



НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.И. ЛОБАЧЕВСКОГО

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МАТЕМАТИКИ И МЕХАНИКИ

КАФЕДРА ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЙ И СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

ЛАБОРАТОРИЯ ITLAV



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***

**Примеры оптимизации приложений
для процессоров с архитектурой RISC-V**



**LOBACHEVSKY
UNIVERSITY**

Волокитин В.Д., Козинов Е.А., Васильев Е.П.,
Кустикова В.Д., Мееров И.Б.

Содержание

- ❑ Введение
- ❑ Тестовая инфраструктура
- ❑ Метод сравнения производительности на разных вычислительных системах
- ❑ Примеры оптимизации приложений
 - Транспонирование матриц
 - Фильтрация изображений
- ❑ Векторизация на RISC-V
- ❑ Заключение

ВВЕДЕНИЕ

Введение

- ❑ **Цель лекции** – изучение и апробация некоторых принципов оптимизации вычислений для процессоров с архитектурой RISC-V на примере реализации алгоритмов транспонирования матриц, а также фильтрации изображений методом Гаусса.

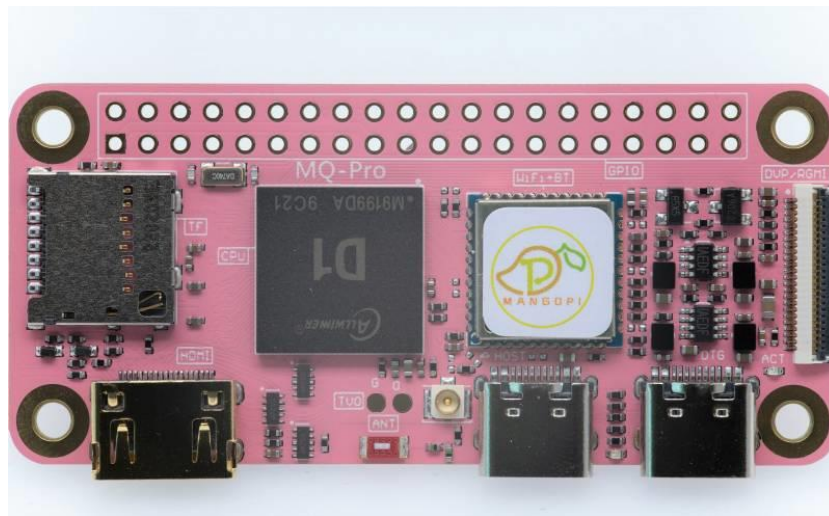
- ❑ **Задачи:**
 - Познакомиться с новой архитектурой RISC-V
 - Провести анализ производительности нескольких устройств с различной архитектурой на известных тестах STREAM
 - Реализовать алгоритмы транспонирования матриц, а также фильтрации изображений методом Гаусса
 - Проверить различные подходы к оптимизации приложений на процессорах с разной архитектурой
 - Произвести анализ результатов оптимизации

ТЕСТОВАЯ ИНФРАСТРУКТУРА

Тестовая инфраструктура. Конфигурация 1

❑ Mango Pi MQ-Pro (D1)

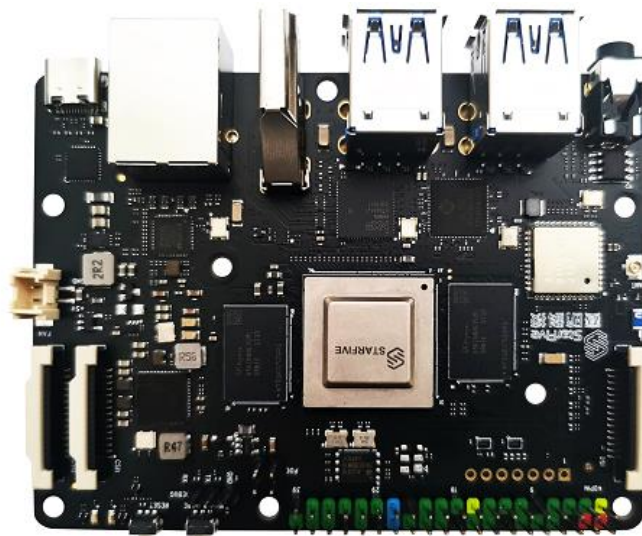
- Процессор Allwinner D1 (1 x XuanTie C906, 1GHz)
- 1GB DDR3L RAM
- Операционная система Ubuntu 22.10 (RISC-V edition)
- Компилятор GCC 12.2



Тестовая инфраструктура. Конфигурация 2

❑ 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



Тестовая инфраструктура. Конфигурация 3

❑ LicheePi 4A (TH1520)

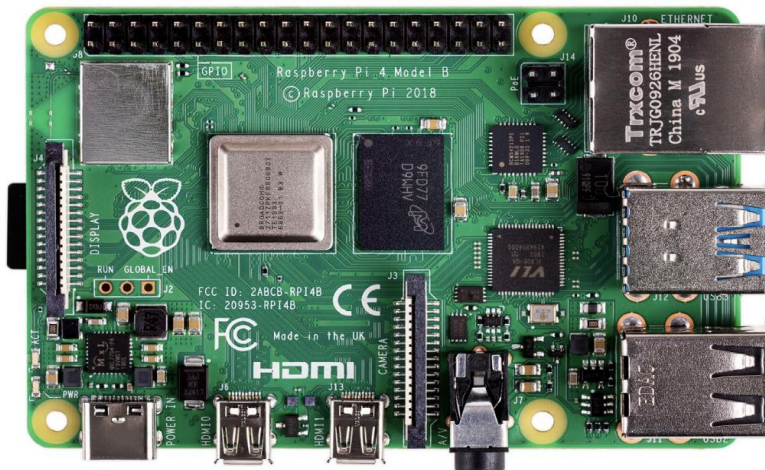
- Процессор T-Head TH1520 (4 x XuanTie C910, 1.85 (2.5) GHz)
- 8-16GB LDDR4X RAM
- Операционная система Debian 12 (RISC-V edition)
- Компилятор GCC 12.2



Тестовая инфраструктура. Конфигурация 4

❑ Raspberry Pi 4 модель B

- Процессор Broadcom BCM2711 (4 x Cortex-A72, up to 1.5 GHz)
- 4GB LPDDR4 RAM
- Операционная система Ubuntu 20.04
- Компилятор GCC 9.4



Тестовая инфраструктура. Конфигурация 5

❑ Сервер с Intel Xeon

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



МЕТОД СРАВНЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ НА РАЗНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ

Методы оценки производительности.

Вычислительные эксперименты.

- ❑ Проводимые вычислительные эксперименты:
 - STREAM – определение пропускной способности памяти (<https://www.cs.virginia.edu/stream/>).
 - Транспонирование матриц – один из основных алгоритмов линейной алгебры. Рассматривается случай плотных матриц.
 - Фильтрация изображения с ядром фильтра Гаусса – пример реализации и оптимизации алгоритма.
- ❑ Для алгоритмов транспонирования матриц и фильтра Гаусса, представлено несколько реализаций:
 - Реализации показывают, как типичные методы оптимизации работы с памятью, хорошо работающие на процессорах x86 и ARM, работают на устройствах RISC-V.

Методы оценки производительности.

Вычислительные эксперименты.

- ❑ Время вычислений.
 - Основной показатель эффективности.
 - Показатель недостаточный, так как сравниваются маломощные устройства в начале жизненного цикла на основе архитектуры RISC-V и процессор с долгой историей развития Intel Xeon.
- ❑ Пропускная способность памяти.
- ❑ Ускорение относительно наивной реализации.
- ❑ Утилизация подсистемы памяти.
 - Метод вычисления.
 - Вычисляем отношение объема данных, которые необходимо переместить между DRAM и CPU, ко времени вычислений.
 - Делим рассчитанное таким образом значение на достигнутую пропускную способность памяти, измеренную бенчмарком STREAM.

ПРИМЕРЫ ОПТИМИЗАЦИИ ПРИЛОЖЕНИЙ

Тесты STREAM

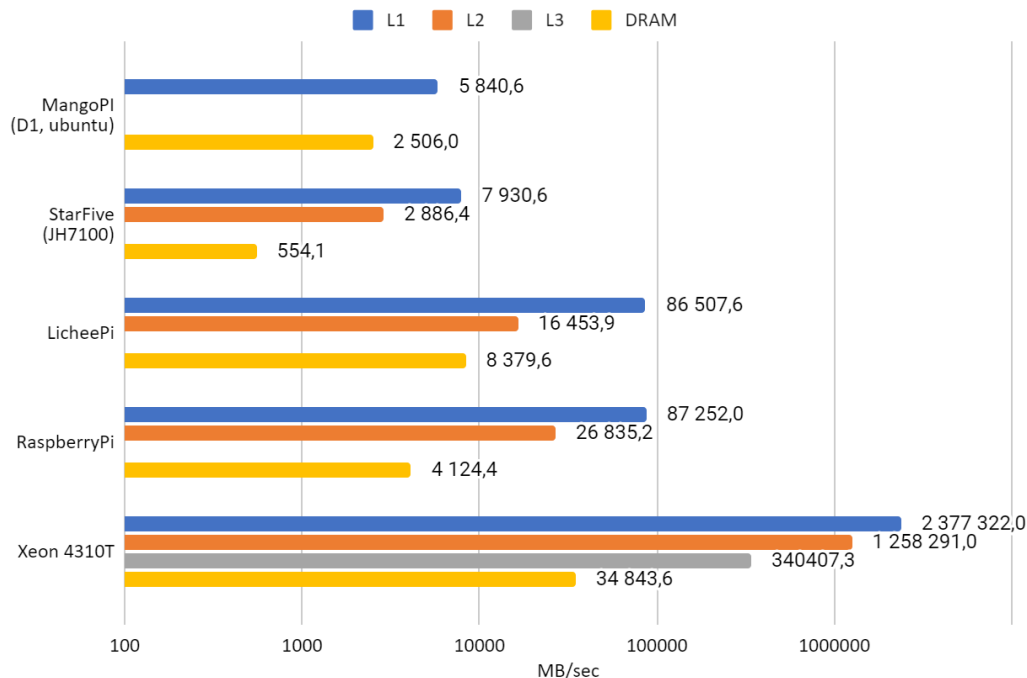
- **Цель запуска STREAM** – измерить эффективность работы памяти.
- STREAM использует 4 теста с разными значениями bytes/iter и FLOPs/iter:
 - **COPY** – простое копирование из одного массива в другой ($a[i]=b[i]$). Эта операция передает 16 байтов за итерацию и не выполняет вычисления с плавающей запятой.
 - **SCALE** – копирование из одного массива в другой с умножением на константу ($a[i]=d*b[i]$). Эта операция передает 16 байт за итерацию и выполняет 1 FLOPs/iter.
 - **SUM** – сумма элементов из двух массивов записывается в третий массив ($a[i]=b[i]+c[i]$). Эта операция передает 24 байта за итерацию, но по-прежнему выполняет 1 FLOPs/iter.
 - **TRIAD** – FMA из элементов двух массивов ($a[i]=b[i]+d*c[i]$) записывается в третий массив. Эта операция передает 24 байта за итерацию и выполняет 2 FLOPs/iter.

❑ Особенности проведения экспериментов:

- В тестах подбирались размеры массивов таким образом, чтобы они не вытеснялись из памяти рассматриваемого уровня (L1, L2, L3, DRAM) и не могли эффективно кэшироваться в более быстрой памяти.
- Для каждого устройства были проведены тесты для каждого уровня памяти (L1, L2, L3, DRAM).
- Запускались многопоточная (для разделяемой памяти) или последовательная (для отдельного ресурса, например L1-кеша) версия STREAM.
 - В последовательных экспериментах результаты умножались на количество ядер.
- Эксперименты запускались несколько раз и выбирались наилучшие достигнутые значения.

Тесты STREAM

□ Пропускная способность памяти на различных устройствах



Mango Pi (Allwinner D1)
Имеет только кэш L1.

StarFive VisionFive (JH7100)
Имеет низкую пропускную
способность DRAM-памяти.

LicheePi 4A (TH1520)
Имеет схожие характеристики
с RaspberryPi 4

Алгоритм транспонирования плотной матрицы

□ Рассматриваемые реализации

- **Naïve** (наивная реализация).
- **Parallel** (параллельная реализация).
- **Blocking** (блочная реализация).
- **Manual_blocking** (использование локальной памяти).
- **Dynamic** (использование динамического планирования в блочной версии кода).

Алгоритм транспонирования плотной матрицы

□ Naïve (наивная реализация)

```
void transpose_baseline(double* mat, int n) {  
    for (int i = 0; i < n; i++)  
        for (int j = i + 1; j < n; j++)  
            std::swap(mat[i * n + j], mat[j * n + i]);  
}
```

□ Parallel (параллельная реализация).

```
void transpose_baseline_omp(double* mat, int n) {  
    #pragma omp parallel for  
    for (int i = 0; i < n; i++)  
        for (int j = i + 1; j < n; j++)  
            std::swap(mat[i * n + j], mat[j * n + i]);  
}
```

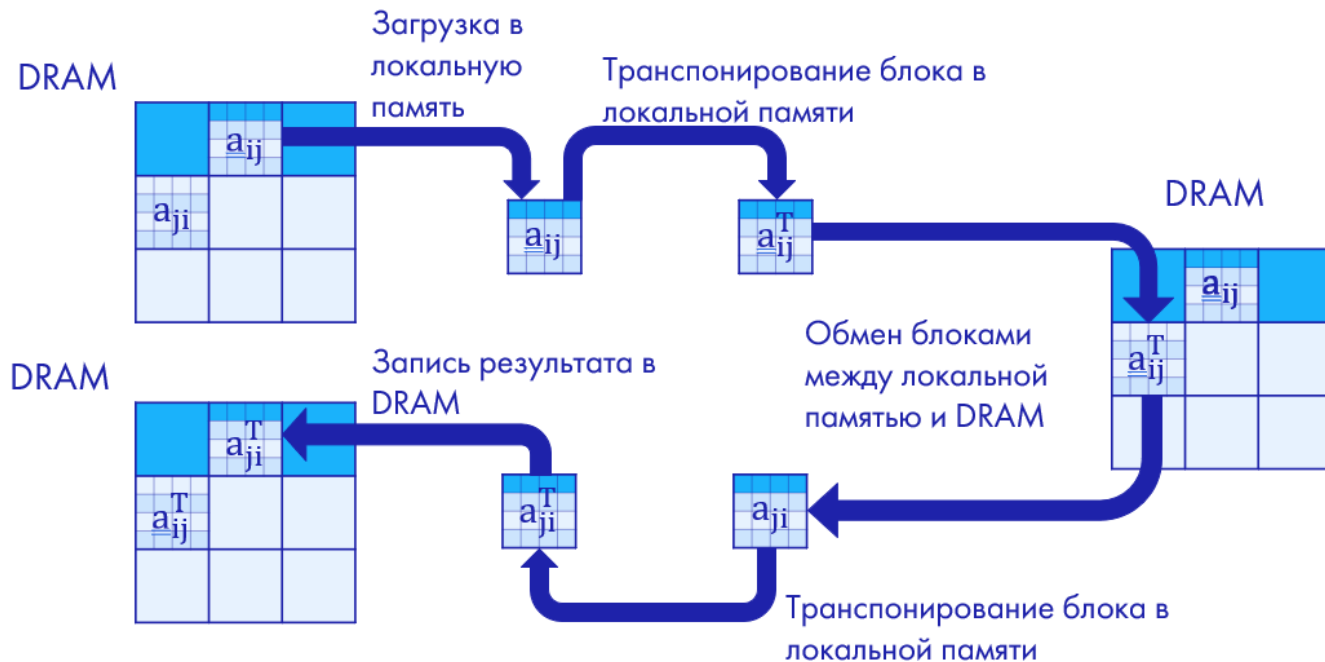
Алгоритм транспонирования плотной матрицы

□ Blocking (блочная реализация)

```
void transpose_block_omp(double* mat, int n) {
    int block_index = 0;    const int block_size = 64;
#pragma omp parallel for
    for (int block_index = 0; block_index < n / block_size; block_index++) {
        // Транспонирование блока на диагонали
        int block_start = block_index * block_size;
        for (int i = block_start; i < block_start + block_size; i++)
            for (int j = i + 1; j < block_start + block_size; j++)
                std::swap(mat[i * n + j], mat[j * n + i]);
        // Транспонирование блоков вне диагонали
        int bj = block_start + block_size;
        for (; bj + block_size < n; bj += block_size)
            for (int i = block_start; i < block_start + block_size; i++)
                for (int j = bj; j < bj + block_size; j++)
                    std::swap(mat[i * n + j], mat[j * n + i]);
        // Обработка границ
        for (int i = block_start; i < block_start + block_size; i++)
            for (int j = bj; j < n; j++)
                std::swap(mat[i * n + j], mat[j * n + i]);
    }
    block_index = n / block_size;
    for (int i = block_index * block_size; i < n; i++)
        for (int j = i + 1; j < n; j++)
            std::swap(mat[i * n + j], mat[j * n + i]);
}
```

Алгоритм транспонирования плотной матрицы

□ Manual_blocking (использование локальной памяти).



Алгоритм транспонирования плотной матрицы

□ **Manual_blocking** (использование локальной памяти).

```
void transpose_cache_omp(double* mat, int n)
{
    int block_index = 0;
    const int block_size = 64;
#pragma omp parallel for
    for (int block_index = 0; block_index < n / block_size; block_index++)
    {
        // Транспонирование блока на диагонали
        int block_start = block_index * block_size;
        for (int i = block_start; i < block_start + block_size; i++)
            for (int j = i + 1; j < block_start + block_size; j++)
                std::swap(mat[i * n + j], mat[j * n + i]);
        ...
    }
}
```

Алгоритм транспонирования плотной матрицы

□ Manual_blocking (использование локальной памяти).

```
... // Транспонирование блоков вне диагонали
double mat_a[block_size * block_size];
for (block_j = block_start + block_size; block_j + block_size < n;
     block_j += block_size) {
    for (int j = block_j, k1 = 0; j < block_j + block_size; j++, k1++)
        for (int k2 = 0; k2 < block_size; k2++)
            mat_a[k1 * block_size + k2] = mat[j * n + block_start + k2];
    for (int k1 = 0; k1 < block_size; k1++)
        for (int k2 = k1 + 1; k2 < block_size; k2++)
            std::swap(mat_a[k1 * block_size + k2], mat_a[k2 * block_size + k1]);
    for (int i = block_start, k2 = 0; i < block_start + block_size; i++, k2++)
        for (int k1 = 0; k1 < block_size; k1++)
            std::swap(mat_a[k2 * block_size + k1], mat[i * n + block_j + k1]);
    for (int k1 = 0; k1 < block_size; k1++)
        for (int k2 = k1 + 1; k2 < block_size; k2++)
            std::swap(mat_a[k1 * block_size + k2], mat_a[k2 * block_size + k1]);
    for (int j = block_j, k1 = 0; j < block_j + block_size; j++, k1++)
        for (int k2 = 0; k2 < block_size; k2++)
            mat[j * n + block_start + k2] = mat_a[k1 * block_size + k2];
... <Обработка границ>
```

Алгоритм транспонирования плотной матрицы

□ **Dynamic** (использование динамического планирования).

```
void transpose_cache_omp_dyn(double* mat, int n)
{
    int block_index = 0;
    const int block_size = 64;
#pragma omp parallel for schedule (dynamic)
    for (int block_index = 0; block_index < n / block_size; block_index++)
    {
        ...
        <Транспонирование Manual_blocking>
        ...
    }
}
```

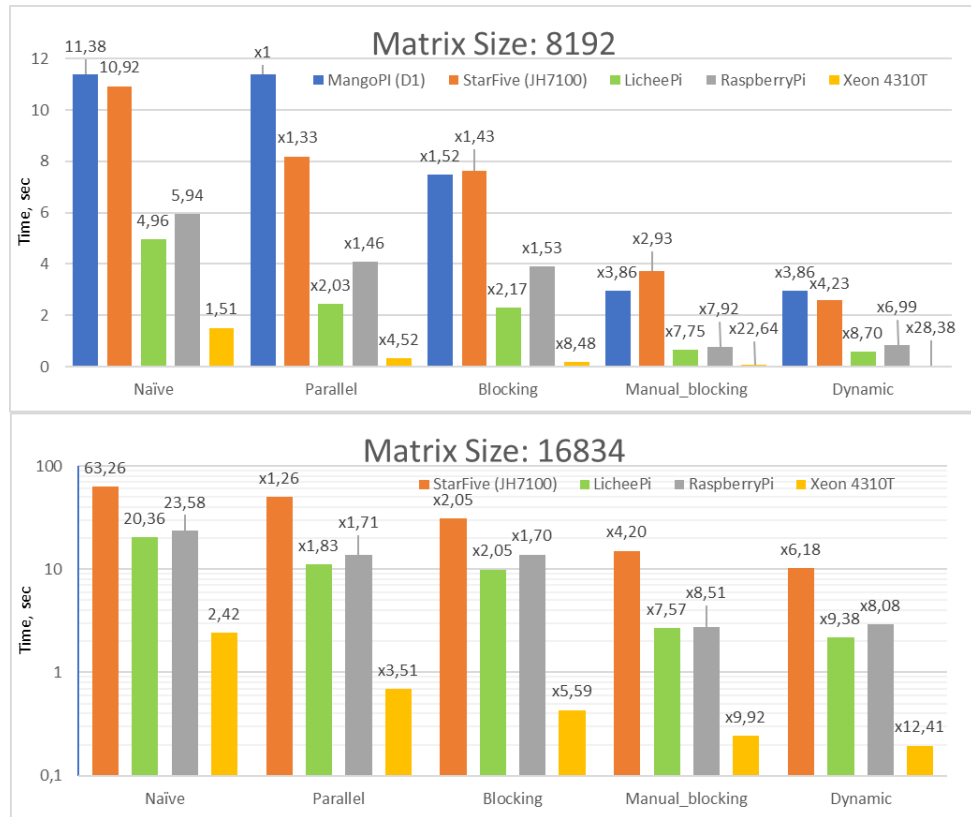
Алгоритм транспонирования плотной матрицы

□ Результаты вычислительных экспериментов

– На графиках

- Время вычислений
- Ускорение относительно базовой реализации

Оптимизации, разработанные для архитектуры x86, хорошо работают и на устройствах RISC-V.



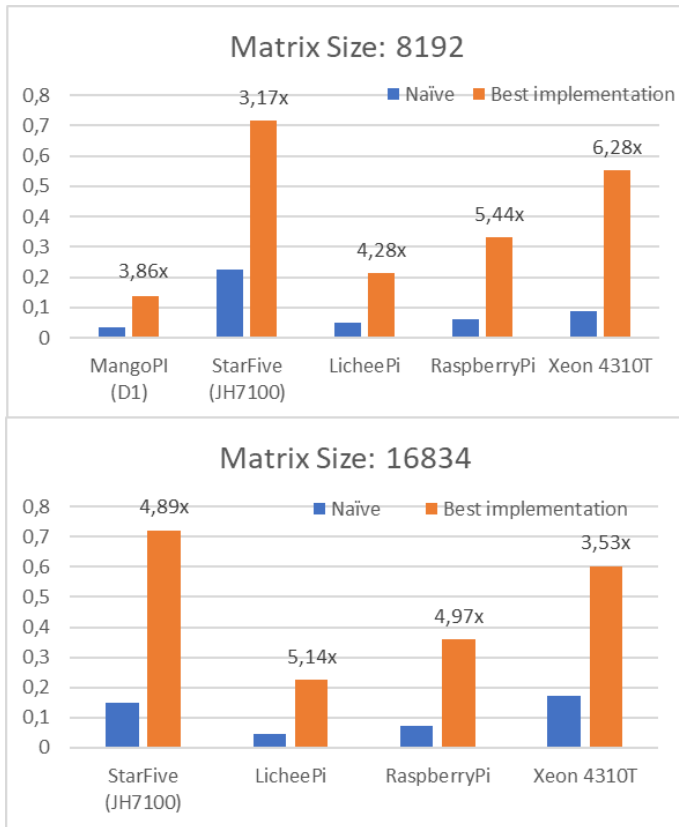
Алгоритм транспонирования плотной матрицы

□ Результаты вычислительных экспериментов

- На графиках
- Утилизация подсистемы памяти (синий – базовая реализация, оранжевый – финальная реализация).

Метрика показывает:

- Насколько эффективно реализовано повторное использование данных, загружаемых из памяти
- Насколько существенно время вычислений зависит от свойств подсистемы памяти
- Оптимальное значение, равное единице, во многих случаях недостижимо, но близость к единице свидетельствует об эффективном использовании памяти.



Алгоритм транспонирования плотной матрицы

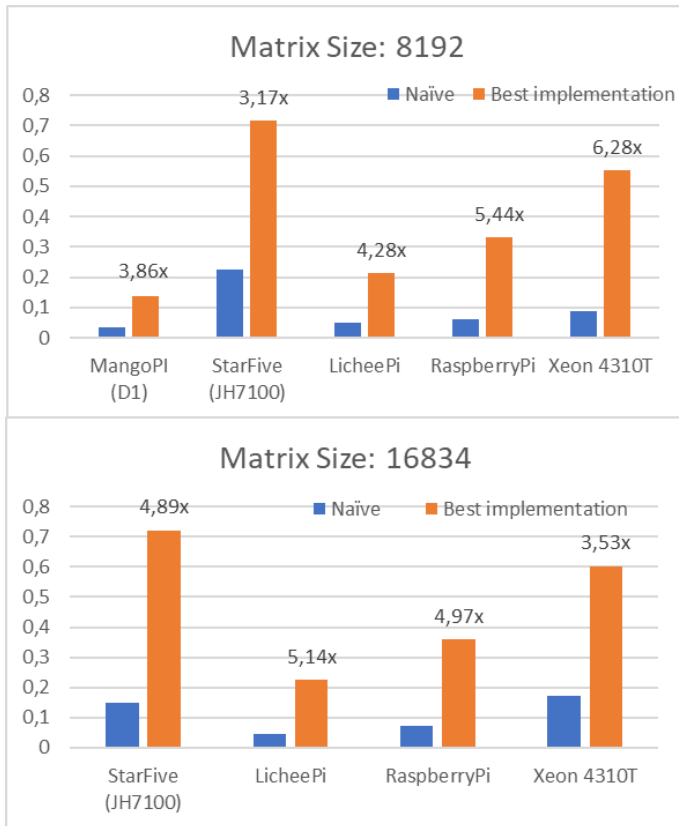
□ Результаты вычислительных экспериментов

Mango Pi показал плохие результаты улучшения.
У Mango Pi только один уровень кэш-памяти не сильно быстрее DRAM, что и влияет на производительность.

StarFive показал хорошие результаты.
У StarFive низкая пропускная способность памяти, но реализовано два канала памяти на два ядра.

Raspberry Pi и Lichee Pi показали схожие результаты.
Низкий абсолютный показатель, вероятно, связан с отсутствием архитектурно-специфических оптимизаций под конкретные платы и процессора.

Все устройства показывают приемлемый рост показателя для достаточно больших матриц.
Оптимизации для архитектуры x86 хорошо работают и на устройствах RISC-V.

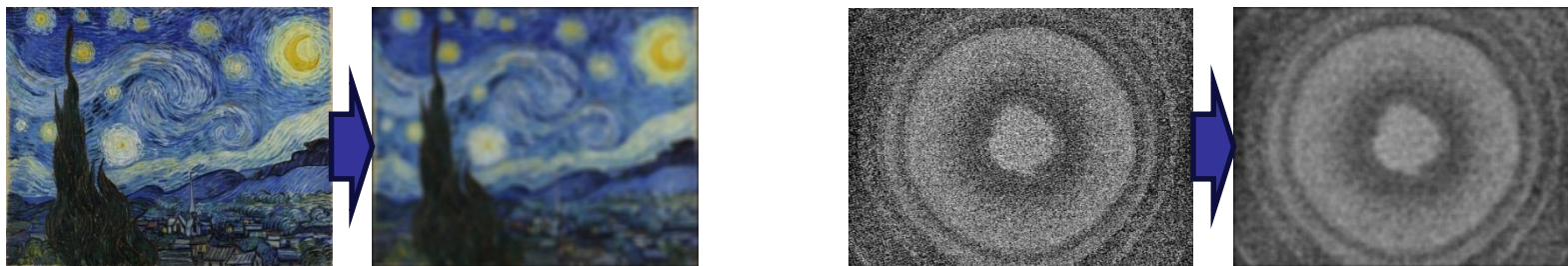


Реализация фильтра Гаусса

□ Причины выбора фильтра Гаусса:

1. Алгоритм необходим во многих алгоритмах компьютерного зрения (CV) для предварительной подготовки входных данных.
2. Существуют эффективные реализации фильтра Гаусса для разных вычислительных архитектур, в частности, в OpenCV (можно сравнить производительность).
3. Дискретная свертка является базовой операцией сверточных нейронных сетей, которые обычно используются в CV-приложениях.

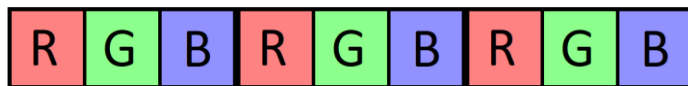
□ Пример применения фильтра Гаусса:



Реализация фильтра Гаусса

□ Методы хранения изображений

- Изображение может быть представлено набором пикселей
- Каждый пиксель имеет свой цвет. Цвет задается одним или несколькими значениями интенсивности
 - В черно-белых изображениях цвет представлен одним значением интенсивности. В цветных – несколькими.
- Структура хранения изображения: последовательный набор интенсивностей для каждого пикселя.



- Значения интенсивности могут быть заданы как целыми числами типа `unsigned char`, так и вещественными числами типа `float`
- При наложении фильтра Гаусса будем использовать вещественный тип данных

Реализация фильтра Гаусса

- Фильтр Гаусса задается ядром:

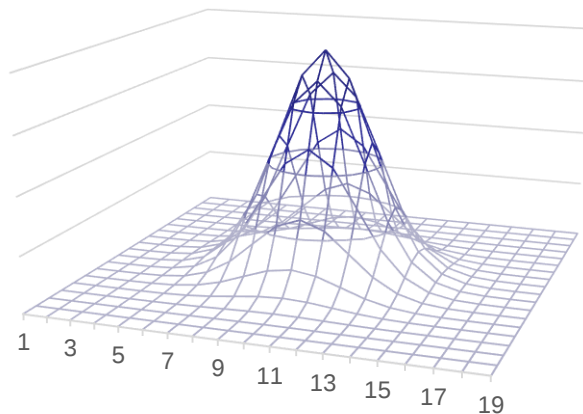
$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- x, y – координаты точки
- σ – среднеквадратическое отклонение (от значения зависит сила размытия)

- Пример матрицы ядра ($N=5, \sigma=2$)

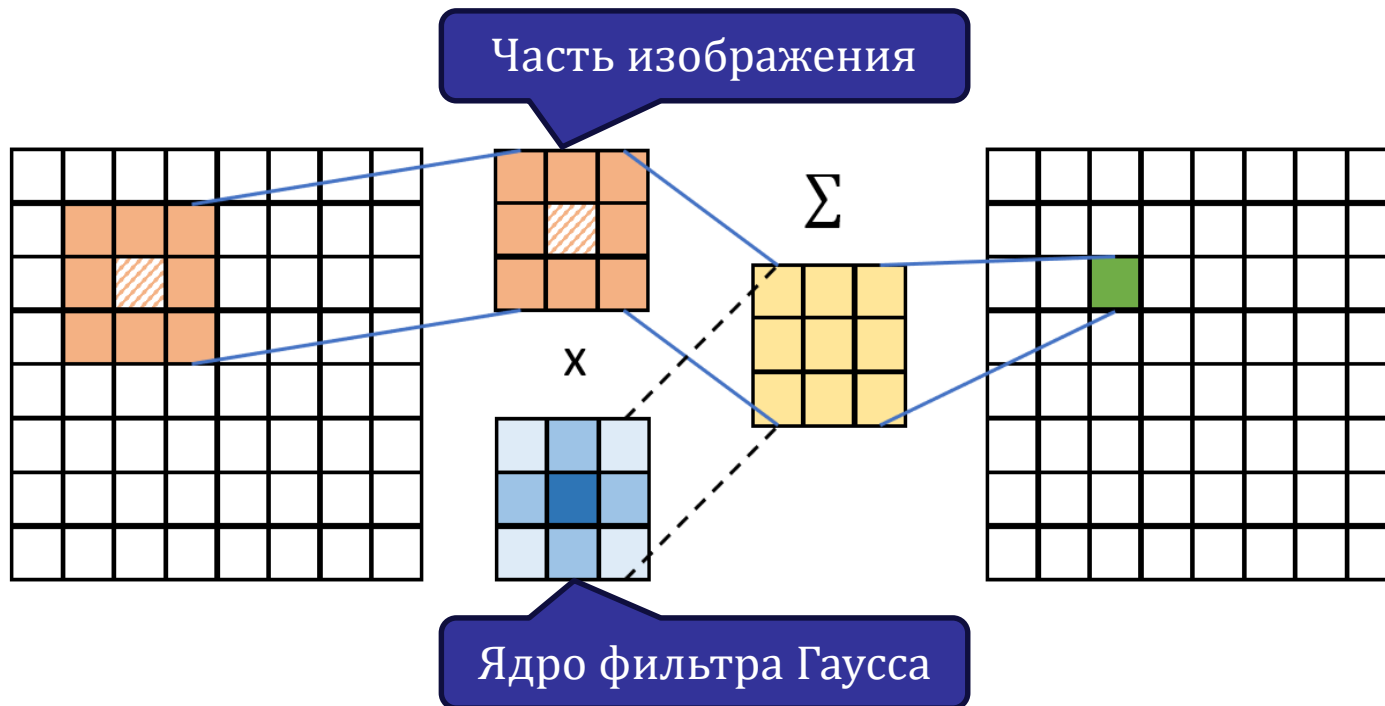
0,015	0,021	0,024	0,021	0,015
0,021	0,031	0,035	0,031	0,021
0,024	0,035	0,040	0,035	0,024
0,021	0,031	0,035	0,031	0,021
0,015	0,021	0,024	0,021	0,015

$N=19$ $\sigma=2$



Реализация фильтра Гаусса

- Общая схема применения фильтра для получения значений каждого пикселя:



Реализация фильтра Гаусса

□ Рассматриваемые реализации

- **Naïve** (наивная реализация).
- **Unit-stride** (последовательный доступ к памяти).
- **1D_kernels** (алгоритмическая оптимизация).
- **Memory** (улучшение доступа к памяти).
- **Parallel** (параллельная реализация).

Реализация фильтра Гаусса

□ Naïve (наивная реализация)

//Заполнение матрицы фильтра

```
float* create2DFilter(int n, float sigma)
{
    float* filter = new float[n * n];
    float coeff = 1.0f / (2.0f * PI * sigma * sigma);
    int middle = n / 2;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            float x = j - middle;
            float y = i - middle;
            filter[i * n + j] = coeff * expf(-(x * x + y * y) / (2.0f * sigma * sigma));
        }
    }
    return filter;
}
```

Реализация фильтра Гаусса

□ Naïve (наивная реализация)

// Базовая реализация фильтра

```
void filterBase(cv::Mat& dist, cv::Mat& src, float sigma, int sizeFilter)
{
```

```
    float* filter = create2DFilter(sizeFilter, sigma);
```

```
    int w = src.size().width;
```

```
    int h = src.size().height;
```

```
    float* srcData = (float *)src.data;
```

```
    float* distData = (float *)dist.data;
```

```
    int middle = sizeFilter / 2;
```

```
    const int cntChannel = 3;
```

```
    ...
```

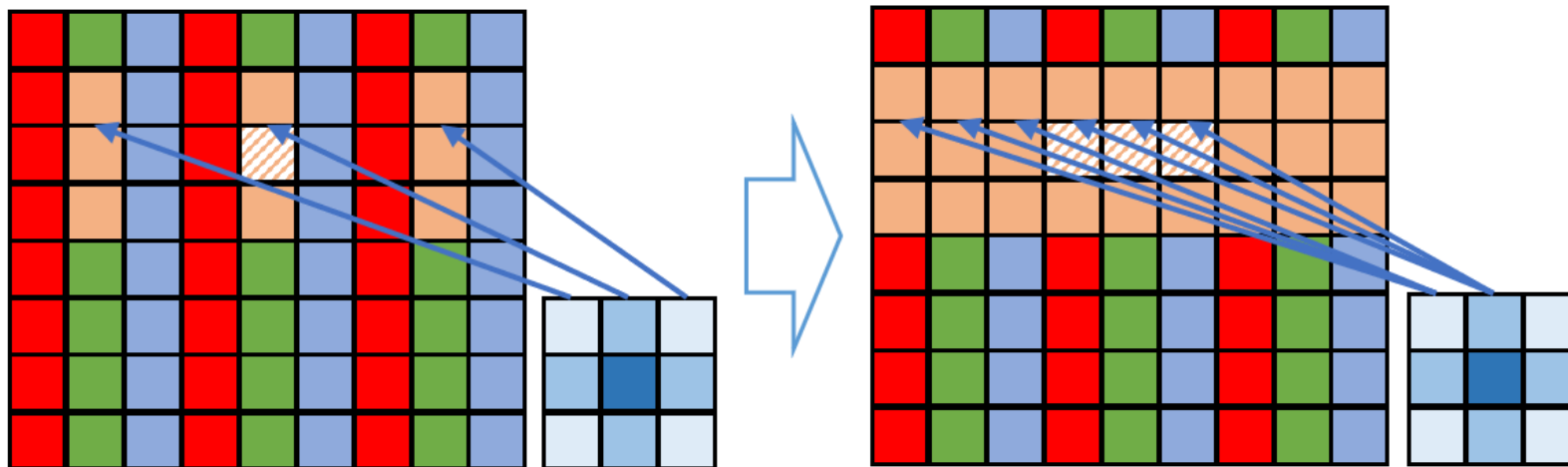
Реализация фильтра Гаусса

□ Naïve (наивная реализация)

```
...
for (int i = 0; i < h - sizeFilter; i++) {
    for (int j = 0; j < w - sizeFilter; j++) {
        for (int c = 0; c < cntChannel; c++) {
            float sum = 0.f;
            for (int i_f = 0; i_f < sizeFilter; i_f++) {
                for (int j_f = 0; j_f < sizeFilter; j_f++) {
                    int pos_i = (i + i_f) * (w * cntChannel);
                    int pos_j = (j + j_f) * (cntChannel) + c;
                    sum += srcData[pos_i + pos_j] * filter[i_f * sizeFilter + j_f];
                }
            }
            int i_d = i + middle, j_d = j + middle;
            distData[(i_d * w + j_d) * cntChannel + c] = sum;
        }
    }
}
delete[] filter;
```

Реализация фильтра Гаусса

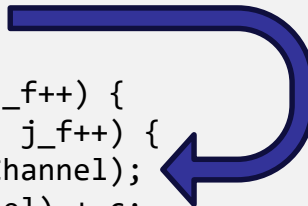
- **Unit-stride** (последовательный доступ к памяти).
 - В базовой версии неоптимальный обход по памяти



Реализация фильтра Гаусса

□ **Unit-stride** (последовательный доступ к памяти).

```
...
for (int i = 0; i < h - sizeFilter; i++) {
    for (int j = 0; j < w - sizeFilter; j++) {
        for (int c = 0; c < cntChannel; c++) {
            float sum = 0.f;
            for (int i_f = 0; i_f < sizeFilter; i_f++) {
                for (int j_f = 0; j_f < sizeFilter; j_f++) {
                    int pos_i = (i + i_f) * (w * cntChannel);
                    int pos_j = (j + j_f) * (cntChannel) + c;
                    sum += srcData[pos_i + pos_j] * filter[i_f * sizeFilter + j_f];
                }
            }
            int i_d = i + middle, j_d = j + middle;
            distData[(i_d * w + j_d) * cntChannel + c] = sum;
        }
    }
}
delete[] filter;
```



Реализация фильтра Гаусса

□ **Unit-stride** (последовательный доступ к памяти).

```
void filterChannel(cv::Mat &dist, cv::Mat& src, float sigma, int sizeFilter)
{
    float* filter = create2DFilter(sizeFilter, sigma);
    int w = src.size().width;
    int h = src.size().height;
    float* srcData = (float *)src.data;
    float* distData = (float*)dist.data;
    int middle = sizeFilter / 2;
    const int cntChannel = 3;
    for (int i = 0; i < h - sizeFilter; i++) {
        for (int j = 0; j < w - sizeFilter; j++) {
            int i_d = i + middle;
            int j_d = j + middle;
            for (int c = 0; c < cntChannel; c++) {
                distData[(i_d * w + j_d) * cntChannel + c] = 0.0f;
            }
        }
    }
}
```

Реализация фильтра Гаусса

□ **Unit-stride** (последовательный доступ к памяти).

```
...
for (int i = 0; i < h - sizeFilter; i++) {
    for (int j = 0; j < w - sizeFilter; j++) {
        int i_d = i + middle;
        int j_d = j + middle;
        for (int i_f = 0; i_f < sizeFilter; i_f++) {
            for (int j_f = 0; j_f < sizeFilter; j_f++) {
                for (int c = 0; c < cntChannel; c++) {
                    int pos_i = (i + i_f) * (w * cntChannel);
                    int pos_j = (j + j_f) * (cntChannel);
                    distData[(i_d * w + j_d) * cntChannel + c] +=
                        srcData[pos_i + pos_j + c] * filter[i_f * sizeFilter + j_f];
                }
            }
        }
    }
}
```

Реализация фильтра Гаусса

□ 1D_kernels (алгоритмическая оптимизация).

- Возможно следующее разложение ядра фильтра Гаусса*:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \right) \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{y^2}{2\sigma^2}} \right)$$

- Пусть W и H – ширина и высота изображения, C – количество каналов, F – размер ядра фильтра.

- Сложность базового алгоритма $O(W \cdot H \cdot C \cdot F^2)$

- Сложность нового алгоритма $O(W \cdot H \cdot C \cdot F)$

- Пример использования разложения:

2	2	4	2	2	1
5	2	4	4	2	0
5	1	1	0	5	1
0	4	0	0	3	3
5	0	5	1	4	0
1	5	3	2	4	2

1	2	1
2	4	2
1	2	1

*	*	*	*	*	*
*	44	43	44	34	*
*	37	24	29	39	*
*	34	22	23	38	*
*	42	39	36	39	*
*	*	*	*	*	*

2	2	4	2	2	1
5	2	4	4	2	0
5	1	1	0	5	1
0	4	0	0	3	3
5	0	5	1	4	0
1	5	3	2	4	2

1
2
1

*	*	*	*	*	*
17	7	13	10	11	2
15	8	6	4	15	5
10	9	6	1	15	7
11	9	13	4	15	5
*	*	*	*	*	*

1	2	1
---	---	---

*	*	*	*	*	*
*	44	43	44	34	*
*	37	24	29	39	*
*	34	22	23	38	*
*	42	39	36	39	*
*	*	*	*	*	*

Реализация фильтра Гаусса

□ **1D_kernels** (алгоритмическая оптимизация).

// Заполнение вектора фильтра

```
float* create1DFilter(int n, float sigma)
{
    float* filter = new float[n];
    float coeff = 1.0f / (sqrtf(2.0f * PI) * sigma);
    int middle = n / 2;

    for (int i = 0; i < n; i++)
    {
        float x = i - middle;
        filter[i] = coeff * expf(-(x * x) / (2.0f * sigma * sigma));
    }

    return filter;
}
```

Реализация фильтра Гаусса

□ 1D_kernels (алгоритмическая оптимизация).

// Инициализация матриц

```
float* filter1D = create1DFilter(sizeFilter, sigma);
cv::Mat tmp;
src.copyTo(tmp);
int w = src.size().width;
int h = src.size().height;
float* srcData = (float*)src.data;
float* tmpData = (float*)tmp.data;
float* distData = (float*)dist.data;

int middle = sizeFilter / 2;
const int cntChannel = 3;

for (int i = 0; i < h; i++) {
    for (int j = 0; j < w * cntChannel; j++) {
        int i_d = i + middle;
        tmpData[i_d * w * cntChannel + j] = 0.0f;
        distData[i_d * w * cntChannel + j] = 0.0f;
    }
}
```

Реализация фильтра Гаусса

□ 1D_kernels (алгоритмическая оптимизация).

```
...
// Первая фаза применения фильтра
for (int i = 0; i < h - sizeFilter; i++)
{
    int i_d = i + middle;
    for (int j = 0; j < w * cntChannel; j++)
    {
        for (int i_f = 0; i_f < sizeFilter; i_f++)
        {
            int pos_i = (i + i_f) * (w * cntChannel);
            int pos_j = j;
            tmpData[i_d * w * cntChannel + j] += srcData[pos_i + pos_j] * filter1D[i_f];
        }
    }
}
...
```

Реализация фильтра Гаусса

□ 1D_kernels (алгоритмическая оптимизация).

```
...  
// Вторая фаза применения фильтра  
for (int i = 0; i < h - sizeFilter; i++)  
{  
    for (int j = 0; j < w - sizeFilter; j++)  
    {  
        int i_d = i + middle;  
        int j_d = j + middle;  
        for (int j_f = 0; j_f < sizeFilter; j_f++)  
        {  
            for (int c = 0; c < cntChannel; c++)  
            {  
                int pos_i = (i) * (w * cntChannel);  
                int pos_j = (j + j_f) * (cntChannel);  
                distData[(i_d * w + j_d) * cntChannel + c] +=  
                    tmpData[pos_i + pos_j + c] * filter1D[j_f];  
            }  
        }  
    }  
}
```

Реализация фильтра Гаусса

❑ Memory (улучшение доступа к памяти).

- Применение горизонтального ядра фильтра Гаусса реализовано неэффективно.
- Изменим порядок циклов, при котором каждый элемент ядра взаимодействует со всей строкой исходной матрицы изображения.

// Первая фаза применения фильтра

```
for (int i = 0; i < h - sizeFilter; i++)
{
    int i_d = i + middle;
    for (int i_f = 0; i_f < sizeFilter; i_f++) {
        for (int j = 0; j < w * cntChannel; j++)
        {
            int pos_i = (i + i_f) * (w * cntChannel);
            tmpData[i_d * w * cntChannel + j] += srcData[pos_i + j] * filter1D[i_f];
        }
    }
}
```

Реализация фильтра Гаусса

□ **Parallel** (параллельная реализация).

- Обе фазы применения фильтра Гаусса могут быть распараллелены по внешнему циклу
 - Итерации внешнего цикла независимые, как следствие, достаточно применить `#pragma omp parallel for`

Реализация фильтра Гаусса

□ Результаты вычислительных экспериментов

– Решаемая задача:

- Размер тестового изображения: 2544x2027
- Размер окна фильтра: 19x19
- Изображение цветное: 3 канала

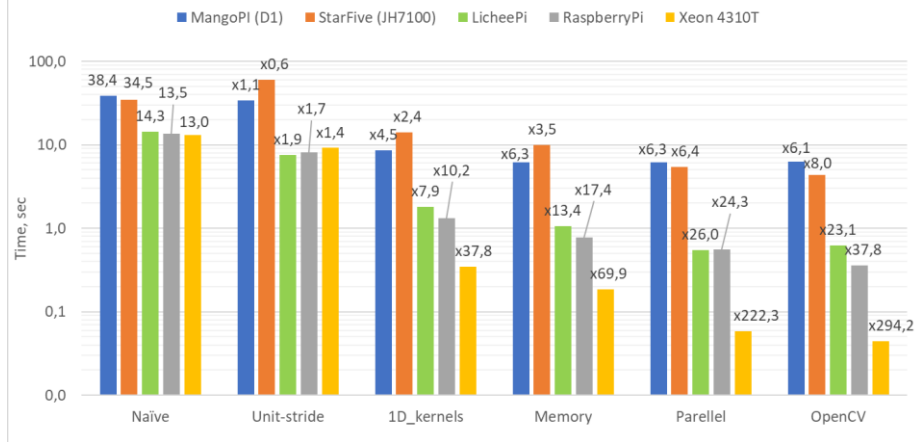
Реализация фильтра Гаусса

□ Результаты вычислительных экспериментов

- На графике время вычислений, а также ускорение относительно базовой реализации

Время вычислений первой модификации алгоритма **«Unit-stride»** явно лучше из-за последовательного доступа к памяти, который намного быстрее благодаря эффективной предварительной выборке данных.

Для **StarFive** упреждающая выборка данных не ускоряет расчеты. Низкая пропускная способность памяти не позволяет вовремя подготовить данные, что и приводит к дополнительным накладным расходам.



Реализация фильтра Гаусса

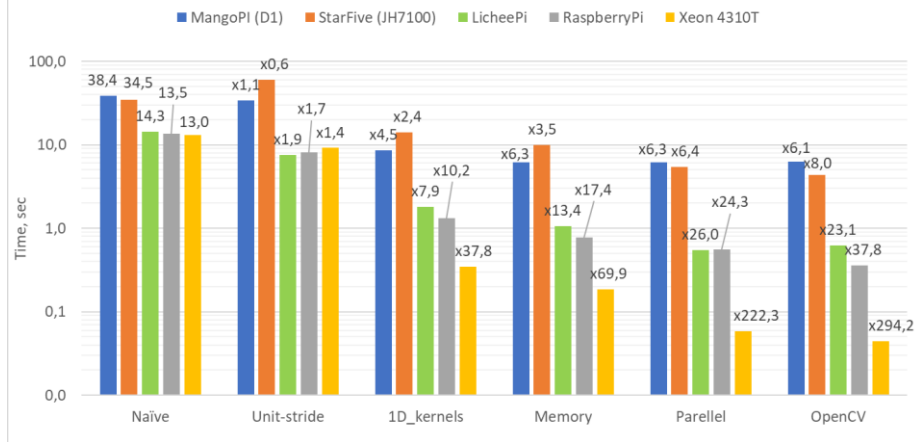
□ Результаты вычислительных экспериментов

- На графике время вычислений, а также ускорение относительно базовой реализации

Модификация «1D_kernels». Видно значительное ускорение.
При размере фильтра F=19 можно было ожидать большего ускорения.

Большого ускорения нет из-за неэффективного обращения к памяти.

Подтверждение этому результаты Модификации «Memory».



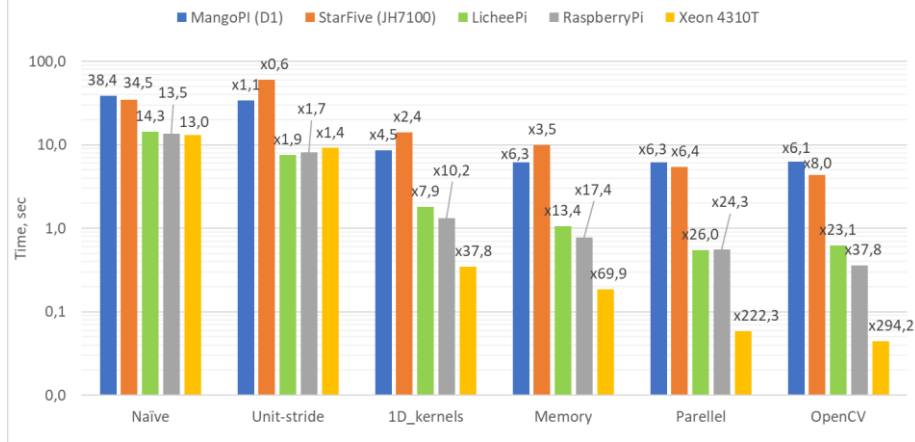
Реализация фильтра Гаусса

□ Результаты вычислительных экспериментов

- На графике время вычислений, а также ускорение относительно базовой реализации

Поскольку задача фильтрации изображений относится к классу **«memory bound»**, ускорение параллельной версии ограничено количеством доступных каналов памяти.

Как и в случае с алгоритмом транспонирования матриц, оптимизации, разработанные для архитектуры x86, хорошо себя демонстрируют и на устройствах RISC-V.



Реализация фильтра Гаусса

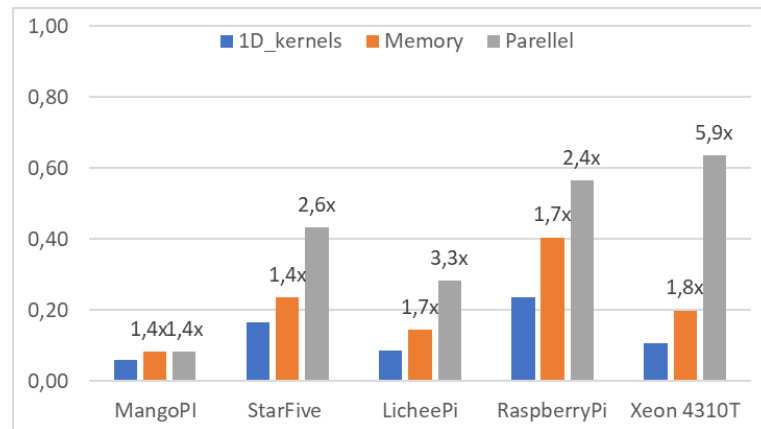
□ Результаты вычислительных экспериментов

- На графике утилизация подсистемы памяти
- При расчете этой метрики используется в качестве основы реализация «1D kernels».

Mango Pi не позволяет обеспечить высокую производительность алгоритма фильтрации изображений из-за отсутствия кэша L2 и медленного кэша L1.

StarFive, LicheePi и Raspberry Pi по ускорению доступа к памяти показывают схожие результаты. Низкая эффективность связана с оптимизацией обходов памяти, а не с переиспользованием данных из кэш-памяти

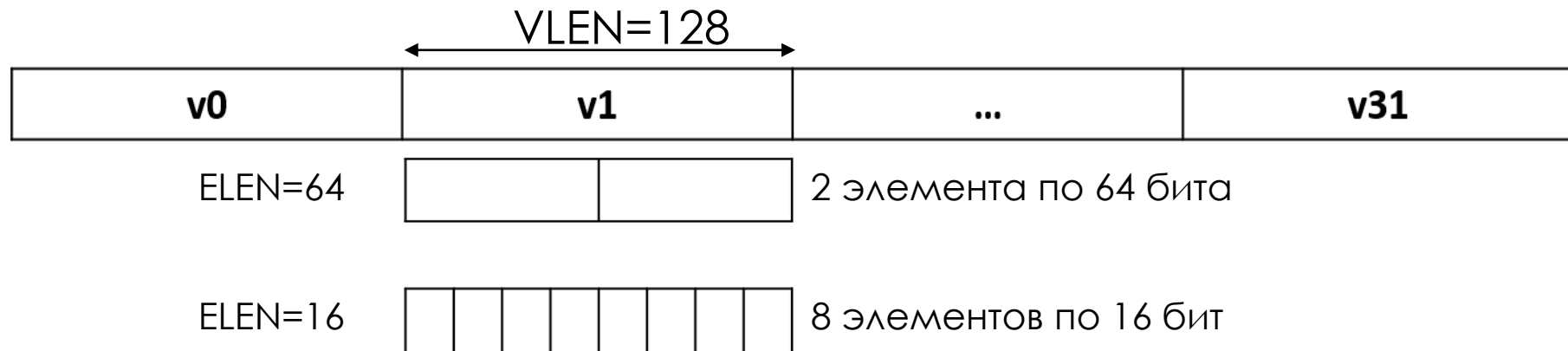
На Intel Xeon 4310T алгоритм обеспечил хорошие значения метрики за счет наличия большого количества каналов памяти, отсутствующих в других рассматриваемых устройствах.



ВЕКТОРИЗАЦИЯ НА RISC-V

RISC-V Vector Extension (RVV)

- RVV определяет 32 векторных регистра (v0-v31) размера VLEN бит
 - VLEN – постоянный параметр, выбранный разработчиком ядра
- Векторы в RVV состоят из элементов
 - Размер элементов в битах – от 8 бит до ELEN бит
 - ELEN – постоянный параметр, выбранный разработчиком ядра



RISC-V Vector Extension (RVV)

- ❑ При работе с векторами в RVV используются два регистра
 - **vtype**: векторный тип
 - **vl**: длина вектора (это не VLEN)
- ❑ vtype описывает тип вектора, который включает в себя:
 - **sew** – стандартная ширина элемента в битах ($8 < sew < ELEN$)
 - **lmul** – множитель длины
- ❑ Позволяет группировать регистры ($lmul = 2^k$, где $-3 \leq k \leq 3$)
 - **vl** количество элементов вектора ($0 \leq vl \leq vlmax(sew, lmul)$)
$$vlmax(sew, lmul) = (VLEN/sew) * lmul$$

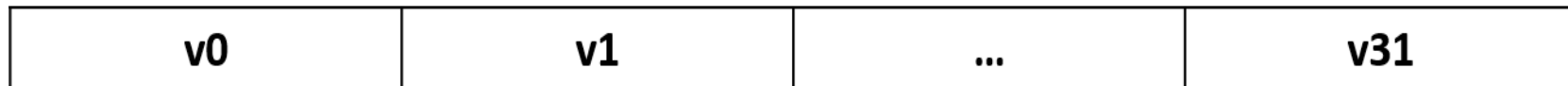
RISC-V Vector Extension (RVV)

□ Вектора объединяются в группы из нескольких векторных регистров. Группа векторов идентифицируется векторным регистром с наименьшим номером в группе

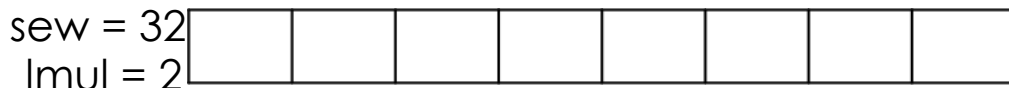
– 16 векторных групп $lmul = 2$: $v0, v2, v4, v6, v8, v10, v12, \dots, v28, v30$

– 8 векторных групп $lmul = 4$: $v0, v4, v8, v12, v16, v20, v24, v28$

– 4 векторные группы $lmul = 8$: $v0, v4, v8, v16$



$$0 \leq vl \leq 16$$



$$0 \leq vl \leq 8$$

RISC-V Vector Extension (RVV)

- ❑ Проект векторных инструкций RVV 0.7.1, представленный T-Head в 2019 г. Поддерживает некоторые ядра T-Head
 - Впоследствии улучшилось до 0.8
 - $VLEN=128$
- ❑ Стандарт RVV 1.0 принят в 2021 г. Сейчас начали появляться платы с ядрами, поддерживающими RVV 1.0
 - Имя расширения «V» резервируется для версии 1.0+
 - $8 \leq VLEN \leq 2^{16}$
- ❑ RVV 0.7.1 и RVV 1.0 очень похожи, но не совместимы на уровне двоичного кода
 - Есть инструменты для конвертации RVV 1.0 в RVV 0.7.1 на уровне ассемблера

OpenCV

- ❑ OpenCV – это библиотека алгоритмов компьютерного зрения с открытым исходным кодом
- ❑ OpenCV использует модульную архитектуру (базовый модуль, обработка изображений, обработка видео и т.д.)
- ❑ Концепция универсальных векторных интринсик в OpenCV:
 - Объявлен набор функций (intrinsics), необходимых в библиотеке и 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

Изменения кода OpenCV

❑ Метрики оценки производительности:

- Время работы
- Ускорение от векторизации

До изменений (~3000 строк)

Длина вектора 128 бит

Типичная реализация универсальных
интринсик:

`vfmadd_vv_f32m1`

`<Команда>_<тип данных>m<кол-во рег.>`

После изменений (~5000 строк)

Длина вектора 512 бит

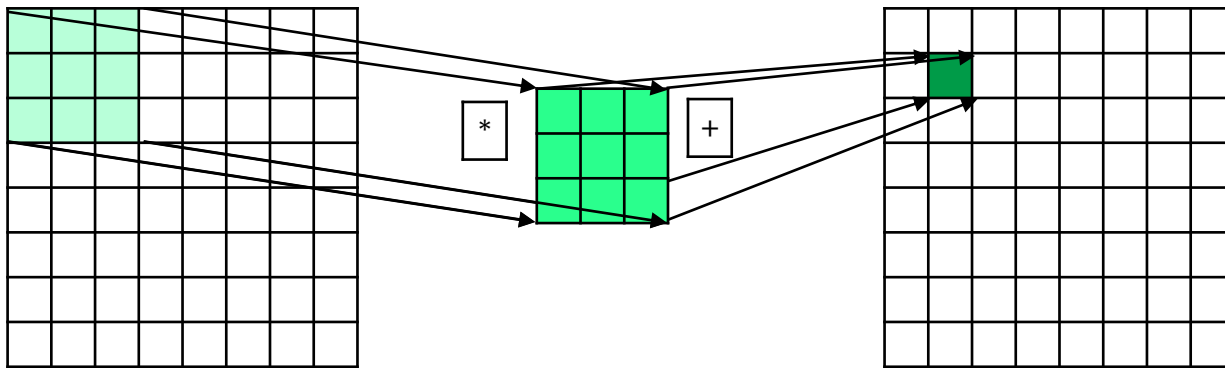
Типичная реализация универсальных
интринсик:

`vfmadd_vv_f32m4`

Также добавлены преобразования типов
`m8->m1->m4`, так как в стандартном
функционале нет прямого преобразования
`m8->m4`

Фильтрация 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 тривиальный 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

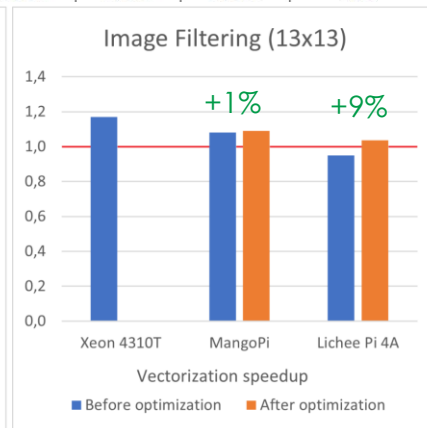
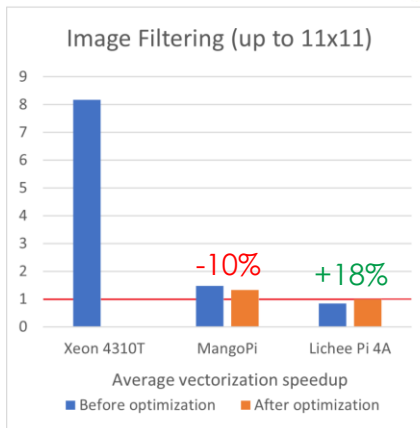
Lichee Pi

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

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

FullHD
1920x1080



Эрозия в OpenCV

- ❑ Эрозия выбирает минимум из некоторой области:

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

- ❑ Эрозия работает с одноканальными изображениями $\Rightarrow$ последовательный доступ к памяти



- ❑ Параллельная версия отсутствует

Эрозия в OpenCV. Результаты

Mango Pi

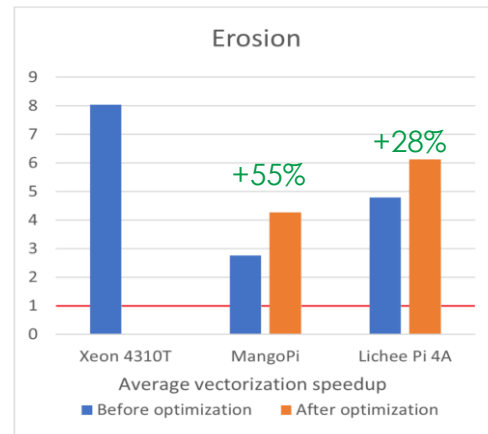
Resolution	Filter size	SeqScalar	SeqVector	Optim	Vectorization speedup	Optimization speedup
1920x1080	1	0,163	0,060	0,043	2,71	1,40
1920x1080	2	0,246	0,073	0,048	3,37	1,51
1920x1080	3	0,320	0,084	0,054	3,79	1,57
3840x2160	1	0,586	0,242	0,157	2,42	1,54
3840x2160	2	0,900	0,297	0,180	3,03	1,65
3840x2160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



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.

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

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



ЗАКЛЮЧЕНИЕ

Заключение

- ❑ В лекции исследованы возможности и перспективы вычислительной архитектуры RISC-V на примере реализации нескольких алгоритмов.
- ❑ В рамках лекции показана вычисленная с помощью теста STREAM пропускная способность памяти. Результаты показали, что существующие прототипы RISC-V по пропускной способности памяти все еще значительно отстают от устройств ARM и x86.
- ❑ Показано, что в алгоритме транспонирования матриц устройства RISC-V демонстрируют отличное использование доступных ресурсов, тогда как при фильтрации изображений память используется менее эффективно.
- ❑ В лекции показано, что типичные методики оптимизации работы с памятью, отработанные за последние десятилетия на ARM и x86, ведут себя ожидаемо на RISC-V, позволяя значительно ускорить вычисления.
- ❑ Показаны варианты RISC-V специфичных оптимизаций при векторизации кода

Литература

1. Volokitin V.D., Kustikova V.D., Kozinov E.A., Linev A.V., Meyerov I.B. Case Study for Running Memory-Bound Kernels on RISC-V CPUs // Parallel Computing Technologies. 17th International Conference, PaCT 2023, Astana, Kazakhstan, August 21–25, 2023, Proceedings.. Springer Lecture Notes in Computer Science (LNCS, volume 14098). 2023. P. 51–65.
2. Козинев Е.А., Сысоев А.В., Волокитин В.Д., Кустикова В.Д., Линев А.В., Мееров И.Б. Оптимизация работы с памятью при решении задач на процессорах архитектуры RISC-V. Н.Новгород: Издательство ННГУ. 2024. 31 с.
3. Volokitin V.D., Vasilyev E.P., Kozinov E.A., Kustikova V.D., Liniov A.V., Rodimkov Yu.A., Sysoyev A.V., Meyerov I.B. Improved Vectorization of OpenCV Algorithms for RISC-V CPU // Lobachevskii Journal of Mathematics. V. 45. № 1. 2024. P. 130-142.

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

