мс) на один слой на GPU NVIDIA Tesla V100 16GB SXM2 при варьировании количества муравьев-агентов для графа с 16 параметрами и 336 слоями (200 прогонов, 500 итераций)

Table 4. Mathematical expectation estimation and confidence interval of execution time (ms) per one layer on NVIDIA Tesla V100 GPU 16GB SXM2 when varying the number of ant agents for a graph with 16 parameters and 336 layers (200 runs, 500 iterations)

Количество муравьев-агентов K Number of ants-agents K		MatrixACO-Basic-Agent-Local					MatrixACO-Optimal-Agent-Local			
		$T1$	$T2$	$T3$	$\sum$	$\sum/K$	$T1+T3$	$T2$	$\sum$	$\sum/K$
100	ME	3.68	106.70	18.55	128.93	1.29	19.85	106.38	126.23	1.26
	CI	±0.15	±0.12	±0.01			±0.17	±0.14		
200	ME	3.50	146.34	34.73	184.57	0.92	36.08	145.54	181.62	0.91
	CI	±0.12	±0.11	±0.01			±0.14	±0.11		
300	ME	3.62	187.85	50.83	242.29	0.81	52.15	186.97	239.12	0.80
	CI	±0.11	±0.09	±0.00			±0.11	±0.09		
400	ME	3.51	232.90	66.82	303.22	0.76	68.34	232.17	300.51	0.75
	CI	±0.12	±0.11	±0.02			±0.13	±0.08		
500	ME	3.67	**270.60**	83.15	357.42	0.71	84.60	**270.26**	354.86	0.71
	CI	±0.08	±0.04	±0.03			±0.05	±0.10		
1000	ME	3.55	274.70	163.47	441.72	0.44	164.78	274.22	439.00	0.44
	CI	±0.17	±0.15	±0.02			±0.17	±0.05		
1500	ME	3.63	276.98	244.11	524.72	0.35	245.27	276.56	521.83	0.35
	CI	±0.09	±0.09	±0.02			±0.11	±0.08		
2000	ME	3.60	279.28	324.43	607.30	0.30	325.39	278.87	604.26	0.30
	CI	±0.03	±0.04	±0.03			±0.02	±0.01		
2500	ME	3.66	280.27	410.21	694.14	0.28	411.05	279.85	690.90	0.28
	CI	±0.07	±0.05	±0.05			±0.03	±0.01		
3000	ME	3.73	280.47	489.09	773.30	0.26	490.52	280.02	770.54	0.26
	CI	±0.08	±0.07	±0.07			±0.08	±0.01		

этапе поиска путей муравьями-агентами и может быть реализован на GPU как выбор наилучшего решения, найденного в одном блоке GPU CUDA. Хотя стратегия элитизма показала высокую эффективность, в данной работе рассматриваются алгоритмы, в которых все муравьи-агенты участвуют в обновлении параметрического графа.

Прямое количественное сравнение абсолютной скорости работы с известными GPU-реализациями ACO для задачи TSP некорректно в силу принципиального различия решаемых задач (в данной работе решалась параметрическая задача в отличие от комбинаторных задач), структур данных и метрик (обычно исследуется время на итерацию TSP, а не время обработки одного слоя параметрического графа). Основным сравнимым результатом данной работы является не абсолютное быстродействие, а демонстрация эффективности предложенной специализированной схемы параллелизма для нового класса задач. Прямое применение TSP-ориентированных GPU-подходов MatrixACO-Layer-Global неэффективно для параметрических задач, в то время как наши модификации, использующие независимость параметров, дают многократное ускорение. Это подтверждает тезис о необходимости разработки специализированных, а не универсальных, параллельных реализаций ACO. Теоретическая вычислительная сложность предложенного алгоритма для первого этапа составляет $O(n\cdot m)$, для второго этапа равна $O(K\cdot n\cdot m)$, а для третьего этапа — $O(n\cdot m+K\cdot n)=O(n\cdot(m+K))$. Общая вычислительная сложность на одну итерацию последовательного матричного ACO составляет $O_{\text{алг}}=O(n\cdot m)+O(K\cdot n\cdot m)+O(n\cdot(m+K))+K\cdot O_{\text{целевой функции}}(1)\approx O(n\cdot(K\cdot m+K+m))$. Учитывая, что для оптимального параметрического графа $m=5$, а K может быть порядка сотен или тысяч муравьев-агентов на одной итерации, доминирующим слагаемым является $O(n\cdot K\cdot m)$, что можно упростить до $O(n\cdot K)$. Таким образом, принципиальное отличие в вычислительной сложности заключается в зависимости от размерности задачи: $O(n\cdot K)$ для параметрического ACO против $O(n^2\cdot K)$ для классического ACO. Это объясняется независимостью параметров: агенту не нужно

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

перебирать уже пройденные вершины (список запретов), а вероятность выбора для каждого параметра вычисляется независимо от других, что снижает степень полинома с квадратичной до линейной. Данный аспект подтверждает обоснованность архитектурных решений, принятых при разработке параллельных GPU-реализаций, и объясняет наблюдаемое на практике сверхлинейное ускорение при росте размерности задачи. Сравнение предложенных модификаций с другими метаэвристическими методами параметрической оптимизации с точки зрения точности найденного решения приведено в [20], но в работе [20] получено только 6-кратное ускорение алгоритма на GPU. Предложенные модификации алгоритма, оптимизированные под особенности архитектуры CUDA GPU, обеспечивают ускорение работы алгоритма более чем в 100 раз.

Представленные реализации алгоритмов, кроме оптимизированной версии и версии с транспонированными матрицами, являются memory bound реализациями, производительность которых ограничена скоростью доступа к памяти, а не вычислительными мощностями. Профилирование с помощью Nsight Systems выявило значительное количество операций `cudaMemcpy` (копирование данных между хостом и устройством) и ядер, работающих с глобальной памятью GPU. Узким местом предложенного алгоритма является необходимость частого обращения каждого муравья-агента к хэш-таблице. Помимо атомарного добавления записи (а для представленных задач все муравьи-агенты находили уникальное решение) в хэш-таблицу, после перемещения муравья-агента атомарно проверяется эффективность найденного им решения. В оптимизированном под CUDA алгоритме учитываются размеры warp'а и буферов под конкретную архитектуру GPU. Данный подход позволил обеспечить существенный (более чем в 10 раз) прирост к скорости выполнения алгоритма. Профиль NVIDIA Nsight Systems демонстрирует стабильную 100% занятость (Occupancy) мультипроцессоров, а временные характеристики выполнения ядер остаются чрезвычайно стабильными (отклонения между итерациями составляют менее 2%). Оптимальная реализация активно использует различные уровни памяти GPU. Функция добавления феромона и процедура нормировки матриц (выполняется на GPU в виде объединенного этапа "$1+3$") задействует 100 КБ статической разделяемой памяти (shared memory) на блок, что близко к аппаратному пределу. Показатель попадания в кэш (Hit Rate) составляет 99.8047%, а пропускная способность глобальной памяти поддерживается на уровне 4.0 ± 0.1 ГБ/с.

Постоянное взаимодействие с памятью GPU и относительно скромный объем параллельных матричных вычислений в ядре алгоритма обуславливают необходимость тщательного сравнения эффективности GPU-версии матричного метода муравьиных колоний с предельно оптимизированной реализацией для центрального процессора. Такая CPU-версия, использующая двухуровневый параллелизм (межъядерный через директивы OpenMP и векторный посредством инструкций SIMD набора AVX2), представляет собой серьезного конкурента, особенно для задач малой и средней размерности. Для параметрического графа с 16 параметрами (336 слоев) GPU-реализация (Tesla V100) оказывается быстрее оптимизированной CPU-версии (Xeon Gold 6130) примерно в 3.4 раза (табл. 5). Однако с ростом размерности задачи преимущество GPU становится все более выраженным. При увеличении числа параметров до 128 достигается ускорение в 4.4 раза, а для задачи с 512 параметрами GPU демонстрирует устойчивое ускорение в 4.5 раза. Этот рост объясняется лучшей способностью архитектуры GPU масштабироваться при обработке тысяч независимых параметров благодаря массовому параллелизму, в то время как производительность CPU-версии, ограниченная числом физических ядер и пропускной способностью оперативной памяти, улучшается менее значительно.

Для комплексной оценки практической ценности метода критически важным является анализ его энергоэффективности — количества решений, получаемых на единицу потребленной энергии. Рассмотрим серверную конфигурацию с процессором Intel Xeon Gold 6130 (оценочная средняя мощность под нагрузкой $P_{\mathrm{CPU}} \approx 105$ Вт) и ускорителем NVIDIA Tesla V100 SXM2 ($P_{\mathrm{GPU}} \approx 265$ Вт). Энергия E, затрачиваемая на решение задачи, может быть оценена как произведение средней мощности устройства на время его работы: $E = P \cdot T$. Для задачи с 16 параметрами (336 слоев графа) абсолютное время выполнения составляет $T_{\mathrm{CPU}} \approx 0.31$ с для CPU и $T_{\mathrm{GPU}} \approx 1.06$ с для GPU (табл. 5). В таком случае находим, что $E_{\mathrm{CPU}} \approx 37.42$ Дж, а $E_{\mathrm{GPU}} \approx 27.54$ Дж. Отсюда получаем, что энергоэффективность составляет $n_{\mathrm{CPU}} \approx 0.027$ Дж$^{-1}$ и $n_{\mathrm{GPU}} \approx 0.036$ Дж$^{-1}$. Это означает, что уже для задачи средней размерности GPU-реализация примерно на 35% энергоэффективнее CPU-версии, несмотря на почти вдвое большее пиковое энергопотребление ускорителя. Преимущество GPU существенно возрастает с увеличением размерности решаемой проблемы. Для конфигурации с 512 параметрами (10752 слоя) время выполнения всех итераций равно $T_{\mathrm{CPU}} \approx 0.49$ с и $T_{\mathrm{GPU}} \approx 0.11$ с. Энергопотребление при этом составляет $E_{\mathrm{CPU}} \approx 559.08$ Дж и $E_{\mathrm{GPU}} \approx 312.31$ Дж.

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

Таблица 5. Оценка математического ожидания времени выполнения (мс) на один слой в сравнении со временем оптимальной модификации OpenMP (200 прогонов, 500 итераций по 500 муравьев-агентов)

Table 5. Mathematical expectation estimation of execution time (ms) per one layer in comparison with time of the optimal modification OpenMP (200 runs, 500 iterations with 500 ant agents)

Количество параметров Number of parameters		2	4	8	16	32	64	128	256	512
Количество слоев графа Number of graph layers		42	84	168	336	672	1344	2688	5376	10752
Intel Xeon Gold 6130 2TB RAM +	CUDA	2.05	1.12	0.58	0.31	0.18	0.16	0.13	0.12	0.11
NVIDIA Tesla V100 16GB SXM2	OpenMP	4.42	2.06	1.50	1.06	0.85	0.62	0.57	0.54	0.49
Intel Core i5-12450H 2.00 GHz 16GB RAM +	CUDA	2.43	1.39	0.85	0.58	0.35	0.29	0.28	0.37	0.26
NVIDIA GeForce RTX 3060 Laptop GPU	OpenMP+AVX2	2.88	1.79	1.22	0.96	0.80	0.60	0.58	0.48	0.47

Таблица 6. Теоретическая оценка энергетической эффективности выполнения оптимальной реализации на CPU и GPU для десктопной и серверной конфигураций

Table 6. Theoretical assessment of the energy efficiency of performing the optimal implementation on CPU and GPU for desktop and server configurations

Количество параметров Number of parameters		2	4	8	16	32	64	128	256	512
Количество слоев графа Number of graph layers		42	84	168	336	672	1344	2688	5376	10752
Intel Xeon Gold 6130 2TB RAM + NVIDIA Tesla V100 16GB SXM2	Ускорение (T_{CPU}/T_{GPU})	2.15	1.84	2.57	3.43	4.68	3.90	4.35	4.27	4.52
	E_{GPU}, Дж	22.83	24.94	26.02	27.54	32.45	56.97	94.05	180.04	312.31
	E_{CPU}, Дж	19.48	18.16	26.50	37.42	60.17	88.04	162.00	304.65	559.08
	n_{GPU}, Дж$^{-1}$	0.044	0.040	0.038	0.036	0.031	0.018	0.011	0.006	0.003
	n_{CPU}, Дж$^{-1}$	0.051	0.055	0.038	0.027	0.017	0.011	0.006	0.003	0.002
	n_{GPU}/n_{CPU}, %	0.85	0.73	1.02	**1.36**	**1.85**	**1.55**	**1.72**	**1.69**	**1.79**
	EDP_{GPU}, Дж·с	1.97	2.35	2.55	2.86	3.97	12.25	33.38	122.32	368.07
	EDP_{CPU}, Дж·с	3.61	3.14	6.69	13.34	34.48	73.83	249.95	883.94	2976.90
	EDP_{CPU}/EDP_{GPU}	**1.84**	**1.34**	**2.62**	**4.66**	**8.68**	**6.03**	**7.49**	**7.23**	**8.09**
Intel Core i5-12450H 2.00 GHz 16GB RAM + NVIDIA GeForce RTX 3060 Laptop GPU	Ускорение (T_{CPU}/T_{GPU})	1.19	1.28	1.44	1.66	2.30	2.05	2.02	1.31	1.74
	E_{GPU}, Дж	8.16	9.36	11.39	15.55	18.70	31.46	61.23	158.54	223.82
	E_{CPU}, Дж	4.60	5.70	7.79	12.26	20.44	30.60	58.76	98.80	185.47
	n_{GPU}, Дж$^{-1}$	0.123	0.107	0.088	0.064	0.053	0.032	0.016	0.006	0.004
	n_{CPU}, Дж$^{-1}$	0.217	0.175	0.128	0.082	0.049	0.033	0.017	0.010	0.005
	n_{GPU}/n_{CPU}, %	0.56	0.61	0.68	0.79	1.09	0.97	0.96	0.62	0.83
	EDP_{GPU}, Дж·с	0.83	1.10	1.62	3.02	4.37	12.37	46.86	314.20	626.18
	EDP_{CPU}, Дж·с	0.56	0.86	1.60	3.96	11.00	24.64	90.87	256.87	905.27
	EDP_{CPU}/EDP_{GPU}	0.67	0.78	0.98	**1.31**	**2.52**	**1.99**	**1.94**	0.82	**1.45**

Соответственно, энергоэффективность GPU-версии ($n_{GPU} \approx 0.0032$ Дж$^{-1}$) становится на 76% выше, чем у CPU ($n_{CPU} \approx 0.0018$ Дж$^{-1}$). Еще более показательной является метрика Energy-Delay Product (EDP), интегрирующая оба ключевых фактора производительности и энергоэффективности: $EDP = E \cdot T$. Для задачи с 16 параметрами $EDP_{CPU} \approx 13.34$ Дж $\cdot$ с, а $EDP_{GPU} \approx 2.86$ Дж $\cdot$ с, что демонстрирует преимущество GPU в 4.6 раза. Для случая 512 параметров это преимущество увеличивается до 8.1 раза ($EDP_{CPU} \approx 2976.90$ Дж $\cdot$ с против $EDP_{GPU} \approx 368.07$ Дж $\cdot$ с). Данная динамика объясняется фундаментальными архитектурными различиями. Процессор, даже использующий многопоточность и векторные инструкции, достигает предела масштабируемости из-за ограниченного числа ядер и пропускной способности подсистемы памяти. В то же время массивно-параллельная архитектура GPU позволяет эффективно скрывать задержки доступа к высокопропускной памяти (HBM2 у V100) за счет выполнения десятков тысяч легковесных потоков. В результате относительное ускорение $S = T_{CPU}/T_{GPU}$ растет с увеличением

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

размерности задачи (с 3.4 до 4.5 раз для рассматриваемого диапазона), что напрямую ведет к прогрессирующему улучшению энергетических показателей. Ускоритель Tesla V100 демонстрирует стабильное 4–4.5-кратное ускорение и преимущество в энергоэффективности на 54–87% по сравнению с 32-ядерным процессором Xeon, а комплексное преимущество по EDP на GPU лучше CPU в 4.6–8.8 раз (табл. 6). На мобильной платформе Intel Core i5-12450H 2.00 GHz (16 GB RAM) +NVIDIA GeForce RTX 3060 Laptop GPU, RTX 3060 фактор ускорения, обеспечиваемый GPU, примерно равен 2. Это дает практическую пользу, однако не является принципиальным прорывом. По показателям энергоэффективности конфигурации оказываются сопоставимы с небольшим преимуществом CPU на малых задачах. Ключевое преимущество мобильного GPU заключается в 2–2.5 раза лучшем показателе EDP для задач от 32 параметров, что свидетельствует об оптимальном балансе между скоростью выполнения и энергозатратами. Метрики стабилизируются после $n = 672$, что указывает на достижение предела эффективного использования ресурсов данного GPU для рассматриваемой задачи. На NVIDIA Tesla V100 все метрики улучшаются с ростом n, что подтверждает отличную масштабируемость алгоритма на GPU.

5. Заключение. Проведенное исследование доказало высокую эффективность разработанных матричных модификаций метода муравьиных колоний для решения параметрических задач оптимизации на GPU с архитектурой CUDA. Использование матричной формы представления графа значительно упрощает реализацию параллельных версий ACO, позволяя достичь высоких показателей масштабируемости и эффективности при обработке больших объемов данных. Ключевым научным результатом стало создание специализированных схем параллелизма, которые используют фундаментальное свойство параметрических задач — независимость выбора значений параметров, что кардинально отличает предложенный подход от традиционных GPU-реализаций ACO, ориентированных на комбинаторные задачи типа TSP. Экспериментальные результаты, полученные на GPU Tesla V100, показали, что оптимизированная модификация MatrixACO-Optimization обеспечивает устойчивое ускорение в 30–45 раз по сравнению с CPU-реализацией при решении задач высокой размерности и более 300 раз по сравнению с однопоточной матричной реализацией. Особенно значимыми являются показатели энергоэффективности: GPU-версия демонстрирует на 76% более высокую энергоэффективность и в 8 раз лучший показатель Energy-Delay Product для задачи с 512 параметрами. Важным выводом работы является доказанная масштабируемость алгоритма. Действительно, с ростом размерности задачи все ключевые метрики продолжают улучшаться на серверных GPU, в то время как производительность CPU-версии ограничивается количеством ядер и пропускной способностью памяти. Полученные результаты убедительно свидетельствуют, что прямое использование “TSP-ориентированных” GPU-подходов для параметрических задач неэффективно, необходима разработка специализированных, а не универсальных, параллельных реализаций метаэвристических алгоритмов оптимизации. Предложенные модификации открывают перспективы для решения сверхбольших параметрических задач в таких областях, как машинное обучение, имитационное моделирование и анализ данных, где требуется оптимизация десятков тысяч параметров. Полученные результаты были применены при создании параметрического оптимизатора для выявления оптимальных значений параметров модели SARIMAX в задаче анализа временных рядов показателей пассажиро- и грузоперевозок авиакомпаниями Российской Федерации. Исследовалось 25 показателей, отражающих как состояние отрасли авиапассажирских перевозок в целом, так и деятельность отдельных авиакомпаний. Предложенные модификации позволили определить оптимальные (с точки зрения критериев MAE, MSE, RMSE) значения параметров модели SARIMA при сильно ограниченном числе измерений. Это ограничение связано с серьезным изменением характера рядов после пандемии SARS-Covid-19 и с особенностями сбора помесячных показателей. Всего имеется 60 измеренных помесячных значений на каждый показатель, что ограничивает применяемые методы из-за недостаточности элементов в обучающей выборке. Тем не менее, полученные значения модели SARIMA хоть и имеют существенную погрешность в предсказаниях помесячных показателей (из-за отсутствия стационарности и сложном представлении ряда), но обеспечивают минимальную ошибку предсказания годовой динамики, что требуется при стратегическом планировании развития отрасли.

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

Список литературы

1. *Colorni A., Dorigo M., Vittorio M.* Distributed Optimization by Ant Colonies // Proceedings of the First European Conference on Artificial Life. Elsevier Publishing. 1991. 134–142.
2. *Dorigo M., Gambardella L.M.* Ant Colony System: a Cooperative Learning Approach to the Traveling Salesman Problem // Evolutionary Computation, IEEE Trans. 1997. **1.1**. 53–66. doi 10.1109/4235.585892.
3. *Bullnheimer B., Hartl R., Strauss C.* An improved Ant System Algorithm for the Vehicle Routing Problem // Annals of Operations Research. 1999. **89**. 319–328. doi 10.1023/A:1018940026670.
4. *Gambardella L.M., Taillard E.D., Dorigo M.* Ant colonies for the quadratic assignment problem // Journal of the Operational Research Society. 1999. **50**, N 2. 167–176.
5. *Huang K.L., Liao C.J.* Ant Colony Optimization Combined with Taboo Search for the Job Shop Scheduling Problem // Computers & Operations Research. 2008. **35**, N 4. 1030–1046. doi 10.1016/j.cor.2006.07.003.
6. *Dorigo M., Maniezzo V., Colorni A.* Positive Feedback as a Search Strategy // Technical Report 91-016, Politecnico di Milano, Italy, 1991.
7. *Gambardella L.M., Dorigo M.* Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem // Machine Learning Proceedings 1995. 252–260. doi 10.1016/B978-1-55860-377-6.50039-6.
8. *Bullnheimer B., Hartl R.F., Strauss C.* Applying the Ant System to the Vehicle Routing Problems // Annals of Operations Research. 1998. **79**. 109–123.
9. *Stützle T., Hoos H.H.* MAX–MIN Ant System // Future Generation Computer Systems. 2000. **16**, N 8. 889–914. doi 10.1016/S0167-739X(00)00043-1.
10. *Cordon O., Fernández de Viana I., Herrera F.* Analysis of the best-worst ant system and its variants on the TSP // Soft Computing. 2002. **9**, N 2–3. 177–192.
11. *Randall M., Lewis A.A.* A Parallel Implementation of Ant Colony Optimization // Journal of Parallel and Distributed Computing. 2002. **62**, N 9. 1421–1432. doi 10.1006/jpdc.2002.1854.
12. *Dorigo M., Stützle T.* Ant Colony Optimization // Cambridge, Massachusetts: MIT Press. 2004, 321
13. *Bai H., OuYang D., Li X., et al.* MAXMIN Ant System on GPU with CUDA // 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC). IEEE Computer Society. 2009. 801–804. doi 10.1109/ICICIC.2009.255.
14. *Chitty D.M.* Applying ACO to Large Scale TSP Instances // F. Chao et al. (Eds.) Advances in Computational Intelligence Systems. UKCI. Intelligent Systems and Computing, Springer, Cham. 2018. **650**. doi 10.1007/978-3-319-66939-7_9.
15. *Skinderowicz R.* The GPU-based parallel Ant Colony System // Journal of Parallel and Distributed Computing. 2016. **98**. 48–60. doi 10.1016/j.jpdc.2016.04.014.
16. *Zhang D., You X., Liu S., Yang K.* Multi-Colony Ant Colony Optimization Based on Generalized Jaccard Similarity Recommendation Strategy // IEEE Access. 2019. **7**. 157303-157317. doi 10.1109/ACCESS.2019.2949860.
17. *Skinderowicz R.* Implementing a GPU-based parallel MAX–MIN Ant System // Future Generation Computer Systems. 2020. **106**. 277–295. doi 10.1016/j.future.2020.01.011.
18. *Huan Z.B., Fu G.T., Fa T.H., et al.* High Performance Ant Colony System Based on GPU Warp Specialization with a Static-Dynamic Balanced Candidate Set Strategy // Future Generation Computer Systems. 2021. **125**. 136–150. doi 10.1016/j.future.2021.06.041.
19. *Sinitsyn I.N., Titov Y.P.* Control of Set of System Parameter Values by the Ant Colony Method // Autom. Remote Control. 2023. **84**. 893–903. doi 10.1134/S0005117923080106.
20. *Sudakov V.A., Titov Y.P.* Matrix-Based ACO for Solving Parametric Problems Using Heterogeneous Reconfigurable Computers and SIMD Accelerators // Mathematics. 2025. **13** № 1284. doi 10.3390/math13081284.
21. *Sudakov V.A., Titov Y.P.* Investigation of the Parametric Graph Model in the Ant Colony Method // Math. Models Comput. Simul. 2025. **17**. 126–136. doi 10.1134/S2070048224700996.
22. *Синицын И.Н., Титов Ю.П.* Исследование алгоритмов циклического поиска дополнительных решений при оптимизации порядка следования гиперпараметров методом муравьиных колоний // Системы высокой доступности. 2023. **19**, N 1. 59–73. http://radiotec.ru/ru/journal/Highly_available_systems/number/2023-1/article/23354. Дата обращения: 25 февраля 2026 г.
23. *Титов Ю.П.* Реализация модификаций алгоритма ACO на CUDA для алгоритма работы на SIMD вычислителе [Электронный ресурс]. https://github.com/kalengul/ACO_SIMD. Дата обращения: 25 февраля 2026 г.

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

24. *Mishra S.K.* Some New Test Functions for Global Optimization and Performance of Repulsive Particle Swarm Method // University Library of Munich, Germany, MPRA Paper. 2006. https://mpra.ub.uni-muenchen.de/2718/. Cited February 25, 2026.
25. *Chetverushkin B.N., Sudakov V.A., Titov Y.P.* Graph Condensation for Large Factor Models // Dokl. Math. 2024. **109**. 246–251. doi 10.1134/S1064562424702090.

Получена *24 сентября 2025 г.*
Принята *18 января 2026 г.*
Опубликована *27 февраля 2026 г.*

Информация об авторе

Юрий Павлович Титов — к.т.н., доцент; Московский авиационный институт (национальный исследовательский университет), Волоколамское шоссе, д. 4, 125080, Москва, Российская Федерация.

References

1. A. Colorni, M. Dorigo and M. Vittorio, "Distributed Optimization by Ant Colonies," *Proceedings of the First European Conference on Artificial Life. Paris, France, January 1991* (Elsevier Publishing, 1991), pp. 134–142.
2. M. Dorigo and L. M. Gambardella, "Ant Colony System: a Cooperative Learning Approach to the Traveling Salesman Problem," Evolutionary Computation, IEEE Transactions. **1.1**, 53–66 (1997). doi 10.1109/4235.585892.
3. B. Bullnheimer, R. Hartl and C. Strauss, "An improved Ant System Algorithm for the Vehicle Routing Problem," Annals of Operations Research **89**, 319–328 (1999). doi 10.1023/A:1018940026670.
4. L. M. Gambardella, E. D. Taillard and M. Dorigo, "Ant colonies for the quadratic assignment problem," Journal of the Operational Research Society. **50** (2), 167–176 (1999).
5. K. L. Huang and C. J. Liao, "Ant Colony Optimization Combined with Taboo Search for the Job Shop Scheduling Problem," Computers & Operations Research. **35** (4), 1030–1046 (2008). doi 10.1016/j.cor.2006.07.003.
6. M. Dorigo, V. Maniezzo and A. Colorni, "Positive Feedback as a Search Strategy," In *Technical Report 91-016, Politecnico di Milano, Italy, 1991.*
7. L. M. Gambardella and M. Dorigo, "Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem," In *Machine Learning Proceedings 1995, Morgan Kaufmann*, pp. 252–260. doi 10.1016/B978-1-55860-377-6.50039-6.
8. B. Bullnheimer, R. F. Hartl and C. Strauss, "Applying the Ant System to the Vehicle Routing Problems," Annals of Operations Research. **79**, 109–123 (1998). doi 10.1007/978-1-4615-5775-3_20.
9. T. Stützle and H. H. Hoos, "MAX–MIN Ant System," Future Generation Computer Systems. **16** (8), 889–914 (2000). doi 10.1016/S0167-739X(00)00043-1.
10. O. Cordon, I. F. de Viana, F. Herrera, "Analysis of the best-worst ant system and its variants on the TSP," Soft Computing. **9** (2–3), 177–192 (2002).
11. M. Randall, A. A. Lewis, "A Parallel Implementation of Ant Colony Optimization," Journal of Parallel and Distributed Computing. **62** (9), 1421–1432 (2002). doi 10.1006/jpdc.2002.1854.
12. M. Dorigo, T. Stützle, *Ant Colony Optimization* (MIT Press, Cambridge, Massachusetts, 2004).
13. H. Bai, D. OuYang, X. Li, at al., "MAXMIN Ant System on GPU with CUDA," In *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), Kaohsiung, Taiwan, December 7–9, 2009* (IEEE Computer Society), pp. 801–804. doi 10.1109/ICICIC.2009.255.
14. D. M. Chitty, "Applying ACO to Large Scale TSP Instances," In: *F. Chao et al. (Eds.) Advances in Computational Intelligence Systems. UKCI 2017* (Intelligent Systems and Computing, Springer, Cham), **650** (2018). doi 10.1007/978-3-319-66939-7_9.
15. R. Skinderowicz, "The GPU-based parallel Ant Colony System," Journal of Parallel and Distributed Computing. **98**, 48–60 (2016) doi 10.1016/j.jpdc.2016.04.014.
16. D. Zhang, X. You, S. Liu, K. Yang, "Multi-Colony Ant Colony Optimization Based on Generalized Jaccard Similarity Recommendation Strategy," IEEE Access. **7**, 157303-157317 (2019) doi 10.1109/ACCESS.2019.2949860.
17. R. Skinderowicz, "Implementing a GPU-based parallel MAX–MIN Ant System," Future Generation Computer Systems. **106**, 277–295 (2020) doi 10.1016/j.future.2020.01.011.

2026, **27** (1), 46–66. doi 10.26089/NumMet.v27r104

18. Z. B. Huan, G. T. Fu, T. H. Fa, at al., "High Performance Ant Colony System Based on GPU Warp Specialization with a Static-Dynamic Balanced Candidate Set Strategy," Future Generation Computer Systems. **125**, 136–150 (2021). doi 10.1016/j.future.2021.06.041.
19. I. N. Sinitsyn, Y. P. Titov, "Control of Set of System Parameter Values by the Ant Colony Method," Autom. Remote Control, **84**, 893–903 (2023). doi 10.1134/S0005117923080106.
20. V. Sudakov, Y. Titov, "Matrix-Based ACO for Solving Parametric Problems Using Heterogeneous Reconfigurable Computers and SIMD Accelerators," Mathematics, **13** (1284), (2025). doi 10.3390/math13081284.
21. V. A. Sudakov, Y. P. Titov, "Investigation of the Parametric Graph Model in the Ant Colony Method," Math. Models Comput. Simul. **17**, 126–136 (2025). doi 10.1134/S2070048224700996.
22. I. N. Sinitsyn, Y. P. Titov, "Investigation of algorithms for cyclic search for additional solutions when optimizing the order of hyperparameters by the ant colony method," High Availability Systems. **19** (1), 59–73 (2023). http://radiotec.ru/en/journal/Highly_available_systems/number/2023-1/article/23354. Cited February 25, 2026.
23. Y. P. Titov, Implementation of ACO Algorithm Modifications on CUDA for SIMD Compute Processor Operation [*Electronic resource*]. https://github.com/kalengul/ACO_SIMD. Cited January 23, 2026.
24. S. K. Mishra, "Some New Test Functions for Global Optimization and Performance of Repulsive Particle Swarm Method," University Library of Munich, Germany, MPRA Paper. **2718** (2006). https://mpra.ub.uni-muenchen.de/2718/. Cited February 25, 2026.
25. B. N. Chetverushkin, V. A. Sudakov, Y. P. Titov, "Graph Condensation for Large Factor Models," Dokl. Math. **109**, 246–251 (2024). doi 10.1134/S1064562424702090.

Received *September 24, 2025*
Accepted *January 18, 2026*
Published *February 27, 2026*

Information about the author

Yurii P. Titov — Ph. D., Associate Professor; Moscow Aviation Institute (National Research University), Volokolamskoe shosse, 4, 125080, Moscow, Russia.