LOBACHEVSKY UNIVERSITY

DEPARTMENT OF HPC AND SYSTEM PROGRAMMING

Сравнение производительности и оптимизация программ для RISC-V процессоров. Примеры

Валентин Волокитин, Иосиф Мееров

При участии Е. Васильева, В. Кустиковой, Е. Козинова, А. Линева, Ю. Родимкова, А. Сысоева.

ННГУ им. Н. И. Лобачевского,

Институт информационных технологий, математики и механики,
каф. Высокопроизводительных вычислений и системного программирования

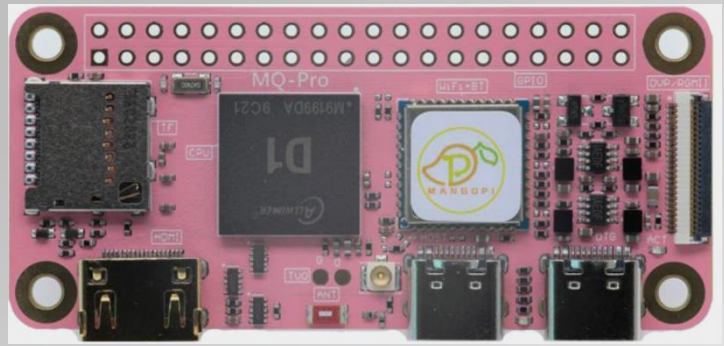
Суперкомпьютерные дни в России, Москва, 2023

Тематика доклада

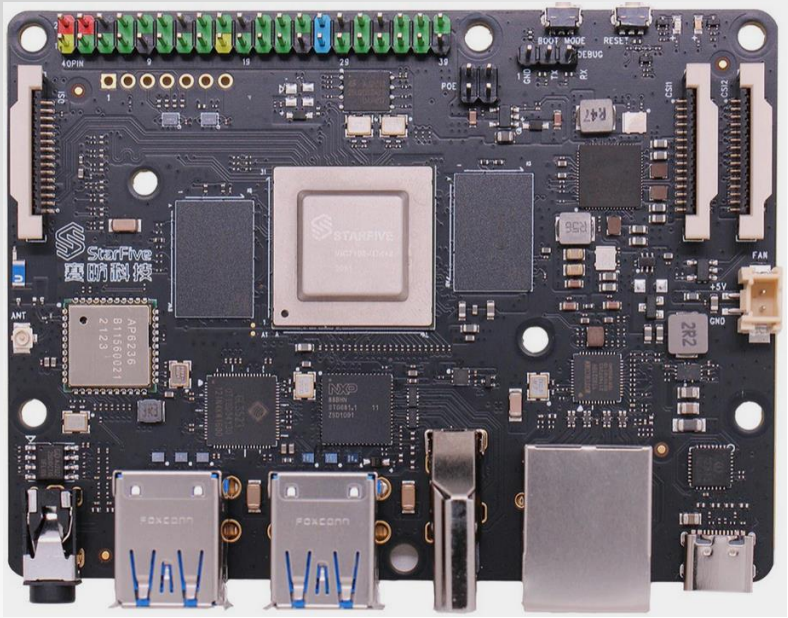
- Мы анализируем производительность доступных устройств RISC-V, демонстрируем подходы к оптимизации кода
- Мы показываем, что несмотря на значительный проигрыш по времени работы, устройства RISC-V быстро прогрессируют и показывают хорошее использование ресурсов
- Мы демонстрируем, что общепринятые подходы к оптимизации работы с памятью хорошо работают и на RISC-V
- Мы изучаем, как улучшить производительность на RISC-V, учитывая особенности архитектуры и набора команд
- В целом, мы демонстрируем потенциал архитектуры RISC-V как одной из перспективных архитектур для HPC

Инфраструктура (первая серия тестов)

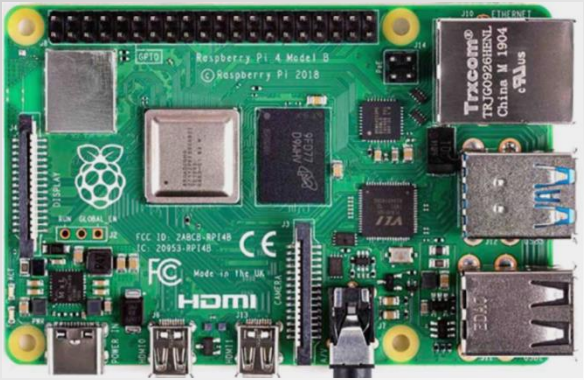
Mango Pi MQ-Pro (D1)



StarFive VisionFive (v1)



Raspberry Pi 4 model B



Intel Xeon 4310T

Mango Pi

- **Mango Pi MQ-Pro (D1) с Allwinner D1 CPU** (1 x XuanTie C906, 1GHz) и 1GB DDR3L RAM.
  - RV64IMAFDCV ISA,
  - 5-stage in-order execution pipeline,
  - L1 2-way set-associative I-Cache and 4-way set-associative D-Cache with a size of 32 KB each and cache line size of 64 bytes,
  - TLB, branch predictor, hardware prefetch for instructions and data,
  - 16, 32, 64-bits integer and floating point scalar and 512-bit vector operation including floating point FMA

- Ubuntu 22.10 OS (RISC-V edition) and GCC 12.2 compiler

StarFive VisionFive

- **StarFive VisionFive (v1) with StarFive JH7100 CPU** (2 x StarFive U74, 1 GHz) и 8 GB LPDDR4 RAM.
  - RV64IMAFDCB ISA,
  - 8-stage dual-issue in-order execution pipeline,
  - L1 2-way set-associative I-Cache and 4-way set-associative D-Cache with a size of 32 KB each, cache line size of 64 bytes,
  - 128 KB 8-way L2 cache
  - TLB, branch predictor, hardware data prefetch,
  - 64-bits integer and 32, 64-bits floating point scalar operation including floating point FMA
- Ubuntu 22.10 OS (RISC-V edition) and GCC 12.2 compiler

ARM и x86 CPUs

- **Raspberry Pi 4 model B** with Broadcom BCM2711 (4 x Cortex-A72, up to 1.5 GHz) processor and 4GB LPDDR4 RAM. Ubuntu 20.04 operating system and GCC 9.4 compiler were installed.

- **Server with 2 x Intel Xeon 4310T** (2 x 10 Ice Lake cores, up to 3.4 GHz) and 64 GB DDR4 RAM. CentOS 7 operating system and GCC 9.5 compiler were installed.

x86: используется только 10 ядер CPU, чтобы исключить влияние NUMA-эффектов, отсутствующих на других устройствах

Бенчмарки

STREAM benchmark

- элементарные операции над векторами
- позволяет определить пропускную способность памяти
- позволит нам интерпретировать результаты сравнения устройств

Транспонирование матрицы (in-place), Гауссова фильтрация

- как техники оптимизации работы с памятью ускоряют код на устройствах RISC-V при реализации алгоритмов, производительность которых ограничена параметрами подсистемы памяти?

Метрики производительности (I)

Время вычислений, Пропускная способность памяти

- Обычно используются в качестве основных метрик
- Однако мы должны помнить, что сравнение включает
  - *Одноядерный маломощный процессор архитектуры RISC-V*, на начальной стадии разработки и внедрения
  - 10-ядерный серверный Xeon, основанный на многолетнем опыте разработки HPC-устройств

Нужны и относительные метрики!

Метрики производительности (II)

- **Время вычислений, Пропускная способность памяти**
- **Улучшение относительно наивной реализации алгоритма в результате применения оптимизаций, типичных для традиционных CPU**
  - Дополнительная метрика, помогает понять, какого улучшения можно добиться в каждом конкретном случае

Метрики производительности (III)

Использование подсистемы памяти

- Вычисляем отношение числа байт данных (DRAM <-> CPU) к времени вычислений
- Делим это значение на достигнутую пропускную способность памяти, измеренную в бенчмарке STREAM
- Результат принадлежит [0,1] и является безразмерным
- Близость к 1 показывает эффективное использование памяти

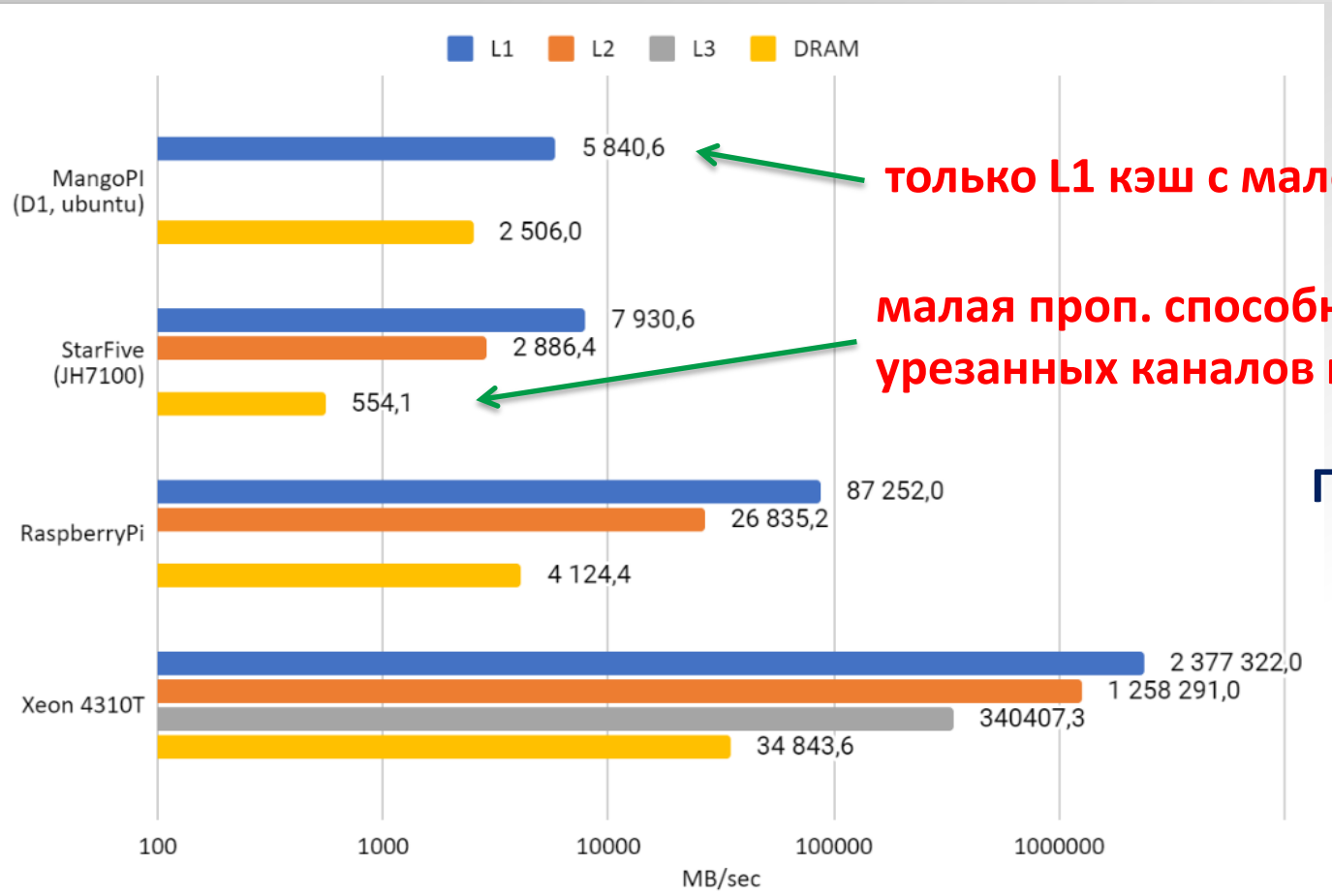
Эта метрика позволяет сравнивать устройства, несмотря на их существенную разницу в производительности

Численные результаты. STREAM

Цель: измерить пропускную способность памяти на 4 CPU

- **COPY** – **a[i]=b[i]**
  - 16 байт за итерацию, нет fp вычислений
- **SCALE** – **a[i]=d*b[i]**
  - 16 байт и 1 FLOP за итерацию
- **SUM** – **a[i]=b[i]+c[i]**)
  - 24 байт и 1 FLOP за итерацию
- **TRIAD** – **a[i]=b[i]+d*c[i]** (FMA)
  - 24 байт и 2 FLOP за итерацию

Численные результаты. STREAM



только L1 кэш с малой проп. способностью

малая проп. способность DRAM (из-за урезанных каналов памяти)

Подсистема памяти на RISC-V пока уступает x86 и ARM

Алгоритм транспонирования матриц

- **Один из основных в линейной алгебре; memory-bound**
- **Наивная реализация (“Naïve”)**
  - То, что реализуют, не думая о производительности
  - Не может быть эффективной на любом из рассматриваемых устройств

```
1:   Transpose_baseline (double * mat, int size)
2:      for (i=0; i < size; i++)
3:            for (j=i+1; j < size; j++)
4:                  mat[i][j] = mat[j][i]
```

Алгоритм транспонирования матриц

- **Распараллеливание (“Parallel”)**
  - Обычный parallel_for
- **Блочный алгоритм (“Blocking”)**
  - Исключает ненужные загрузки данных и обеспечивает лучшее использование кэш-памяти

```
1:  Transpose_block (double * mat, int size)
2:    parallel_for (i_blk=0; i_blk<size; i_blk+=blk_size)
3:      for (j_blk=i_blk; j_blk<size; j_blk+=blk_size)
4:        for (i=i_blk; i<i_blk+blk_size; i++)
5:          for (j=j_blk +1; j<j_blk+blk_size; j++)
6:            mat[i][j] = mat[j][i]
```

Алгоритм транспонирования матриц

- **Улучшенный доступ к памяти (“Manual_blocking”)**
  - Улучшает предыдущую оптимизацию -> последовательный доступ
  - Блоки подгружаются в кэш вручную, далее транспонируются и т.д.

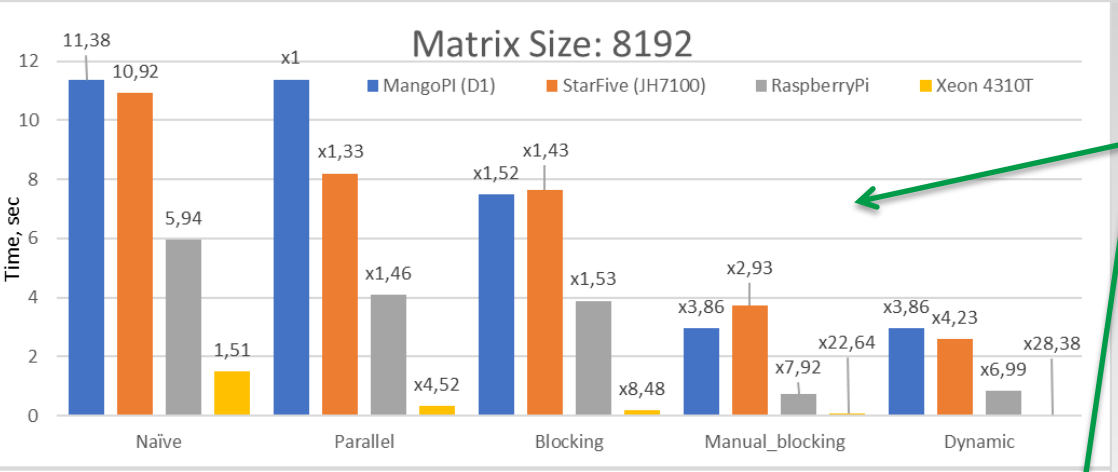
```
1:  Transpose_improvedMemAccess (double * mat, int size)
2:    parallel_for(i_blk=0; i_blk<size; i_blk+=blk_ size)
3:      double cache_blk[blk_size*blk_size]
4:      for (j_blk=i_blk; j_blk<size; j_blk+=blk_size)
5:        load_block_to_cache (i_blk, j_blk)
6:        transpose_block_in_cache()
7:        swap_block (j_blk, i_blk)
8:        transpose_block_in_cache()
9:        store_block (i_blk, j_blk)
```

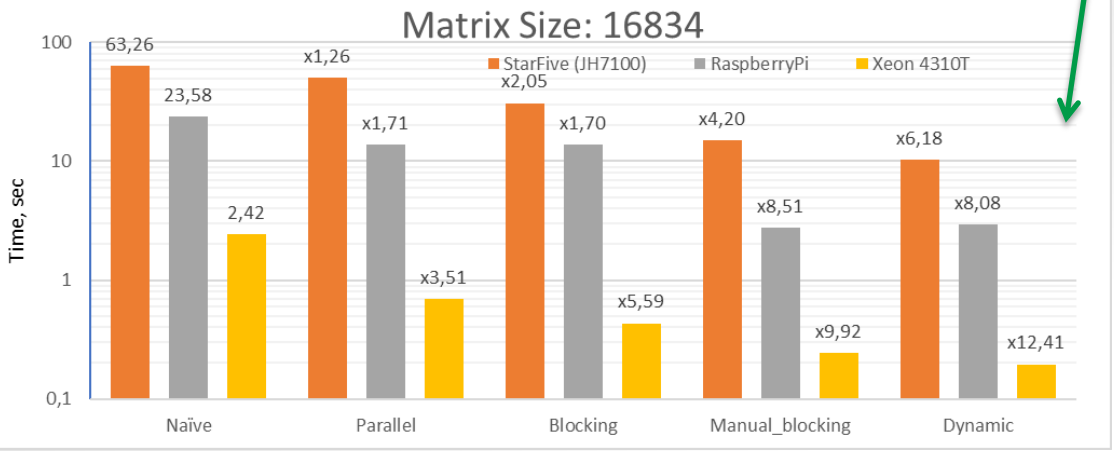
Алгоритм транспонирования матриц

- **Динамическое планирование (“Dynamic”)**
  - Итоговая версия кола
  - Отличается динамическим планированием в параллельном цикле
  - Устраняет дисбаланс при обходе строк треугольной матрицы, которые имеют разную длину

Naïve Implementation (“Naïve”)
Parallelization (“Parallel”)
Better Data Reuse: Cache Blocking (“Blocking”)
Improved Memory Access (“Manual_blocking”)
Dynamic Scheduling (“Dynamic”)

Время вычислений

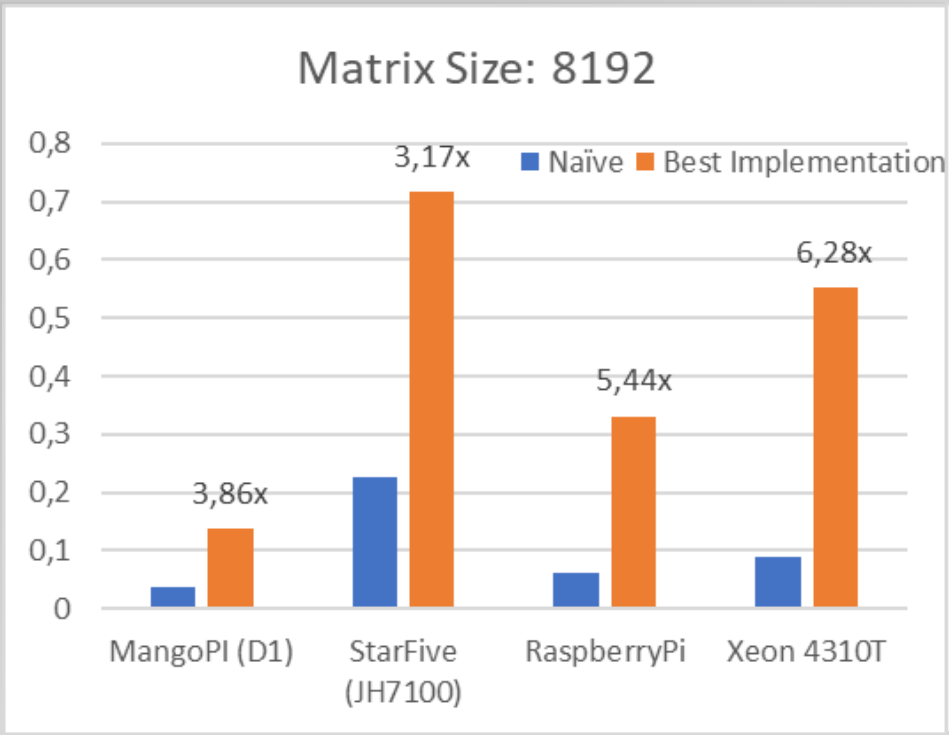


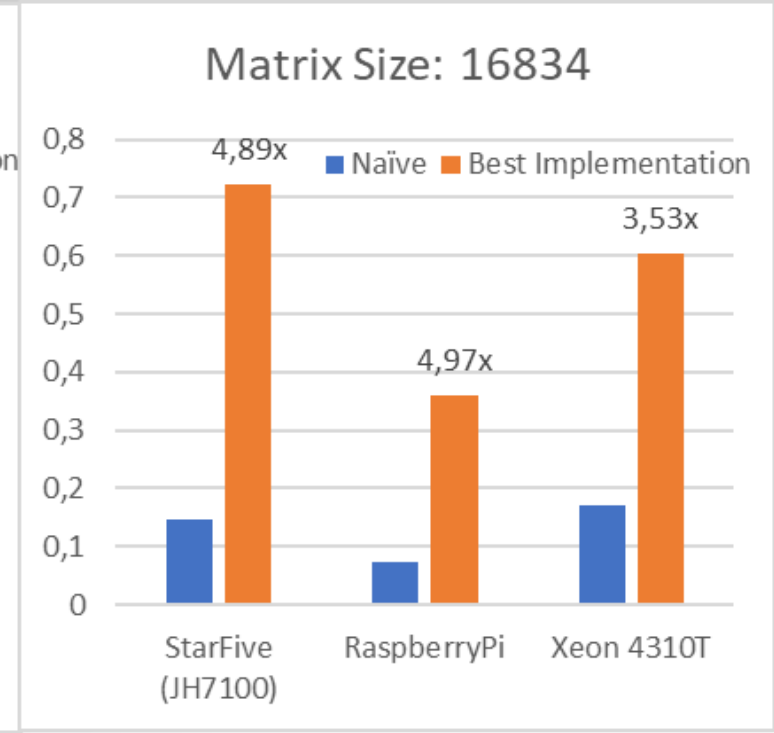


Оптимизации, проверенные на x86, **отлично работают на RISC-V устройствах**.

Raspberry Pi имеет преимущество по скорости, но меньше, чем мог бы, исходя из параметров памяти -> **в устройствах RISC-V пропускная способность памяти задействуется лучше.**

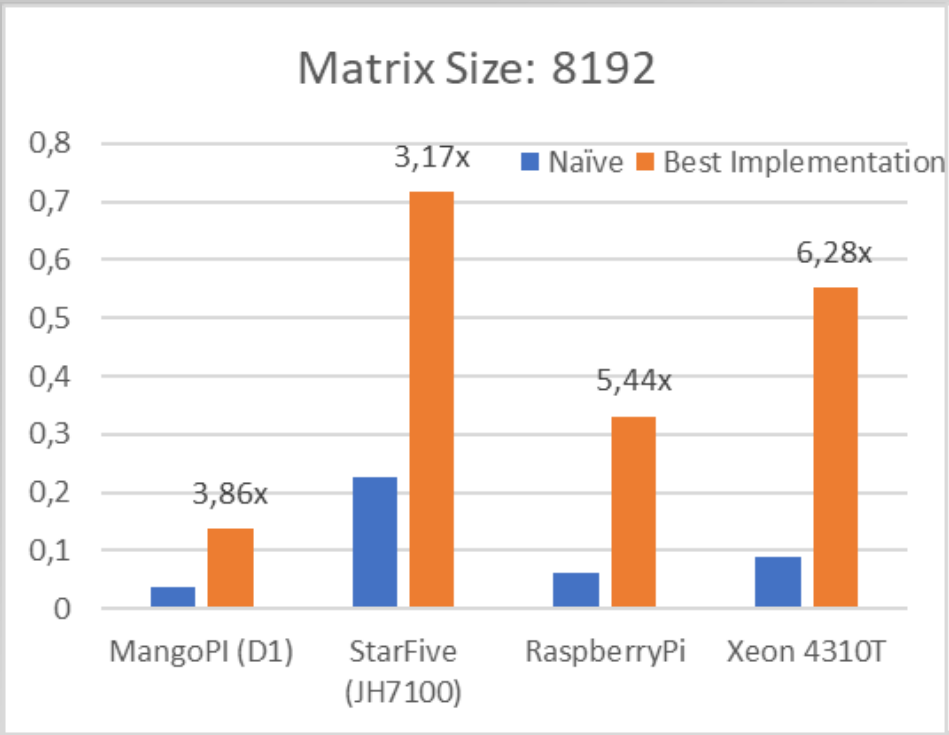
Утилизация пропускной способности памяти

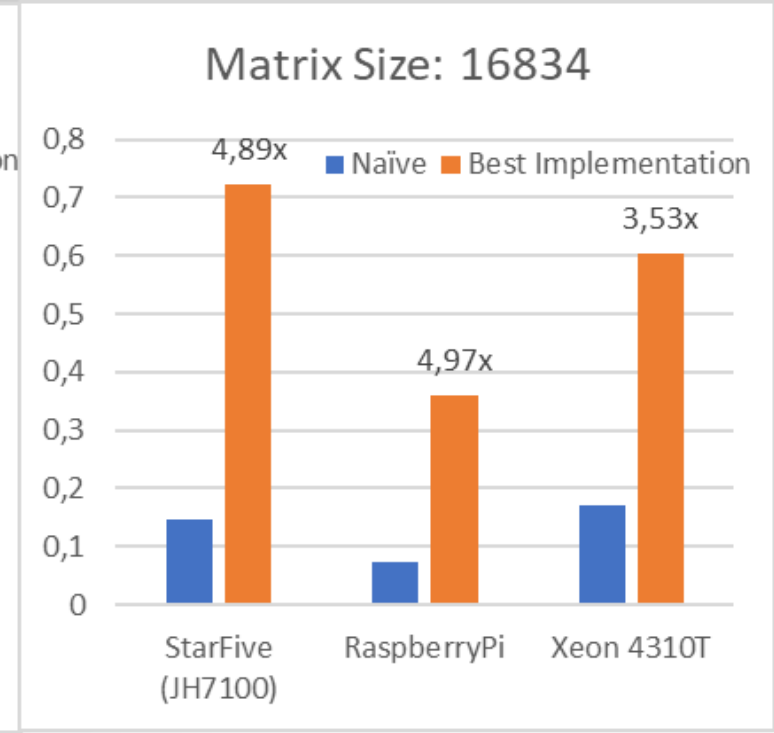




Метрика показывает, насколько эффективно переиспользуются данные, загруженные из памяти, и насколько время работы зависит от свойств подсистемы памяти

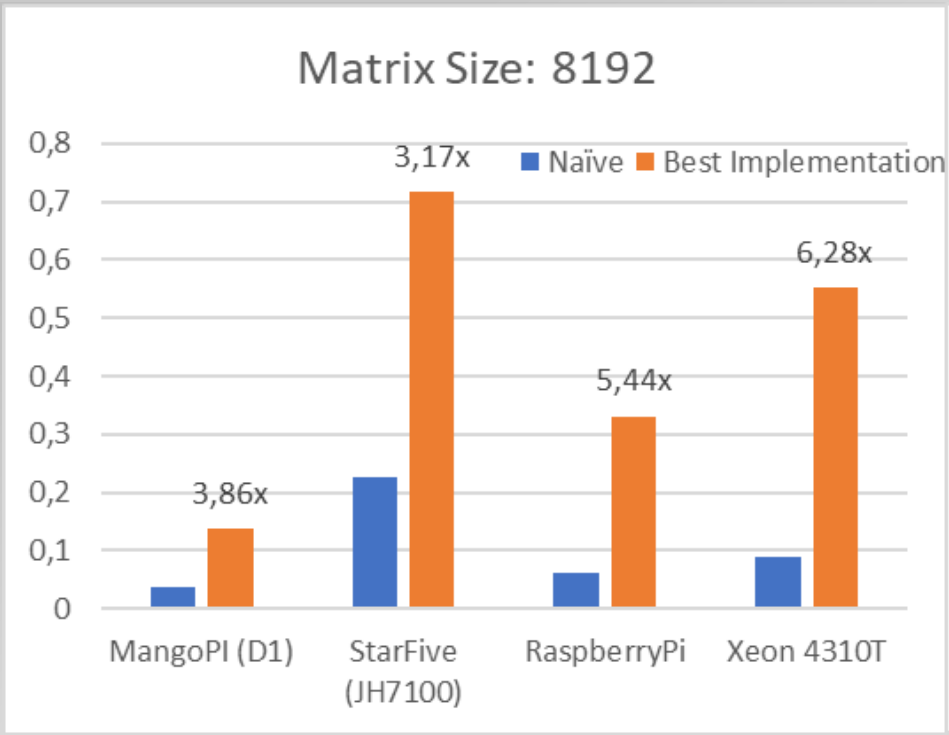
Утилизация пропускной способности памяти

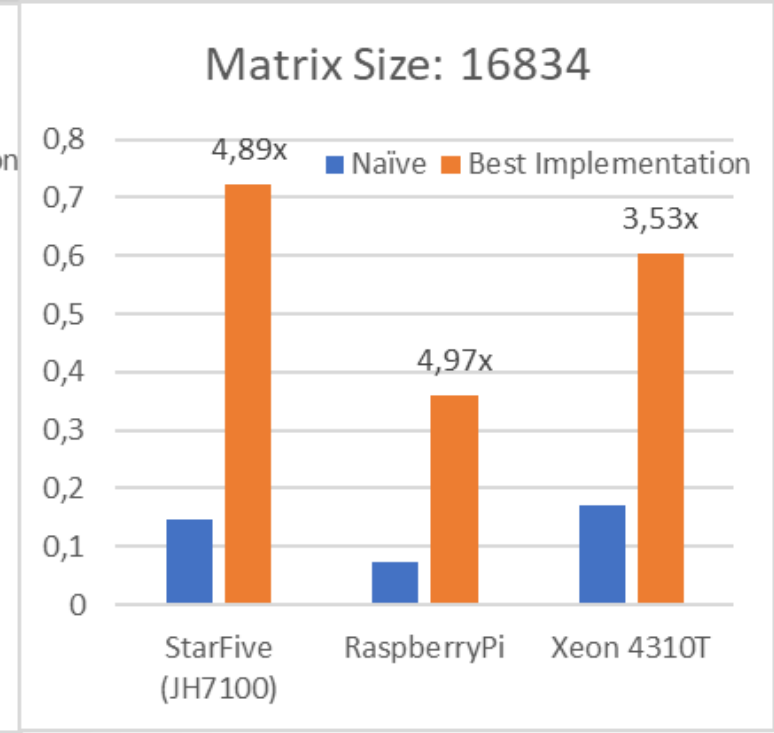




Raspberry Pi: пропускная способность памяти используется не эффективно (м.б. помогут ARM-специфичные оптимизации, но мы не внедряли специальных оптимизаций для других устройств).

Утилизация пропускной способности памяти





StarFive (JH7100): хорошая утилизация пропускной способности памяти.

Mango Pi (D1): неважная утилизация пропускной способности памяти (только 1 уровень кэш-памяти с не слишком большим улучшением относительно DRAM).

Промежуточные итоги

- C и C++ коды легко м.б. портированы на RISC-V устройства
- Общепринятые подходы к оптимизации кода на x86 хорошо работают на устройствах RISC-V (балансировка, блочная обработка)

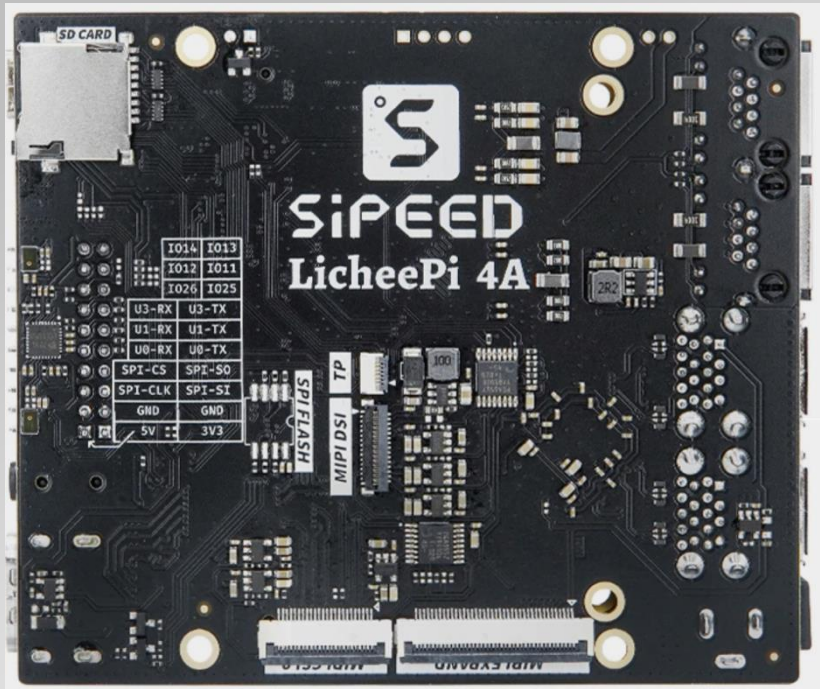
!!! Очень хорошо для разработчиков

- Несмотря на ожидаемо большую разницу в вычислительных возможностях, процессоры **RISC-V достигают хорошего использования подсистемы памяти** за счет общепринятых подходов к оптимизации

!!! Хороший потенциал для дальнейшей разработки

Прошло несколько месяцев

- Сравнительно новое устройство – Lichee Pi 4A.



RISC-V 64GCV C910*4@2GHz

· Each core contains 64KB I cache and 64KB D Cache

· Shared 1MB L2 Cache

· 8GB 64bits LPDDR4

https://wiki.sipeed.com/hardware/en/lichee/th1520/lp4a.html

Векторизация алгоритмов OpenCV на RISC-V

- **OpenCV:** универсальные интринсики (см. доклад М. Милащенко) – сильная идея, делающая код переносимым, нужна лишь их реализация для соответствующей платформы
- **Вопрос:** как улучшить векторизацию на RISC-V?

- **Идея:** использовать «блочные» векторные команды (аналог 512-битных векторов в 128-битном RVV 0.7.1)

Адаптация универсальных интринсиков для RISC-V

- **Вопрос:** как улучшить векторизацию на RISC-V?
- **Идея:** использовать «блочные» векторные команды (аналог 512-битных векторов в 128-битном RVV 0.7.1)

- **Реализация:** модификация файла универсальных интринсиков для RISC-V (**не нужно вносить изменения в реализацию алгоритмов OpenCV!**)

Вычислительные эксперименты

- Intel Xeon Silver (10 ядер)
- Lichee Pi 4A (4 ядра и 8 ГБ ОЗУ) и Mango Pi (1 ядро), векторные расширения RVV 0.7.1.
- Алгоритмы OpenCV:
  - Фильтрация
  - Эрозия
  - Bag-of-words и SVM

Результаты

- Lichee Pi 4A почти на порядок быстрее, чем Mango Pi
- Отставание от Xeon в несколько раз (но надо учесть, что оно будет больше, если алгоритмы будут хорошо распараллелены)
- Ускорение на Lichee Pi 4A от улучшенной векторизации составляет **десятки процентов** (в зависимости от алгоритма)

Выводы

- Очевидный прогресс устройств RISC-V
- Быстрое адаптация C и C++ кодов
- Стандартные техники оптимизации работают ожидаемым образом
- Существует возможность специфичных оптимизаций

- **Ждем продолжения!**

Контакты

- Иосиф Мееров, к.т.н., доцент,
  зав. каф. высокопроизводительных вычислений и системного программирования, ННГУ, Нижний Новгород
  meerov@vmk.unn.ru

- Статья #1: https://link.springer.com/content/pdf/10.1007/978-3-031-41673-6_5?pdf=chapter%20toc
- Статья #2: на рецензировании, препринт в процессе публикации.

Проект поддержан программой академического лидерства ННГУ «Приоритет-2030»

Backup slides (дополнительные детали)

Gaussian Blur algorithm (just brief description)

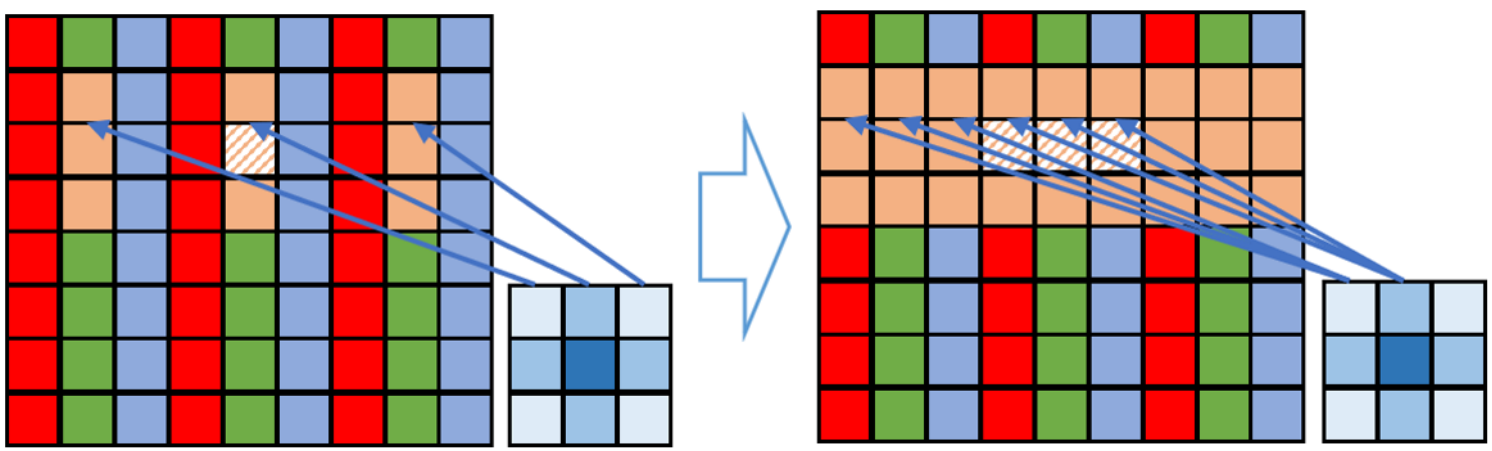
- Let there be an image (tensor) containing one or three channels at the input.
- Each pixel contains one or three intensity values, respectively, each in the range from 0 to 255, or from 0 to 1, if normalization is performed.
- The problem of filtering involves passing through the image from left to right and from top to bottom, applying the Gaussian filter kernel to the pixels, and calculating a discrete convolution.
- **The output** is an image that has the same spatial dimensions as the input one and contains updated intensity values.

Gaussian Blur algorithm (just brief description)

- Necessary in many computer vision (CV) algorithms
- There are efficient implementations of the Gaussian filter for different computing architectures, in particular, in OpenCV
- **!!!** Discrete convolution is a basic operation of convolutional neural networks. The performance of convolutions significantly affects the time of a direct pass through the neural network, which is critical in the implementing deep neural network models in real applications
- **!!! Therefore, the filtering task is the first step towards deep neural networks inference optimization on RISC-V architectures**

Gaussian Blur algorithm. Implementations

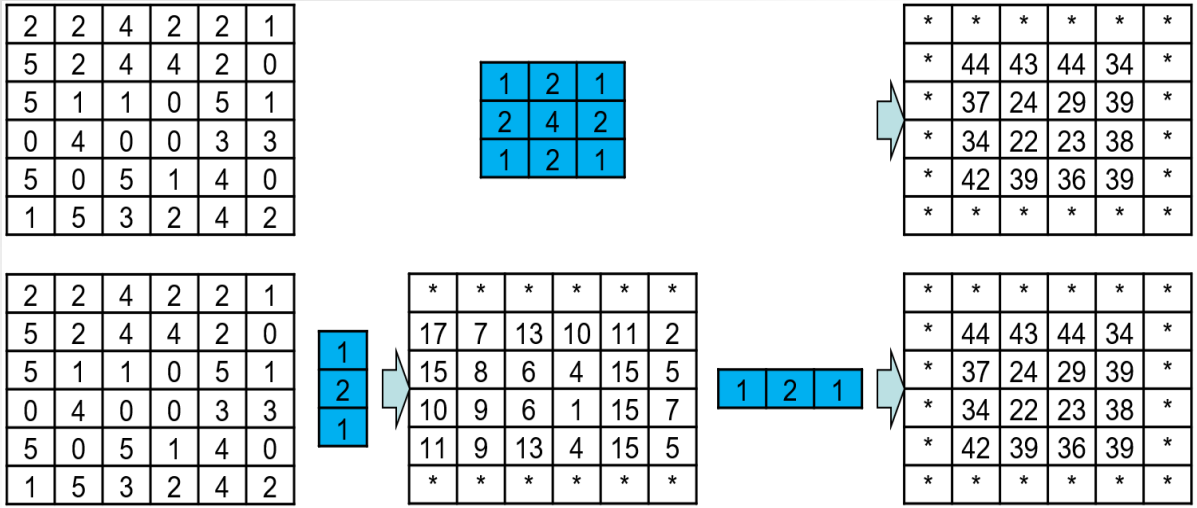
- **Naïve Implementation (“Naïve”).** As a basic implementation, we use an algorithm in which the Gaussian filter kernel is used to sequentially calculate the intensities of each pixel of the resulting image row by row
- **Unit-stride Access (“Unit-stride”).** As a first modification, we change the order of the loops so that the loop through the image channels is inside the filter kernel application loop => **unit-stride access**



Gaussian Blur algorithm. Implementations

One-dimensional kernels ("1D_kernels"). We rearrange the computations based on the following representation of the Gaussian filter: $G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}\right)\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{y^2}{2\sigma^2}}\right)$

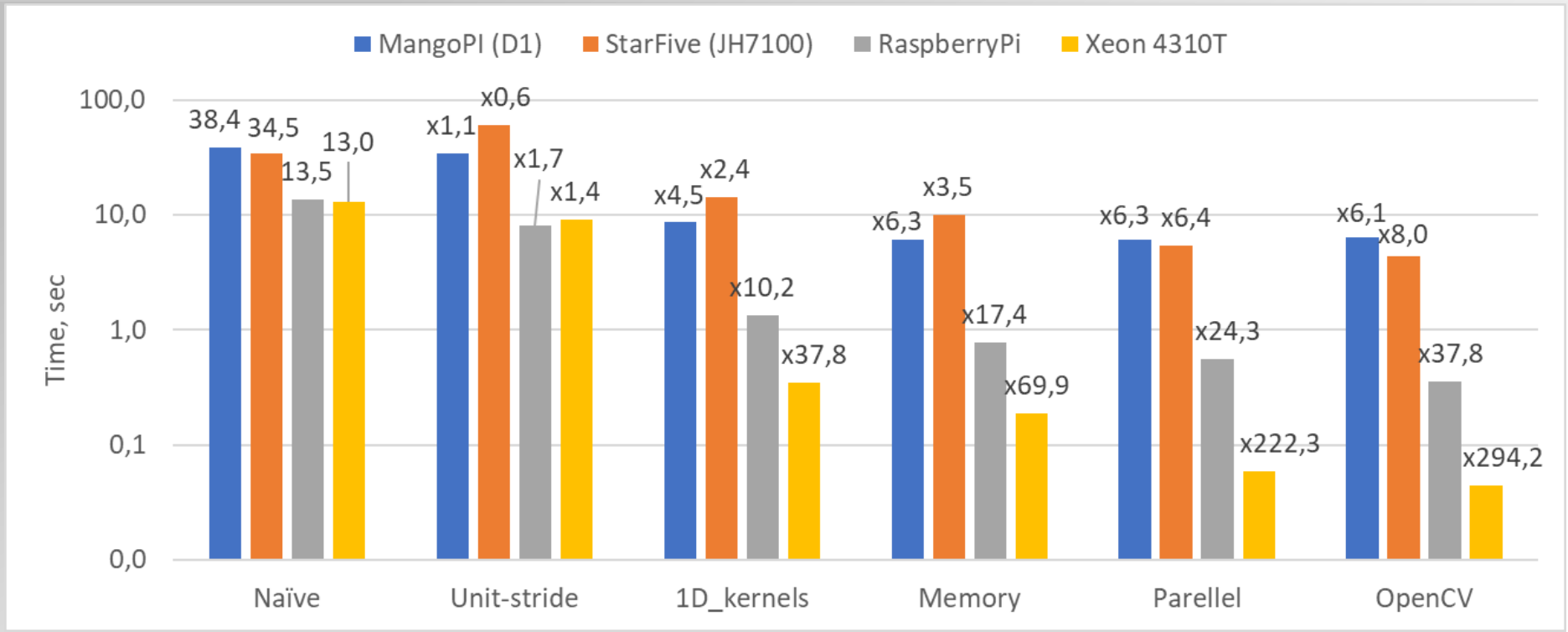
- Now we can successively apply two 1D Gaussian filter kernels instead of using a 2D kernel



Gaussian Blur algorithm. Implementations

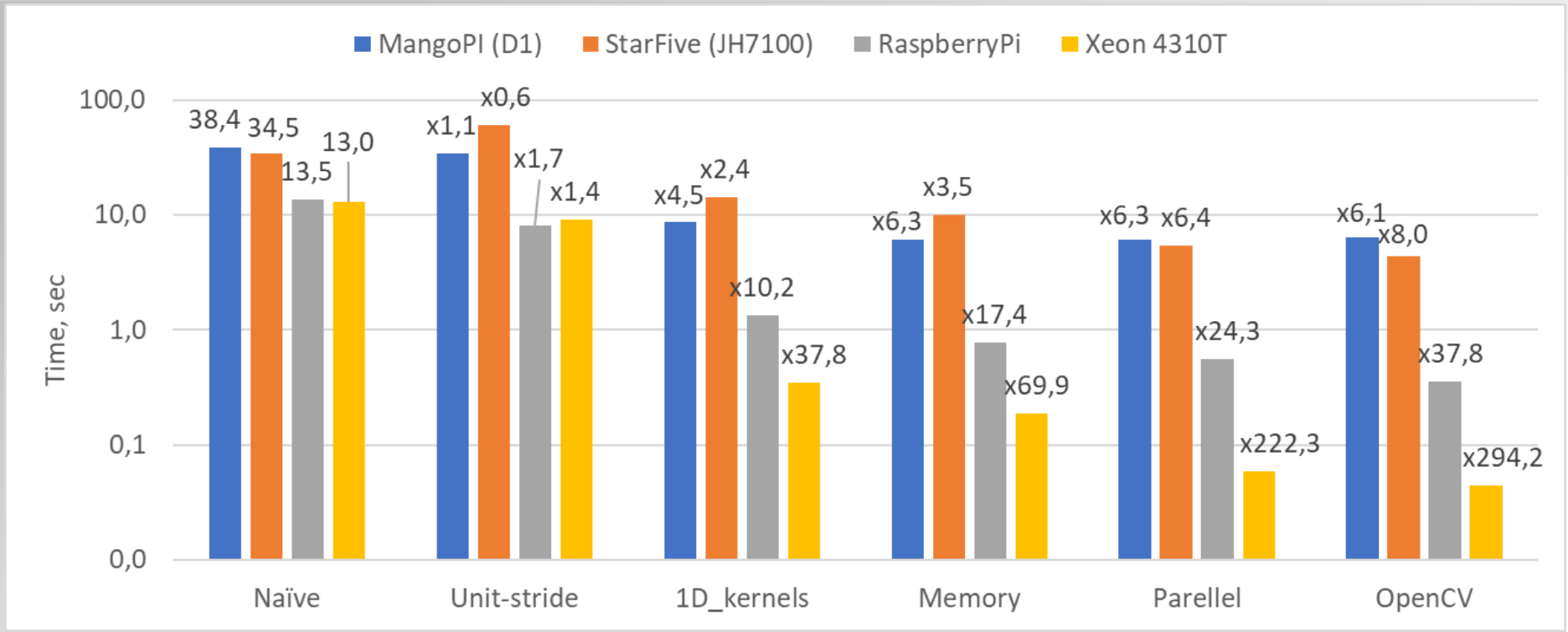
- **Improving Memory Access (“Memory”).** We use the order of loops, in which each element of the kernel interacts with the entire row from the original image matrix => improved memory access pattern
- **Parallel Implementation (“Parallel”).** The computations are independent and well-balanced, therefore we parallelize the algorithm trivially by using `#pragma parallel for` from OpenMP.
- **“OpenCV”** is a reference (optimized) implementation from OpenCV.

Numerical results. Computation time



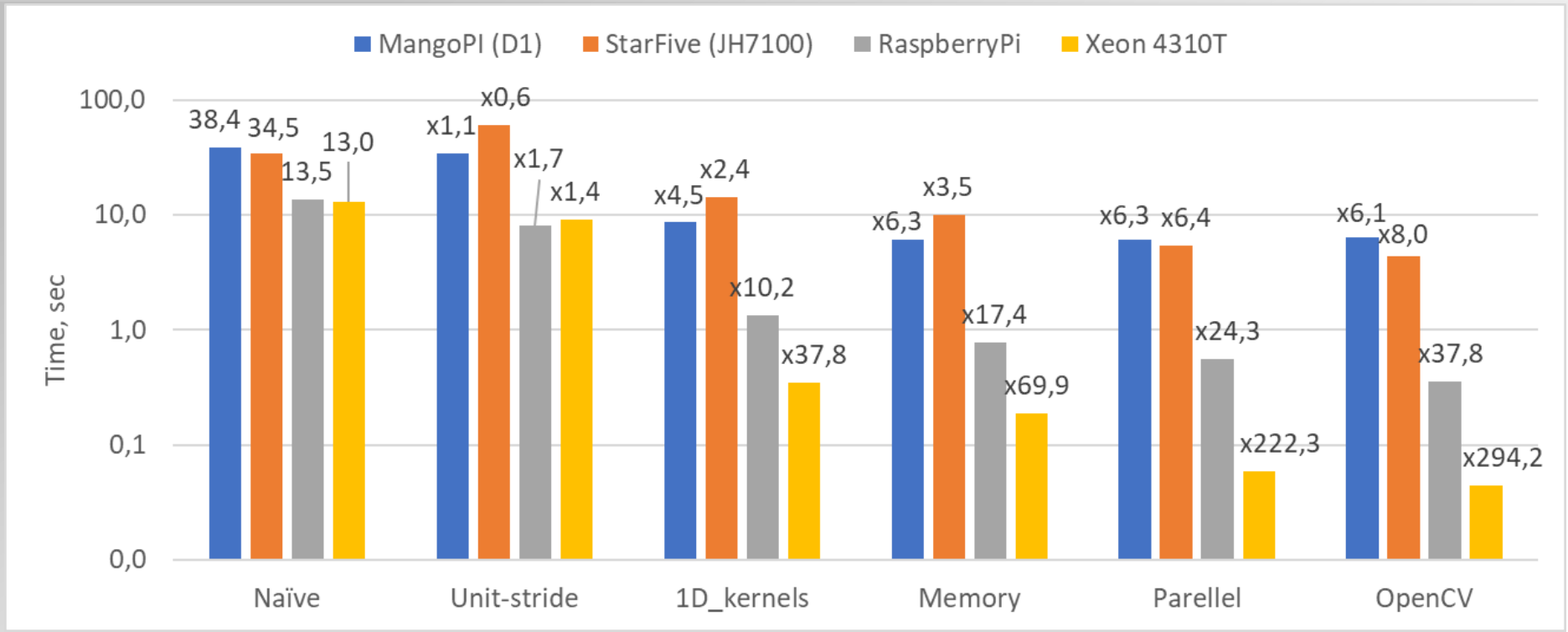
Naïve implementation lags behind OpenCV by several orders of magnitude, regardless of the device architecture

Numerical results. Computation time



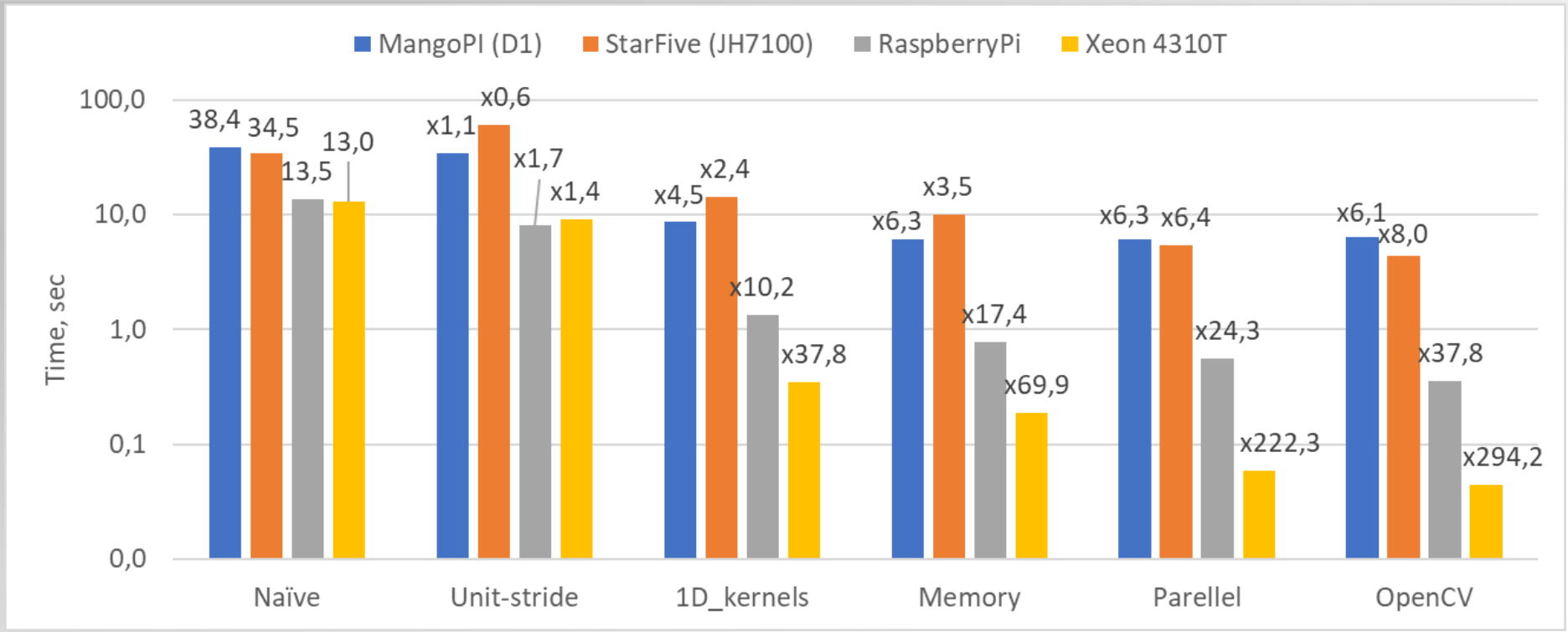
Computation time of the “Unit-stride” version is obviously better because of **sequential memory access** which us much faster due to an efficient data prefetch

Numerical results. Computation time



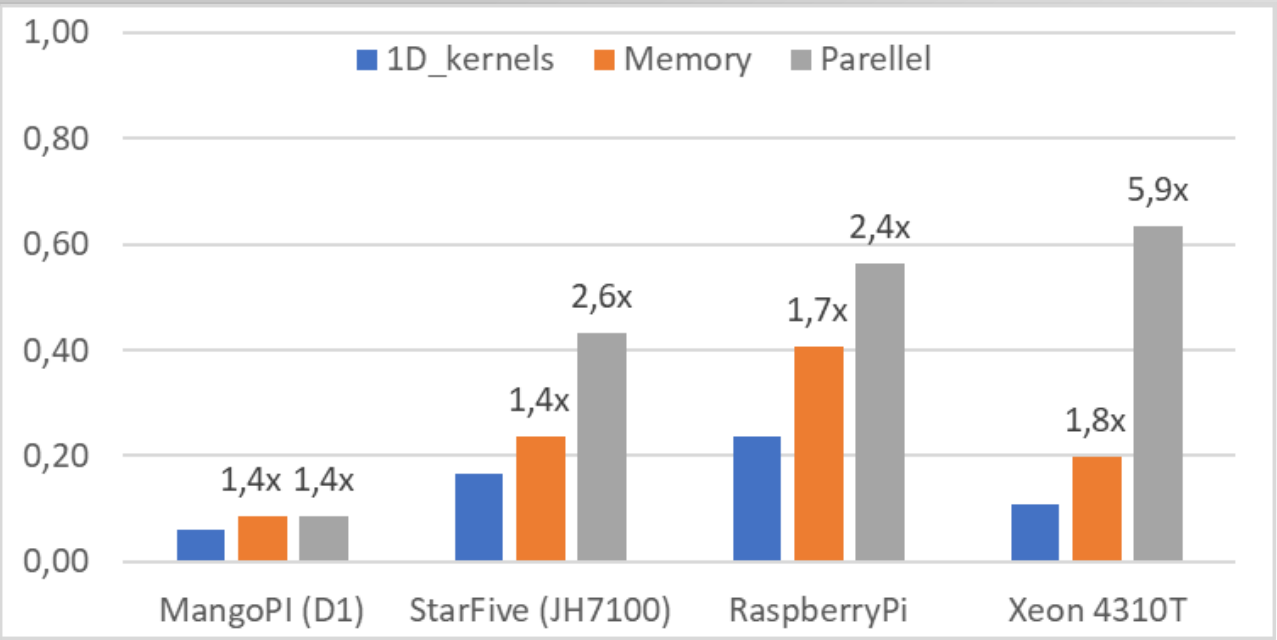
"1D_kernels“ and **“Memory”** implementations are faster, as expected. Large speedup on Xeon is due to effective vectorization of the code by the compiler.

Numerical results. Computation time



Speedup of the **“Parallel”** implementation is limited by the number of cores and memory channels.

Numerical results. Memory bandwidth utilization



Intel Xeon 4310T: the parallel algorithm provided an increase in the memory bandwidth usage metric due to the presence of a larger number of memory channels, which are not available in other devices

Mango Pi does not allow for high performance of the image filtering algorithm due to the lack of L2 cache and slow L1 cache.

StarFive lags behind RaspberryPi in memory access performance, but overall, the results are comparable.