ПРОГРАММИРОВАНИЕ ДЛЯ RISC-V

ЗИМНЯЯ ШКОЛА

ОПТИМИЗАЦИЯ АЛГОРИТМА FFT (БЫСТРОГО ПРЕОБРАЗОВАНИЯ ФУРЬЕ)

Иван Рябинин, Андрей Соколов

YADRO

Знакомьтесь – закон Мура*

«Производительность процессоров должна была удваиваться каждые 18 месяцев из-за сочетания роста количества транзисторов и увеличения тактовых частот процессоров»

* Вернее, его производные – это, например, прогноз Дэвида Хауса из Intel

Алгоритмическая оптимизация

Как достичь более высокой производительности?

Воспользоваться более эффективным алгоритмом!

Как его получить?

Три главных приема алгоритмической оптимизации

Посчитай заранее

- Проверь, можно ли заранее вычислить коэффициенты
- Если коэффициентов много, попробуй свести их в таблицу и вместо вычисления подбирай индекс
- Убери проверки, если знаешь их результат заранее

Выкинь лишнее

- Переиспользуй результаты повторяющихся вычислений
- Замени короткий цикл на линейный код
- Подбери максимально простые операции (например, замени деление на константу умножением на обратную величину)

Подсмотри у друга

- Проверь, не решена ли эта задача до тебя в какой-либо книге или статье
- Попробуй адаптировать алгоритм решения похожей задачи
- Примени к нему приемы I и II

Преобразование Фурье

Спектральный анализ

Обработка цифровых сигналов

$$\partial$$

Решение дифференциальных уравнений

Дискретное преобразование Фурье (DFT)

DFT – один из основных алгоритмов обработки цифровых сигналов

- В качестве входа требует дискретную функцию (в нашем случае комплексную)
- Для длин 2^n называется **быстрым** преобразованием Фурье (**FFT**)

Задача

Оптимизировать FFT для некоторых длин*

* В рамках практикума ограничимся длинами в 2, 4, 8 и 16

Прямое определение DFT

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j$$

$$w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i\sin\left(\frac{2\pi jk}{n}\right)$$

Разрабатываем DFT по определению

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j \qquad w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i\sin\left(\frac{2\pi jk}{n}\right)$$

Вычисление DFT по определению сводится к нескольким простым ходам:

- Заводим два вложенных цикла
- Вычисляем аргумент экспоненты
- Считаем комплексное произведение входных данных и экспоненты
- Возвращаем результат

Разрабатываем DFT по определению

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j \qquad w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i\sin\left(\frac{2\pi jk}{n}\right)$$

```
extern void refDftFwd(const cfloat32_t *pSrc, cfloat32_t *pDst, uint32_t length)
{
    for (uint32_t i = 0; i < length; i++)
    {
        float pDstRe = 0.0f;
        float pDstIm = 0.0f;
        for (uint32_t j = 0; j < length; j++)
        {
            float arg = (_2PI * j * i) / length;
            pDstRe += ( pSrc[j].re * cos(arg) + pSrc[j].im * sin(arg));
            pDstIm += (-pSrc[j].re * sin(arg) + pSrc[j].im * cos(arg));
        }
        pDst[i].re = pDstRe;
        pDst[i].im = pDstIm;
    }
}
```

Разрабатываем DFT по определению

В общем виде тогда DFT имеет сложность

$$O(N^2)$$

Как ее понизить?

Разрабатываем FFT2

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j \qquad w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i\sin\left(\frac{2\pi jk}{n}\right)$$

```
extern void refDftFwd(const cfloat32_t *pSrc, cfloat32_t *pDst, uint32_t length = 2)
{
    for (uint32_t i = 0; i < length; i++)
    {
        float pDstRe = 0.0f;
        float pDstIm = 0.0f;
        for (uint32_t j = 0; j < length; j++)
        {
            float arg = (_2PI * j * i) / length;
            pDstRe += ( pSrc[j].re * cos(arg) + pSrc[j].im * sin(arg));
            pDstIm += (-pSrc[j].re * sin(arg) + pSrc[j].im * cos(arg));
        }
        pDst[i].re = pDstRe;
        pDst[i].im = pDstIm;
    }
}
```

Разрабатываем FFT2

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j \qquad w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i\sin\left(\frac{2\pi jk}{n}\right)$$

```
extern void dft2Fwd(const cfloat32_t *pSrc, cfloat32_t *pDst)
{
    for (uint32_t i = 0; i < 2; i++)          // итераций мало - разворачиваем
    {
        float pDstRe = 0.0f;
        float pDstIm = 0.0f;
        for (uint32_t j = 0; j < 2; j++)      // итераций мало - разворачиваем
        {
            float arg = (_2PI * j * i) / 2;   // может принимать только значения ±pi
            pDstRe += ( pSrc[j].re * cos(arg) + pSrc[j].im * sin(arg));
            pDstIm += (-pSrc[j].re * sin(arg) + pSrc[j].im * cos(arg));
        }
        pDst[i].re = pDstRe;
        pDst[i].im = pDstIm;
    }
}
```

Разрабатываем FFT2

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j \qquad w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i\sin\left(\frac{2\pi jk}{n}\right)$$

```
extern void dft2Fwd(const cfloat32_t *pSrc, cfloat32_t *pDst)
{
    pDst[1].re = pSrc[0].re - pSrc[1].re;
    pDst[1].im = pSrc[0].im - pSrc[1].im;
    pDst[0].re = pSrc[0].re + pSrc[1].re;
    pDst[0].im = pSrc[0].im + pSrc[1].im;
}
```

Разрабатываем FFT4

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j \qquad w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i\sin\left(\frac{2\pi jk}{n}\right)$$

```
extern void refDftFwd(const cfloat32_t *pSrc, cfloat32_t *pDst, uint32_t length = 4)
{
    for (uint32_t i = 0; i < length; i++)
    {
        float pDstRe = 0.0f;
        float pDstIm = 0.0f;
        for (uint32_t j = 0; j < length; j++)
        {
            float arg = (_2PI * j * i) / length;
            pDstRe += ( pSrc[j].re * cos(arg) + pSrc[j].im * sin(arg));
            pDstIm += (-pSrc[j].re * sin(arg) + pSrc[j].im * cos(arg));
        }
        pDst[i].re = pDstRe;
        pDst[i].im = pDstIm;
    }
}
```

Разрабатываем FFT4

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j \qquad w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i \sin\left(\frac{2\pi jk}{n}\right)$$

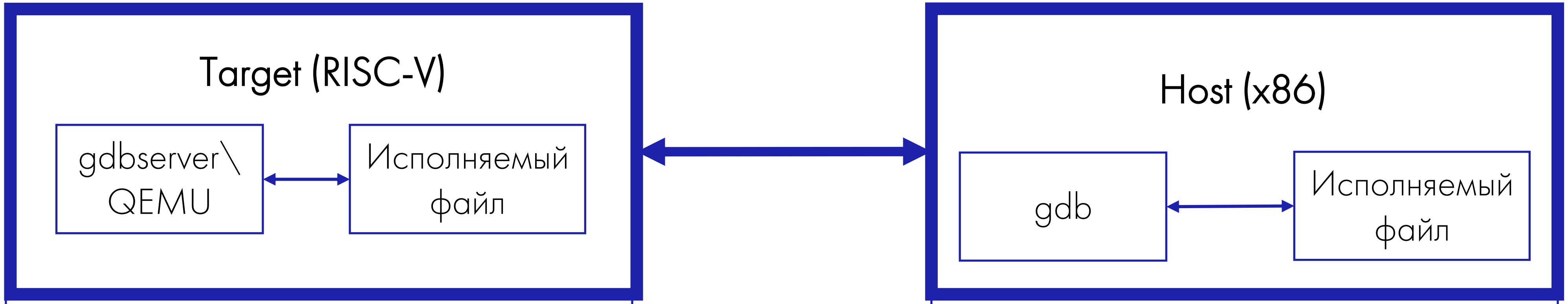
```
extern void dft4Fwd(const cfloat32_t *pSrc, cfloat32_t *pDst)
{
    for (uint32_t i = 0; i < 4; i++)          // итераций мало - разворачиваем
    {                                          // в два приема
        float pDstRe = 0.0f;
        float pDstIm = 0.0f;
        for (uint32_t j = 0; j < 4; j++)      // итераций мало - разворачиваем
        {                                      // в два приема
            float arg = (_2PI * j * i) / 2;   // может принимать значения ±pi, ±pi/2
            pDstRe += ( pSrc[j].re * cos(arg) + pSrc[j].im * sin(arg));
            pDstIm += (-pSrc[j].re * sin(arg) + pSrc[j].im * cos(arg));
        }
        pDst[i].re = pDstRe;
        pDst[i].im = pDstIm;
    }
}
```

Разрабатываем FFT4

```
extern void dft4Fwd(const cfloat32_t *pSrc, cfloat32_t *pDst)
{
    cfloat32_t tmpDst[4];
    tmpDst[0].re = pSrc[0].re   + pSrc[2].re;
    tmpDst[0].im = pSrc[0].im   + pSrc[2].im;
    tmpDst[1].re = pSrc[0].re   - pSrc[2].re;
    tmpDst[1].im = pSrc[0].im   - pSrc[2].im;
    tmpDst[2].re = pSrc[1].re   + pSrc[3].re;
    tmpDst[2].im = pSrc[1].im   + pSrc[3].im;
    tmpDst[3].re = pSrc[1].re   - pSrc[3].re;
    tmpDst[3].im = pSrc[1].im   - pSrc[3].im;
    pDst[0].re   = tmpDst[0].re ? tmpDst[2].re;
    pDst[0].im   = tmpDst[0].im ? tmpDst[2].im;
    pDst[1].re   = tmpDst[1].re ? tmpDst[3].im;
    pDst[1].im   = tmpDst[1].im ? tmpDst[3].re;
    pDst[2].re   = tmpDst[0].re ? tmpDst[2].re;
    pDst[2].im   = tmpDst[0].im ? tmpDst[2].im;
    pDst[3].re   = tmpDst[1].re ? tmpDst[3].im;
    pDst[3].im   = tmpDst[1].im ? tmpDst[3].re;
}
```

Отладка!

Удаленная отладка GDB



1

На x86 (симулятор QEMU):

```
qemu-riscv64 -g <port> <exec>
```

На RISC-V плате:

```
gdbserver :<port> <exec>
```

2

```
riscv64-unknown-linux-gnu-gdb <exec>
```

3

```
target remote <ip>:<port>
```

[docs\How2Debug.md](docs\How2Debug.md)

Основные команды GDB

Выполнение

- `r\run` – старт выполнения программы
- `c\continue` – продолжить выполнение до конца или breakpoint
- `n\next` – выполнение следующей строчки без захода в функции
- `s\step` – выполнение следующей строчки с заходом в функции

Просмотр переменных\памяти

- `p\print` – просмотр переменной или регистра
- `x` ("examine")– просмотр памяти

Breakpoints

- `b\break` – установка breakpoint

Other

- `bt\backtrace` – просмотр стека вызовов

GDB cheatsheet

Разрабатываем FFT4

```
extern void dft4Fwd(const cfloat32_t *pSrc, cfloat32_t *pDst)
{
    cfloat32_t tmpDst[4];
    tmpDst[0].re = pSrc[0].re   + pSrc[2].re;
    tmpDst[0].im = pSrc[0].im   + pSrc[2].im;
    tmpDst[1].re = pSrc[0].re   - pSrc[2].re;
    tmpDst[1].im = pSrc[0].im   - pSrc[2].im;
    tmpDst[2].re = pSrc[1].re   + pSrc[3].re;
    tmpDst[2].im = pSrc[1].im   + pSrc[3].im;
    tmpDst[3].re = pSrc[1].re   - pSrc[3].re;
    tmpDst[3].im = pSrc[1].im   - pSrc[3].im;
    pDst[0].re   = tmpDst[0].re + tmpDst[2].re;
    pDst[0].im   = tmpDst[0].im + tmpDst[2].im;
    pDst[1].re   = tmpDst[1].re + tmpDst[3].im;
    pDst[1].im   = tmpDst[1].im - tmpDst[3].re;
    pDst[2].re   = tmpDst[0].re - tmpDst[2].re;
    pDst[2].im   = tmpDst[0].im - tmpDst[2].im;
    pDst[3].re   = tmpDst[1].re - tmpDst[3].im;
    pDst[3].im   = tmpDst[1].im + tmpDst[3].re;
}
```

Разрабатываем FFT произвольной длины

$$y_k = \sum_{j=0}^{n-1} w^{jk} x_j \qquad w^{jk} = e^{-i\frac{2\pi jk}{n}} = \cos\left(\frac{2\pi jk}{n}\right) + i\sin\left(\frac{2\pi jk}{n}\right)$$

Сложность кода растет быстро – нам необходимо сменить подход!

Попробуем освоить алгоритм Кули-Тьюки.

Он "проще" – его сложность

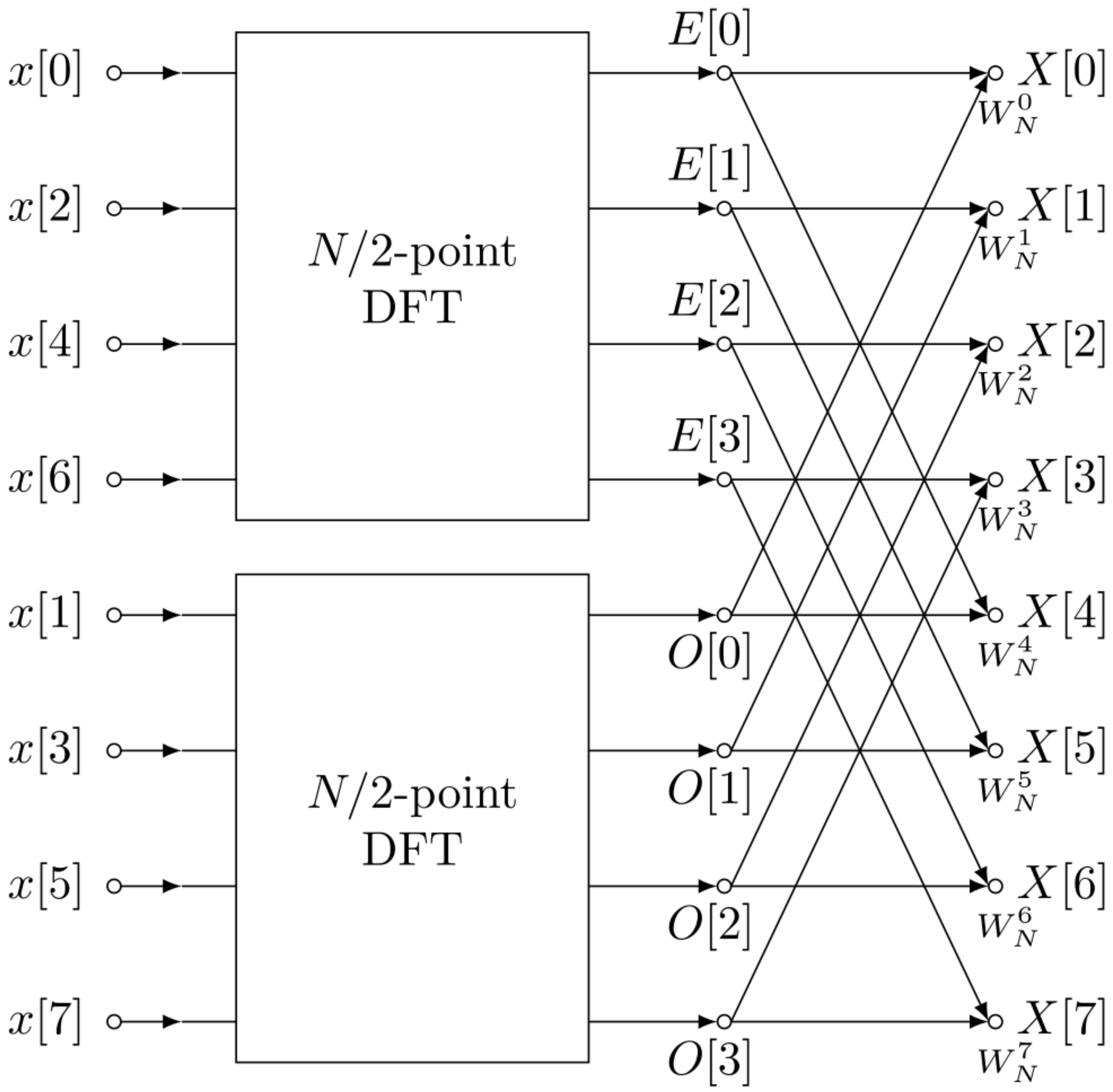
$$O(N\log(N))$$

Разрабатываем DFT произвольной длины

Алгоритм Кули-Тьюки позволяет рассматривать разработку алгоритмов DFT как своего рода конструктор – например, мы можем собрать **FFT8** из **FFT4** и **FFT2**.

Общая схема для DFT длин n_1 и n_2:

1. Представляем входные данные как двумерный массив $n_1 \times n_2$
2. Вычисляем n_1 DFT длины n_2
3. Домножаем промежуточный результат на тригонометрические коэффициенты
4. Вычисляем n_2 DFT длины n_1
5. Упорядочиваем результат



Подробная постановка задачи

1. Написать быстрое преобразование Фурье общего вида по алгоритму Кули-Тьюки
    - Опционально: по Кули-Тьюки собрать **FFT4** из **FFT2**
2. Выбрать длину **FFT** (8 или 16) и алгоритмически оптимизировать код для Кули-Тьюки
    - Опционально: разработать код для обеих длин
3. Поэкспериментировать с RVV оптимизациями: разработать векторный код для **FFT2** и **FFT4**
    - Опционально: попробовать распространить оптимизацию на **DFT8** или **DFT16**
    - Опционально: попробовать сравнить с OpenMP

Формат зачета

- Ваша цель – получить максимальную производительность для **FFT** на **8** и/или **16** точек

- По результатам мы составим турнирные таблицы в **общем зачете** (лучший результат по длине с любыми приемами оптимизации) и в **алгоритмическом зачете** (без RVV и параллелизации)