


Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

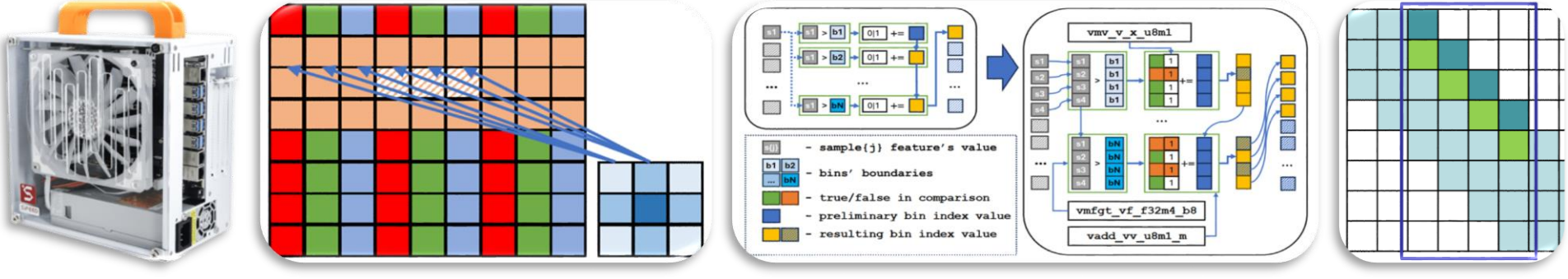
Кафедра высокопроизводительных вычислений и системного программирования

Лаборатория ITLab

Нижегородский государственный университет им. Н.И. Лобачевского
Институт информационных технологий, математики и механики
Кафедра высокопроизводительных вычислений и системного программирования
Лаборатория ITLab

Введение в анализ производительности и оптимизацию программ

Оптимизация программ для процессоров RISC-V. Примеры



Иосиф Мееров, Валентин Волокитин
+ Васильев Е., Воденеева А., Козинов Е., Кустикова В., Линев А., Панова Е., Пирова А., Сысоев А. и др.

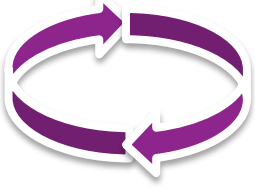
Содержание

- Предметная область
- Мотивация
- Цель лекции
- Инфраструктура
- Оптимизация некоторых алгоритмов библиотеки **OpenCV**
- Оптимизация пакета машинного обучения **CatBoost**
- Оптимизация базовых алгоритмов линейной алгебры в **OpenBLAS**
- Адаптация прямого решателя **MUMPS**
- Основные выводы
- Вместо заключения

Предметная область

Анализ производительности и оптимизация программ

- Что такое **оптимизация программ**?
- **Цель и критерии**
  - скорость
  - память
  - сложность разработки
- **Жизненный цикл**
  - бенчмарки и инфраструктура
  - эксперименты
  - выявление узких мест
  - анализ и улучшения
  - оценка результата



Что нужно знать?

- **Математика**
- **Алгоритмы и структуры данных**
- **Архитектуры, ОС, трансляторы**
- **Параллельные вычисления**
- **Инструментарий**
  - **компиляторы**
  - **профилировщики**
  - **отладчики**
  - **библиотеки**

Методика почти не зависит от архитектуры

Особенности междисциплинарных проектов

Предметная область. Цель лекции

Наша кафедра

- ***Тематика:*** **разработка наукоемкого ПО в междисциплинарных проектах, анализ и оптимизация производительности**
  - Вычислительна физика (лазерная физика, квантовая динамика…)
  - Вычислительная биология
  - Финансовая математика
  - Компьютерная графика, компьютерное зрение, ML & DL
  - Алгоритмы на графах и разреженная алгебра
- ***В основном ориентировано на x86-64 (C, C++, SYCL, Kokkos, Parallel Studio…)***

Почему мы заинтересовались устройствами RISC-V?

Мотивация

Почему мы заинтересовались устройствами RISC-V?

- Свободная, открытая, перспективная архитектура
- Первые устройства RISC-V *достаточно* легко использовать
- Системное ПО приемлемого качества и постоянно развивается
- Программные библиотеки *часто* могут быть построены и использованы без модификаций (*достижение высокой производительности может быть непростым делом*)

RISC-V это глоток свежего воздуха!

- **Ложка дегтя:**
  - Не хватает инструментов для анализа производительности
  - Не хватает документации и методических статей
  - Не хватает устройств HPC-класса

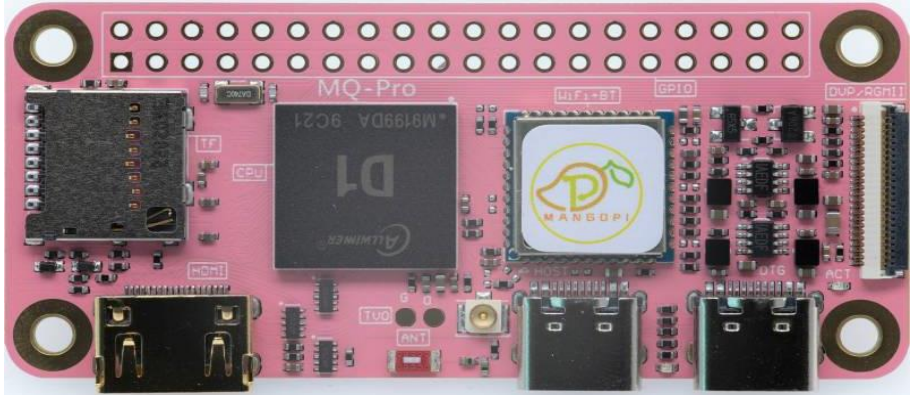
Цель лекции

- **Цель лекции** – рассмотреть различные приемы оптимизации программ для процессоров архитектуры RISC-V
  - Изложение ведется на конкретных примерах
  - Все примеры взяты из научных проектов кафедры
  - Лекция носит обзорный характер, далее каждый из примеров рассматривается детально
  - Из лекции можно понять, как особенности архитектуры RISC-V влияют на производительность, и как их использовать для оптимизации конкретных приложений

Инфраструктура

Инфраструктура: Mango Pi

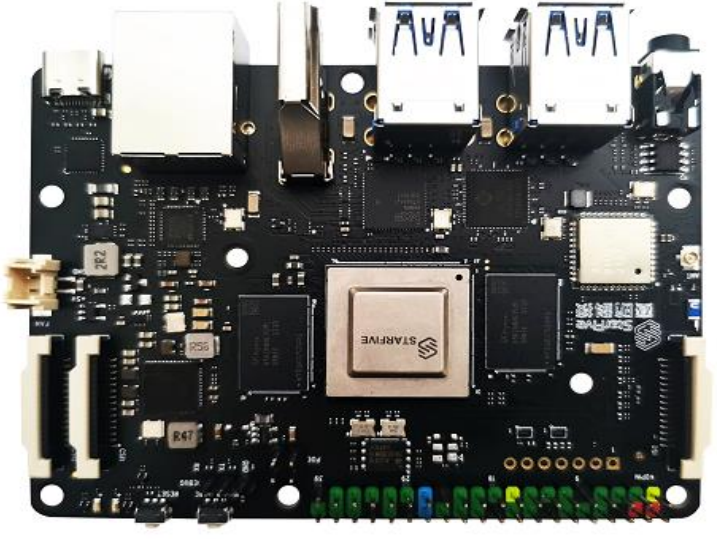
- Mango Pi MQ-Pro (D1)
  - Процессор Allwinner D1 (1 x XuanTie C906, 1GHz)
  - 1GB DDR3L RAM
  - Операционная система Ubuntu 22.10 (RISC-V edition)
  - Компилятор GCC 12.2



* рисунок https://mangopi.org/mangopi_mqpro

Инфраструктура: StarFive VisionFive (v1)

- StarFive VisionFive (v1)
  - Процессор StarFive JH7100 (2 x StarFive U74, 1 GHz)
  - 8 GB LPDDR4 RAM
  - Операционная система Ubuntu 22.10 (RISC-V edition)
  - Компилятор GCC 12.2



* рисунок https://www.starfivetech.com/en/site/boards

Инфраструктура: LicheePi 4A

- LicheePi 4A (TH1520)
  - Процессор T-Head TH1520 (4 x XuanTie C910, 1.85 (2.5) GHz, RVV 0.7.1)
  - 8-16GB LDDR4X RAM
  - ОС Debian 12 (RISC-V edition), компилятор GCC 12.2

* рисунок https://sipeed.com/licheepi4a

Инфраструктура: Banana Pi

- SpacemiT K1 X60
  - Процессор SpacemiT K1 (8 x X60, 2 GHz, 256-bit RVV 1.0)
  - 16 GB RAM
  - Операционная система Bianbu 1.0.15
  - Компилятор GCC RISC-V 14.2.0

https://www.spacemit.com/en/spacemit-x60-core/

Инфраструктура: Intel Xeon

- Сервер с Intel Xeon
  - Процессор 2xIntel Xeon 4310T (2 x 10 Ice Lake, up to 3.4 GHz)
  - 64 GB DDR4 RAM
  - Операционная система CentOS 7
  - Компилятор GCC 9.5

Суперкомпьютер «Лобачевский»

- НОЦ СКТ-Приволжье
- Суперкомпьютер «Лобачевский» – основа для выполнения междисциплинарных проектов

Часть I.
Оптимизация некоторых алгоритмов библиотеки OpenCV для RISC-V

Обзор OpenCV

- **OpenCV** – библиотека алгоритмов компьютерного зрения с открытым исходным кодом
- Модульная архитектура
- Концепция **универсальных векторных интринсиков**:
  - Объявлен набор функций (intrinsics), работающих с векторами
  - Существует ***базовая реализация*** с использованием обычных циклов (может быть автоматически векторизована компилятором)
  - Для каждой платформы векторы преобразуются в стандартные типы векторов платформы
  - Для каждой платформы универсальные встроенные функции реализуются с использованием встроенных функций платформы
- **Облегчается адаптация для конкретных платформ!**

Изменения кода

До изменений (~3000 строк)

Длина вектора 128 бит

Типичная реализация универсальных интринсиков:
vfmadd_vv_f32m1

После изменений (~5000 строк)

Длина вектора 512 бит

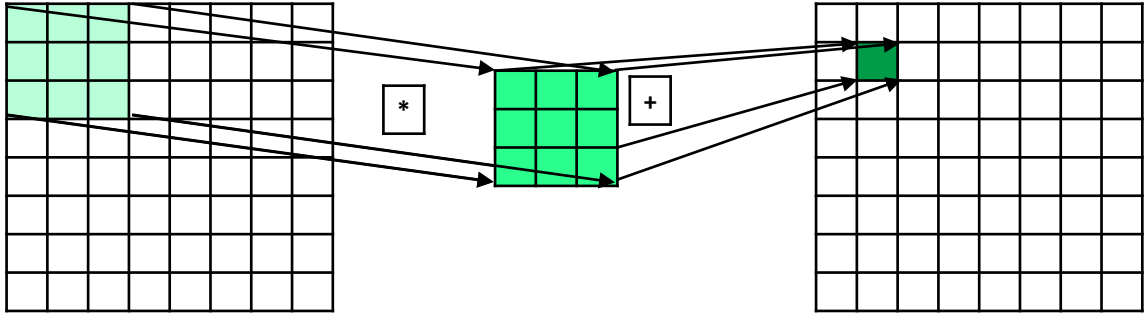
Типичная реализация универсальных интринсиков:
vfmadd_vv_f32m4

Также добавлены преобразования типов m8->m1->m4, так как в стандартной реализации нет прямого преобразования m8->m4

- **Метрики оценки производительности:**
  - Время работы
  - Ускорение от векторизации

Алгоритм фильтрации в OpenCV

- В данном тесте мы использовали общий **алгоритм фильтрации**
- OpenCV имеет оптимизацию для специальных фильтров (фильтр Гаусса, медианный фильтр и т. д.)
- В OpenCV для ядер размером больше 11 используется алгоритм с преобразованием Фурье. Для ядер меньшего размера используется стандартный оптимизированный алгоритм
- Параллельная версия отсутствует



Фильтрация в OpenCV. Результаты

Mango Pi

Kernel size	SeqScalar	SeqVector	**Optim**	Vectorization speedup	**Optimization speedup**
3x3	1,26	0,69	**0,76**	1,84	**0,90**
5x5	2,61	1,42	**1,79**	1,84	**0,80**
7x7	4,34	2,72	**3,29**	1,59	**0,83**
9x9	6,52	6,00	**5,97**	1,09	**1,01**
11x11	6,55	6,26	**5,98**	1,05	**1,05**
13x13	6,55	6,08	**6,00**	1,08	**1,01**

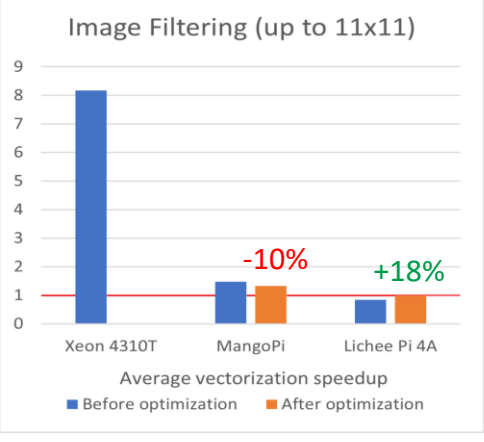
Lichee Pi

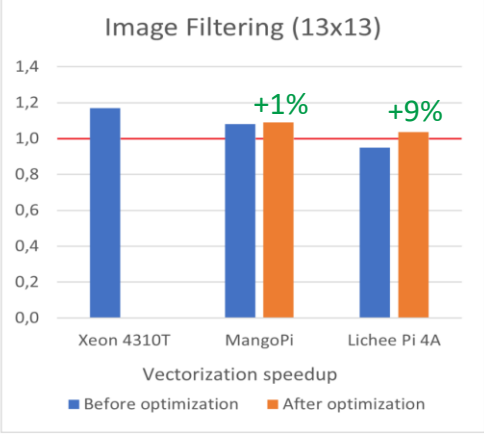
Kernel size	SeqScalar	SeqVector	**Optim**	Vectorization speedup	**Optimization speedup**
3x3	0,19	0,20	**0,14**	0,97	**1,41**
5x5	0,31	0,44	**0,44**	0,71	**1,00**
7x7	0,48	0,78	**0,71**	0,62	**1,11**
9x9	0,61	0,64	**0,59**	0,95	**1,08**
11x11	0,61	0,65	**0,60**	0,94	**1,09**
13x13	0,61	0,65	**0,60**	0,95	**1,09**

FullHD 1920x1080

x86

Kernel size	SeqScalar	SeqVector	Vectorization speedup
3x3	0,058	0,010	6,06
5x5	0,168	0,022	7,70
7x7	0,376	0,040	9,35
9x9	0,641	0,065	9,89
11x11	1,071	0,095	11,24
13x13	0,177	0,152	1,17





Алгоритм эрозии в OpenCV

- Эрозия выбирает минимум из некоторой области:

$$\texttt{dst}(x, y) = \min_{(x',y')\colon \texttt{element}(x',y')\neq 0} \texttt{src}(x + x', y + y')$$

- Эрозия работает с одноканальными изображениями ⇒ последовательный доступ к памяти

- Параллельная версия отсутствует

Алгоритм эрозии в OpenCV. Результаты

Mango Pi

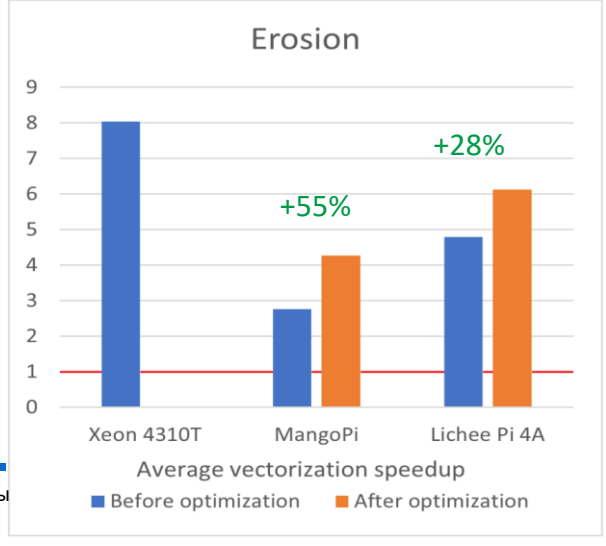
Resolution	Filter size	SeqScalar	SeqVector	Optim	Vectorization speedup	Optimization speedup
1920x1080	1	0,163	0,060	0,043	**2,71**	**1,40**
1920x1080	2	0,246	0,073	0,048	**3,37**	**1,51**
1920x1080	3	0,320	0,084	0,054	**3,79**	**1,57**
3840x2160	1	0,586	0,242	0,157	**2,42**	**1,54**
3840x2160	2	0,900	0,297	0,180	**3,03**	**1,65**
3840x2160	3	1,189	0,361	0,201	**3,29**	**1,79**

Lichee Pi

Resolution	Filter size	SeqScalar	SeqVector	Optim	Vectorization speedup	Optimization speedup
1920x1080	1	0,020	0,004	0,004	**4,56**	**1,17**
1920x1080	2	0,031	0,005	0,004	**5,91**	**1,27**
1920x1080	3	0,049	0,006	0,004	**7,93**	**1,37**
3840x2160	1	0,060	0,017	0,015	**3,47**	**1,17**
3840x2160	2	0,095	0,023	0,016	**4,21**	**1,39**
3840x2160	3	0,147	0,026	0,018	**5,56**	**1,44**

x86

Resolution	Filter size	SeqScalar	SeqVector	Vectorization speedup
1920x1080	1	0,007	0,001	**5,17**
1920x1080	2	0,010	0,001	**6,78**
1920x1080	3	0,013	0,001	**8,60**
3840x2160	1	0,025	0,005	**5,59**
3840x2160	2	0,037	0,005	**7,61**
3840x2160	3	0,050	0,005	**9,92**



Алгоритмы Bag-of-words и SVM в OpenCV…

- **Обучение** включает в себя следующие этапы:
  - Обнаружение ключевых точек на каждом изображении набора обучающих данных (SIFT)
  - Построение дескрипторов ключевых точек
  - Кластеризация дескрипторов ключевых точек. В результате кластеризации находятся центры построенных кластеров
  - Построение нормализованной гистограммы встречаемости «слов» для каждого обучающего изображения
  - Обучение классификатора (SVM) с использованием рассчитанного описания признаков изображения

Алгоритмы Bag-of-words и SVM в OpenCV…

- **Тестирование** включает в себя следующие этапы:
  - Обнаружение ключевых точек с использованием того же алгоритма
  - Построение нормализованной гистограммы встречаемости «слов» для каждого изображения тестовой выборки. Словарь получается на этапе обучения
  - Прогнозирование класса изображения с использованием обученной модели SVM

Алгоритмы Bag-of-words и SVM в OpenCV. Результаты

Mango Pi

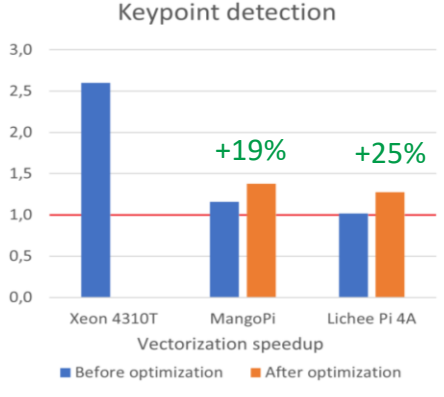
SVM step	SeqScalar	SeqVector	Optim	**Vectorization speedup**	**Optimization speedup**
keypoint detection	269,67	231,86	194,38	**1,16**	**1,19**
feature generation	388,95	338,99	298,82	**1,15**	**1,13**
prediction	3,27	3,25	3,24	**1,01**	**1,00**

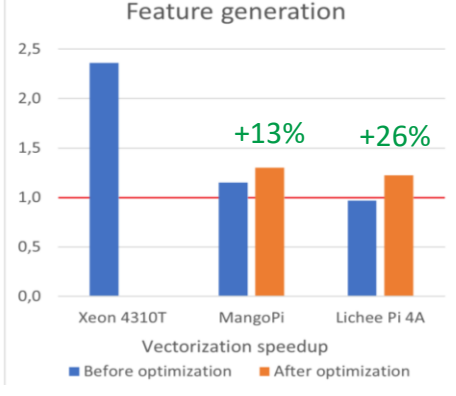
x86

SVM step	SeqScalar	ParScalar	SeqVector	ParVector	**Vectorization speedup**
keypoint detection	9,21	9,07	3,54	3,84	**2,60**
feature generation	10,85	8,63	5,10	3,67	**2,36**
prediction	0,06	0,06	0,06	0,06	**0,99**

Lichee Pi

SVM step	SeqScalar	ParScalar	SeqVector	ParVector	Optim	**Vectorization speedup**	**Optimization speedup**
keypoint detection	26,82	24,59	25,71	24,16	19,27	**1,02**	**1,25**
feature generation	36,02	25,96	36,71	26,76	21,30	**0,97**	**1,26**
prediction	0,30	0,30	0,30	0,30	0,30	**1,00**	**1,00**





Часть II.
Оптимизация пакета CatBoost для RISC-V

Цель

Ключевой вопрос: как оценивать производительность больших программных комплексов и какие техники могут улучшить производительность на устройствах RISC-V?

- Рассматриваем **CatBoost**, один из широко распространенных пакетов ML, основанный на *деревьях решений*
- Рассматриваем алгоритмы CatBoost как **«черный ящик»**, не погружаясь глубоко в логику алгоритмов (*типично для performance engineering)*
- Фокусируемся на **аспектах производительности** и ищем пути **ускорить код**, *обнаружив* и *векторизовав* основные **вычислительно-трудоемкие циклы** при работе на выбранных наборах данных

Цель

Ключевой вопрос: как оценивать производительность больших программных комплексов и какие техники могут улучшить производительность на устройствах RISC-V?

- Рассматриваем алгоритмы CatBoost как **«черный ящик»**, не погружаясь глубоко в логику алгоритмов (*типично для performance engineering)*
- Фокусируемся на **аспектах производительности** и ищем способы **ускорить код**, *обнаружив* и *векторизовав* основные **вычислительно-трудоемкие циклы** при работе на выбранных наборах данных

Чем обусловлен такой план?

Что такое CatBoost?

- **CatBoost** реализует алгоритм, основанный на ***градиентном бустинге деревьев решений***, пакет разработан **Yandex**
- Используется в *поисковых и рекомендательных системах,* персональных помощниках, ПО для автомобилей и др., в частности, в Yandex, CERN…
- Библиотека с *открытым исходным кодом* (Apache 2.0)
- Решает задачи классификации, регрессии и др.
- Python, R, Java и C++ APIs
- Оптимизирован для вычислений на CPU и GPU
- Включает средства для тренировки и вывода моделей, анализа качества и визуализации

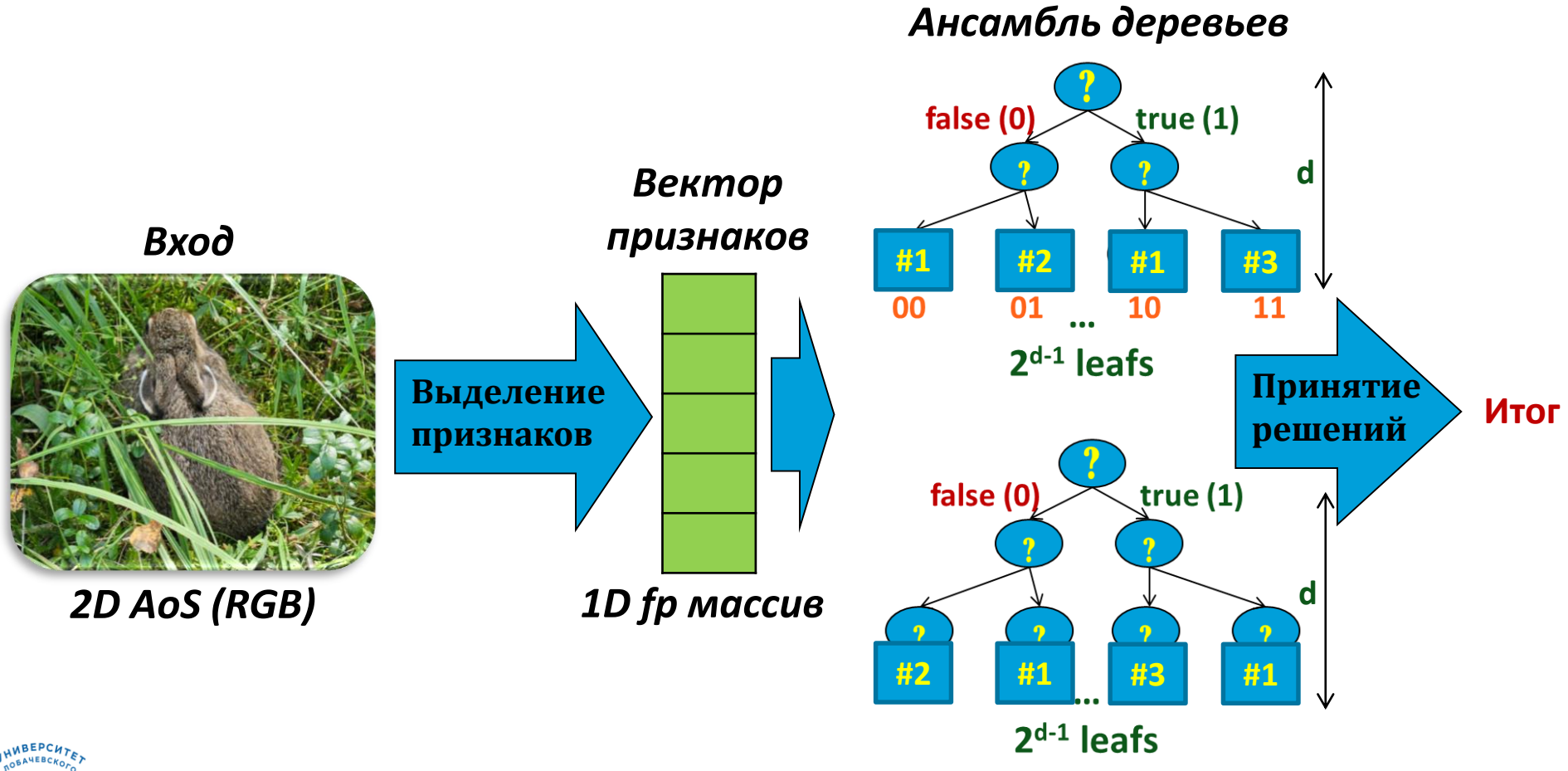
Градиентный бустинг деревьев решений

- **CatBoost реализует градиентный бустинг деревьев решений**

- **Идея бустинга** – построить сильные предсказательные модели, используя ансамбли слабых моделей
- Алгоритм использует «забывчивые деревья решений» (*oblivious decision trees)* с малой глубиной в качестве слабых моделей
- **Забывчивое дерево** – упрощенная модель дерева решений, где все узлы на одном уровне проверяют одно и то же условие
- В ходе тренировки, деревья добавляются к ансамблю, и каждое дерево пытается исправить ошибки предыдущего

Алгоритм CatBoost

Пример:
задача классификации



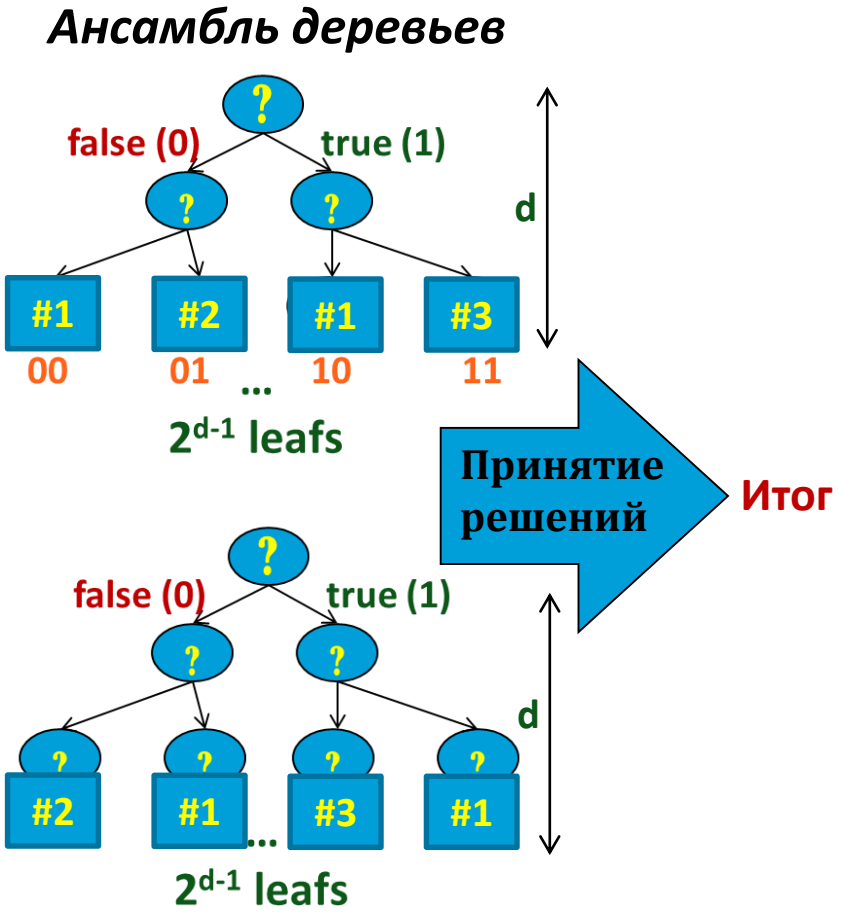
Алгоритм CatBoost

Пример:
задача классификации

Забывчивое дерево **это упрощенное дерево (на одном уровне все узлы содержат одно и то же условие)**

Оптимизация (реализовано Yandex)
- не нужно хранить деревья напрямую
- не нужны традиционные проверки
! Достаточно отсортировать все натренированные пороги, сделать бинаризацию и использовать битовые операции, чтобы найти результирующий лист дерева

Наша цель: ***улучшить производительность на RISC-V CPUs, не меняя алгоритм и структуры данных***



Наборы данных

- Используем несколько распространенных наборов данных
- Нет гарантии, что набор *репрезентативен*, но, по крайней мере, мы покрываем несколько разных сценариев

Набор данных	Строки/Столбцы	Число классов	Функция потерь	Темп обучения	Глубина дерева
MQ2008	9630 x 46	-	YetiRank	0.02	6
Santander customer transaction	400k x 202	2	LogLoss	0.01	1
Covertype	464.8k x 54	7	MultiClass	0.50	8
YearPredictionMSD	515k x 90	-	MAE	0.30	6
image-embeddings	5649 x 512	20	MultiClass	0.05	4

Наборы данных (краткое описание)

- *“Covertype”*
  - 52 целочисленных и бинарных признака, представляющих дикие местности и типы почв
  - требуется прогнозировать тип лесного покрова (7 классов)
  - набор данных был разделен на обучающую и тестовую выборку в соотношении 70:30
- *“Santander customer transaction”*
  - 200 ненормализованных признаков, бинарная классификация
  - тренировочная и тестовая выборки имеют по 200 000 элементов каждая
- *“YearPredictionMSD”*
  - 90 ненормализованных признаков, извлеченных из песен
  - требуется прогнозировать год выхода песни
  - 463 715 сэмплов в тренировочной выборке и 51 630 в тестовой

Наборы данных (краткое описание)

- “MQ2008”
  - 46 признаков для решения задачи ранжирования
  - содержит 9 630 тренировочных сэмплов и 2 874 тестовых сэмплов
- “image-embeddings”
  - подмножество набора данных PASCAL VOC 2007 (20 классов)
  - не содержит изображений с объектами нескольких классов
  - содержит 9 630 тренировочных сэмплов и 2 874 тестовых сэмплов
  - эмбеддинги генерируются для изображений с использованием натренированной модели *resnet34* из библиотеки TorchVision
  - для получения эмбеддингов, из модели убран последний классификационный слой

Возможности для оптимизации

- ***x86 CPUs***: множество средств для анализа и оптимизации оптимизации (VTune, Advisor, компиляторы C, C++, Fortran…)
- ***RISC-V CPUs***:
  - Компиляторы на ранней стадии разработки (*весьма неплохого качества, тем не менее*)
  - Набор средств для оптимизации существенно ограничен
  - Их функциональность скудная (могут лишь искать хотспоты)
- Как оптимизировать большие коды? *(стандартно для perf. eng.)*
  - Найти хотспоты (вычислительно-трудоемкие участки кода)
  - Попробовать векторизовать вычислительные циклы
  - Проверить и улучшить эффективность распараллеливания
- ***Что мы сделали***: вручную векторизовали код на **RVV 0.7.1**

Как найти хотспоты на устройствах RISC-V?

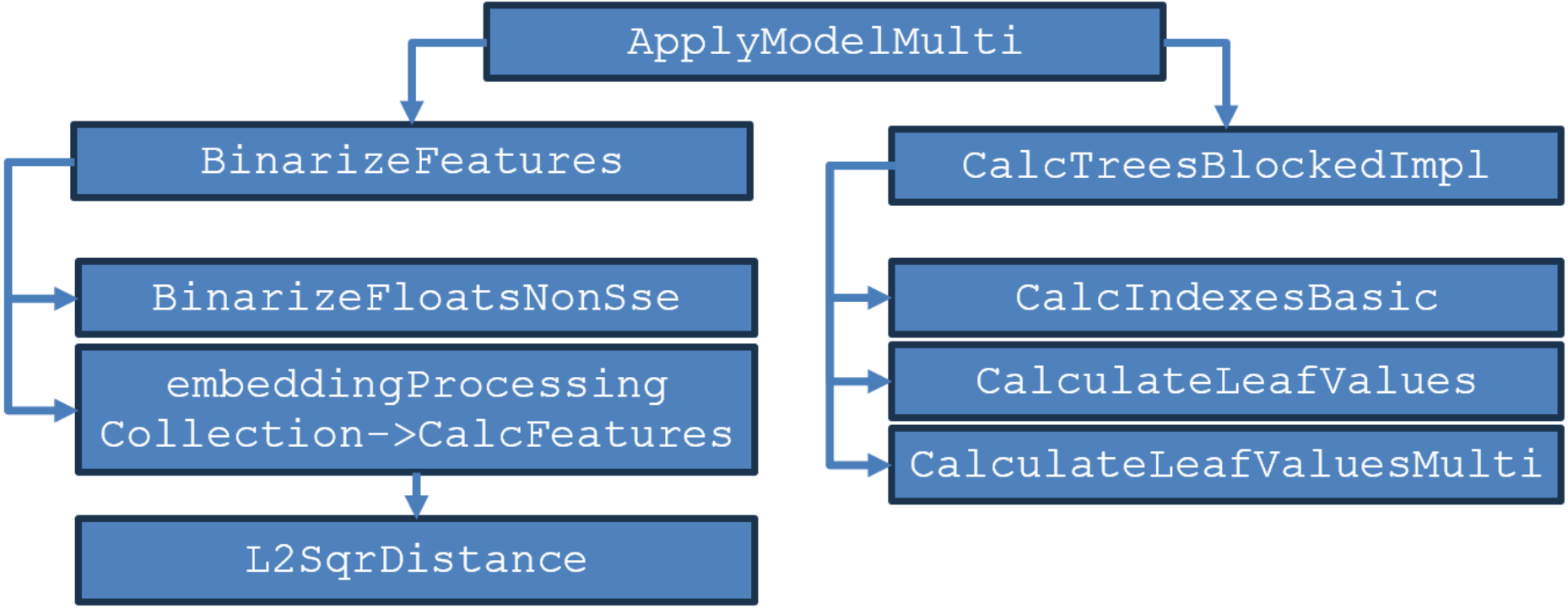
- Выяснилось, что использовать ***perf*** из Linux для анализа CatBoost **проблематично***, т.к. код содержит *интерфейс* на Python и *динамические библиотеки* на C++
- **Технология профилирования:**
  - Найти точку входа – наиболее нагруженную функцию **(perf)**
  - Изучить тело функции, вставить замеры времени и аккумулировать времена в глобальную переменную **(свой таймер)**
  - Проанализировать результаты, продвинуться вглубь, построить граф вызова функций

**** Судя по всему, проблематично, но возможно*** ☺

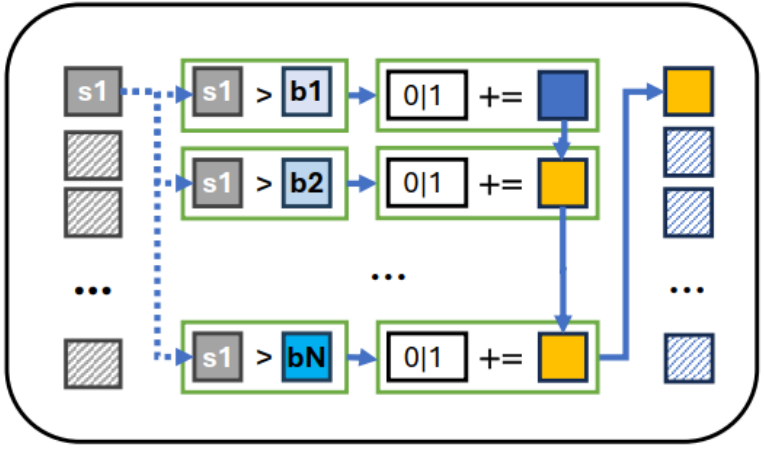
Как найти хотспоты на устройствах RISC-V?

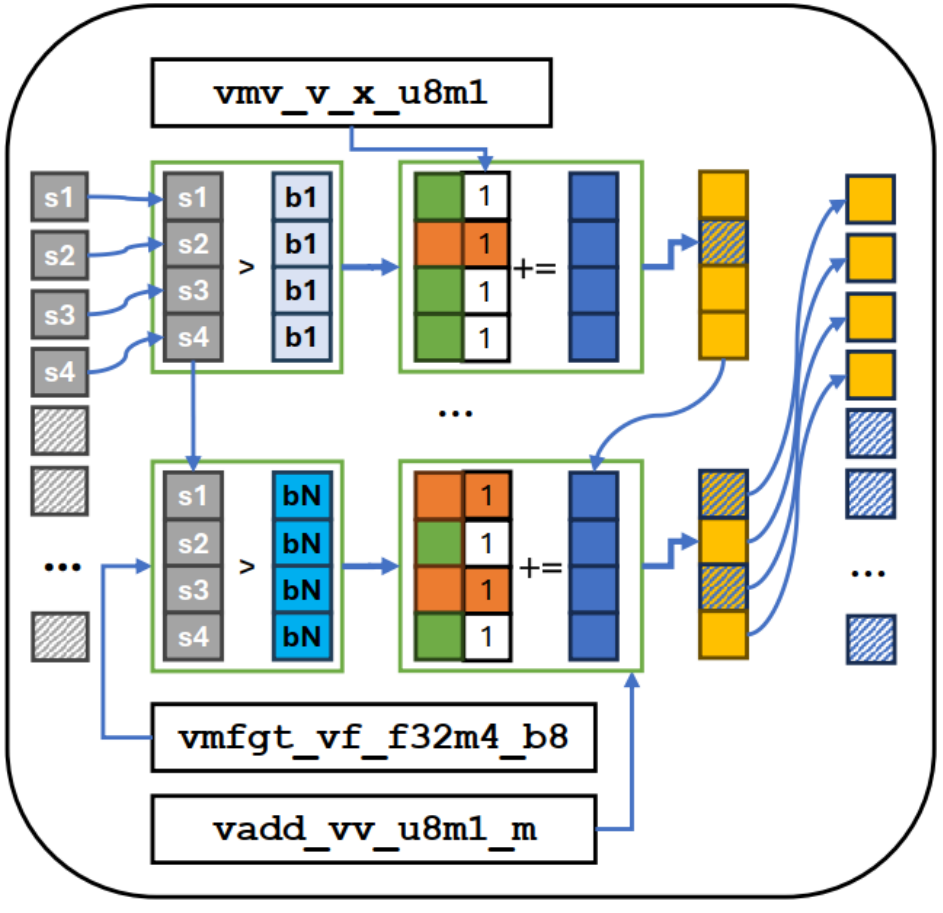
- **Технология профилирования:**
  - Найти точку входа – наиболее нагруженную функцию **(perf)**
  - Изучить тело функции, вставить замеры времени и аккумулировать времена в глобальную переменную **(свой таймер)**
  - Проанализировать результаты, продвинуться вглубь, построить граф вызова функций
- **Проверка:**
  - Использовать тот же подход на x86 CPUs и сравнить с результатами профилировки имеющимися на x86 средствами
  - Сравнить запуски с профилировкой и без нее на RISC-V CPUs
  - Провести ***итоговые эксперименты*** **без профилировки** на **полных наборах данных**

Кандидаты для векторизации



Функция BinarizeFloatsNonSse()





s{j} - sample{j} feature's value

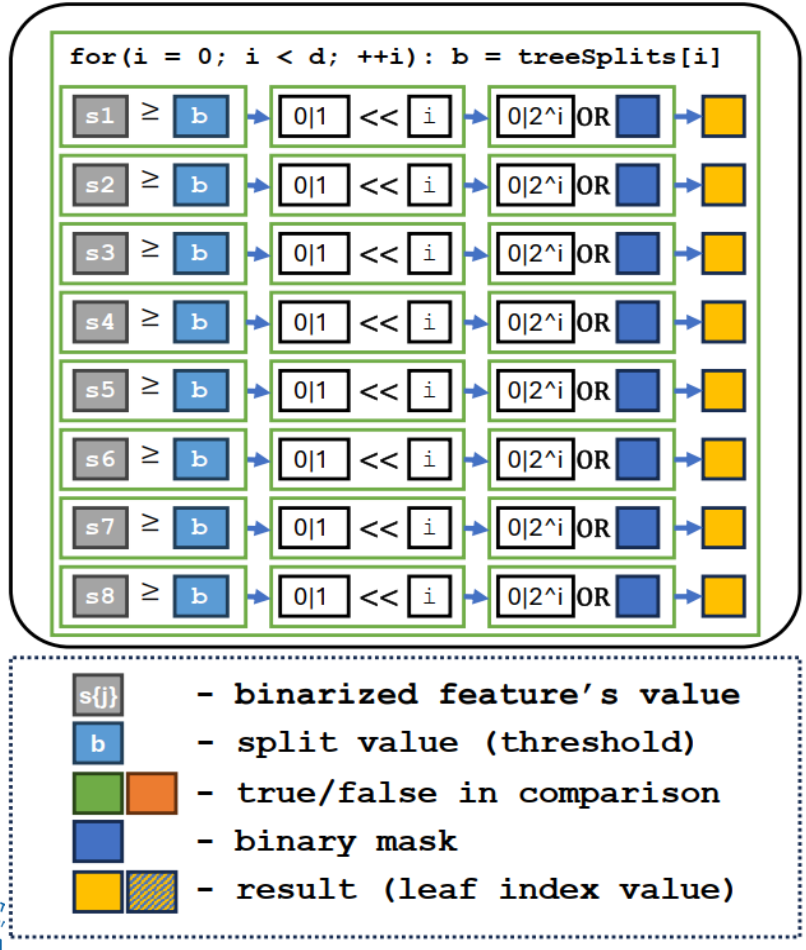
b1 b2 ... bN - bins' boundaries

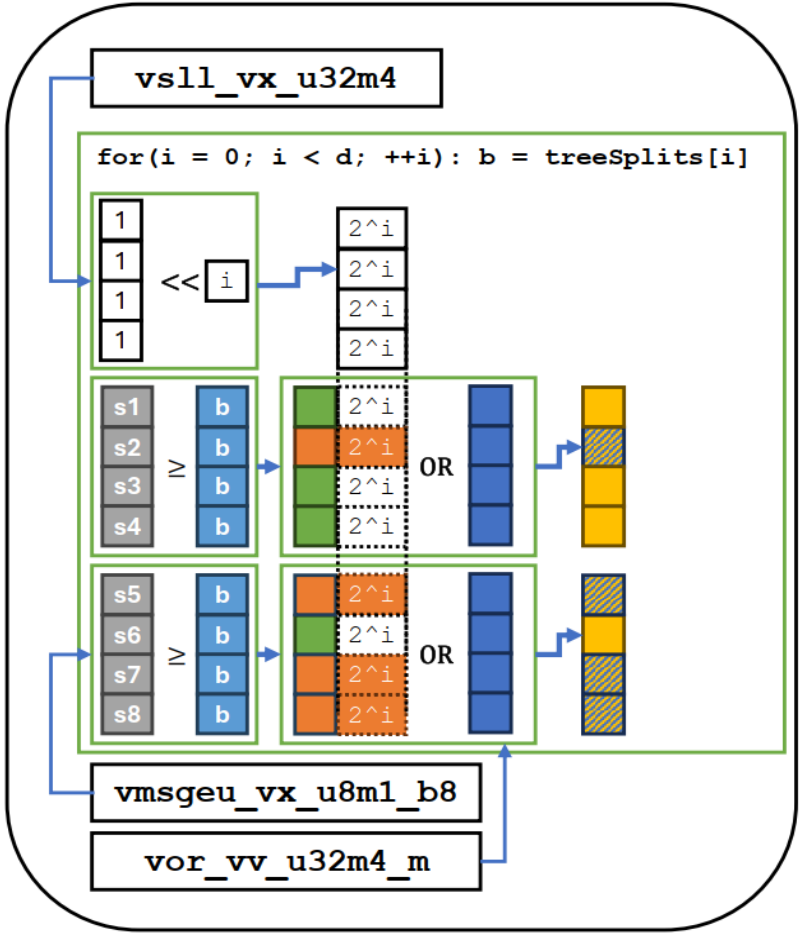
- true/false in comparison

- preliminary bin index value

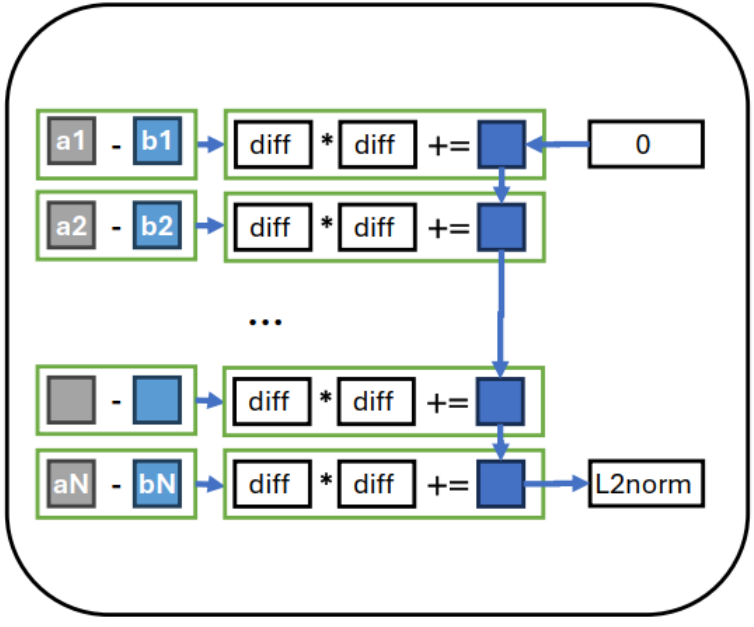
- resulting bin index value

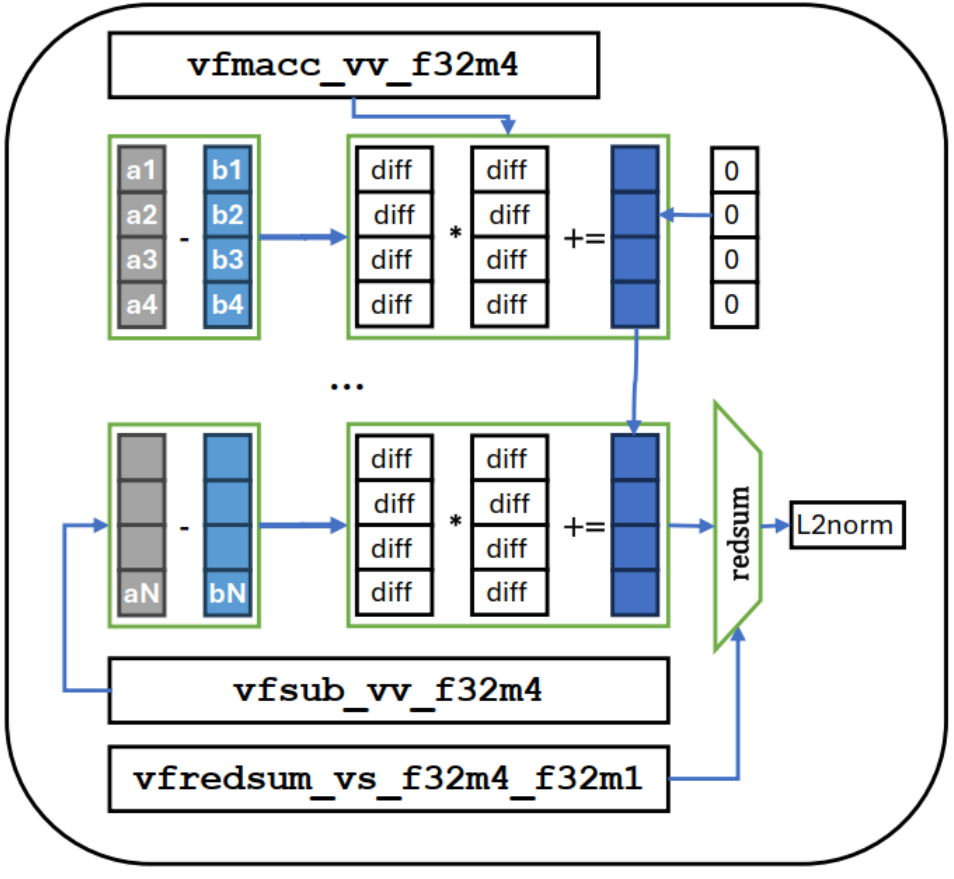
Функция CalcIndexesBasic()





Функция L2SqrDistance()





a{j} b{j} - source vectors

Векторизация для RISC-V CPUs: Идея

- Векторное расширение RVV 0.7.1 позволяет ***выбрать число векторных регистров, используемых в одной операции***
- Например, мы можем использовать **блок из 4х 128-битных регистров** и соответствующие векторные операции, как будто архитектура содержит **512-битные регистры**
- Для этого нужно указать суффикс **m4** в наименованиях интринсиков
- Таким образом можно **значительно** повысить производительность, но определить наилучший вариант (m1, m2, m4, m8) пока можно только экспериментально
- Мы использовали наилучший вариант

Оборудование

- **X86 (***тренировка моделей, подготовка наборов данных, вычисление точности, сравнение производительности***)**:
  - Intel Xeon Silver 4310T (2 CPUs с 10 ядрами, 20 ядер всего), 64 GB RAM
- **RISC-V**: **Mini-cluster Lichee Cluster 4A**
  - 7 план с RISC-V TH1520 CPUs на базе ядер C910
  - Каждый процессор содержит 4 ядра с поддержкой RVV 0.7.1
  - Каждая плата имеет 16GB RAM

Предварительные результаты (1)

Profiling results of CatBoost prediction on the `YearPredictionMSD` dataset on RISC-V CPU. The code was run in a serial mode. Time is given in seconds.

Function/metric	Call count	Baseline		Optimized		Speedup
		time	% total time	time	% total time	
CalcTreesBlockedImpl	8	1.35	89.41%	0.39	79.82%	**3.43**
CalcIndexesBasic	79992	1.02	67.60%	0.07	15.11%	**13.68**
CalculateLeafValues	79992	0.21	13.70%	0.20	40.56%	**1.03**
BinarizeFloatsNonSse	720	0.09	5.63%	0.03	6.05%	**2.85**
Other (profiler, auxiliary func ...)		0.07	4.95%	0.07	14.14%	-
Total time		**1.51**		**0.49**		**3.06**

Предварительные результаты (2)

Profiling results of CatBoost prediction on the `Covertype` dataset on RISC-V CPU. The code was run in a serial mode. Time is given in seconds.

Function/metric	Call count	Baseline		Optimized		Speedup
		time	% total time	time	% total time	
CalcTreesBlockedImpl	8	1.45	95.70%	0.81	93.17%	**1.79**
CalcIndexesBasic	39520	0.70	46.40%	0.06	6.33%	**12.76**
CalculateLeafValues Multi	39520	0.67	44.44%	0.69	78.64%	**0.98**
BinarizeFloatsNonSse	432	0.02	1.24%	0.01	1.49%	**1.45**
Other (profiler, auxiliary func ...)		0.05	3.05%	0.05	5.34%	-
Total time		**1.52**		**0.87**		**1.74**

Предварительные результаты (3)

Profiling results of CatBoost prediction on the `image-embedding` dataset on RISC-V CPU. The code was run in a serial mode. Time is given in seconds.

Function/metric	Call count	Baseline		Optimized		Speedup
		time	% total time	time	% total time	
CalcTreesBlockedImpl	8	1.60	8.15%	1.17	18.54%	**1.36**
CalcIndexesBasic	38064	0.35	1.76%	0.04	0.56%	**9.72**
CalculateLeafValues Multi	38064	1.18	6.05%	1.07	16.95%	**1,11**
BinarizeFeatures	1	17.93	91.60%	5.10	80.70%	**3.51**
BinarizeFloats NonSse	312	0.03	0.13%	0.00	0.07%	**5.44**
embeddingProcessing Collection		17.91	91.48%	5.10	80.63%	**3.51**
Other (profiler, auxiliary func ...)		0.05	0.25%	0.05	0.76%	-
Total time		**19.58**		**6.33**		**3.10**

Результаты на полных наборах данных

Final comparison results. The code was run in a multithreaded mode. Time is given in seconds. An accuracy is same in all runs, therefore it is shown only once for each dataset.

DataSet	Accuracy	Time (x86)	Time (RISC-V) Baseline	Time (RISC-V) Optimized	Speedup
Santander customer transaction	0.911	0.17	16.07	7.65	**2.10**
Covertype	0.960	0.42	59.41	30.60	**1.94**
YearPredictionMSD	9.168	0.06	16.30	2.79	**5.84**
MQ2008	0.850	0.02	0.55	0.50	**1.10**
image-embeddings	0.802	0.18	16.66	6.00	**2.78**

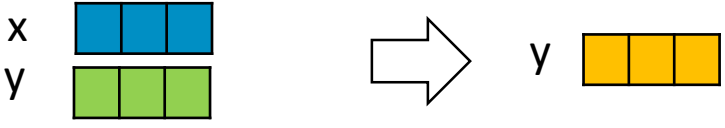
Производительность: выводы

- Время на сервере x86 приведено только для информации, детальное сравнение с x86 пока не имеет большого смысла
- Получено значительное (до 6x) ускорение
- **Два возможных ограничения:**
  - Ускорение достигается при обработке «пачки» входных данных (на одном сэмпле ускорение не ожидается)
  - Хотспоты и распределение нагрузки зависят от ***набора данных*** и ***решаемой задачи***. Мы исследовали различные модели, но в некоторых других сценариях может потребоваться ***дополнительная оптимизация*** для эффективного использования ресурсов RISC-V CPU

Часть III.
Оптимизация базовых алгоритмов линейной алгебры в OpenBLAS для RISC-V

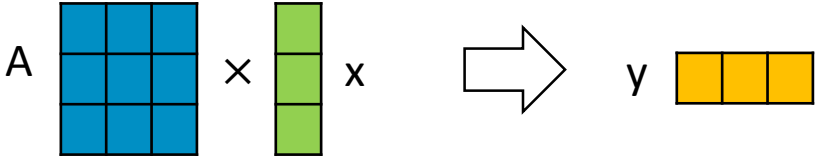
Интерфейс BLAS

- **BLAS** (Basic Linear Algebra Subprograms) – интерфейс библиотек для базовых операций линейной алгебры (Lawson, Hanson at el., 1979)
- Включает функции трех уровней, реализации для 4 типов данных
  - **Уровень 1** – вектор-векторные операции (16 функций)



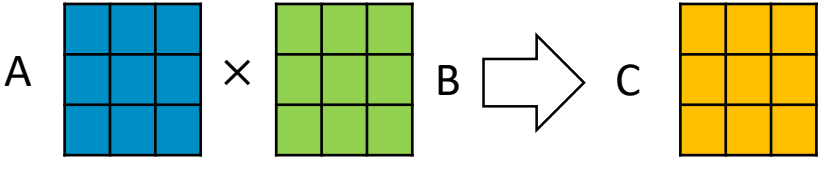
$$axpy$$
$$y \leftarrow \alpha * x + y$$

  - **Уровень 2** – матрично-векторные операции (25 функций)



$$gemv$$
$$y \leftarrow \alpha * op(A) * x + \beta * y$$

  - **Уровень 3** – матричные операции (9 функций)



$$gemm$$
$$C \leftarrow \alpha * op(A) * op(B) + \beta * C$$

Реализации стандарта BLAS

- Много реализаций стандарта для различных архитектур:
  - Intel oneAPI MKL, Goto BLAS, Netlib BLAS, ATLAS (*процессоры x86*), ACML (*процессоры AMD*), cuBLAS, CLBlast (*GPU*)…
  - **OpenBLAS** – оптимизированная реализация BLAS с открытым исходным кодом
    - Содержит низкоуровневые оптимизированные реализации функций для различных типов процессоров, в том числе, для процессоров Intel, AMD и RISC-V
    - Поддерживается большим сообществом разработчиков
- **Как сравнивать разные реализации?**
- **Как выявить функции, реализацию которых можно улучшить?**

Есть ли потенциал для оптимизации?

- **Потенциал для оптимизации есть (почти) всегда**
- Разработана **система тестирования** корректности и производительности для алгоритмов BLAS
- Выбраны **тестовые данные** (матрицы SuiteSparse, сгенерированные матрицы). Матрицы разбиты на «маленькие», «средние», «большие»

- Проведены масштабные эксперименты на суперкомпьютере (Intel x86) для сравнения производительности **OpenBLAS vs. MKL**
- **Обнаружено, что работа с ленточными матрицами организована в OpenBLAS недостаточно эффективно**

Функции BLAS 2 для ленточных матриц

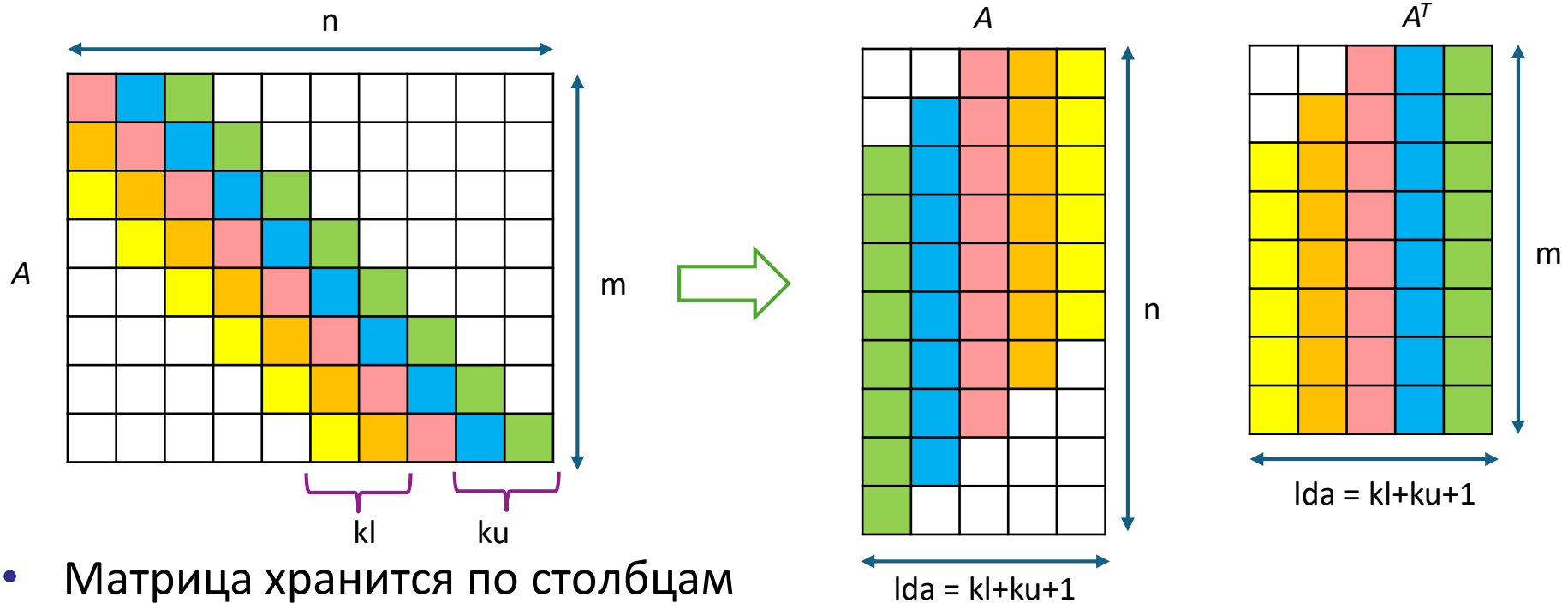
- **GBMV**: $y = \alpha \times op(A) \times x + \beta \times y$, где x, y – вектора, α, β – скаляры, A – ленточная матрица с kl нижними и ku верхними диагоналями размера $m \times n$, $op(A) = A$ или $op(A) = A^T$.
- **SBMV**: $y = \alpha \times A \times x + \beta \times y$, где x, y – вектора, α, β – скаляры, A – симметричная ленточная матрица с k побочными диагоналями, размера $n \times n$ (хранится только верхний или нижний треугольник).
- **TBMV**: $x = A \times x$, где x, y – вектора, A – треугольная ленточная матрица с k побочными диагоналями, размера $n \times n$ (хранится только верхний или нижний треугольник).
- **TBSV**: решение СЛАУ $op(A)x = b$, где x, b – вектора, A – треугольная ленточная матрица с k побочными диагоналями размера $n \times n$, $op(A) = A$ или $op(A) = A^T$ (хранится только верхний или нижний треугольник).

Программная реализация

- Базовая версия – библиотека OpenBLAS.
- В библиотеку OpenBLAS были встроены векторные реализации функций для процессоров x86 (набор инструкций AVX-512) и RISC-V (наборы инструкций rvv 0.7.1, rvv 1.0).
  - Точность одинарная и двойная
  - Последовательная версия
  - Параллельная версия (для некоторых алгоритмов)

Формат хранения

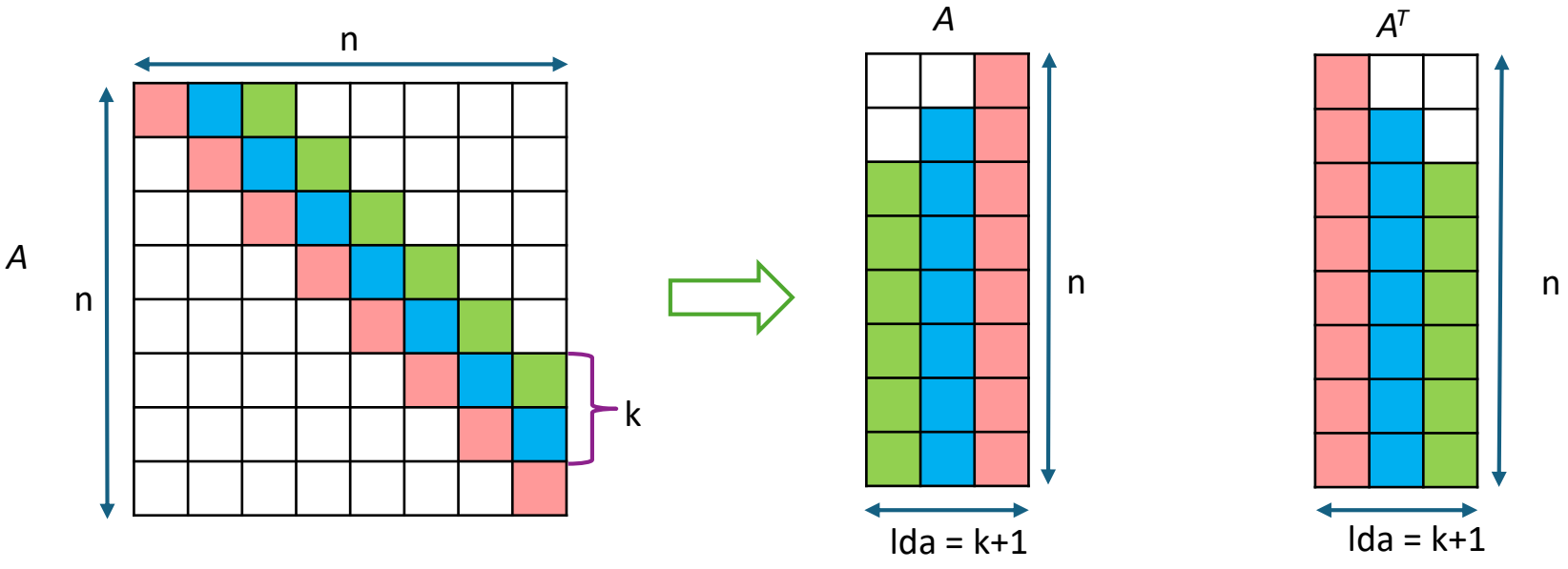
- Ленточная матрица общего вида



- Матрица хранится по столбцам

Формат хранения

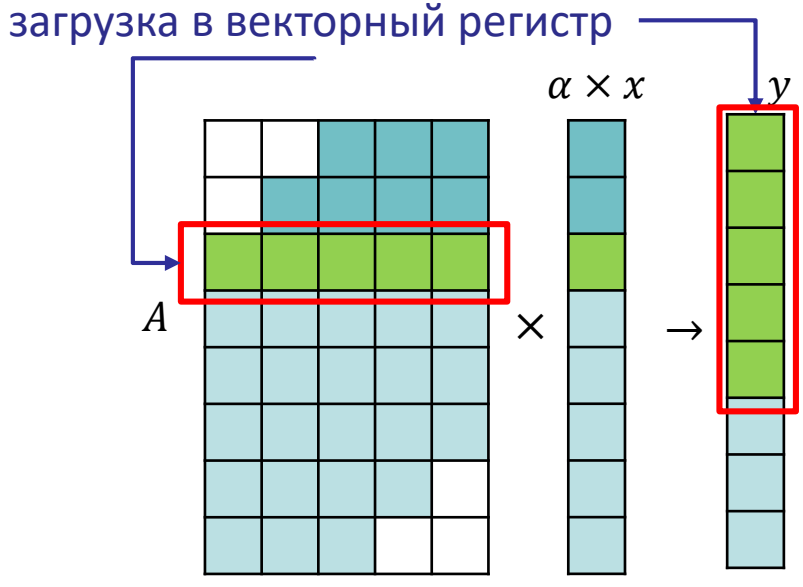
- Треугольная или симметричная ленточная матрица



Умножение ленточной матрицы общего вида на вектор (функция GBMV)

- Функция GBMV: $y = \alpha \times op(A) \times x + \beta \times y$,
  где x, y – вектора, α, β – скаляры, A – ленточная матрица с kl нижними и ku верхними диагоналями размера $m \times n$, $op(A) = A$ или $op(A) = A^T$.

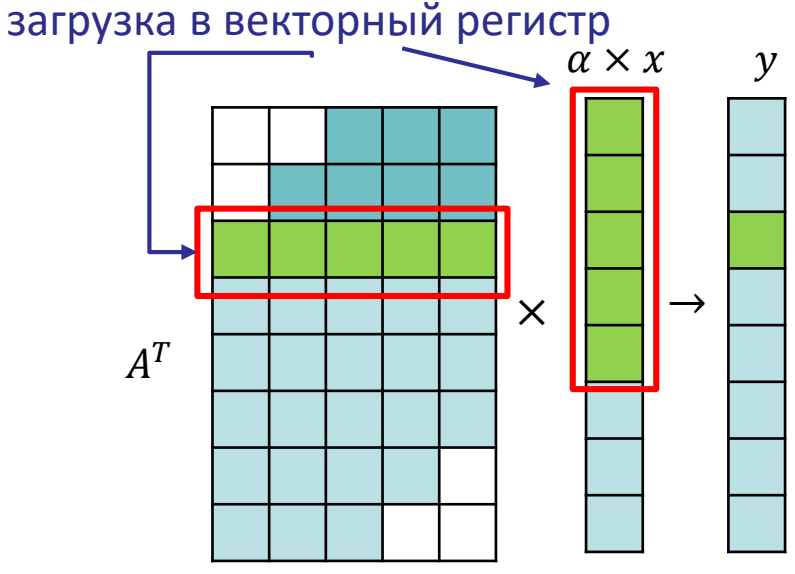
- Не транспонированная матрица
- Основная вычислительная операция:
  - AXPY: $y = \alpha \times x + y$,
    где x, y – векторы,
    α – скаляр
- В OpenBLAS векторизация выполнена в AXPY
- Эффективность векторизации зависит от числа диагоналей матрицы



Умножение ленточной матрицы общего вида на вектор (функция GBMV)

- Функция GBMV: $y = \alpha \times op(A) \times x + \beta \times y$,
  где x, y – вектора, α, β – скаляры, A – ленточная матрица с kl нижними и ku верхними диагоналями размера $m \times n$, $op(A) = A$ или $op(A) = A^T$.

- Транспонированная матрица
- Основная вычислительная операция – DOT:
  $res = \sum_{i=1}^{n} x_i y_i$,
  x, y – векторы
- В OpenBLAS векторизация выполнена в DOT



Умножение ленточной матрицы общего вида на вектор (функция GBMV)

Базовая реализация: при малом числе диагоналей вызываются скалярные версии AXPY и DOT

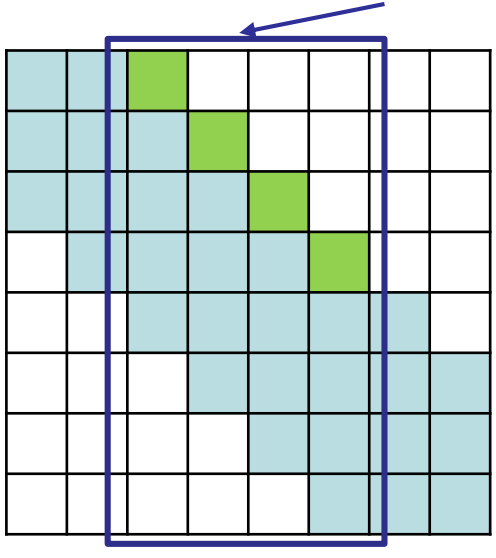
```
void GBMV(BLASLONG m, BLASLONG n, BLASLONG ku, BLASLONG kl, FLOAT alpha,
          FLOAT *a, BLASLONG lda, FLOAT *X, FLOAT *Y) {
  BLASLONG offset_u = ku; BLASLONG offset_l = ku + m;
  for (i = 0; i < MIN(n, m + ku); i++) {
    start = MAX(offset_u, 0);
    end   = MIN(offset_l, ku + kl + 1);
    length  = end - start;
#ifndef TRANS
    AXPYU_K(length, 0, 0, alpha * X[i], a + start, 1, Y +start -offset_u,
            1, NULL, 0);
#else
    Y[i] += alpha * DOTU_K(length, a + start, 1, X +start - offset_u, 1);
#endif
    offset_u --; offset_l --; a += lda;
}
```

Умножение ленточной матрицы общего вида на вектор (функция GBMV)

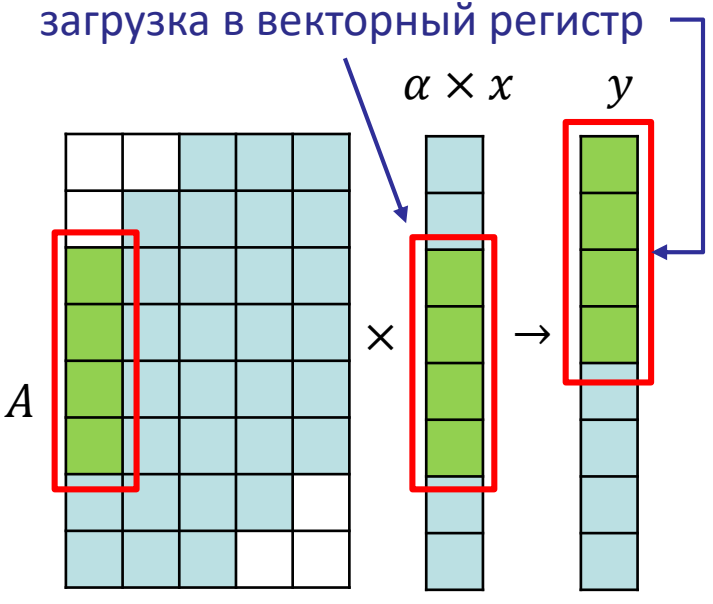
- **Новый алгоритм**:

Разделить матрицу на полосы, обход выполнять по диагоналям матрицы

ширина полосы = длине векторного регистра



загрузка в векторный регистр

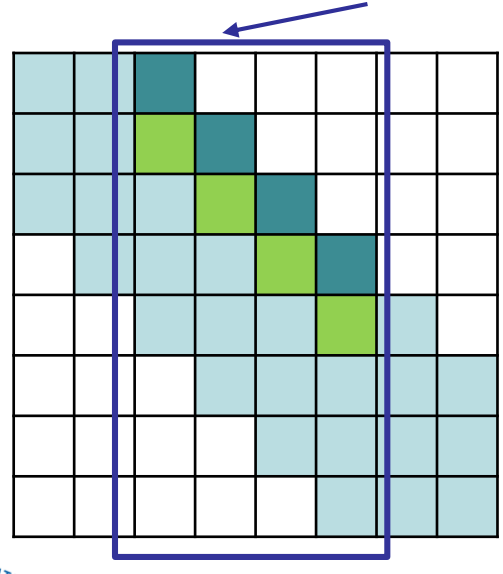


Умножение ленточной матрицы общего вида на вектор (функция GBMV)

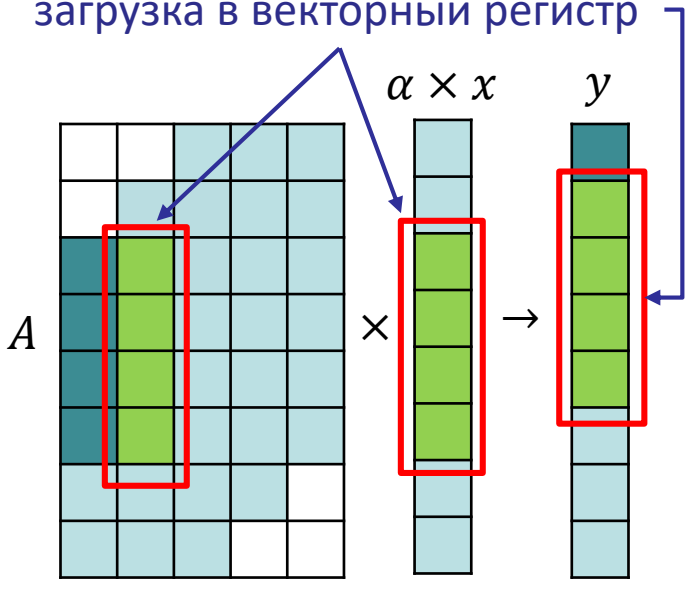
- **Новый алгоритм**:

Разделить матрицу на полосы, обход выполнять по диагоналям матрицы

ширина полосы = длине векторного регистра



загрузка в векторный регистр



Умножение ленточной матрицы общего вида на вектор (функция GBMV)

Оптимизированная реализация, $op(A) = A$, x86, AVX-512

```
void GBMV(BLASLONG jstart, BLASLONG jend, BLASLONG ku, BLASLONG kl, FLOAT alpha,
          FLOAT *a, BLASLONG lda, FLOAT *X, FLOAT *Y) {
   // BLOCK_SIZE = 8 для double, BLOCK_SIZE = 16 для float
   __m512i vindex = _mm512_set_epi64(7*lda, 6*lda, 5*lda, 4*lda, 3*lda,
                    2*lda, lda, 0); // индексы сдвигов для загрузки диагонали
   __m512d diag_i, x_copy, y_copy; // переменные-регистры
   __m512d factor = _mm512_set1_pd(alpha); // загрузка коэффициента в регистр
   for (j = jstart; j < jend - BLOCK_SIZE; j += BLOCK_SIZE) { // обход по блокам
       x_copy = _mm512_loadu_pd(X + j); // загрузка вектора X в регистр
       x_copy = _mm512_mul_pd(x_copy, factor); // x_copy *= factor
       for (i = 0; i < ku+kl+1; i++) { // обход матрицы по диагоналям
          y_copy = _mm512_loadu_pd(Y + j - ku + i); // загрузка вектора Y в регистр
          diag_i = _mm512_i64gather_pd(vindex, a + i, SCALE); // загрузка диагонали
          // y_copy += diag_i*x_copy
          y_copy = _mm512_fmadd_pd(diag_i, x_copy, y_copy);
          _mm512_storeu_pd(Y + j - ku + i, y_copy); // сохранение Y
       }
       a += lda * BLOCK_SIZE; // сдвиг указателя на следующий блок
   }
}
```

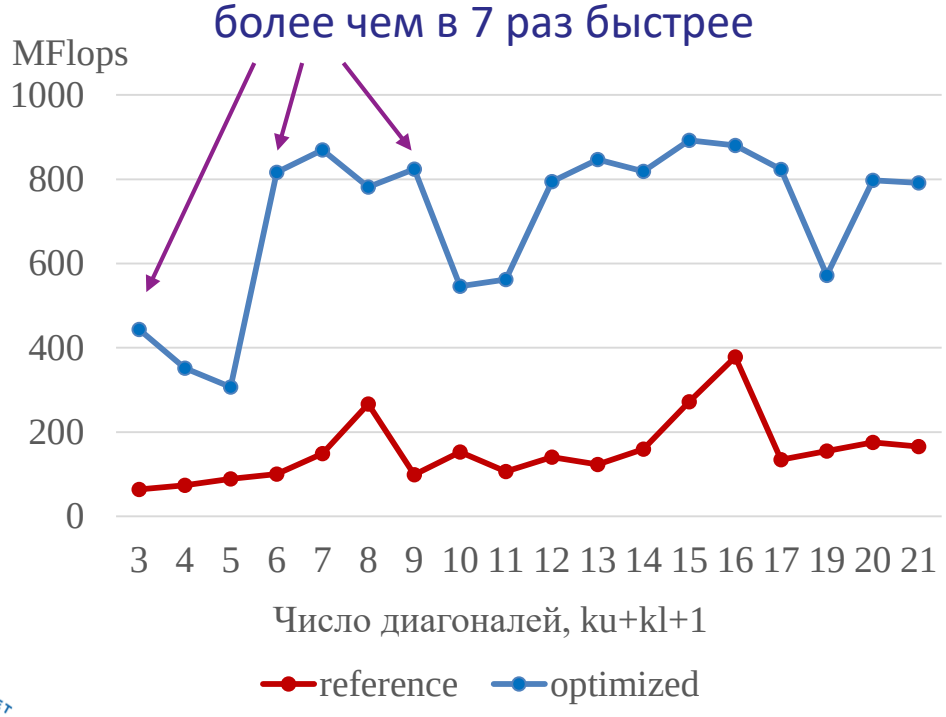
Умножение ленточной матрицы общего вида на вектор (функция GBMV)

Оптимизированная реализация, $op(A) = A$, RISC-V, rvv-0.7.1

```
void GBMV(BLASLONG jstart, BLASLONG jend, BLASLONG ku, BLASLONG kl, FLOAT alpha,
          FLOAT *a, BLASLONG lda, FLOAT *X, FLOAT *Y) {
   size_t VL = vsetvl_e64m4(BLOCK_SIZE); // определяем длину регистра
   BLASLONG stride = lda*sizeof(double); // шаг по массиву a для загрузки диагонали
   vfloat64m4_t diag_i, x_copy, y_copy;  // переменные-регистры
   for (j = jstart; j + VL < jend; j += VL) {
      x_copy = vle64_v_f64m1(X + j, VL); // загрузка вектора X в регистр
      x_copy = vfmul_vf_f64m1(x_copy, alpha, VL); // x_copy *= alpha
      for (i = 0; i < ku+kl+1; i++) {        // обход матрицы по диагоналям
         // загрузка вектора Y в регистр
          y_copy = vle64_v_f64m1(Y + j - ku + i, VL);
          diag_i = vlse64_v_f64m1(a + i, stride, VL); // загрузка диагонали
          // y_copy += diag_i*x_copy
          y_copy = vfmacc_vv_f64m1(y_copy, diag_i, x_copy, VL);
          vse64_v_f64m1(Y + j - ku + i, y_copy, VL); // сохранение Y
      }
      a += lda * VL; // сдвиг указателя на следующий блок
   }
}
```

Вычислительные эксперименты. GBMV на RISC-V

Производительность GBMV при $n = m =$ 1 млн., двойная точность, транспонированная матрица, длина регистра 512 бит



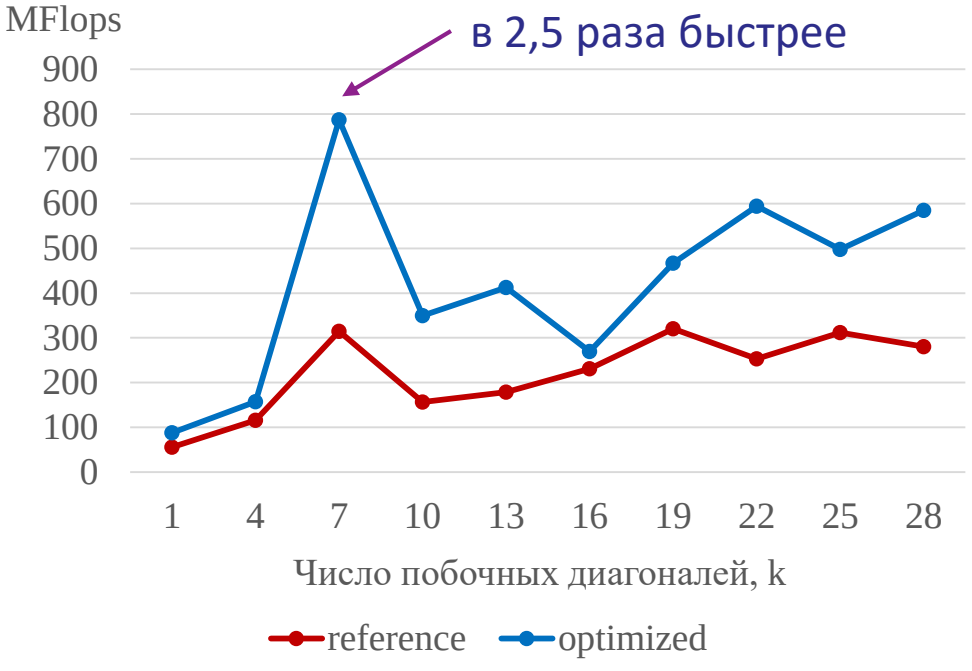
- Новый алгоритм в среднем в 5 раз быстрее базовой реализации
- Наилучшее ускорение при 3, 6, 9 диагоналях
- При числе диагоналей, кратном 8, новый алгоритм работает быстрее в 2,3 – 3 раза

Умножение симметричной ленточной матрицы на вектор (функция SBMV)

- **SBMV:** $y = \alpha \times A \times x + \beta \times y$, где x, y – вектора, α, β – скаляры, A – симметричная ленточная матрица с k побочными диагоналями, размера $n \times n$.
- Отличия от GBMV:
  - На каждой итерации вычисляется произведение для одного столбца матрицы A, вызываются обе функции: AXPY и DOT.
- Оптимизированный алгоритм: идея аналогична GBMV

Вычислительные эксперименты. SBMV на RISC-V

Производительность SBMV при $n = 1$ млн., двойная точность, длина регистра 512 бит



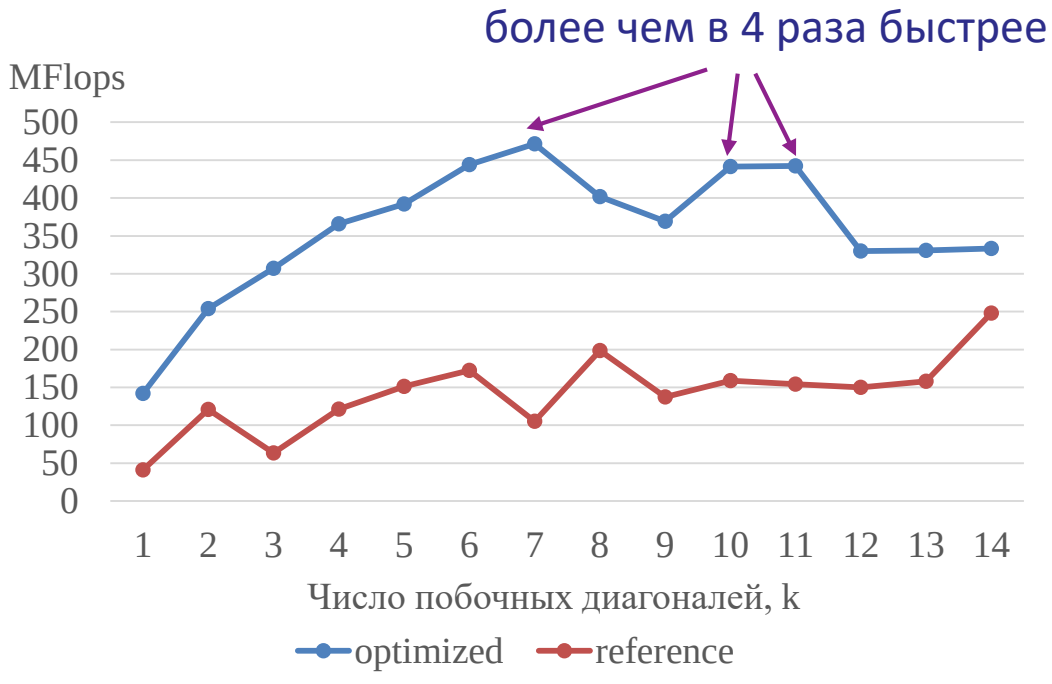
- Новый алгоритм в среднем в 1,8 раз быстрее базовой реализации
- Наилучшее ускорение при $k < 8$
- Наименьшее ускорение при k, кратном 8
- При фиксированном k и увеличении n производительность алгоритма не изменяется

Умножение треугольной ленточной матрицы на вектор (функция TBMV)

- **TBMV:** $x = op(A) \times x$, где x – вектор, A – треугольная ленточная матрица с k побочными диагоналями, размера $n \times n$, $op(A) = A$ или $op(A) = A^T$.
- Отличия от GBMV:
  - 4 варианта реализации в зависимости от вида треугольника и $op(A)$
  - для нижне-треугольной матрицы обход снизу вверх, для верхне-треугольной – сверху вниз
  - диагональный элемент обрабатывается отдельно
- Оптимизированный алгоритм: идея аналогична GBMV

Вычислительные эксперименты. TBMV на RISC-V

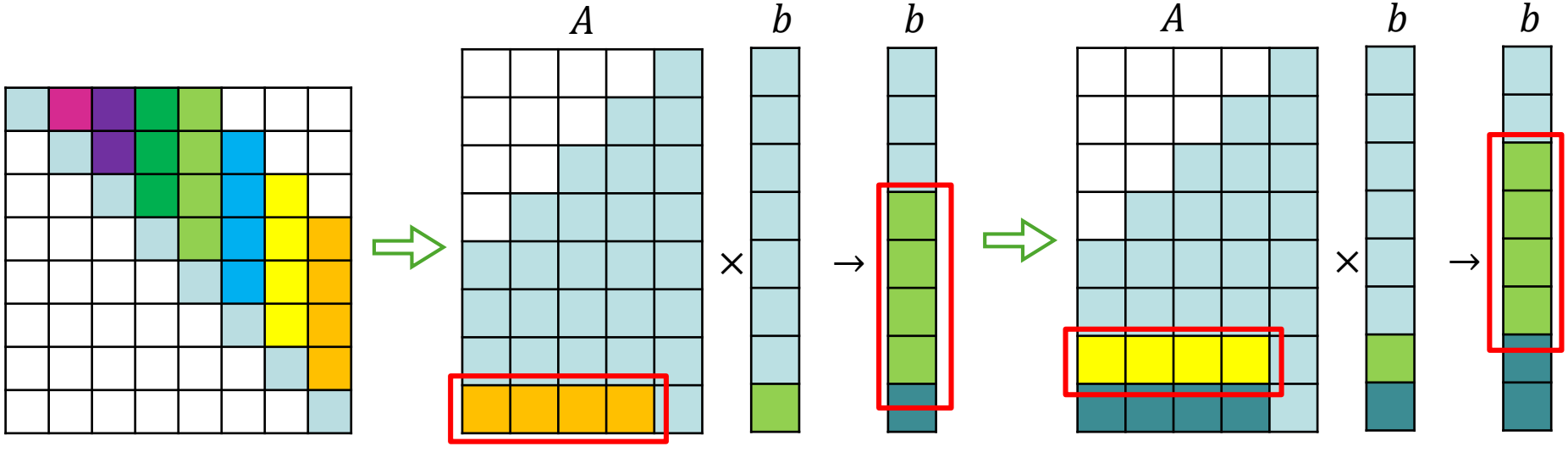
Производительность TBMV при $n = 1$ млн., double precision, верхний треугольник, транспонированная матрица, длина регистра 512 бит



- Новый алгоритм в среднем в 2,8 раз быстрее базовой реализации
- Наилучшее ускорение при $k < 8$
- Наименьшее ускорение при k, кратном 8
- При фиксированном k и увеличении n производительность алгоритма не изменяется

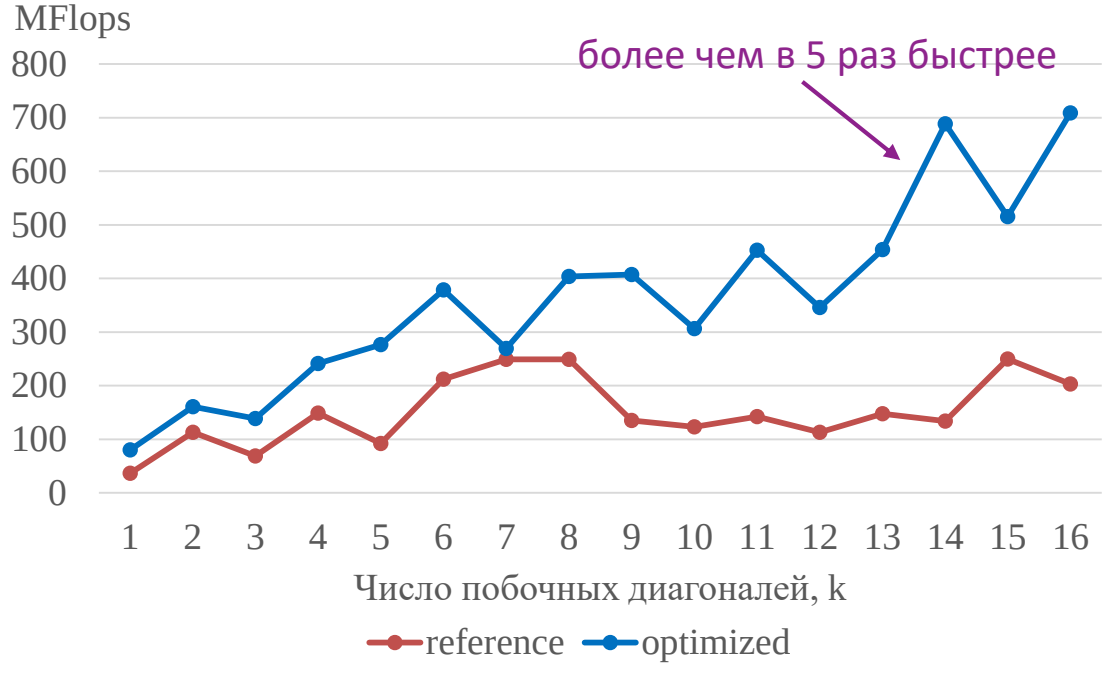
Решение СЛАУ с треугольной ленточной матрицей (функция TBSV)

- Функция TBSV: решение СЛАУ $op(A)x = b$, где x, b – вектора, A – треугольная ленточная матрица с k побочными диагоналями, размера $n \times n$, $op(A) = A$ или $op(A) = A^T$.



Вычислительные эксперименты. TBSV на RISC-V

Производительность TBSV при $n = 500\,000$, двойная точность, верхний треугольник, транспонированная матрица, длина регистра 128 бит.



- Новый алгоритм работает в среднем в 2,5 раза быстрее базовой реализации
- Наибольшее ускорение при числе диагоналей 8 < k < 16
- При большей длине регистра производительность оптимизированной версии близка к базовой реализации

Выводы

- Даже в оптимизированных под целевую архитектуру библиотеках часто есть возможность для дополнительного ускорения
- В данном случае это ускорение существенно (в несколько раз)
- Основная причина ускорения – более эффективная векторизация

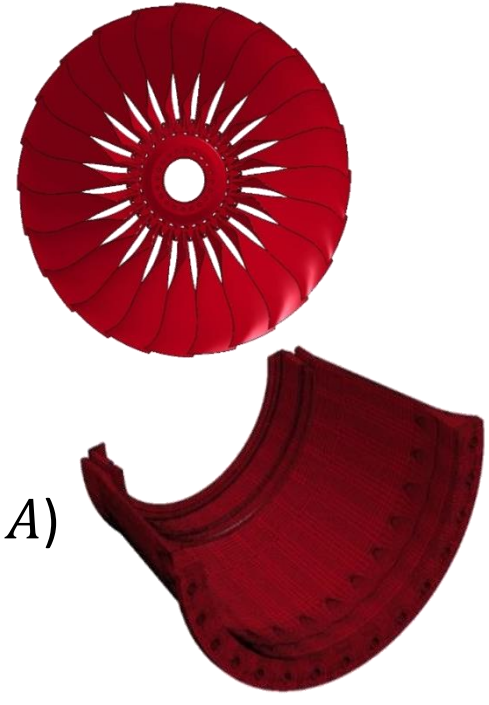
- В ряде случаев результат можно дополнительно улучшить за счет оптимизации работы с памятью (см. отдельную лекцию)

Часть IV.
Адаптация прямого решателя MUMPS и переупорядочивателей для RISC-V

Прямой метод решения СЛАУ

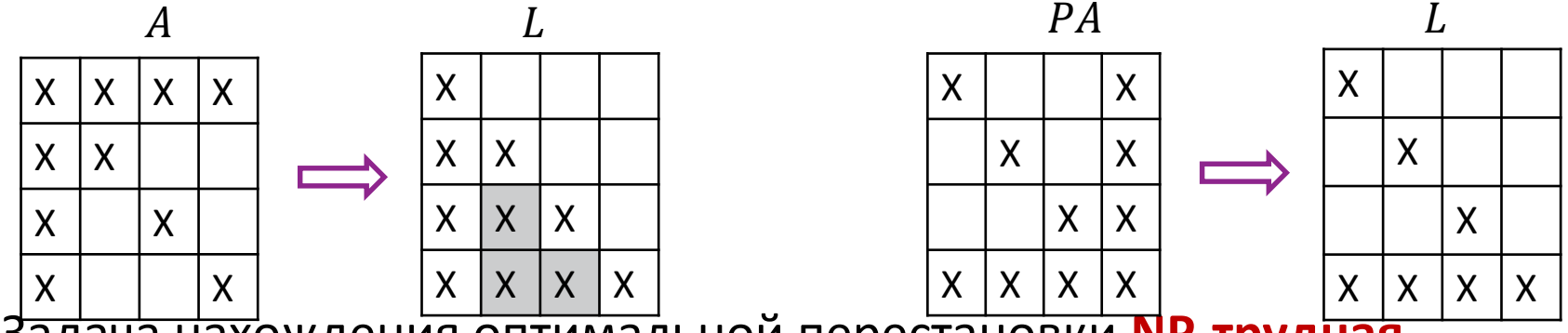
- При выполнении численного моделирования конечно-элементными методами часто возникает задача решения **системы линейных алгебраических уравнений** (**СЛАУ**) с ***разреженной*** матрицей большого порядка.
- Пусть дана система линейных уравнений: $Ax = b$
  A – симметричная положительно определенная разреженная матрица,
  b – плотный вектор,
  x – вектор неизвестных.
- Прямой метод решения: $A = LDL^T$
  L – нижнетреугольная матрица (*фактор* матрицы A)
- Переход к решению треугольных систем:

$$LDy = b, L^T x = y$$



Проблема заполнения фактора

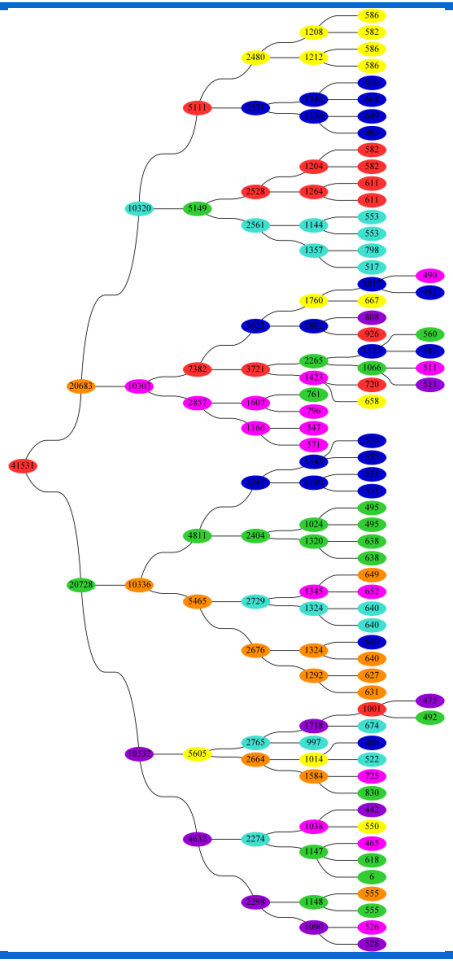
- При прямом решении СЛАУ с разреженной матрицей возникает проблема ***заполнения*** – существенного увеличения числа ненулевых элементов фактора матрицы.
- Уменьшить заполнение можно с помощью **симметричной перестановки**: $Ax = b \Leftrightarrow (PAP^T)(Px) = Pb,$
- P – матрица перестановки

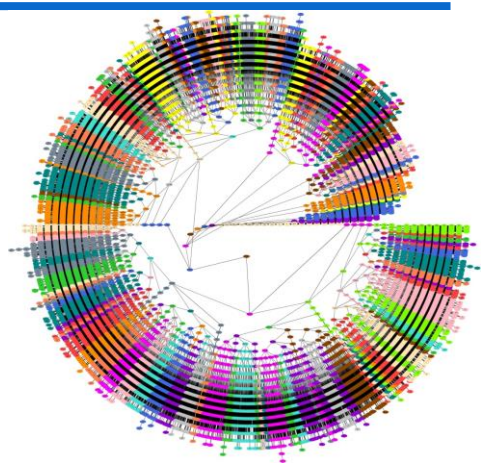


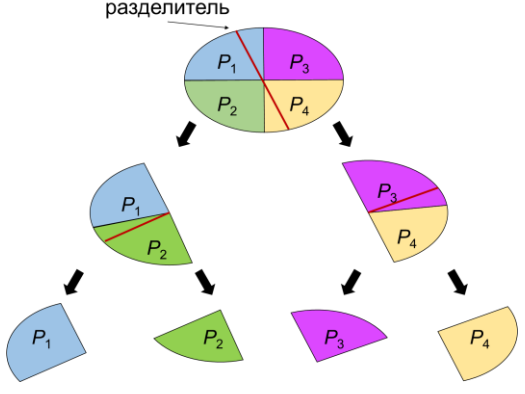
- Задача нахождения оптимальной перестановки **NP-трудная** (*Yannakakis*, 1981), решается эвристическими методами (ND, MD…)

Проблема заполнения фактора

- Задача нахождения оптимальной перестановки **NP-трудная** (*Yannakakis*, 1981), решается эвристическими методами (ND, MD…)
- Наиболее распространенная реализация – пакет **Metis** (ParMetis, mt-Metis)
- **Разработка ННГУ** – пакет MORSy (PMORSy, DMORSy)







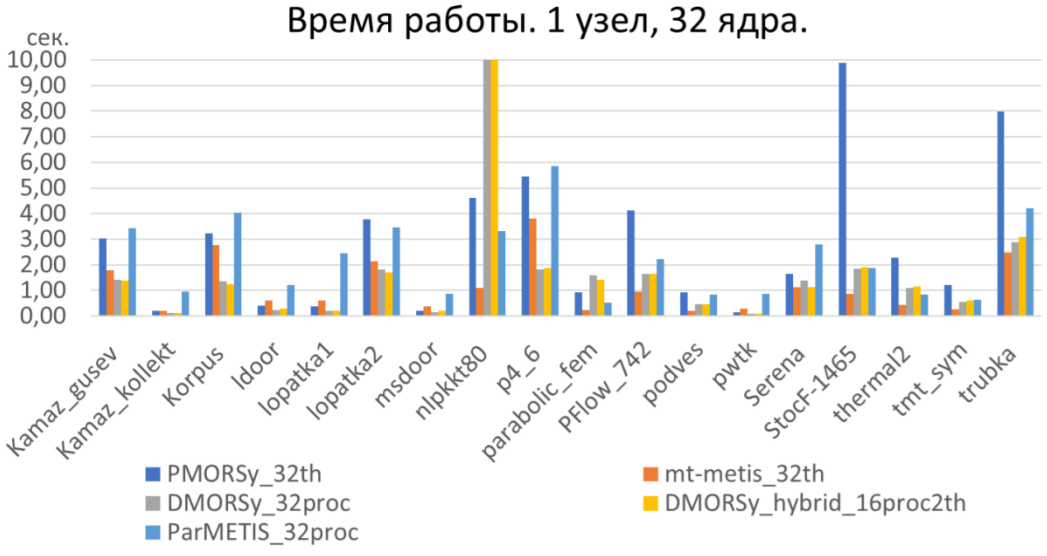
Тестируемые приложения

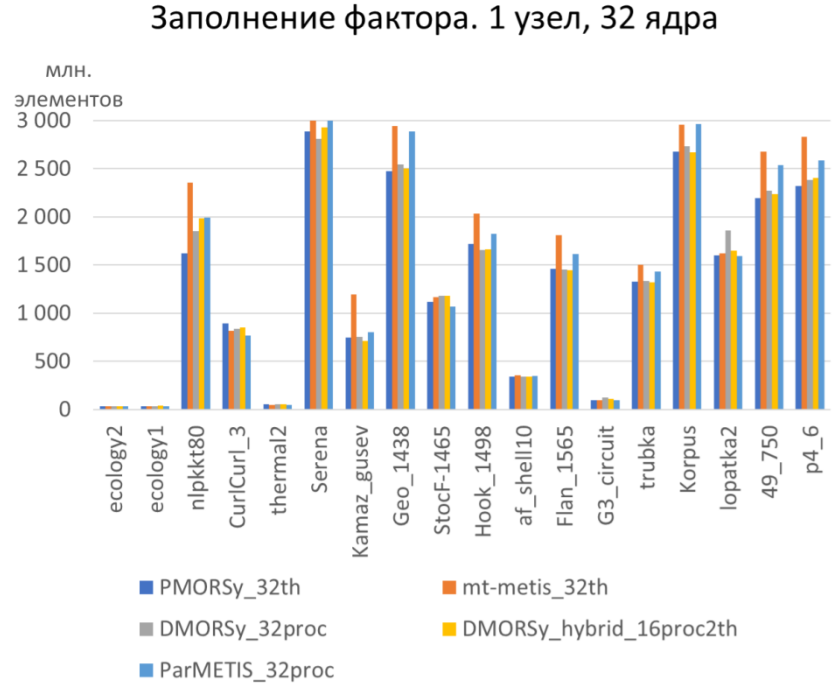
- **DMORSy** – переупорядочиватель разреженных матриц (ННГУ)
  - многоуровневый метод вложенных сечений
  - гибридная MPI + OpenMP схема параллелизма
- **ParMETIS** – переупорядочиватель разреженных матриц с открытым исходным кодом
  - многоуровневый метод вложенных сечений
  - параллелизм для распределенной памяти, технология MPI
- **MUMPS** – свободно распространяемый прямой решатель СЛАУ
  - гибридная MPI + OpenMP схема параллелизма, используется BLAS и переупорядочиватели
- **Запустятся ли MUMPS и Metis/MORSy на RISC-V?**
- **Какую производительность можно ожидать?**

Тестовое окружение

- **RISC-V**:
  - мини-кластер из четырех 4-ядерных процессоров CPU TH1520, частотой 1.848Ghz, память 8 GB, операционная система Debian 12
  - компиляторы gcc v. 13.2.0, gfortran; решатель MUMPS v. 5.6.2, библиотеки OpenBLAS v. 0.3.26 и ParMETIS v. 4.0.3
- **x86**:
  - Кластер МСЦ РАН: Intel Xeon Gold 6154 (Skylake), 3.0 GHz, 2 x 18 ядер, память 192 GB, Intel Parallel Studio XE 2017 CE
  - решатель MUMPS v. 5.4.1, MKL BLAS
- Матрицы: симметричные матрицы из коллекции SuiteSparse (https://sparse.tamu.edu/) порядком от 0.5 млн до 1.5 млн строк и заполнением порядка $1 \times 10^{-6} - 1 \times 10^{-5}$ ненулевых элементов

Сравнение переупорядочивателей, x86



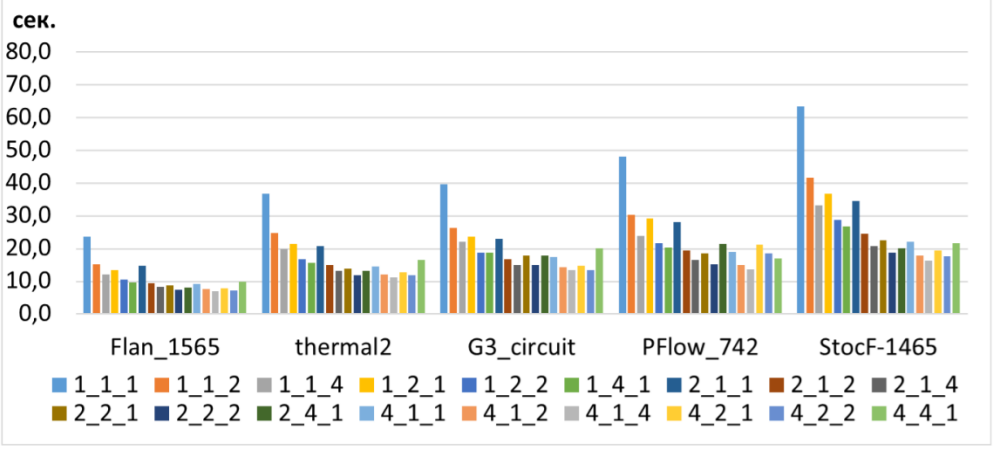


- DMORSy работает быстрее, чем ParMETIS, на матрицах порядка менее 1 млн. строк и матрицах с регулярной структурой.
- По заполнению фактора на половине тестовых матриц DMORSy опережает ParMETIS на ~8%, на других матрицах DMORSy уступает ParMETIS на ~12%.

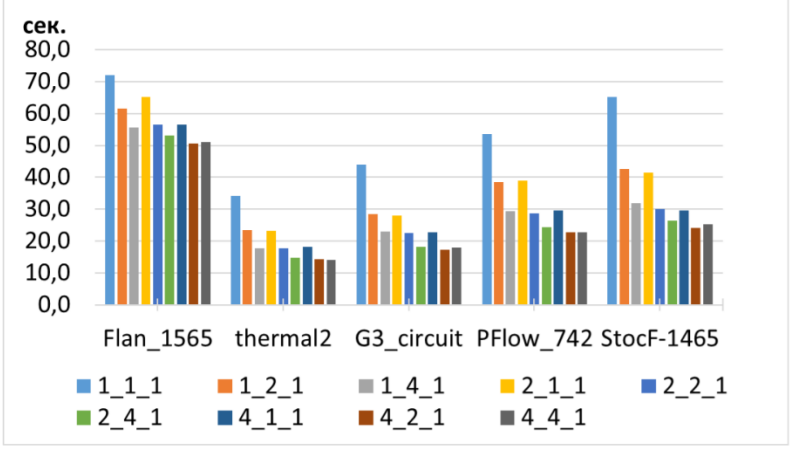
Сравнение переупорядочивателей, RISC-V

- Параметры переупорядочивателей фиксированы
- **DMORSy**: 4 узла x (1 процесс x 4 потока), среднее ускорение – 2.8 раз
- **ParMETIS**: 4 узла x (2 или 4 процесса x 1 поток), среднее ускорение – 1.8 раз

Время работы DMORSy (параметры по умолчанию)



Время работы ParMETIS

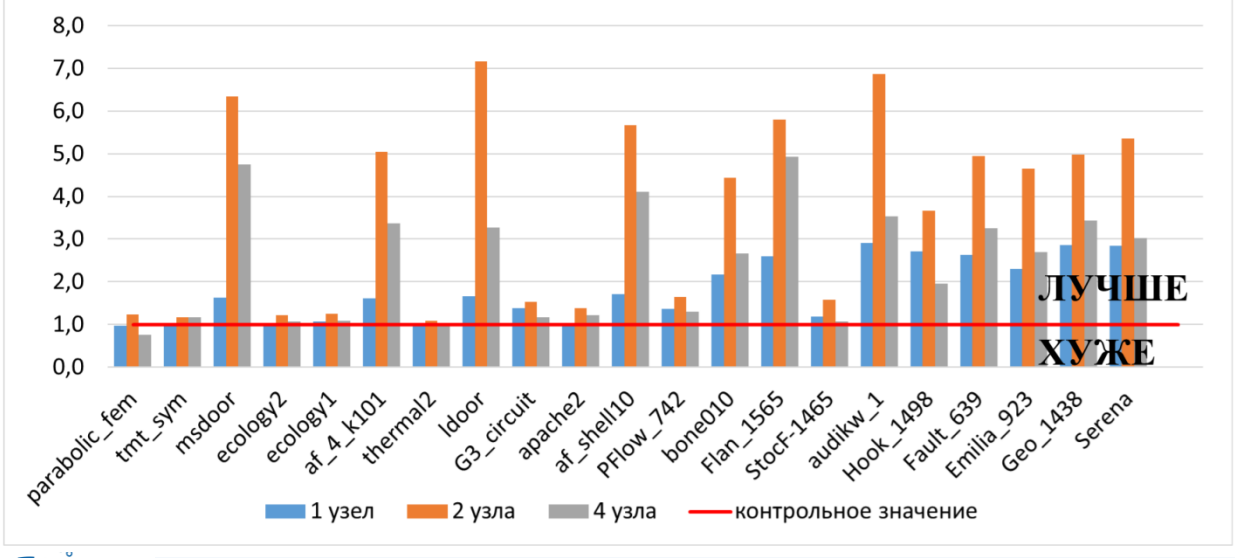


X Y Z, где X – число узлов, Y – число процессов на узел, Z – число потоков на процесс

Сравнение переупорядочивателей, RISC-V

- Сравнение DMORSy и ParMETIS по двум критериям: заполнение фактора после применения перестановки (чем меньше, тем лучше), время работы
- Параметры DMORSy, дающие наименьшее заполнение

Время работы ParMETIS относительно DMORSy, разы

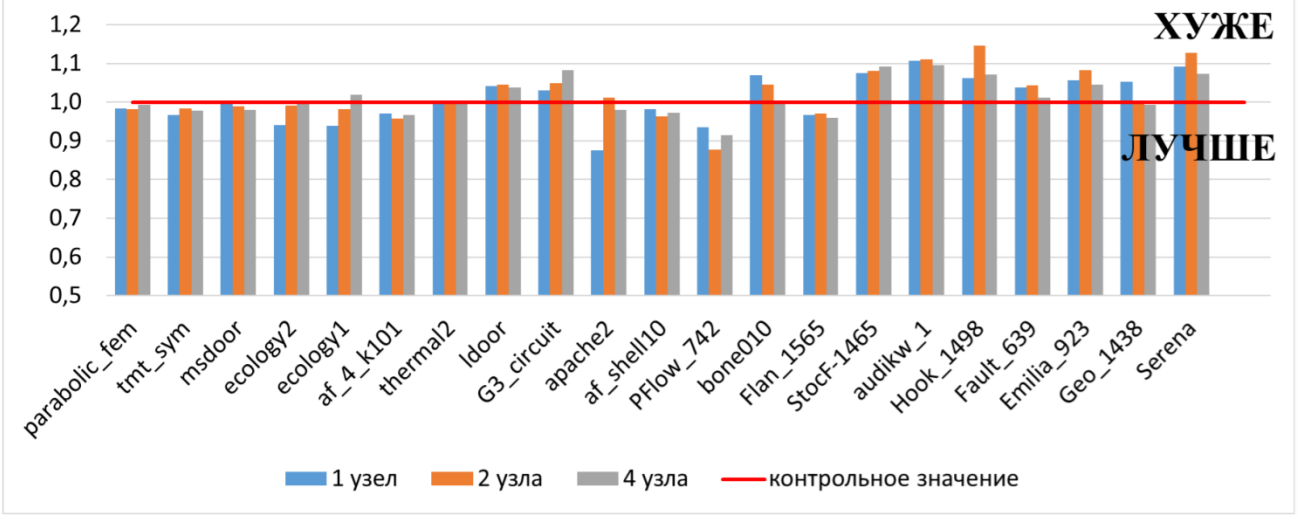


- В среднем, на двух узлах DMORSy работает быстрее ParMETIS в 3,7 раз, на четырех узлах – в 2,4 раза.
- ParMETIS плохо масштабируется, на 2/3 тестовых матриц ускоряется в среднем в 1,8 раз на 4х узлах
- Среднее ускорение DMORSy 1,6 раз на 4х узлах

Сравнение переупорядочивателей, RISC-V

- Сравнение DMORSy и ParMETIS по двум критериям: заполнение фактора после применения перестановки (чем меньше, тем лучше), время работы
- Параметры DMORSy, дающие наименьшее заполнение

Заполнение фактора, полученное DMORSy, относительно ParMETIS, разы

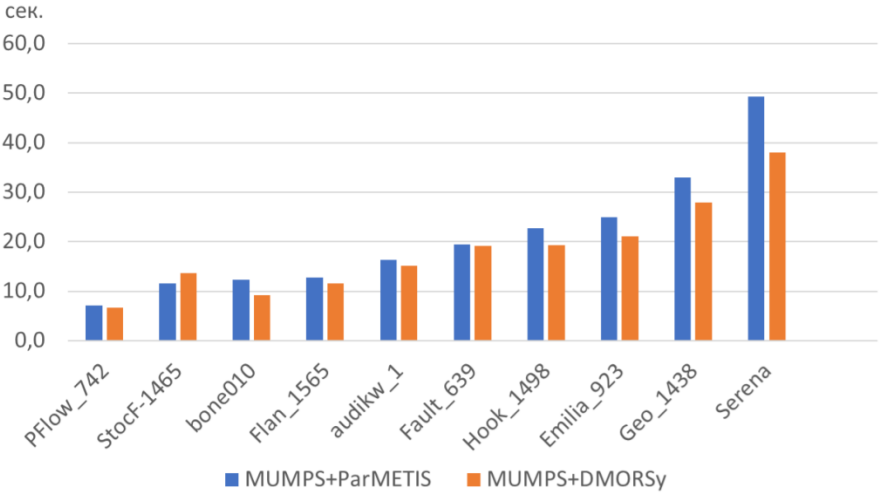


Близкие результаты:

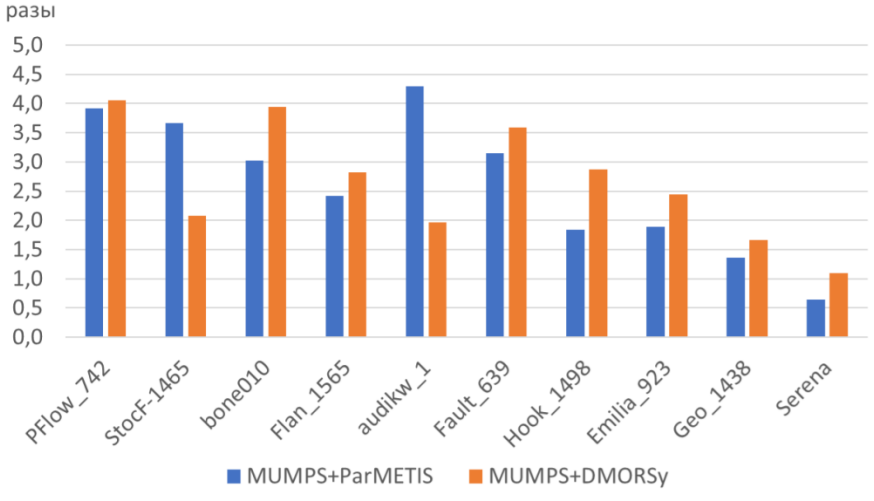
- на половине тестовых матриц DMORSy опережает ParMETIS на ~3%
- на других матрицах DMORSy уступает ParMETIS на ~6%
- DMORSy дает лучшие перестановки на более разреженных матрицах

Вычислительные эксперименты для решателя СЛАУ, x86

Время работы MUMPS
на 1 узле кластера МСЦ, 32 ядра



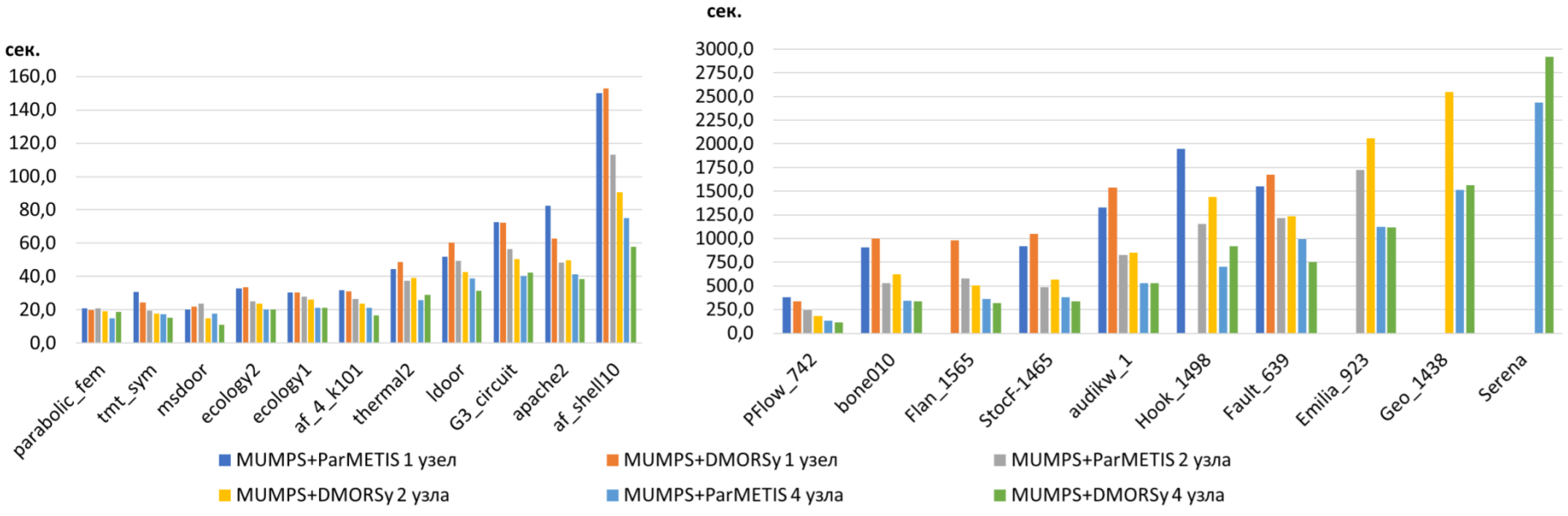
Ускорение MUMPS относительно
последовательной версии, 32 ядра



На большинстве тестовых матриц MUMPS с DMORSy работал быстрее, чем с ParMETIS, в среднем – на 26%. Опережение получено за счет сокращения времени переупорядочения и времени численной факторизации. Среднее ускорение MUMPS – 2,8 раз

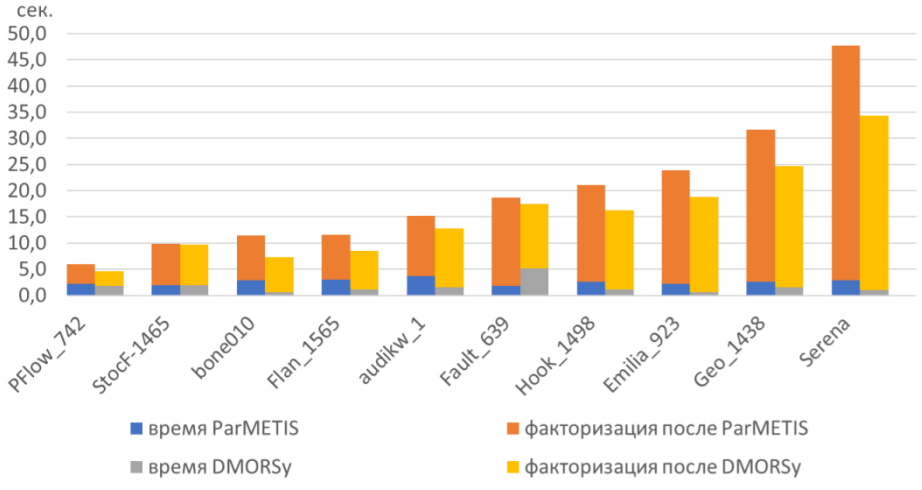
Вычислительные эксперименты для решателя СЛАУ, RISC-V

Время работы решателя MUMPS, 1 – 4 узла x (4 ядра)



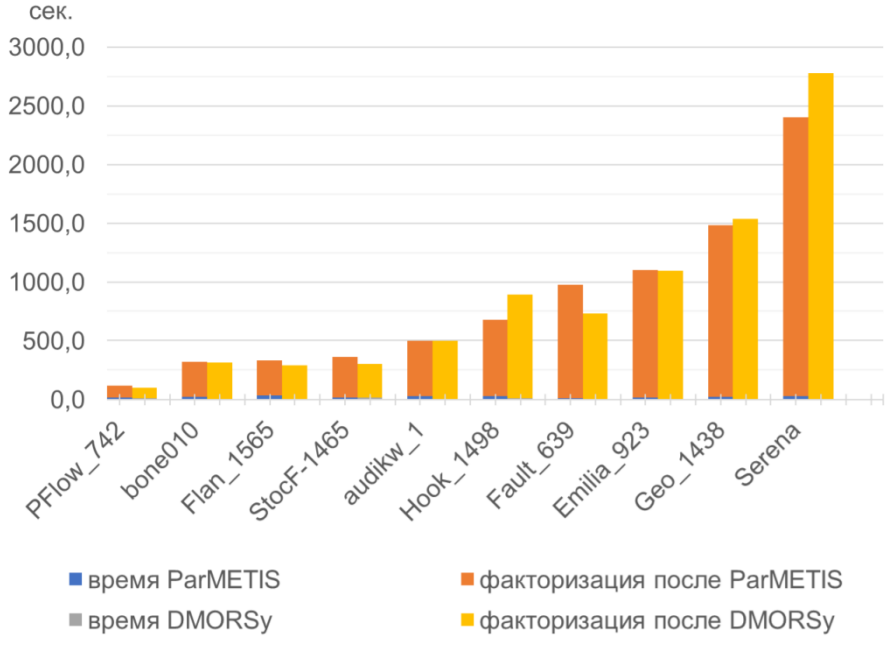
Вычислительные эксперименты для решателя СЛАУ, RISC-V

Соотношение времени переупорядочения и времени решения СЛАУ, 32 ядра, x86



Время факторизации в ~4–30 больше, чем время переупорядочения

Соотношение времени переупорядочения и времени решения СЛАУ, RISC-V, 16 ядер



Время факторизации до 100 раз больше, чем время переупорядочения

Основные выводы

Выводы по рассмотренным примерам

- RISC-V – достаточно зрелая архитектура, программно-аппаратное окружение подходит для использования и постепенно движется в сторону HPC
- Компилятор приемлемого качества (доступно распараллеливание, начиная с RVV 1.0 – автовекторизация)
- Большие пакеты легко переносятся на RISC-V простой пересборкой (*наилучшая производительность не гарантируется, но это типично*)
- Стандартные приемы оптимизации работают ожидаемым образом
- Присутствует специфика RISC-V, которую необходимо учитывать
- Компиляторы и профилировщики *пока* не позволяют раскрыть все возможности устройств – необходимо дальнейшее развитие
- Пока непонятно, что делать с проблемой “memory wall”

Литература

- **Первые эксперименты с устройствами RISC-V, методика сравнения производительности**

  Volokitin V. et al. ***Case Study for Running Memory-Bound Kernels on RISC-V CPUs*** // International Conference on Parallel Computing Technologies. – Springer LNCS. – Cham: Springer Nature Switzerland, 2023. – С. 51-65.

- **Улучшенная векторизация в OpenCV**

  Volokitin V. D. et al. Improved vectorization of OpenCV algorithms for RISC-V CPUs // Lobachevskii Journal of Mathematics. – 2024. – Т. 45. – №. 1. – С. 130-142.

- **Векторизация вывода в CatBoost**

  Kozinov E., Vasiliev E., Gorshkov A., Kustikova V., Maklaev A., Volokitin V., Meyerov I. Vectorization of Gradient Boosting of Decision Trees Prediction in the CatBoost Library for RISC-V Processors // Springer LNCS. – 2024. (Принята к печати) Препринт: https://arxiv.org/abs/2405.11062

Литература

Оптимизация/бенчмаркинг вывода в искусственных нейронных сетях

- Mukhin I., Rodimkov Y., Vasiliev E., Volokitin V., Sidorova A., Kozinov E., Meyerov I., Kustikova V. *Benchmarking Deep Learning Inference on RISC-V CPUs* // Springer Lecture Notes in Computer Science. – 2024. – Принята к печати.
- Sidorova A., Mukhin I., Kustikova V. *Optimizing Deep Learning Inference on RISC-V CPUs within the OpenVINO Toolkit* // Springer Communications in Computer and Information Science (CCIS). – 2024. – Принята к печати.

Литература

Высокопроизводительные вычисления и линейная алгебра

- Пирова А.Ю., Мееров И.Б. *Анализ производительности прямого решателя СЛАУ с разреженной матрицей на мини-кластере с процессорами архитектуры RISC-V* // Суперкомпьютерные дни в России – Москва: МАКС Пресс, 2024. – С. 34-44.
- Воденеева А.А., Ковалев К. И., Козинов Е.А., Пирова А.Ю., Устинов А.В., Волокитин В.Д., Мееров И.Б. *Оптимизация функций умножения плотной ленточной матрицы на вектор* // ММ и СКТ – Н. Новгород: Изд-во ННГУ, 2024. – С. 27-30.
- Козинов Е.А., Пирова А.Ю., Волокитин В.Д., Панова Е.А., Линев А.В., Мееров И.Б. *Анализ производительности OpenBLAS на процессорах с архитектурой RISC-V и x86* // ММ и СКТ – Нижний Новгород: Изд-во ННГУ, 2024. – С. 75-78.
- Pirova A. et al. Performance optimization of BLAS algorithms with band matrices for RISC-V processors //Future Generation Computer Systems. – 2025. – P. 107936.

Контакты

Нижегородский государственный университет
http://www.unn.ru

Институт информационных технологий, математики и механики
http://www.itmm.unn.ru

Кафедра высокопроизводительных вычислений и системного программирования