2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

doi 10.26089/NumMet.v27r327

УДК 004.4'2;
004.272.2

Исследование применимости автотюнеров в задачах суперкомпьютерного кодизайна на примере MLKAPS

Т. Е. Загоруйко
Московский государственный университет имени М. В. Ломоносова,
Научно-исследовательский вычислительный центр,
Москва, Российская Федерация
ORCID: 0009-0004-5619-3670, e-mail: tzag2001@gmail.com

А. С. Антонов
Московский государственный университет имени М. В. Ломоносова,
Научно-исследовательский вычислительный центр,
Москва, Российская Федерация
ORCID: 0000-0003-2820-7196, e-mail: asa@parallel.ru

Аннотация: В статье представлен обзор нескольких современных автотюнеров с акцентом на их функциональность, показана их применимость к решению задач в рамках концепции суперкомпьютерного кодизайна. Приводятся результаты экспериментального исследования эффективности автотюнера MLKAPS применительно к задаче оптимизации параметров теста производительности HPL. Проведен сравнительный анализ автотюнера MLKAPS с методом полного перебора параметров. Продемонстрировано, что MLKAPS при условии более обширного пространства поиска всех возможных конфигураций позволяет существенно сократить время поиска оптимальной конфигурации. Полученные результаты демонстрируют практическую применимость автотюнеров для автоматизации настройки параллельных приложений в условиях доступности ограниченных ресурсов суперкомпьютерных комплексов.

Ключевые слова: автотюнер, суперкомпьютерный кодизайн, Kernel Tuner, KTT, GPTune, HYPERF, VibeCodeHPC, MLKAPS, HPL, оптимизация.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 25–11–00181, https://rscf.ru/project/25-11-00181/. При получении результатов использовалось оборудование Центра коллективного пользования сверхвысокопроизводительными вычислительными ресурсами МГУ имени М. В. Ломоносова [1].

Для цитирования: Загоруйко Т.Е., Антонов А.С. Исследование применимости автотюнеров в задачах суперкомпьютерного кодизайна на примере MLKAPS // Вычислительные методы и программирование. 2026. **27**, № 3. 417–433. doi 10.26089/NumMet.v27r327.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

Investigation of the applicability of autotuners in supercomputer co-design problems using MLKAPS

Tatiana E. Zagoruiko
Lomonosov Moscow State University, Research Computing Center, Moscow, Russia
ORCID: 0009-0004-5619-3670, e-mail: tzag2001@gmail.com

Alexander S. Antonov
Lomonosov Moscow State University, Research Computing Center, Moscow, Russia
ORCID: 0000-0003-2820-7196, e-mail: asa@parallel.ru

Abstract: This paper provides an overview of a subset of modern autotuners, focusing on their functionality and demonstrating their applicability to solving problems within the framework of supercomputer co-design concept. The results of an experimental study of the MLKAPS autotuner's effectiveness in relation to the problem of optimizing HPL benchmark parameters are presented. A comparative analysis of the MLKAPS autotuner with the full iteration method is conducted. It is demonstrated that MLKAPS, given a larger search space of all possible configurations, can significantly reduce the search time for the optimal configuration. The obtained results demonstrate the practical applicability of autotuners for automating the tuning of parallel applications in conditions of availability of limited resources of supercomputer complexes.

Keywords: autotuner, supercomputer co-design, Kernel Tuner, KTT, GPTune, HYPERF, VibeCodeHPC, MLKAPS, HPL, optimization.

Acknowledgements: The study was supported by a grant from the Russian Science Foundation No. 25–11–00181, https://rscf.ru/en/project/25-11-00181/. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University [1].

For citation: T. E. Zagoruiko, A. S. Antonov, "Investigation of the applicability of autotuners in supercomputer co-design problems using MLKAPS," Numerical Methods and Programming. **27** (3), 417–433 (2026). doi 10.26089/NumMet.v27r327.

1. Введение. В условиях стремительного роста объемов вычислений и усложнения решаемых прикладных задач особую актуальность приобретает проблема эффективного использования ресурсов высокопроизводительных вычислительных систем. Ключевыми факторами эффективности при этом являются не только время выполнения программ на суперкомпьютерах, но и совокупные трудозатраты, связанные с разработкой, оптимизацией и сопровождением параллельных приложений.

Оптимизация параллельных программ представляет собой сложную междисциплинарную задачу и требует от разработчика глубоких знаний триады суперкомпьютерного кодизайна [2], включающей алгоритмические решения, особенности программной реализации и характеристики целевой вычислительной архитектуры. Данная триада суперкомпьютерного кодизайна представляется одной из ключевых особенностей разработки современных суперкомпьютерных приложений, поскольку их эффективность во многом определяется не только качеством написанного кода, но и корректностью выбранного для реализации алгоритма, выбора параметров параллелизации, организации памяти, выбора способов распределения вычислений и других характеристик, которые напрямую зависят от доступной разработчику архитектуры вычислительной системы. Использование современных моделей параллельного программирования (таких как OpenMP, MPI, CUDA и др.) предполагает необходимость учета множества параметров, влияющих на производительность, включая число вычислительных потоков/процессов, организацию памяти, параметры декомпозиции задачи и специфику архитектуры.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

Неоптимальный выбор данных параметров приводит к снижению эффективности вычислений, увеличению времени выполнения программ и, как следствие, к росту затрат на использование ресурсов суперкомпьютеров. Дополнительную сложность вносит разнообразие современных вычислительных архитектур, каждая из которых предъявляет собственные требования к настройке и оптимизации программного кода. В результате либо существенно возрастает нагрузка на высококвалифицированных специалистов, либо возникает необходимость проведения большого числа экспериментальных запусков с целью подбора эффективных конфигураций, что также требует значительных временны́х и вычислительных ресурсов.

Одним из перспективных направлений решения указанных проблем является использование технологий автотюнинга параллельных программ. Автотюнеры представляют собой программные инструменты, предназначенные для автоматизированного поиска оптимальных или близких к оптимальным конфигураций параметров исполнения программ с учетом особенностей конкретной вычислительной архитектуры. При этом цели автотюнинга не ограничиваются исключительно минимизацией времени выполнения: в зависимости от условий задачи и ограничений архитектуры оптимизация может быть ориентирована, например, на снижение потребления памяти или энергозатрат.

Поскольку автотюнеры работают с программной реализацией и направлены на оптимизацию различных целевых метрик, которые зависят как от выбранного разработчиком алгоритма, так и от доступной ему архитектуры, можно говорить о том, что автотюнеры работают в рамках парадигмы суперкомпьютерного кодизайна и всегда важно рассматривать данные инструменты с позиции всей триады (алгоритм–реализация–архитектура), а не какой-либо ее части в отрыве от других. В противном случае результаты таких исследований будут слишком узконаправленными, маловоспроизводимыми и субъективными.

Применение автотюнеров позволяет частично формализовать и автоматизировать процессы оптимизации, снижая требования к глубине специализированных знаний разработчика и повышая воспроизводимость получаемых результатов. Благодаря этому автотюнинг рассматривается как универсальный инструмент, потенциально применимый как в научных исследованиях, так и в индустриальных проектах, связанных с разработкой и эксплуатацией высокопроизводительных параллельных приложений.

Целью данной статьи является демонстрация различных современных подходов автотюнинга суперкомпьютерных приложений и исследование их применимости в задачах суперкомпьютерного кодизайна на примере MLKAPS.

В рамках работы обозреваются современные автотюнеры, ориентированные на оптимизацию HPC-приложений (High Performance Computing), с описанием их внутренней организации, предоставляемого функционала и отличительных функций, которые выделяют конкретные автотюнеры на фоне других подобных инструментов. Также в рамках статьи приведено описание апробации одного из автотюнеров — MLKAPS — для ознакомления с тонкостями внедрения такого инструмента на практике.

Статья направлена на демонстрацию потенциальной пользы, которую могут предоставить автотюнеры, и перспективности данного направления для дальнейших исследований, поскольку в силу небольшой распространенности темы автотюнеров ряд инструментов довольно быстро теряет поддержку разработчиков и остается в статусе исследовательского проекта, а не развивается далее как практический инструмент, который можно внедрить и использовать в рамках реальных проектов.

1.1. Эволюция кодизайна в высокопроизводительных вычислениях. Концепция кодизайна в контексте HPC представляет собой совместную разработку аппаратных решений для достижения максимальной производительности при минимальных затратах ресурсов. В работе [3] подчеркивается, что такой подход особенно важен в условиях роста сложности архитектур суперкомпьютеров и все более специализированных вычислительных задач. Автор выделяет четыре ключевых аспекта кодизайна: совместное проектирование архитектуры, системного ПО, компиляторов и прикладного кода.

Подтверждение важности взаимодействия между уровнями архитектуры, программной реализации и алгоритма можно найти и в исследовании [4], где анализируется влияние распределения вычислений на многопроцессорных системах с общей памятью на производительность приложений.

1.2. Обзор существующих решений. Для понимания современных подходов к автотюнингу HPC-приложений полезно рассмотреть существующие исследования, посвященные автотюнингу, поскольку анализ данных работ позволяет определить основные направления развития автотюнеров, способы их классификации и сравнения, и прочее.

Развитие технологий автотюнинга за последние два десятилетия привело к появлению широкого спектра инструментов, основанных на различных подходах к оптимизации параллельных программ. Современные автотюнеры отличаются как по целевым моделям параллельного программирования и вычис-

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

лительным архитектурам, так и по методам поиска оптимальных конфигураций и способам интеграции с прикладным кодом.

В данной работе под понятием “модель параллельного программирования” подразумевается версия авторов [5, 6] — это совокупность программных средств и механизмов, обеспечивающих параллельное исполнение вычислений, включая использование специализированных библиотек, директив, языковых расширений и других средств распараллеливания.

Различия в подходах и механизмах реализации автотюнеров обуславливают возможность их классификации по ряду признаков, среди которых можно выделить следующие:

- Поддерживаемые модели параллельного программирования, для которых они могут быть применимы: ytopt (X. Wu) [7–9], HYPERF (J. Park) [10, 11] или MLKAPS (M. Jam) [12, 13] используются для программ, разработанных по технологии OpenMP, а CLTune (C. Nugteren, V. Codreanu) [14, 15], Kernel Tuner (B. van Werkhoven) [16–18] и KTT (F. Petrovič) [19–21], в свою очередь, — для программ, использующих технологии OpenCL и CUDA.
- Целевые вычислительные архитектуры. Данный критерий тесно связан с используемой моделью параллельного программирования: автотюнеры, ориентированные на технологию OpenMP, как правило, применяются для оптимизации программ, исполняемых на многоядерном центральном процессоре, тогда как инструменты для OpenCL и CUDA нацелены на использование графических ускорителей.
- Методы подбора оптимальной конфигурации. Под понятием “конфигурация” понимается комбинация параметров, которые перебирает автотюнер в пространстве поиска. В свою очередь, понятие “пространство поиска” обозначает все возможные комбинации таких параметров (число потоков, на которых запускается программа, размер рабочей группы в случае OpenCL, размеры тайлов и прочее). Ранние автотюнеры, например ATLAS (R. C. Whaley, J. J. Dongarra) [22, 23], основывались на полном переборе пространства поиска, что обеспечивало высокое качество результатов, но сопровождалось значительными временны́ми затратами. Современные инструменты используют методы поиска, позволяющие отсекать некоторые заведомо “проигрышные” конфигурации, например, Kernel Tuner может применять случайный поиск, метод имитации отжига, генетические алгоритмы и другие, а Orion (A. B. Hayes) [24] использует метод многорукого бандита (Multi-Armed Bandit). Подобные методы позволяют существенно сократить время перебора конфигураций.
- И другие.

В основном значительная часть исследований по теме автотюнинга посвящена классификации подходов к автотюнингу и методам исследования пространства поиска. Например, в работе [25] предложена классификация автотюнеров по ряду признаков, включая методы исследования пространства поиска, область применения настройки, этап выполнения оптимизации и используемые модели оценки конфигураций. Автор рассматривает автотюнинг как многоуровневую задачу, затрагивающую как программную реализацию, так и особенности вычислительной архитектуры. В обзоре [26] систематизированы исследования, посвященные применению методов машинного обучения в задачах автотюнинга на уровне компиляторов. Авторы выделяют различные подходы к анализу характеристик программ, сокращению пространства поиска и использованию машинного обучения для прогнозирования производительности. В статье [27] предлагается классификация по множеству различных признаков: представление автотюнеров (библиотечное, компиляторное и др.), типы решений (алгоритм, вариант кода и др.), способы интеграции в приложение и др. Авторы также обсуждают проблемы масштабируемости, вычислительных затрат и ограниченной переносимости результатов автотюнинга между различными архитектурами. Отдельный интерес могут представлять работы [28, 29], поскольку в них затрагивается направление исследований, связанное с созданием инфраструктуры для накопления и повторного использования данных автотюнинга. Авторы предлагают платформу Collective Mind, предназначенную для систематизации результатов оптимизации и обмена информацией между исследователями. Важная тема сравнения автотюнеров между собой, а также причин искажения результатов при представлении эффективности инструментов автотюнинга, рассмотрена в [30].

2. Обзор автотюнеров. Рассмотренные исследования формируют первичное и общее представление об автотюнерах, но практическое применение данных инструментов и подробное описание подходов к их разработке можно найти в конкретных программных инструментах. В связи с этим далее рассматривается некоторый набор современных автотюнеров с подробным описанием их внутренней организации, особенностей предоставляемого функционала и прочего.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

2.1. Kernel Tuner. Kernel Tuner [16–18] — фреймворк для тестирования и автоматической настройки ядер CUDA, OpenCL и выполняемых на стороне хоста вычислительных ядер, разработанных на языке C с использованием, например, OpenMP с поддержкой множества различных методов поиска. Фреймворк спроектирован с возможностью дальнейшего расширения функциональных возможностей.

На вход Kernel Tuner подается пользовательский скрипт на языке Python, в котором описаны расположение исходного кода, требования к его вызову и параметры для дальнейшей настройки. После этого автотюнер занимается компиляцией и сравнительным анализом вычислительных ядер, а также поиском оптимальной конфигурации для каждого ядра. В рамках использования Kernel Tuner определим следующие термины, которыми будем оперировать далее:

- оптимизация кода — изменения, вносимые программистом в код вычислительного ядра с целью повышения его производительности, но в данном случае под оптимизацией подразумевается поиск, выполняемый автотюнером;
- настраиваемые параметры ядра — параметризованные оптимизации кода, которые могут принимать различные значения и могут быть как включены, так и отключены.

Работа с Kernel Tuner состоит из трех блоков: пользовательский ввод, настройка ядра и бэкенд.

§ 2.1.1. Пользовательский ввод. В рамках пользовательского ввода пользователь предоставляет на вход автотюнеру код ядра (он может быть передан либо как путь к файлу с исходным кодом, либо непосредственно в виде строковой переменной внутри скрипта), а также скрипт на языке Python для дальнейшей настройки, который использует функции, разработанные в пользовательском интерфейсе — одной из частей настройки ядра.

§ 2.1.2. Настройка ядра. Настройка ядра состоит из нескольких частей (логических блоков): *пользовательский интерфейс*, *стратегии* и *исполнители*.

Пользовательский интерфейс предоставляет пользователю набор функций, с помощью которого он может описать код ядра и что именно он планирует в нем настраивать. В данном наборе наиболее распространены следующие функции:

- `run_kernel` — компилирует и запускает определенную конфигурацию ядра, а также управляет выделением памяти на GPU и копированием данных в память GPU и обратно; возвращает выходные данные ядра GPU в виде списков массивов библиотеки `Numpy` языка Python;
- `tune_kernel` — используется для настройки ядра GPU с учетом набора настраиваемых параметров.

Обе упомянутые выше функции поддерживают множество различных параметров, позволяющих, например, осуществить выбор устройства и/или платформы для настройки, выбор компилятора и его параметров и другое.

Стратегии отвечают за выбор конфигураций ядра из пространства поиска, которые будут скомпилированы и протестированы на следующем этапе исполнителями. По умолчанию будет использоваться метод полного перебора, при котором рассматриваются все варианты в пространстве поиска. Но можно также использовать такие методы поиска, как случайный поиск, имитация отжига, рой частиц, генетические алгоритмы и другие.

Исполнители компилируют и тестируют конфигурации, выбранные стратегиями. На данный момент поддерживается только последовательный исполнитель, который использует только один последовательный процесс Python. У исполнителя есть свой набор функций, которые отвечают за выделение памяти на GPU, перемещение данных на GPU и обратно и прочее.

Рассмотренные блоки настройки ядра фактически являются не столько последовательными этапами Kernel Tuner, сколько разбиением различных функций данного автотюнера, которые описываются в скрипте Python, по смыслу их использования.

§ 2.1.3. Бэкенд. У Kernel Tuner всего три бэкенда: PuCUDA, PyOpenCL и GCC. Первые два бэкенда отвечают за настройки ядер CUDA и OpenCL соответственно, а последний — за настройку функций на языке C (внутри он может вызывать компиляторы GCC и NVCC).

Также в Kernel Tuner реализованы так называемые “наблюдатели” — программируемые хуки, расширяющие возможности автотюнера, которые позволяют контролировать, что именно он измеряет. Фактически наблюдатели позволяют привлечь некоторые дополнительно установленные устройства к замерам

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

различных характеристик, которые в дальнейшем будут применяться для оценки конфигураций, например:

- `PowerSensorObserver` — наблюдатель, предназначенный для работы со специализированным электроизмерительным прибором PowerSensor2 [31], позволяющим измерять потребляемую устройствами PCIe электрическую мощность и передавать показания на хост по интерфейсу USB;
- `NVMLObserver` — наблюдатель, применяемый для работы с NVML (NVIDIA Management Library) [32], позволяющий отслеживать энергопотребление, расход энергии, частоту ядра и памяти GPU, а также напряжение и температуру GPU для всех конфигураций во время тестирования производительности.

Кроме того, при необходимости пользователи могут создать собственных наблюдателей.

Kernel Tuner возвращает результаты тестирования в виде списка словарей и словарь `env`, в котором содержится важная информация об аппаратном и программном обеспечении, на котором проводилось тестирование.

2.2. KTT. Kernel Tuning Toolkit (KTT) [19–21] — фреймворк, позволяющий автоматически настраивать вычислительные ядра, написанные на CUDA, OpenCL и Vulkan. Как заявляют авторы, KTT обеспечивает взаимодействие между хостом и ускорителем, создает пространство поиска параметров настройки и осуществляет в нем поиск, проверяет результаты и время выполнения настроенных ядер, позволяет выполнять динамическую настройку во время работы программы, профилировать автоматически настроенные ядра и многое другое.

Во время автономной настройки ядра с помощью KTT создается ядро, указываются его аргументы (входные и выходные данные), определяются параметры настройки и выполняется непосредственно запуск. Значения параметров передаются в исходный код ядра в виде определений препроцессора, и их можно использовать для изменения функциональности ядра.

Фреймворк KTT представляет собой библиотеку, которую можно интегрировать в пользовательский код, и его применение можно разбить на выполнение следующих задач:

1. Инициализация KTT. На данном этапе требуется создать экземпляр тюнера, являющегося одним из основных классов KTT, поскольку дальнейшие этапы тесно связаны с конкретным экземпляром тюнера. Обязательными параметрами являются индекс платформы, индекс устройства и используемый вычислительный API (например, CUDA).
2. Определение ядра. Требуется загрузить исходный код в тюнер с помощью создания определения ядра, при котором указываются имя функции ядра и его исходный код. Фактически в рамках KTT вычислительное ядро соответствует конкретной функции. Например, для
   `__kernel void conv(`<аргументы функции>`) {`<тело функции>`}`
   определение будет выглядеть следующим образом:
   `tuner.AddKernelDefinitionFromFile("conv", ...).`
3. Определение входных и выходных данных ядра. KTT поддерживает в качестве форм аргументов ядра буферы, скалярные значения и константные аргументы памяти. После предоставления данных будет возвращен дескриптор аргумента, с помощью которого можно присваивать аргументы определениям ядра, введенным на предыдущем этапе.
4. Настройка параметров. При определении нового параметра требуется указывать его имя, на которое можно будет ссылаться в исходном теле ядра, и значения, которые этот параметр может принимать. Например, `tuner.AddParameter(kernel, "unroll_factor", std::vector<uint64_t>{1, 2, 4})`.
5. Валидация выходных данных. Прежде чем определить оптимальную конфигурацию, важно также убедиться, что при указанных параметрах вычислительное ядро срабатывает корректно.
6. Лаунчер ядра. Функция, которая определяет, что происходит при запуске ядра. Если требуются дополнительные манипуляции с вычислительным ядром (например, часть операций выполняется на хосте, или функцию требуется запустить несколько раз, или использовать при запуске сразу несколько функций ядра), то пользователю необходимо определить собственный лаунчер. В рамках KTT лаунчер имеет доступ к его низкоуровневому вычислительному интерфейсу, который позволяет запускать функции ядра, менять число потоков, модифицировать буферы и прочее.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

7. Применение искателей. Они определяют порядок выбора и запуска конфигураций ядра при автономной и динамической настройке. На данный момент KTT поддерживает три искателя:
   - `DeterministicSearcher`, который исследует конфигурации в одном и том же порядке при условии отсутствия изменений в параметрах настройки, порядка их добавления и значений;
   - `RandomSearcher`, исследующий конфигурации в случайном порядке;
   - `McmcSearcher`, использующий метод Монте-Карло с цепями Маркова для более точного прогнозирования эффективных конфигураций.

   При этом KTT также допускает создание и использование пользовательских искателей при необходимости.

Кроме времени выполнения KTT способен собирать другую информацию о запусках ядра, например: использование глобальной памяти, количество выполненных инструкций и прочее. Но по умолчанию сбор метрик профилирования отключен.

2.3. GPTune. GPTune (GP от Gaussian Process) [33–35] — фреймворк автоматической настройки для решения основной задачи оптимизации "черного ящика", основанный на таком суррогатном моделировании, как регрессия гауссовского процесса с поддержкой таких новых функций автотюнинга, как многозадачное обучение, трансферное обучение и другие. Разработан специально для автотюнинга приложений HPC, которые могут содержать множество настраиваемых параметров и требовать значительных вычислительных ресурсов для оценки. Стоит отметить, что GPTune отличается формальным математическим описанием работы.

GPTune поддерживает параллелизм как с общей, так и с распределенной памятью. Доступны следующие три режима:

1. Режим запуска MPI (по умолчанию). Накладывает ограничения на компиляцию приложения пользователя, поскольку оно должно быть скомпилировано с теми же зависимостями, что и GPTune.
2. Режим RCI (Reverse Communication Interface). Применяется для обхода проблем режима по умолчанию (в ситуациях недоступности версии MPI, с которой был скомпилирован GPTune, для приложения пользователя или наличия более быстрых MPI-реализаций, например SpectrumMPI, CrayMPICH и прочие).
3. Упрощенный режим. Не зависит от OpenMPI и mpi4py. Напоминает режим по умолчанию за тем исключением, что при его использовании не поддерживается параллелизм с распределенной памятью при моделировании, поиске и пакетном выполнении целевых функций.

GPTune предоставляет возможность автоматической настройки с одной целью (single-objective) и многими (multi-objective), применяя к ним как многозадачное обучение (MLA, Multi-task Learning Autotuning), так и обучение на одной задаче (SLA, Single-task Learning Autotuning).

Далее опишем некоторые определения, которые ввели авторы GPTune. Они понадобятся для дальнейшего описания алгоритмов:

1. $\mathbb{IS}$ (task parameter Input Space) — пространство задач.
2. $T = \{t_1, \ldots, t_n\}$, $T \in \mathbb{IS}$ — множество задач. Каждая задача представляет собой комбинацию входных параметров, которые задают условия проводимого эксперимента и не участвуют в оптимизации, как, например, размеры матрицы.
3. $\mathbb{PS}$ (tuning Parameter Space) — пространство параметров оптимизации.
4. $X = \{x_1, \ldots, x_m\}$, $X \in \mathbb{PS}$ — множество конфигураций.
5. $\mathbb{OS}$ (Output Space) — пространство целевых метрик.
6. $Y = \{y_1, \ldots, y_s\}$, $Y \in \mathbb{OS}$ — множество значений целевой функции $y(t, x)$.

§ 2.3.1. Настройка с одной целью. Основной алгоритм GPTune для оптимизации одной метрики. Состоит из четырех этапов: семплинга, моделирования, поиска и выдачи результата.

На этапе семплинга из всего пространства поиска $\mathbb{PS}$ выбираются случайным образом произвольные точки x_i, и для каждой задачи t_j производится запуск приложения и измерение целевой метрики y_k. На выходе данного этапа будет получено начальное обучение суррогатной модели.

На этапе моделирования производится обновление гиперпараметров модели LCM (Linear Coregionalization Model) для $y(t, x)$, на выходе из которого получаем апостериорное среднее и апостериорную

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

дисперсию, которые передаются на вход этапу оптимизации. В рамках данного этапа производится поиск параметров x, которые максимизируют функцию ожидаемого улучшения (Expected Improvement, EI) для задачи t. Для найденных новых параметров x' вычисляется значение $y'(t, x')$.

Этапы моделирования и оптимизации продолжаются до тех пор, пока не будет исчерпано допустимое число запусков, заданное пользователем. На выходе всего алгоритма будут оптимальные параметры для каждой задачи и значение целевой метрики.

§ 2.3.2. Многоцелевая настройка. Фактически алгоритм идентичен настройке с одной целью за исключением того, что y становится вектором $y = \{y^1, \ldots, y^i\}$ (например, требуется оптимизировать не только время выполнения, но и энергопотребление) и во время этапа поиска выбираются точки, которые улучшают хотя бы одну метрику и при этом не ухудшают другие.

2.4. HYPERF. Фреймворк HYPERF предназначен для автотюнинга HPC с интерфейсом C/C++ на основе `pragma` и механизмом автоматической настройки на основе расписания (schedule) [10, 11]. Ключевой идеей основанного на `pragma` подхода к настройке является отделение семантики программы от оптимизации. Автотюнер управляет только пользовательскими директивами (например, технологии OpenMP) с выбранными параметрами для генерации кандидатов с помощью низкоуровневого компилятора с поддержкой `pragma`. Это расширяет модель программирования OpenMP, позволяя пользователям указывать циклы для делегирования в HYPERF.

Фреймворк предназначен для области HPC, а именно для оптимизации научного и инженерного кода, чувствительного к пропускной способности памяти.

Фактически HYPERF состоит из двух основных компонентов:

1. Драйвера автоматической настройки OpenMP C/C++, который управляет сквозным (end-to-end) процессом компиляции, координируя интерфейс OpenMP C/C++, TVM-HPC и низкоуровневый компилятор. Принимает программы на OpenMP в качестве входных данных, компилирует их в абстрактные синтаксические деревья (AST), а затем применяет два разных пути компиляции для областей кода, помеченных `#pragma omp autotune`, и остального кода. Для первых областей кода с указанием `pragma` драйвер вызывает OpenMP C/C++ `-to-TIR` транслятор для генерации расписаний TIR (TensorIR, IR — Internal Representation) и запускает TVM-HPC для выполнения автотюнинга. Для остальной части AST драйвер заменяет автонастроенные циклы вызовами описанных функций и компилирует AST стандартным низкоуровневым компилятором. Затем он вызывает компоновщик для генерации окончательного исполняемого файла.
2. Средства автоматической настройки по расписанию для HPC на основе TVM, которое фокусируется на выполнении поисковой оптимизации переведенных программ TIR из драйвера. В частности, выполняет канонизацию TIR, чтобы гарантировать совместимость сгенерированных TIR с автотюнингом, и расширяет автоматическую настройку для поддержки за счет ослабления структурных ограничений и зависимостей, первоначально применявшихся для тензорных операций DL.

Для генерации корректных и совместимых TIR были переопределены некоторые существующие клаузы и введены новые специально для использования директивы `autotune for`:

- `map(`<type> `:` <locator_lst>`)`: `locator_lst` — список смежных разделов массива, где каждый раздел состоит из массива (одномерного или многомерного), в размерностях которого указаны пары значений нижней границы и длины для каждого измерения, разделенные двоеточием. Аргумент <type> указывает направление движения данных (на данный момент поддерживается только `tofrom`).
- `reduction` с точки зрения описания для пользователя идентичен стандартному `reduction`. Используется для определения свойств оси блока, необходимых для создания корректного TIR.
- `private` для пользователя идентичен стандартному `private`. Эта информация используется на этапе канонизации TIR для идентификации и замены вре́менных буферов TIR.
- `struct_info(lst)` — список структурных элементов, к которым осуществляется доступ в области OpenMP. Их требуется передавать в качестве отдельных входных параметров в сгенерированной функции TIR, поскольку изначально TIR не поддерживает данный тип.

2.5. VibeCodeHPC. VibeCodeHPC [36, 37] — мультиагентная система на основе больших языковых моделей (LLMs) для автотюнинга HPC-программ на суперкомпьютерах. Представляет собой автотюнер на основе Claude Code [38] — инструмента для помощи в написании кода на базе LLM, работающий в

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

среде интерфейса командной строки; получает инструкции от пользователей, на основе которых может самостоятельно генерировать, редактировать, выполнять и отлаживать код.

Перечислим основные функции автотюнера VibeCodeHPC.

1. Автотюнинг с учетом уникальных условий разработки на суперкомпьютерах.
2. Совместная работа нескольких агентов LLM с разными ролями. В качестве таких LLM-агентов могут выступать:
   - Менеджер проекта (Project Manager, PM) — агент верхнего уровня, контролирующий весь проект. На основе инструкций пользователя он управляет проектом, дает указания подчиненным агентам, контролирует их работу, а также служит интерфейсом для диалога с пользователем.
   - Системный инженер (System Engineer, SE) — агент промежуточного уровня управления, который руководит работой программистов в соответствии с указаниями PM, координирует работу нескольких программистов, собирает статистическую информацию и формирует отчеты.
   - Программист (Programmer, PG) — агент, отвечающий за выполнение реальных задач по программированию. Разворачивается для каждой специализированной области, вроде оптимизации CPU, GPU, распределенного распараллеливания с использованием MPI или оптимизации алгоритмов. Шаблоны развертывания готовятся в соответствии с описанной конфигурацией суперкомпьютера.
   - Непрерывный поставщик (Continuous Deliverer, CD) — агент, отвечающий за контроль версий сгенерированного кода, отправку на GitHub, создание документов и прочее. Обеспечивает непрерывную разработку и публикацию результатов.
3. Долгосрочная автономная работа, которая обеспечивается благодаря мониторингу активности агентов и механизму динамического развертывания.

VibeCodeHPC состоит из следующих компонентов.

1. ИИ-агента для написания кода на основе интерфейса командной строки: представляет собой агента на основе LLM, выполняющего автономное программирование. На данный момент используется Claude Code.
2. Функционала для работы и разработки на суперкомпьютерах: предыдущий компонент, как правило, разрабатывается с учетом требований к программированию общего назначения, поэтому для работы с операциями и функциями, характерными для суперкомпьютера, требуется дополнительный функционал. VibeCodeHPC предоставляет следующие функции для работы с суперкомпьютером:
   - управление SSH-подключением (организация и поддержка удаленного доступа к суперкомпьютерам);
   - работа с системой batch заданий (отправка заданий, мониторинг состояния выполнения и получение результатов);
   - управление программными модулями (загрузка компиляторов и библиотек с помощью команды модуля окружения);
   - предоставление информации об оборудовании (системные сведения, например, конфигурация кластера, доступность GPU и прочее);
   - предоставление информации о программной среде (сведения о доступных компиляторах, библиотеках и прочем).
3. Функционала многоагентной совместной работы: доработка функциональности Claude Code со стороны разработчиков VibeCodeHPC в рамках автотюнера для обеспечения совместной работы нескольких агентов. Реализованы следующие функции:
   - настройка конфигурации и ролей агентов (определение области ответственности каждого агента и правил работы);
   - механизм взаимодействия между агентами (передача и получение сообщений между агентами);
   - правила работы с несколькими агентами (правила управления совместной работой);
   - функция мониторинга работы агентов (отслеживание состояния активности каждого агента в режиме реального времени);
   - функция динамического развертывания агентов (добавление и удаление агентов по мере необходимости).

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

Работа VibeCodeHPC организована следующим образом.

1. Запуск системы. VibeCodeHPC запускают в локальном терминале пользователя под управлением Linux, с которого осуществляется доступ к суперкомпьютеру и запускается Claude Code.
2. Предоставление задач и требований. Пользователь предоставляет VibeCodeHPC код, который требуется оптимизировать для повышения производительности и, при необходимости, документ с описанием требований (используется только при необходимости автономной оптимизации кода без участия пользователя). В документе должны быть указаны:
   - *целевое оборудование*: архитектура процессора, наличие GPU, конфигурации узла и прочее;
   - *целевые показатели производительности*: время выполнения, пропускная способность, энергоэффективность и прочее;
   - *ограничения задачи*: неоптимизируемые части, запрещенные библиотеки, требования к точности и прочее.
3. Выполнение автономной оптимизации. После предоставления документа VibeCodeHPC запускает процесс оптимизации. Генерация кода выполняется на локальном компьютере, после чего он передается на суперкомпьютер по протоколу SFTP (SSH File Transfer Protocol). Для эффективного управления сеансами SSH и SFTP VibeCodeHPC использует Desktop Commander MCP [39] (особенно полезен для суперкомпьютеров, требующих двухфакторной аутентификации). Выполнение заданий контролируется планировщиками пакетных заданий PBS (Portable Batch System) и Slurm. VibeCodeHPC работает до тех пор, пока не будет достигнута поставленная пользователем цель, но при необходимости пользователь может в любой момент прервать процесс оптимизации. Также VibeCodeHPC допускает прямое взаимодействие с пользователем во время работы, что позволяет пользователю постепенно корректировать направление оптимизации.
4. Мониторинг и подтверждение результатов. Пользователь может отслеживать работу каждого агента в режиме реального времени: VibeCodeHPC автоматически создает логи, которые позволяют отслеживать ход оптимизации. После достижения поставленной цели создается итоговый отчет, в котором обобщаются процесс оптимизации, достигнутые результаты и примененные методы.

3. Автотюнер MLKAPS. Проведенный обзор показывает, что современные автотюнеры существенно различаются по множеству разных признаков (поддерживаемые модели параллельного программирования, сложность реализации, способы интеграции и прочее). Для проведения апробации в данной работе был выбран автотюнер MLKAPS, поскольку он ориентирован на широко используемые при разработке HPC-приложений и известные большему числу разработчиков модели параллельного программирования OpenMP и MPI. Кроме того, MLKAPS предоставляет понятное описание практического использования данного инструмента, достаточный набор примеров и не требует от разработчика узкоспециализированных знаний в области программирования или конкретной вычислительной архитектуры, что является важной составляющей для распространения автотюнеров среди более широкой аудитории разработчиков и удобства их внедрения в реальные проекты.

3.1. Описание. MLKAPS (Machine Learning for Kernel Accuracy and Performance Studies) — инструмент автоматической настройки, предназначенный для поиска оптимальных конфигураций параметров для различных входных данных и условий выполнения с использованием методов машинного обучения и адаптивного семплирования [12, 13]. Основная цель MLKAPS заключается в автоматизации процесса выбора параметров, обеспечивающих оптимальные значения заданной целевой метрики.

На вход MLKAPS подается описание двух типов параметров: входных параметров и параметров проектирования, а также вычислительное ядро, для которого производится оптимизация. В качестве ядра может выступать как отдельная функция, так и программный модуль. Пользователь также задает целевую функцию оптимизации, которой может быть, например, минимизация времени выполнения, минимизация энергопотребления или максимизация точности вычислений.

Результатом работы MLKAPS является дерево решений, определяющее выбор оптимальных значений параметров проектирования в зависимости от конкретных значений входных параметров. Полученное дерево решений генерируется в виде кода на языке C и может быть интегрировано непосредственно в исходный код параллельной программы.

Входные параметры соответствуют спецификации задачи и не могут быть настроены MLKAPS. Данные параметры или определяются пользователем (например, размеры матрицы), или соответству-

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

ют фиксированным параметрам среды выполнения (например, доступный объем памяти или количество ядер).

Оптимальные комбинации параметров проектирования, которые подбирает MLKAPS, могут относиться к ядру (например, вариант алгоритма или количество используемых потоков для версии OpenMP) или к среде выполнения (например, флаги компилятора или размещение потоков).

Настройка MLKAPS производится по следующим четырем этапам: семплирование, моделирование, оптимизация, построение дерева решения. Далее рассмотрим подробно каждый из них.

§ 3.1.1. Семплирование. Полный перебор всех возможных конфигураций параметров, как правило, является вычислительно дорогостоящим и практически неосуществимым. В связи с этим на этапе семплирования производится отбор ограниченного подмножества конфигураций, которые в дальнейшем используются для построения и обучения модели. MLKAPS поддерживает несколько стратегий семплирования:

- *Равномерное заполнение пространства параметров (space-filling).* Простейшая стратегия, обеспечивающая равномерное покрытие пространства конфигураций и одинаковую вероятность выбора каждой из них.
- *Иерархическое семплирование по дисперсии (HVS, Hierarchical Variance Sampling).* Итеративная стратегия, использующая метод LHS (Latin Hypercube Sampling) для формирования начальной выборки. Далее пространство параметров рекурсивно разбивается на области, для которых с использованием дерева решений оцениваются размер и дисперсия целевой метрики. Выборки распределяются по областям пропорционально произведению их размера на дисперсию. В [12] отмечено, что данная стратегия не видит разницы между отклонениями, вносимыми хорошими и плохими конфигурациями.
- *GA-адаптивное семплирование.* Стратегия, разработанная специально для MLKAPS, направленная на концентрацию вычислительных ресурсов в областях пространства параметров, содержащих перспективные конфигурации. В отличие от равномерного исследования всего пространства, данная стратегия преимущественно отбирает точки в окрестностях оптимальных решений.

Рассмотрим подробнее этап семплирования.

1. Входные данные.
   - $X = [x_1, \ldots, x_n]$, где X — пространство конфигураций, $x_1, \ldots, x_n$ — конфигурации. Отметим, что в конфигурации попадают как входные параметры, так и параметры проектирования. Например, если даны параметры $threads \in \{1, 2, 4, 8\}$, $block\_size \in \{16, 32, 64\}$, то $X = [(1, 16), (1, 32), \ldots, (8, 64)]$.
   - Целевая метрика (время выполнения программы, ускорение, энергопотребление и прочее).
   - “Бюджет” (допустимое количество запусков).
2. Ход этапа.

   В соответствии с выбранной стратегией семплирования производится отбор подмножества конфигураций $X' \subset X$. Основной целью этапа является минимизация числа реальных запусков программы при сохранении информативности выборки, необходимой для последующего построения модели.
3. Результат.

   $\text{Res} = \{(x, y) : x \in X', y \in Y\}$, где X' — подмножество множества X, Y — множество результатов целевой метрики, $y_1, \ldots, y_n$ — значения целевой метрики.

   Например, для заданной конфигурации x_1 после запуска целевая метрика время $y_1 = 15$ мс.

§ 3.1.2. Моделирование и оптимизация. На данном этапе производится построение и уточнение суррогатной модели, аппроксимирующей поведение оптимизируемой программы. В MLKAPS в качестве такой модели используется ансамбль деревьев решений с градиентным бустингом (Gradient Boosting Decision Trees, GBDT), реализованный в библиотеке LightGBM. С точки зрения машинного обучения данный этап формулируется как задача регрессии.

1. Входные данные.
   - Res — наблюдения, полученные на этапе семплирования;
   - тип целевой функции: максимизация или минимизация.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

2. Ход этапа.

   Целью этапа является построение суррогатной модели, которая приближает зависимость значения целевой метрики от конфигурации параметров. Модель не имеет доступа к исходному коду программы и обучается исключительно на основе наблюдаемых данных. Полученная аппроксимация позволяет быстро оценивать качество новых конфигураций без выполнения реальных запусков.

   В процессе обучения настраиваются гиперпараметры модели (например, параметры регуляризации или уровень шума в данных) таким образом, чтобы минимизировать расхождение между предсказаниями модели и наблюдаемыми значениями целевой метрики.

   После обучения модель используется для прогнозирования качества ранее не исследованных конфигураций. Выбранные лучшие конфигурации запускаются на реальной системе, а полученные результаты возвращаются в обучающую выборку, формируя механизм обратной связи и итеративного улучшения модели.

   Например, пусть даны:

   - $X' = [(1, 16), (2, 64), (8, 32)]$;
   - $Y = [200, 140, 160]$.

   На основе этих данных модель может выявить, что при числе потоков $t \leqslant 1$ и $t \geqslant 8$ значение целевой метрики ухудшается. Данные наблюдения передаются на этап *Оптимизации* для выбора новых конфигураций, которые затем проверяются экспериментально.

   Обучение модели может быть завершено, например, при выполнении одного из следующих условий:

   - сходимости гиперпараметров;
   - достижения заданного “бюджета”.

3. Результат.

   Обученная суррогатная модель.

§ 3.1.3. Построение дерева решений.

1. Входные данные.

   Обученная суррогатная модель.

2. Ход этапа.

   На основе всех накопленных данных строится итоговое дерево решений, учитывающее как входные параметры задачи, так и параметры проектирования. Это позволяет адаптировать выбор оптимальных параметров к различным условиям, например, различным размерам задачи или характеристикам архитектур вычислительных систем.

   Внутренние узлы дерева соответствуют условиям на входные параметры, тогда как листья содержат оптимальные значения параметров проектирования, рекомендуемые для соответствующих условий выполнения.

3. Результат.

   Финальная конфигурация, преобразованная в код на языке C.

3.2. Применение MLKAPS. Апробация автотюнера MLKAPS проводилась на базе суперкомпьютерного комплекса “Ломоносов-2” [1] на 5 вычислительных узлах, на каждом из которых по 14 вычислительных ядер. В качестве тестируемой задачи был взят широко известный тест HPL (High Performance Linpack) для настройки параметров исполнения MPI-приложения с 4 выделенными параметрами, приведенными в табл. 1.

На вход автотюнеру предоставлялись исходные параметры запуска теста и диапазоны допустимых значений параметров конфигурации, описанных в рамках конфигурационного файла в формате json. При этом модификация исходного кода HPL не производилась. Автотюнинг осуществлялся на уровне исполнения программы. В качестве целевой метрики использовалось время выполнения HPL. Запуски MLKAPS на суперкомпьютере и обработка HPL.dat описывались в нескольких bash-скриптах.

При доступных 70 вычислительных ядрах мы рассматриваем 64 допустимые комбинации P × Q, диапазон N в 30000 значений и 206 вариантов NB, что при всего 4 настраиваемых параметрах дает теоретический размер пространства поиска в 395 520 000 возможных конфигураций в рамках использования MLKAPS.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

Таблица 1. Параметры задачи

Table 1. Task parameters

Переменная Variable	Описание Description	Значение для MLKAPS Value for MLKAPS	Значение для скрипта Value for script
N	Размер матрицы (N × N) Matrix size (N × N)	[30000, 60000]	40000
NB	Размер блока (NB × NB) Block size (NB × NB)	[50, 256]	[64, 256] с шагом 8 [64, 256] with step 8
P	Количество строк в решетке процессов Number of rows in the process grid	[1, 8]	[1, 8]
Q	Количество столбцов в решетке процессов Number of columns in the process grid	[1, 8]	[1, 8]

Таблица 2. Результаты апробации MLKAPS для N = 40000

Table 2. Results of MLKAPS testing for N = 40000

Описание Description	MLKAPS	Скрипт Script
Размер охватываемого пространства конфигураций Size of the covered configuration space	395 520 000	1600
Время работы скриптов Script execution time	∼ 4 ч ∼ 4 h	∼ 16 ч ∼ 16 h
Время исполнения оптимума Optimum execution time	42.31 с 42.31 s	39.57 с 39.57 s
Оптимум NB Optimum NB	123	192
Оптимум P × Q Optimum P × Q	8 × 6	8 × 8

Для анализа работы MLKAPS проводились также обособленные запуски HPL через скрипт, который перебирает все параметры в указанных пределах, приведенных в табл. 1 в столбце “Значение для скрипта”. Поскольку полный перебор требует на порядки большее число ядро-часов процессорного времени, было принято решение рассматривать меньшие диапазоны значений N, NB, P, Q. Итого вышло 1600 возможных конфигураций.

Поскольку MLKAPS оперировал при N в диапазоне от 30000 до 60000, им было предложено несколько вариантов оптимальных конфигураций в зависимости от размера задачи. Но в связи с тем, что полным перебором охватить те же размеры задачи не представлялось возможным из-за ограниченных ресурсов, далее рассматривалась только конфигурация, предложенная для N = 40000.

Исходя из результатов в табл. 2, видно, что потеря ∼ 7% в эффективности (42.31 с против 39.57 с) компенсируется экономией используемого процессорного времени в 4 раза и рассмотрением в ∼ 250 тысяч большего размера пространства конфигураций.

4. Заключение. Проведен обзор актуальных в настоящее время автотюнеров, предназначенных для разных условий работы, задач и уровня погружения пользователя. Особое внимание в обзорах было уделено функциональности и особенностям работы каждого автотюнера.

Более подробно рассмотрен автотюнер MLKAPS и результаты работы с ним на суперкомпьютерном комплексе “Ломоносов-2”, на основе которых сделан вывод, что использование MLKAPS для настройки параметров теста HPL позволяет достичь компромисса, близкого к оптимальному с практической точки зрения. Жертвуя точностью, пользователь может сэкономить ресурсы суперкомпьютера и получить корректные рекомендации для целого диапазона размеров решаемой задачи.

Таким образом, проведенный анализ продемонстрировал, что в условиях ограниченных вычислительных квот и необходимости частого переконфигурирования приложений автотюнеры являются эффективным инструментом для решения задач суперкомпьютерного кодизайна, позволяя автоматизировать поиск оптимальных конфигураций с учетом не только программной реализации, но и особенностей конкретной вычислительной архитектуры.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

Список литературы

1. *Voevodin Vl.V., Antonov A.S., Nikitenko D.A., et al.* Supercomputer Lomonosov-2: large scale, deep monitoring and fine analytics for the user community // Supercomputing Frontiers and Innovations. 2019. **6**, No. 2. 4–11. doi 10.14529/jsfi190201.
2. *Antonov A.S., Maier R.V., Nikitenko D.A., Voevodin V.V.* An approach to solving the problem of supercomputer co-design // Lobachevskii J. Math. 2024. **45**, N 7. 2965–2973. doi 10.1134/S1995080224603680.
3. *Ang J.A.* High performance computing co-design strategies // MEMSYS'15: Proceedings of the 2015 International Symposium on Memory Systems, USA, Washington, October 5–8, 2015. New York: ACM, 2015. pp. 51–52. doi 10.1145/2818950.2818959.
4. *Kasim H., March V., Zhang R., See S.* Survey on parallel programming model // Network and Parallel Computing IFIP International Conference (NPC 2008), China, Shanghai, October 18–20, 2008. Berlin: Springer, 2008. pp. 266–275. doi 10.1007/978-3-540-88140-7_24.
5. *Романенко А.А.* Особенности адаптации программ под GPU с использованием технологии OpenACC. Новосибирск: РИЦ НГУ, 2016.
6. *Боресков А.В., Харламов А.А., Марковский Н.Д. и др.* Параллельные вычисления на GPU. Архитектура и программная модель CUDA. М.: Издательство Московского университета, 2015.
7. *Wu X., Kruse M., Balaprakash P., et al.* Autotuning PolyBench benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization // Concurrency and Computation: Practice and Experience. 2022. **34**, N 20. Article Number e6683. doi 10.1002/cpe.6683.
8. *Wu X., Balaprakash P., Kruse M., et al.* ytopt: autotuning scientific applications for energy efficiency at large scales // Concurrency and Computation: Practice and Experience. 2025. **37**, N 1. Article Number e8322. doi 10.1002/cpe.8322.
9. GitHub — ytopt-team/ytopt: ytopt: machine-learning-based autotuning and hyperparameter optimization framework using Bayesian Optimization · GitHub. https://github.com/ytopt-team/ytopt. Cited July 9, 2026.
10. *Park J., Shin Y., Lee J., et al.* HYPERF: end-to-end autotuning framework for high-performance computing // HPDC'25: Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing, USA, July 20–23, 2025. New York: ACM, 2025. pp. 1–14. doi 10.1145/3731545.3731588.
11. GitHub — SNU-CODElab/HYPERF · GitHub. https://github.com/SNU-CODElab/HYPERF. Cited July 9, 2026.
12. *Jam M.E., Petit E., de Oliveira Castro P., et al.* MLKAPS: machine learning and adaptive sampling for HPC kernel auto-tuning // ACM Trans. Archit. Code Optim. 2025. **22**, N 4. 1–22. doi 10.1145/3774418.
13. GitHub — MLCGO/MLKAPS: MLKAPS: Machine Learning for Kernel Accuracy and Performance Studies · GitHub. https://github.com/MLCGO/MLKAPS. Cited July 9, 2026.
14. *Nugteren C., Codreanu V.* CLTune: a generic auto-tuner for OpenCL kernels // 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, Italy, Turin, September 23–25, 2015. IEEE Press, 2015. pp. 195–202. doi 10.1109/MCSoC.2015.10.
15. GitHub — CNugteren/CLTune: CLTune: An automatic OpenCL & CUDA kernel tuner · GitHub. https://github.com/CNugteren/CLTune. Cited July 9, 2026.
16. *van Werkhoven B.* Kernel Tuner: a search-optimizing GPU code auto-tuner // Future Generation Computer Systems. 2019. **90**. 347–358. doi 10.1016/j.future.2018.08.004.
17. Design documentation — Kernel Tuner 1.3.1 documentation. https://kerneltuner.github.io/kernel_tuner/stable/design.html. Cited July 13, 2026.
18. GitHub — KernelTuner/kernel_tuner: Kernel Tuner · GitHub. https://github.com/KernelTuner/kernel_tuner. Cited July 13, 2026.
19. *Petrovič F., Střelák D., Hozzová J., et al.* A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with kernel tuning toolkit // Future Generation Computer Systems. 2020. **108**. 161–177. doi 10.1016/j.future.2020.02.069.
20. GitHub — HiPerCoRe/KTT: Kernel Tuning Toolkit · GitHub. https://github.com/HiPerCoRe/KTT. Cited July 13, 2026.
21. KTT — Kernel Tuning Toolkit. https://hipercore.github.io/KTT/index.html. Cited July 13, 2026.
22. *Whaley R.C., Dongarra J.J.* Automatically tuned linear algebra software // SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, USA, Orlando, November 7–13, 1998. IEEE Press, 1998. p. 38. doi 10.1109/SC.1998.10004.
23. Automatically Tuned Linear Algebra Software (ATLAS). https://math-atlas.sourceforge.net/. Cited July 13, 2026.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

24. *Hayes A.B., Li L., Chavarría-Miranda D., et al.* Orion: a framework for GPU occupancy tuning // Middleware'16: Proceedings of the 17th International Middleware Conference, Italy, Trento, December 12–16, 2016. New York: ACM, 2016. pp. 1–13. doi 10.1145/2988336.2988355.
25. *Mustafa D.* A survey of performance tuning techniques and tools for parallel applications // IEEE Access. 2022. **10**. 15036–15055. doi 10.1109/ACCESS.2022.3147846.
26. *Ashouri A.H., Killian W., Cavazos J., et al.* A survey on compiler autotuning using machine learning // ACM Computing Surveys (CSUR). 2018. **51**, N 5. 1–42. doi 10.1145/3197978.
27. *Balaprakash P., Dongarra J., Gamblin T., et al.* Autotuning in high-performance computing applications // Proceedings of the IEEE. 2018. **106**, N 11. 2068–2083. doi 10.1109/JPROC.2018.2841200.
28. *Fursin G., Miceli R., Lokhmotov A., et al.* Collective mind: towards practical and collaborative auto-tuning // Scientific Programming. 2014. **22**, N 4. 309–329. doi 10.3233/SPR-140396.
29. GitHub — gfursin/cm-ctuning-code-source: Collective Mind Repository (cTuning; Benchmarks) · GitHub. https://github.com/gfursin/cm-ctuning-code-source. Cited July 13, 2026.
30. *Willemsen F.-J., Schoonhoven R., Filipovič J., et al.* A methodology for comparing optimization algorithms for auto-tuning // Future Generation Computer Systems. 2024. **159**. 489–504. doi 10.1016/j.future.2024.05.021.
31. *Romein J.W., Veenboer B.* PowerSensor 2: a fast power measurement tool // 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), UK, Belfast, April 2–4, 2018. IEEE Press, 2018. pp. 111–113. doi 10.1109/ISPASS.2018.00020.
32. NVIDIA Management Library (NVML) | NVIDIA Developer. https://developer.nvidia.com/management-library-nvml. Cited July 14, 2026.
33. *Liu Y., Sid-Lakhdar W.M., Marques O., et al.* GPTune: multitask learning for autotuning exascale applications // PPoPP'21: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Republic of Korea, February 27, 2021. New York: ACM, 2021. pp. 234–246. doi 10.1145/3437801.3441621.
34. *Cho Y., Demmel J.W., Dinh G., et al.* GPTune User Guide. https://gptune.lbl.gov/documentation/gptune-user-guide/. Cited July 16, 2026.
35. GitHub — gptune/GPTune · GitHub. https://github.com/gptune/GPTune. Cited July 14, 2026.
36. *Hayashi S., Morita K., Mukunoki D., et al.* VibeCodeHPC: an agent-based iterative prompting auto-tuner for HPC code generation using LLMs. 2025. doi 10.48550/arXiv.2510.00031.
37. GitHub — Katagiri-Hoshino-Lab/VibeCodeHPC: CLI-based multi-agents for auto-tuning (e.g. HPC code optimazation loops) supporting local LLMs · GitHub. https://github.com/Katagiri-Hoshino-Lab/VibeCodeHPC. Cited July 14, 2026.
38. GitHub — anthropics/claude-code: Claude Code is an agentic coding tool that lives in your terminal, understands your codebase, and helps you code faster by executing routine tasks, explaining complex code, and handling git workflows - all through natural language commands · GitHub. https://github.com/anthropics/claude-code. Cited July 14, 2026.
39. GitHub — wonderwhy-er/DesktopCommanderMCP: This is MCP server for Claude that gives it terminal control, file system search and diff file editing capabilities · GitHub. https://github.com/wonderwhy-er/DesktopCommanderMCP. Cited July 14, 2026.

Получена 29 апреля 2026 г. *Принята* 22 июня 2026 г. *Опубликована* 20 июля 2026 г.

Информация об авторах

Татьяна Евгеньевна Загоруйко — студент магистратуры; Московский государственный университет имени М. В. Ломоносова, Научно-исследовательский вычислительный центр, Ленинские горы, 1, стр. 4, 119234, Москва, Российская Федерация.

Александр Сергеевич Антонов — к.ф.-м.н., вед. науч. сотр.; Московский государственный университет имени М. В. Ломоносова, Научно-исследовательский вычислительный центр, Ленинские горы, 1, стр. 4, 119234, Москва, Российская Федерация.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

References

1. Vl. V. Voevodin, A. S. Antonov, D. A. Nikitenko, et al., "Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community," Supercomput. Front. Innov. **6** (2), 4–11 (2019). doi 10.14529/jsfi190201.
2. A. S. Antonov, R. V. Maier, D. A. Nikitenko, and V. V. Voevodin, "An Approach to Solving the Problem of Supercomputer Co-design," Lobachevskii J. Math. **45** (7), 2965–2973 (2024). doi 10.1134/S1995080224603680.
3. J. A. Ang, "High Performance Computing Co-Design Strategies," in *MEMSYS'15: Proceedings of the 2015 International Symposium on Memory Systems, USA, Washington, October 5–8, 2015* (ACM, New York, 2015), pp. 51–52. doi 10.1145/2818950.2818959.
4. H. Kasim, V. March, R. Zhang, and S. See, "Survey on Parallel Programming Model," in *Network and Parallel Computing IFIP International Conference (NPC 2008), China, Shanghai, October 18–20, 2008* (Springer, Berlin, 2008), pp. 266–275. doi 10.1007/978-3-540-88140-7_24.
5. A. A. Romanenko, *Features of Software Adaptation for GPU Using OpenACC Technology* (RIC NGU, Novosibirsk, 2016) [in Russian].
6. A. V. Boreskov, A. A. Kharlamov, N. D. Markovsky, et al., *Parallel computing on the GPU. CUDA Architecture and Software model* (Izdat. Moskow. Universiteta, Moscow, 2015) [in Russian].
7. X. Wu, M. Kruse, P. Balaprakash, et al., "Autotuning PolyBench Benchmarks with LLVM Clang/Polly Loop Optimization Pragmas Using Bayesian Optimization," Concurrency and Computation: Practice and Experience **34** (20), Article Number e6683 (2022). doi 10.1002/cpe.6683.
8. X. Wu, P. Balaprakash, M. Kruse, et al., "ytopt: Autotuning Scientific Applications for Energy Efficiency at Large Scales," Concurrency and Computation: Practice and Experience **37** (1), Article Number e8322 (2025). doi 10.1002/cpe.8322.
9. GitHub — ytopt-team/ytopt: ytopt: machine-learning-based autotuning and hyperparameter optimization framework using Bayesian Optimization · GitHub. https://github.com/ytopt-team/ytopt. Cited July 9, 2026.
10. J. Park, Y. Shin, J. Lee, et al., "HYPERF: End-to-End Autotuning Framework for High-Performance Computing," in *HPDC'25: Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing, USA, July 20–23, 2025* (ACM, New York, 2025), pp. 1–14. doi 10.1145/3731545.3731588.
11. GitHub — SNU-CODElab/HYPERF · GitHub. https://github.com/SNU-CODElab/HYPERF. Cited July 9, 2026.
12. M. E. Jam, E. Petit, P. de Oliveira Castro, et al., "MLKAPS: Machine Learning and Adaptive Sampling for HPC Kernel Auto-tuning," ACM Trans. Archit. Code Optim. **22** (4), 1–22 (2025). doi 10.1145/3774418.
13. GitHub — MLCGO/MLKAPS: MLKAPS: Machine Learning for Kernel Accuracy and Performance Studies · GitHub. https://github.com/MLCGO/MLKAPS. Cited July 9, 2026.
14. C. Nugteren and V. Codreanu, "CLTune: A Generic Auto-Tuner for OpenCL Kernels," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, Italy, Turin, September 23–25, 2015* (IEEE Press, 2015), pp. 195–202. doi 10.1109/MCSoC.2015.10.
15. GitHub — CNugteren/CLTune: CLTune: An automatic OpenCL & CUDA kernel tuner · GitHub. https://github.com/CNugteren/CLTune. Cited July 9, 2026.
16. B. van Werkhoven, "Kernel Tuner: A search-optimizing GPU code auto-tuner," Future Generation Computer Systems **90**, 347–358 (2019). doi 10.1016/j.future.2018.08.004.
17. Design documentation — Kernel Tuner 1.3.1 documentation. https://kerneltuner.github.io/kernel_tuner/stable/design.html. Cited July 13, 2026.
18. GitHub — KernelTuner/kernel_tuner: Kernel Tuner · GitHub. https://github.com/KernelTuner/kernel_tuner. Cited July 13, 2026.
19. F. Petrovič, D. Střelák, J. Hozzová, et al., "A Benchmark Set of Highly-Efficient CUDA and OpenCL Kernels and its Dynamic Autotuning with Kernel Tuning Toolkit," Future Generation Computer Systems **108**, 161–177 (2020). doi 10.1016/j.future.2020.02.069.
20. GitHub — HiPerCoRe/KTT: Kernel Tuning Toolkit · GitHub. https://github.com/HiPerCoRe/KTT. Cited July 13, 2026.
21. KTT — Kernel Tuning Toolkit. https://hipercore.github.io/KTT/index.html. Cited July 13, 2026.
22. R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, USA, Orlando, November 7–13, 1998* (IEEE Press, 1998), p. 38. doi 10.1109/SC.1998.10004.
23. Automatically Tuned Linear Algebra Software (ATLAS). https://math-atlas.sourceforge.net/. Cited July 13, 2026.

2026, **27** (3), 417–433. doi 10.26089/NumMet.v27r327

24. A. B. Hayes, L. Li, D. Chavarría-Miranda, et al., "Orion: A Framework for GPU Occupancy Tuning," in *Middleware'16: Proceedings of the 17th International Middleware Conference, Italy, Trento, December 12–16, 2016* (ACM, New York, 2016), pp. 1–13. doi 10.1145/2988336.2988355.
25. D. Mustafa, "A Survey of Performance Tuning Techniques and Tools for Parallel Applications," IEEE Access **10**, 15036–15055 (2022). doi 10.1109/ACCESS.2022.3147846.
26. A. H. Ashouri, W. Killian, J. Cavazos, et al., "A Survey on Compiler Autotuning Using Machine Learning," ACM Computing Surveys (CSUR). **51** (5), 1–42 (2018). doi 10.1145/3197978.
27. P. Balaprakash, J. Dongarra, T. Gamblin, et al., "Autotuning in High-Performance Computing Applications," Proceedings of the IEEE. **106** (11), 2068–2083 (2018). doi 10.1109/JPROC.2018.2841200.
28. G. Fursin, R. Miceli, A. Lokhmotov, et al., "Collective mind: Towards practical and collaborative auto-tuning," Scientific Programming **22** (4), 309–329 (2014). doi 10.3233/SPR-140396.
29. GitHub — gfursin/cm-ctuning-code-source: Collective Mind Repository (cTuning; Benchmarks) · GitHub. https://github.com/gfursin/cm-ctuning-code-source. Cited July 13, 2026.
30. F.-J. Willemsen, R. Schoonhoven, J. Filipovič, et al., "A methodology for comparing optimization algorithms for auto-tuning," Future Generation Computer Systems **159**, 489–504 (2024). doi 10.1016/j.future.2024.05.021.
31. J. W. Romein and B. Veenboer, "PowerSensor 2: A Fast Power Measurement Tool," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), UK, Belfast, April 2–4, 2018* (IEEE Press, 2018), pp. 111–113. doi 10.1109/ISPASS.2018.00020.
32. NVIDIA Management Library (NVML) | NVIDIA Developer. https://developer.nvidia.com/management-library-nvml. Cited July 14, 2026.
33. Y. Liu, W. M. Sid-Lakhdar, O. Marques, et al., "GPTune: Multitask Learning for Autotuning Exascale Applications," in *PPoPP'21: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Republic of Korea, February 27, 2021* (ACM, New York, 2021), pp. 234–246. doi 10.1145/3437801.3441621.
34. Y. Cho, J. W. Demmel, G. Dinh, et al., "GPTune user guide," https://gptune.lbl.gov/documentation/gptune-user-guide/. Cited July 16, 2026.
35. GitHub — gptune/GPTune · GitHub. https://github.com/gptune/GPTune. Cited July 14, 2026.
36. S. Hayashi, K. Morita, D. Mukunoki, et al., "VibeCodeHPC: An Agent-Based Iterative Prompting Auto-Tuner for HPC Code Generation Using LLMs," (2025). doi 10.48550/arXiv.2510.00031.
37. GitHub — Katagiri-Hoshino-Lab/VibeCodeHPC: CLI-based multi-agents for auto-tuning (e.g. HPC code optimazation loops) supporting Local LLMs · GitHub. https://github.com/Katagiri-Hoshino-Lab/VibeCodeHPC. Cited July 14, 2026.
38. GitHub — anthropics/claude-code: Claude Code is an agentic coding tool that lives in your terminal, understands your codebase, and helps you code faster by executing routine tasks, explaining complex code, and handling git workflows - all through natural language commands · GitHub. https://github.com/anthropics/claude-code. Cited July 14, 2026.
39. GitHub — wonderwhy-er/DesktopCommanderMCP: This is MCP server for Claude that gives it terminal control, file system search and diff file editing capabilities · GitHub. https://github.com/wonderwhy-er/DesktopCommanderMCP. Cited July 14, 2026.

Received	*Accepted*	*Published*
April 29, 2026	*June 22, 2026*	*July 20, 2026*

Information about the authors

Tatiana E. Zagoruiko — Master's Student; Lomonosov Moscow State University, Research Computing Center, Leninskie Gory, 1, building 4, 119234, Moscow, Russia.

Alexander S. Antonov — Ph.D., Leading Researcher; Lomonosov Moscow State University, Research Computing Center, Leninskie Gory, 1, building 4, 119234, Moscow, Russia.