

Оптимизация программ

Основы информатики

Компьютерные основы программирования

u.to/DbCmFA

На основе CMU 15-213/18-243:

Introduction to Computer Systems

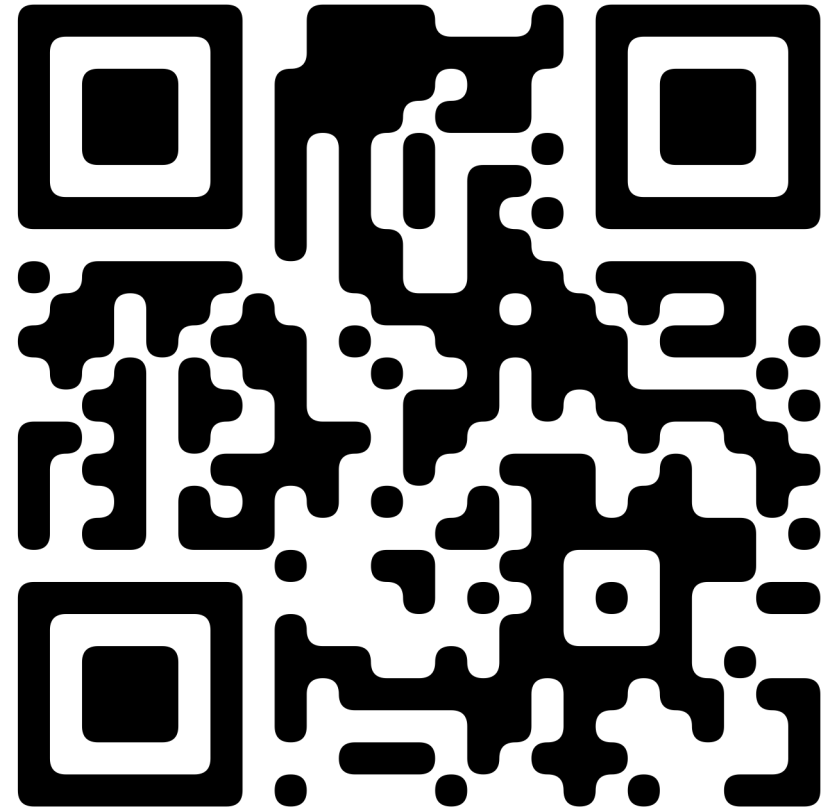
u.to/XoKmFA

Лекция 8, 31 марта, 2023

Лектор:

Дмитрий Северов, кафедра информатики 608 КПП

cs.mipt.ru/wp/?page_id=346



Оптимизация программ

- **Обзор**
- **Оптимизации полезные в общем случае**
 - Вынос кода, предварительные вычисления
 - Уменьшение «стоимости» операций
 - Использование общности подвыражений
 - Исключение обращений к процедурам
- **Помехи оптимизации**
 - Обращения к процедурам
 - Псевдонимы памяти
- **Использование параллелизма команд**
- **Работа с условными конструкциями**

О реальной производительности

- *Производительность зависит не только от асимптотической сложности алгоритма*

- **Постоянные множители тоже имеют значение!**

- Легко увидеть 10-кратное изменение быстродействия кода в зависимости от качества его написания
- Оптимизация должна производиться на разных уровнях:
 - алгоритмы, представление данных, процедуры, циклы

- **Для оптимизации требуется системное понимание**

- Как код компилируется и исполняется
- Как работают современные процессоры и памяти
- Как измерить быстродействие программ и найти узкие места
- Как улучшить быстродействие не ухудшив модульности и общности

Оптимизирующие компиляторы

- **Обеспечивают эффективное отображение кода на машину**
 - распределение регистров
 - выбор и упорядочение кода (планирование)
 - исключение неисполняемого кода
 - исключение мелких недостатков
- **Не улучшают (обычно) асимптотическую сложность**
 - выбор наилучшего алгоритма в целом – задача программиста
 - улучшение оценки O -большое (часто) более важно
 - но постоянные множители тоже имеют значение
- **Затрудняются преодолеть помехи оптимизации**
 - возможность возникновения синонимов памяти
 - возможность побочного эффекта от вызова процедуры

Ограничения компиляторов

■ Фундаментальное ограничение

- Сохранять поведение программы **в любых условиях**
- Часто не позволяет выполнять оптимизации, которые изменяют поведение только в условиях, неактуальных для вашей задачи

■ Очевидное программисту поведение может быть скрыто языком и стилем написания программ

- например подразумеваемый диапазон данных может быть более ограничен, чем предполагается определением типа

■ Почти весь анализ выполняется внутри процедур

- Анализ программы в целом в большинстве случаев слишком дорог
- Новые версии GCC выполняют межпроцедурный анализ внутри файла
 - Но не между различными файлами

■ Почти весь анализ основан на *статической* информации

- Компилятору проблематично предвидеть ввод во время выполнения

■ **В сомнительной ситуации, компилятор должен быть консервативен**

Оптимизации, полезные в общем случае

- Оптимизации, которые должны быть сделаны независимо от процессора и/или компилятора
- Вынос кода
 - Уменьшать количество повторений одинаковых вычислений
 - если они всегда выдают одинаковый результат
 - особенно важно выносить код из цикла

```
void set_row(double *a, double *b,  
            long i, long n)  
{  
    long j;  
    for (j = 0; j < n; j++)  
        a[n*i+j] = b[j];  
}
```



```
long j;  
int ni = n*i;  
for (j = 0; j < n; j++)  
    a[ni+j] = b[j];
```

Вынос кода компилятором (-O1)

```
void set_row(double *a, double *b,  
            long i, long n)  
{  
    long j;  
    for (j = 0; j < n; j++)  
        a[n*i+j] = b[j];  
}
```

```
long j;  
long ni = n*i;  
double *rowp = a+ni;  
for (j = 0; j < n; j++)  
    *rowp++ = b[j];
```

```
set_row:  
    testq    %rcx, %rcx           # Test n  
    jle      .L1                  # If 0, goto done  
    imulq    %rcx, %rdx           # ni = n*i  
    leaq     (%rdi,%rdx,8), %rdx   # rowp = A + ni*8  
    movl     $0, %eax             # j = 0  
    .L3:                                     # loop:  
    movsd    (%rsi,%rax,8), %xmm0  # t = b[j]  
    movsd    %xmm0, (%rdx,%rax,8)  # M[A+ni*8 + j*8] = t  
    addq     $1, %rax             # j++  
    cmpq     %rcx, %rax           # j:n  
    jne      .L3                  # if !=, goto loop  
    .L1:                                     # done:  
    rep ; ret
```

Снижение «стоимости»

- Замена дорогостоящих операций более простыми
- Сдвиг и сложение вместо умножения и деления

$16 * x \quad \rightarrow \quad x \ll 4$

- Выгода зависит от машины
- Зависит от «стоимости» команд умножения и деления
 - На Intel Nehalem, целочисленное умножение требует 3 цикла ЦП

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        a[n*i + j] = b[j];
```



```
int ni = 0;  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
    ni += n;  
}
```


Использование общности подвыражений

- Повторное использование частей выражений
- Компиляторы не изощряются в арифметических преобразованиях

```
/* Сумма соседей of i,j */  
up =    val[(i-1)*n + j  ];  
down =  val[(i+1)*n + j  ];  
left =  val[i*n          + j-1];  
right = val[i*n          + j+1];  
sum = up + down + left + right;
```

3 умножения: $i*n$, $(i-1)*n$, $(i+1)*n$

```
leaq    1(%rsi), %rax    # i+1  
leaq    -1(%rsi), %r8    # i-1  
imulq   %rcx, %rsi      # i*n  
imulq   %rcx, %rax      # (i+1)*n  
imulq   %rcx, %r8      # (i-1)*n  
addq    %rdx, %rsi      # i*n+j  
addq    %rdx, %rax      # (i+1)*n+j  
addq    %rdx, %r8      # (i-1)*n+j
```

```
long inj = i*n + j;  
up =    val[inj - n];  
down =  val[inj + n];  
left =  val[inj - 1];  
right = val[inj + 1];  
sum = up + down + left + right;
```

1 умножение: $i*n$

```
imulq   %rcx, %rsi      # i*n  
addq    %rdx, %rsi      # i*n+j  
movq    %rsi, %rax      # i*n+j  
subq    %rcx, %rax      # i*n+j-n  
leaq    (%rsi,%rcx), %rcx # i*n+j+n
```

Блокировка оптимизаций №1: процедуры

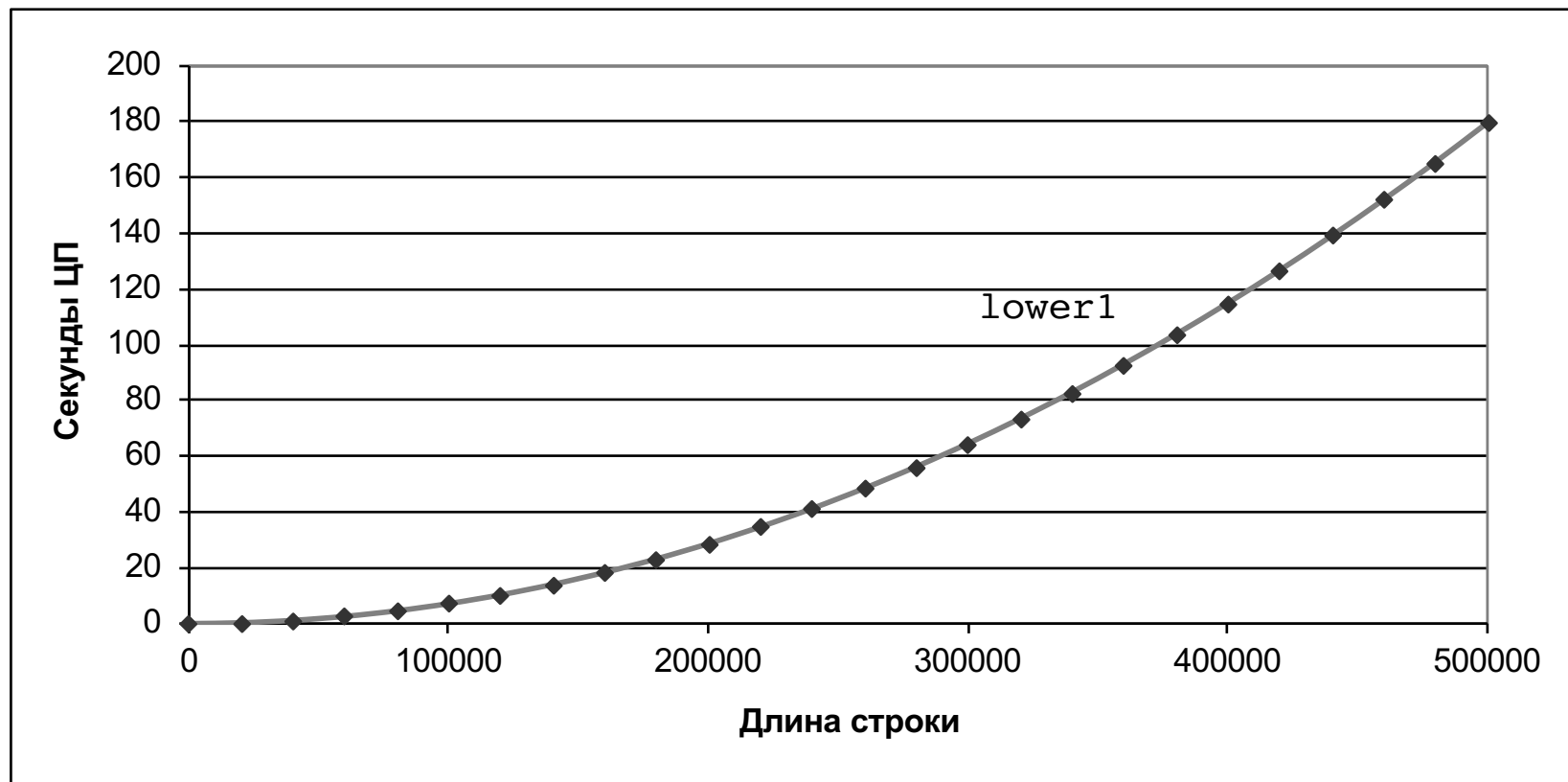
- Процедура преобразования строки в нижний регистр

```
void lower(char *s)
{
    int i;
    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Взята из лабораторной работы

Быстродействие преобразования

- Время учетверяется при удвоении длины строки
- Квадратичная зависимость



Преобразуем цикл в goto

```
void lower(char *s)
{
    int i = 0;
    if (i >= strlen(s))
        goto done;
loop:
    if (s[i] >= 'A' && s[i] <= 'Z')
        s[i] -= ('A' - 'a');
    i++;
    if (i < strlen(s))
        goto loop;
done:
}
```

- `strlen` выполняется на каждой итерации

Обращение к strlen

```
/* Свой вариант strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

■ Быстродействие strlen

- Единственный способ определить длину строки – просканировать её по всей длине в поисках символа конца строки

■ Общая сложность для строки длиной N

- N обращений к strlen
- Требуется N, N-1, N-2, ..., 1 операций
- Полная сложность - $O(N^2)$

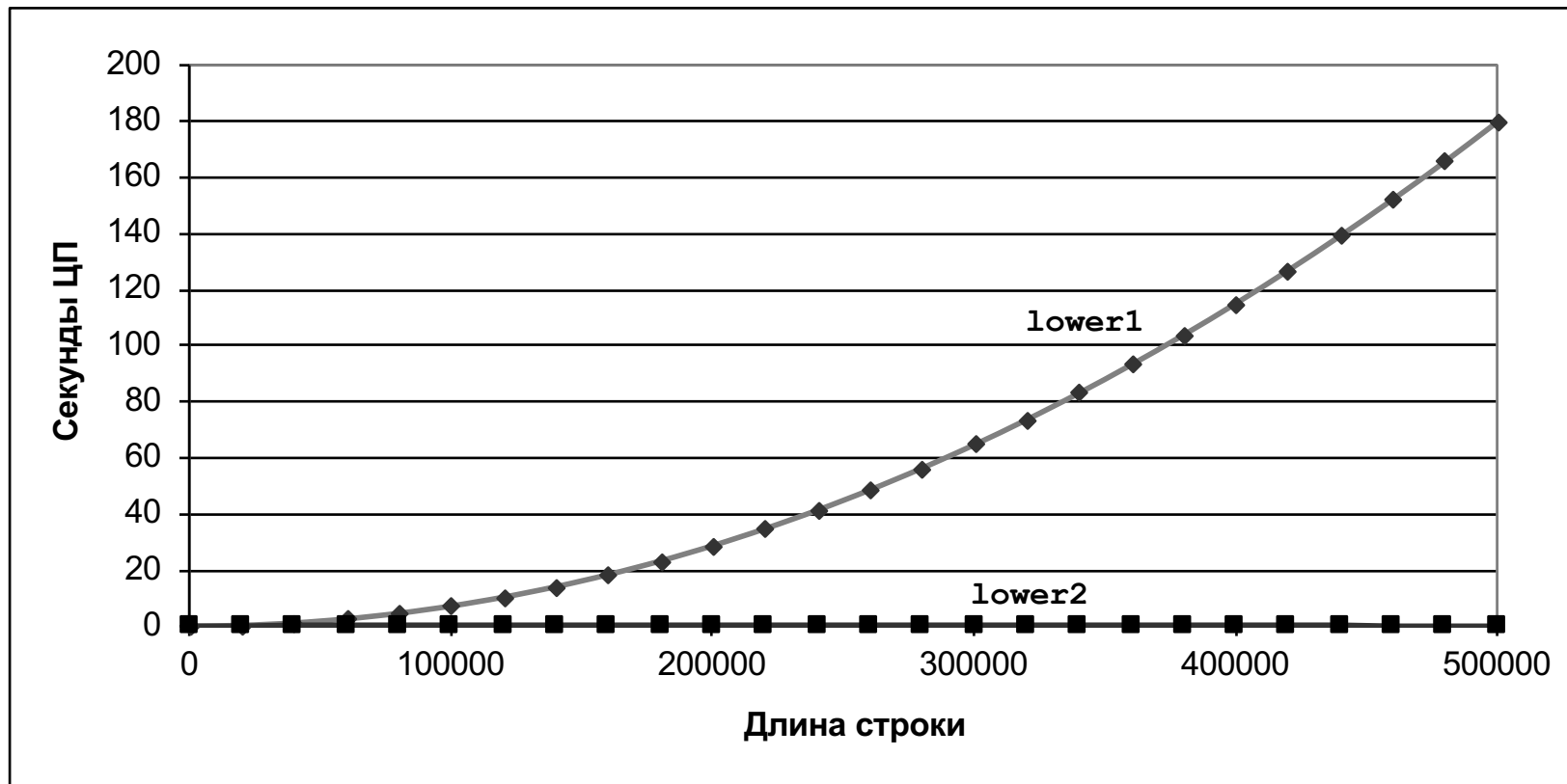
Улучшение производительности

```
void lower(char *s)
{
    int i;
    int len = strlen(s);
    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}
```

- Вынос из цикла обращения к `strlen`
- Возможно, т.к. никакой результат не зависит от итерации
- Вариант выноса кода

Быстродействие преобразования

- Время удваивается при удвоении длины строки
- Линейная сложность lower2



Помеха оптимизации: вызовы процедур

■ Почему компилятор не может вынести `strlen` из цикла?

- Процедура может иметь побочный эффект
 - Изменение общего состояния при каждом вызове
- Функция может возвращать разные результаты для одинаковых аргументов
 - В зависимости от других частей общего состояния
 - Вызывающая может взаимодействовать с вызываемой помимо аргументов

■ Предупреждение:

- Компилятор интерпретирует вызов процедуры как «чёрный ящик»
- Слабая оптимизация в его окрестности

■ Лекарство:

- Вынос кода вручную
- Используйте `inline` функции
 - GCC делает это с ключом `-O1`
 - Внутри одного файла

```
int lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    lencnt += length;
    return length;
}
```


О памяти

```
/* Суммировать строки матрицы n X n
   и сохранить в векторе b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 внутренний цикл
.L4:
    movsd    (%rsi,%rax,8), %xmm0      # загрузка ЧПТ
    addsd    (%rdi), %xmm0             # сложение ЧПТ
    movsd    %xmm0, (%rsi,%rax,8)      # выгрузка ЧПТ
    addq     $8, %rdi
    cmpq     %rcx, %rdi
    jne      .L4
```

- Код обращается к **b[i]** на каждой итерации
- Почему компилятор не может оптимизировать это выносом кода?

Псевдонимы памяти

```
/* Суммировать строки матрицы n X n
   и сохранить в векторе b */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
{ 0, 1, 2,
  4, 8, 16},
  32, 64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

Значения B:

init: [4, 8, 16]

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]

- Код изменяет `b[i]` на каждой итерации
- Компилятор должен учитывать возможность влияния этого изменения на поведение всей программы целиком

Исключение псевдонимов

```
/* Суммировать строки матрицы n X n
   и сохранить в векторе b */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L10:
    addsd    (%rdi), %xmm0      # загрузка и суммирование ЧПТ
    addq     $8, %rdi
    cmpq     %rax, %rdi
    jne      .L10
```

- Компилятор обнаружил, что нет необходимости сохранять промежуточный результат в памяти!

Помеха оптимизации: псевдонимы памяти

■ Псевдонимы

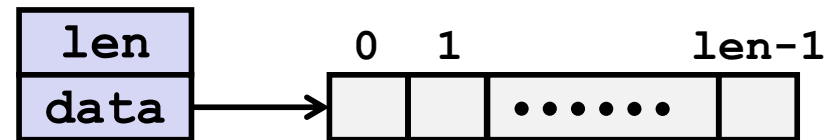
- Два различных обращения в память, обращаются к одной ячейке
- Легко случается именно в Си
 - разрешена адресная арифметика
 - прямой доступ к структурам хранения
- Возьмите за привычку использование локальных переменных
 - для накопления в циклах
 - **как способ исключить возможность псевдонимов как помеху оптимизации**

Использование параллелизма команд

- **Необходимо общее понимание конструкции современных процессоров**
 - Аппаратура может выполнить несколько команд параллельно
- **Быстродействие ограничено зависимостями данных**
- **Простые изменения могут привести к радикальному улучшению быстродействия**
 - Компиляторы часто неспособны выполнить такие изменения
 - Отсутствие ассоциативности и дистрибутивности у арифметики с плавающей точкой

Пример оценки: тип данных в векторах

```
/* структура данных для вектора */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```



■ Типы данных

- Используем различные определения `data_t`
 - `int`
 - `long`
 - `float`
 - `double`

```
/* выбирает элемент вектора
и сохраняет в val */
int get_vec_element
(*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Вычисление оценок

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Вычислить сумму
или произведение
элементов вектора

■ Варианты типов данных

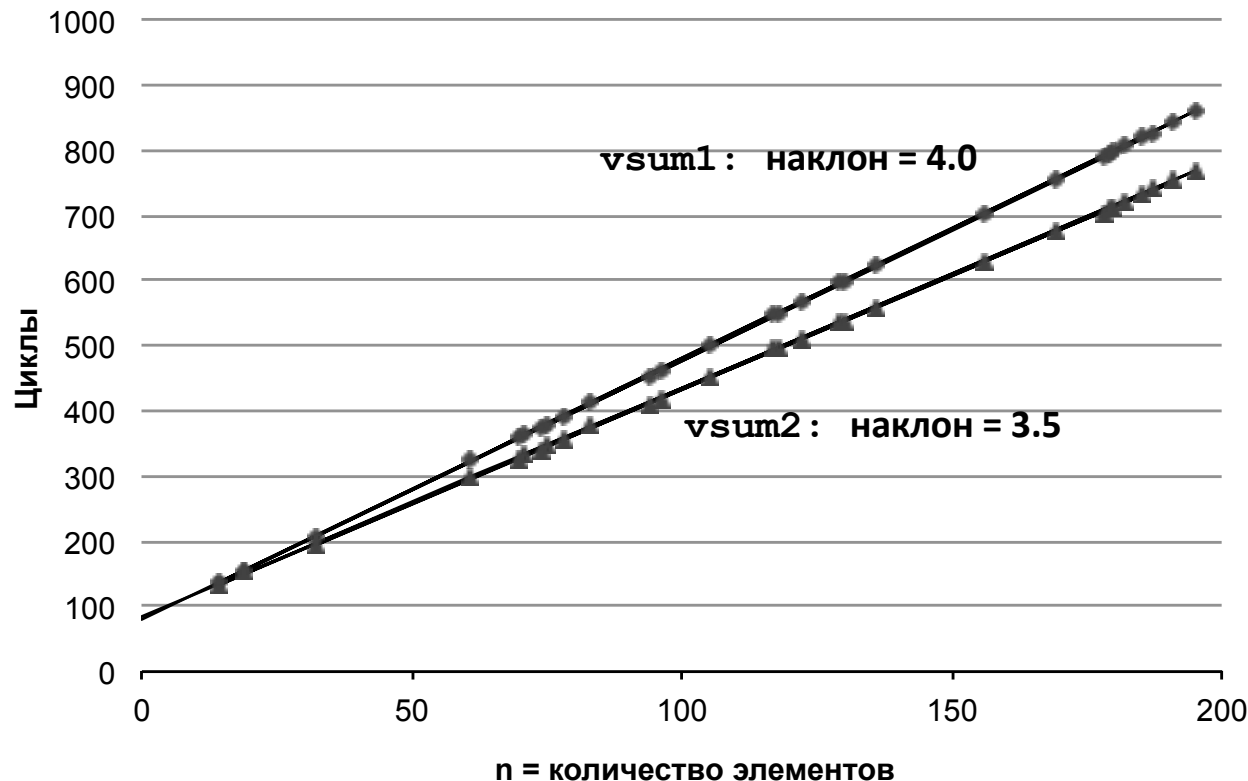
- int
- double

■ Варианты операций

- Различные определения
OP и IDENT
- + / 0
- * / 1

Циклов на элемент, Cycles Per Element (CPE)

- Удобный способ отображения быстродействия программы, работающей с массивами или списками
- Длина = n
- В нашем случае: **CPE = циклов на операцию**
- $T = \text{CPE} * n + \text{накладные}$
 - CPE – наклон прямой



Оценки быстродействия

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Вычислить сумму
или произведение
элементов вектора

Тип данных	int		double	
Операция	сложение	умножение	сложение	умножение
Combine1 неоптимизирована	22.68	20.02	19.98	20.18
Combine1 -O1	10.12	10.12	10.17	11.14

Простейшие оптимизации

```
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

- Вынос обращения к `vec_length` из цикла
- Исключение контроля границ в каждой итерации
- Накопление во временной переменной

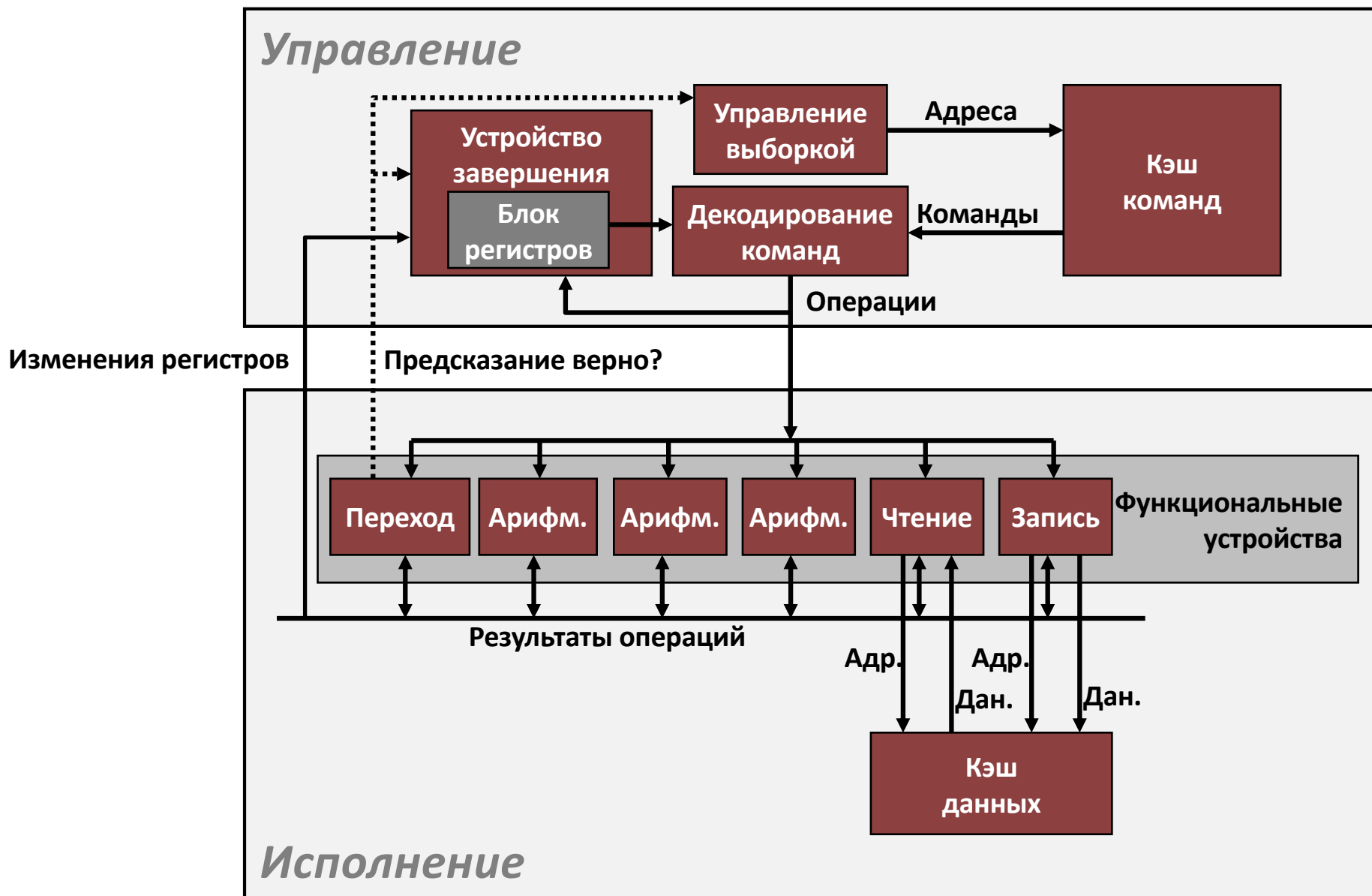
Эффект простейших оптимизаций

```
void combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP d[i];
    *dest = t;
}
```

Тип данных	int		double	
Операция	сложение	умножение	сложение	умножение
Combine1 -O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- Исключены причины накладных в цикле

Конструкция современного ЦП

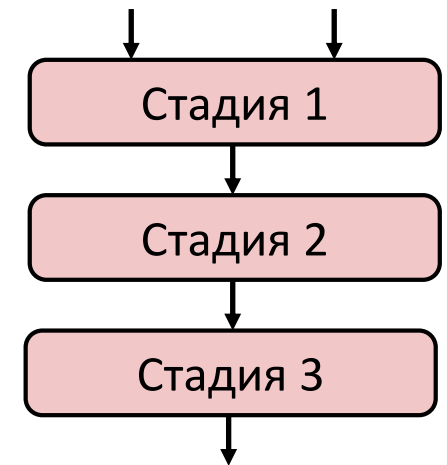


Суперскалярный процессор

- **Определение:** суперскалярный процессор может выбирать и исполнять *несколько команд за такт*. Инструкции выбираются из последовательного потока команд и обычно планируются динамически.
- **Преимущество:** без затрат на программирование, суперскалярный процессор может использовать *параллелизм команд*, который есть в большинстве программ
- Большинство современных ЦП - суперскалярные
- Intel: начиная с Pentium (1993)

Конвейерные функциональные устройства

```
long mult_eg(long a, long b, long c) {  
    long p1 = a*b;  
    long p2 = a*c;  
    long p3 = p1 * p2;  
    return p3;  
}
```



Время							
	1	2	3	4	5	6	7
Стадия 1	a*b	a*c			p1*p2		
Стадия 2		a*b	a*c			p1*p2	
Стадия 3			a*b	a*c			p1*p2

- Вычисления делятся на стадии
- Результат одной стадии передаётся на следующую
- Стадия i может начать новое вычисление как только результат передан стадии $i+1$
- Например, 3 умножения завершаются за 7 циклов, а не за 9

Процессор микроархитектуры Haswell

■ Несколько команд могут выполняться параллельно

2 загрузки с вычислением адресов

1 выгрузка с вычислением адресов

4 целочисленных операции

2 умножения с плавающей точкой

1 сложение с плавающей точкой

1 деление с плавающей точкой

■ Некоторые команды занимают > 1 такта, но могут быть конвейеризованы

<i>Команда</i>	<i>Задержка</i>	<i>Тактов/выборку</i>
Загрузка / Выгрузка	4	1
Целочисленное умножение	3	1
Целочисленное деление	3-30	3-30
Умножение с плавающей точкой	5	1
Сложение с плавающей точкой	3	1
Деление с плавающей точкой	3-15	3-15

x86-64 компиляция Combine4

■ Внутренний цикл (Вариант: целочисленное умножение)

```
.L519:                                # Цикл:
    imull                            (%rax,%rdx,4) , %ecx
    # t = t * d[i]
    addq $1, %rdx                    # i++
    cmpq %rdx, %rbp                  # Сравнение length:i
    jg    .L519                      # If >, goto Loop
```

Тип данных	int		double	
Операция	сложение	умножение	сложение	умножение
Combine4	1.27	3.01	3.01	5.01
Предел по задержке	1.00	3.00	3.00	5.00

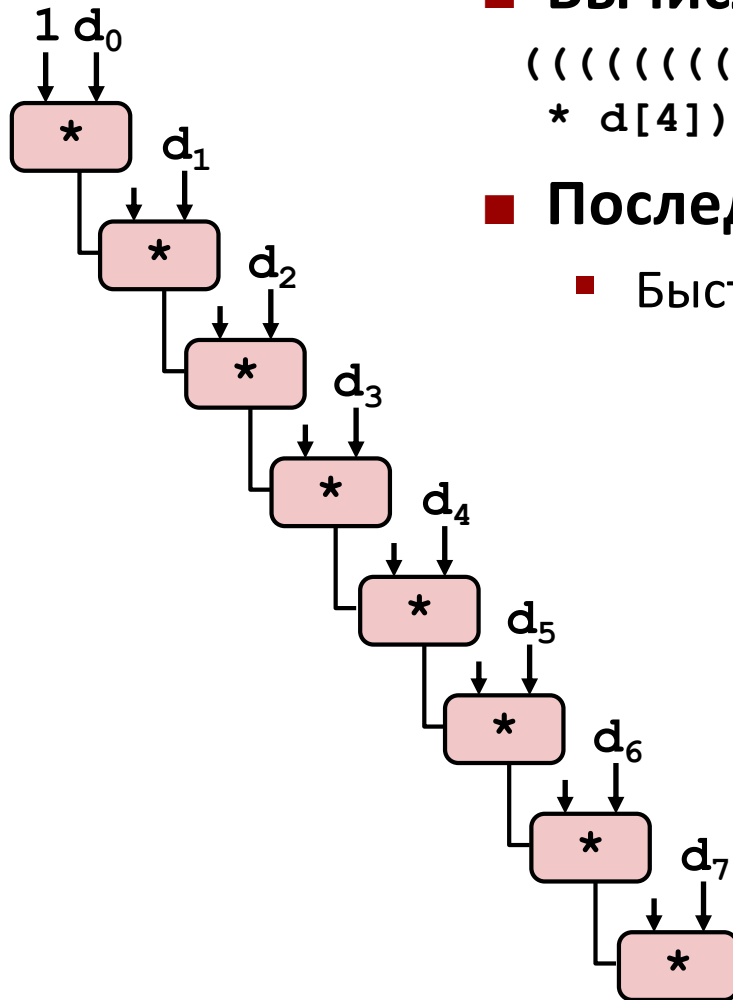
Combine4 = последовательные вычисления (OP = *)

■ Вычисления (length=8)

$(((((1 * d[0]) * d[1]) * d[2]) * d[3]) * d[4]) * d[5]) * d[6]) * d[7])$

■ Последовательная зависимость

- Быстродействие определяется задержкой OP



Разворачивание циклов (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Учёт 2-х элементов за раз */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Учёт оставшихся элементов */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Выполняет в 2 раза больше работы за итерацию

Эффект разворачивания циклов

Тип данных	int		double	
Операция	сложение	умножение	сложение	умножение
Combine4	1.27	3.01	3.01	5.01
Разворачивание 2x	1.01	3.01	3.01	5.01
Предел по задержке	1.00	3.00	3.00	5.00

- Помогло с целочисленным умножением

- Достигнут предел по задержке

- Остальные не улучшились. *Почему?*

- Остаётся последовательная зависимость

```
x = (x OR d[i]) OR d[i+1];
```

Разворачивание циклов с изменением порядка операций (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Учёт 2-х элементов за раз */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Учёт оставшихся элементов */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Сравните с предыдущим

$x = (x \text{ OP } d[i]) \text{ OP } d[i+1];$

- Может ли это изменить результат вычислений?
- Да, для плавающей точки. *Почему?*

Эффект переупорядочения

Тип данных	int		double	
Операция	сложение	умножение	сложение	умножение
Combine4	1.27	3.01	3.01	5.01
Разворачивание 2х	1.01	3.01	3.01	5.01
Разворачивание 2х, с изменением порядка	1.01	1.51	1.51	2.51
Предел по задержке	1.00	3.00	3.00	5.00
Предел по выборке	0.50	1.00	1.00	0.50

■ Почти 2-кратное ускорение int *, double +, double *

- Причина: нарушение последовательной зависимости

```
x = x OP (d[i] OP d[i+1]);
```

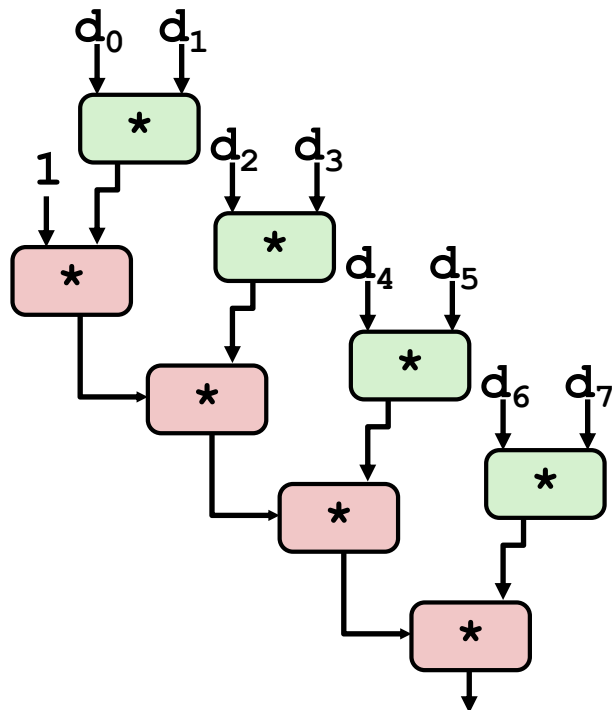
- Почему это происходит?

2 ФУ для double *
2 ФУ для загрузки

4 ФУ для int +
2 ФУ для загрузки

Переупорядоченные вычисления

```
x = x OP (d[i] OP d[i+1]);
```



■ Что изменилось:

- Операции следующей итерации могут начинаться раньше (нет зависимости)

■ Быстродействие в целом

- N элементов, D тактов задержки на операцию
- Должно быть $(N/2+1)*D$ тактов:
CPE = D/2

Разворачивание циклов с раздельными накопителями (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Учёт 2-х элементов за раз */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Учёт оставшихся элементов */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

■ Другой вариант переупорядочивания

Эффект раздельных накопителей

Тип данных	int		double	
Операция	сложение	умножение	сложение	умножение
Combine4	1.27	3.01	3.01	5.01
Разворачивание 2х	1.01	3.01	3.01	5.01
Разворачивание 2х, с изменением порядка	1.01	1.51	1.51	2.51
Разворачивание 2х, 2 накопителя	0.81	1.51	1.51	2.51
Предел по задержке	1.00	3.00	3.00	5.00
Предел по выборке	0.50	1.00	1.00	0.50

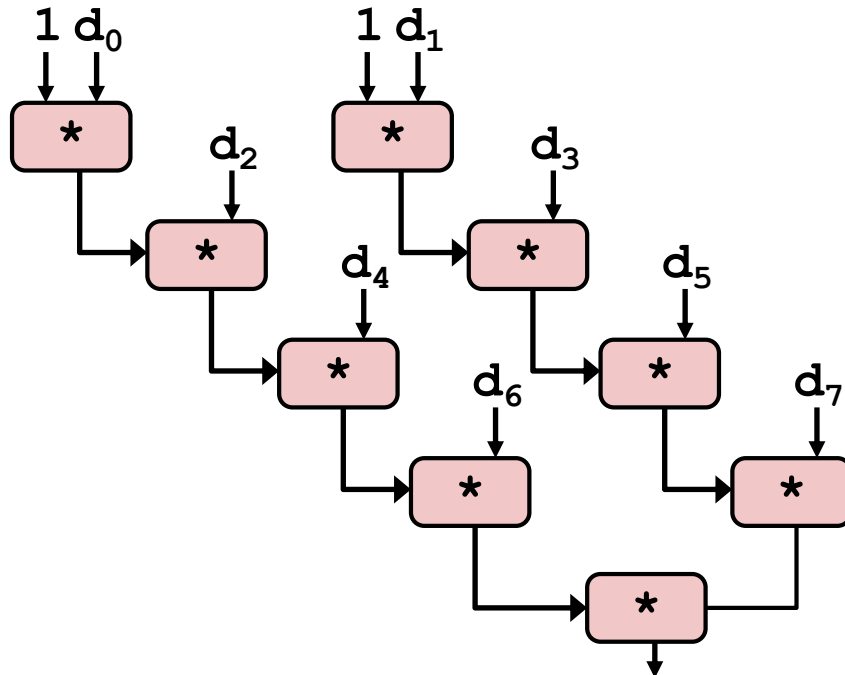
- Int + задействует 2 устройства загрузки

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

- 2-кратное ускорение для int *, double +, double *

Раздельные накопители

```
x0 = x0 OP d[i];  
x1 = x1 OP d[i+1];
```



■ Что изменилось:

- Два независимых “потока” операций

■ Быстродействие в целом

- N элементов, D тактов задержки на операцию
- Должно быть $(N/2+1)*D$ тактов:
 $CPE = D/2$
- CPE совпадает с ожидаемым!

Что теперь?

Разворачивание и разделение

■ Идея

- Можно развернуть до любой степени L
- Можно накапливать K результатов параллельно
- L должно быть кратно K

■ Ограничения

- Не получится превзойти ограничения пропускной способности исполнительных устройств
- Большие накладные расходы для малых длин
 - Последовательное завершение итераций

Разворачивание и разделение: double *

■ Вариант

- Intel Haswell
- Умножение double
- Предел по задержке: 5.00. Предел по выборке: 0.50

Накопители

FP *	Кратность разворачивания L							
К	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Разворачивание и разделение: int +

■ Вариант

- Intel Haswell
- Сложение int
- Предел по задержке: 1.00. Предел по выборке: 0.5

Накопители

Int +	Кратность разворачивания L							
К	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

Достижимое быстродействие

Тип данных	int		double	
Операция	сложение	умножение	сложение	умножение
Лучшая версия	0.54	1.01	1.01	0.52
Предел по задержке	1.00	3.00	3.00	5.00
Предел по выборке	0.50	1.00	1.00	0.50

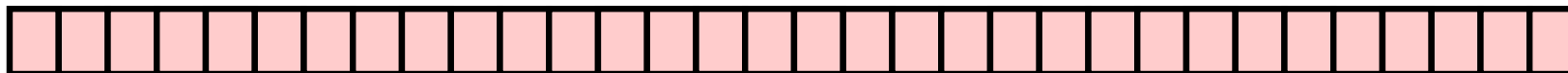
- Ограничено только пропускной способностью исполнительного устройства
- 42-кратное ускорение исходного, неоптимизированного кода

Программирование с AVX2

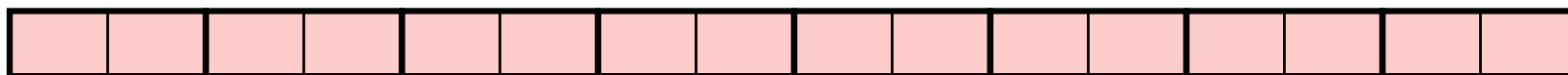
Регистры YMM

■ 16 штук по 32 байта

■ 32 однобайтных целых



■ 16 16-битных целых



■ 8 32-битных целых



■ 8 ЧПТ одинарной точности



■ 4 ЧПТ двойной точности



■ 1 ЧПТ одинарной точности



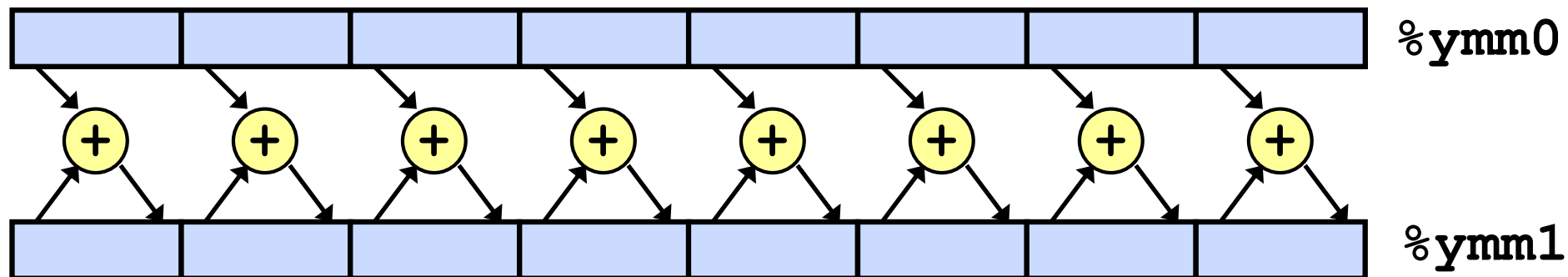
■ 1 ЧПТ двойной точности



Команды SIMD

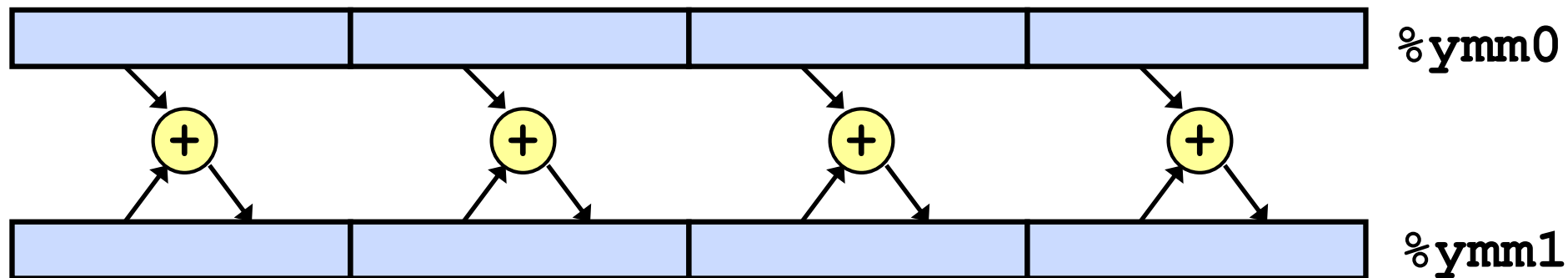
■ Одинарная точность

`vaddsd %ymm0, %ymm1, %ymm1`



■ Двойная точность

`vaddpd %ymm0, %ymm1, %ymm1`



Используя векторные инструкции

Тип данных	int		double	
Операция	сложение	умножение	сложение	умножение
Скалярный оптимум	0.54	1.01	1.01	0.52
Векторный оптимум	0.06	0.24	0.25	0.16
Предел по задержке	0.50	3.00	3.00	5.00
Предел по выборке	0.50	1.00	1.00	0.50
Векторная выборка	0.06	0.12	0.25	0.12

■ Использование команд SSE

- Параллельные операции над несколькими элементами данных
- Детали OPT:SIMD на сайте:
<http://csapp.cs.cmu.edu/public/waside.html>

Об условных переходах

■ Проблема

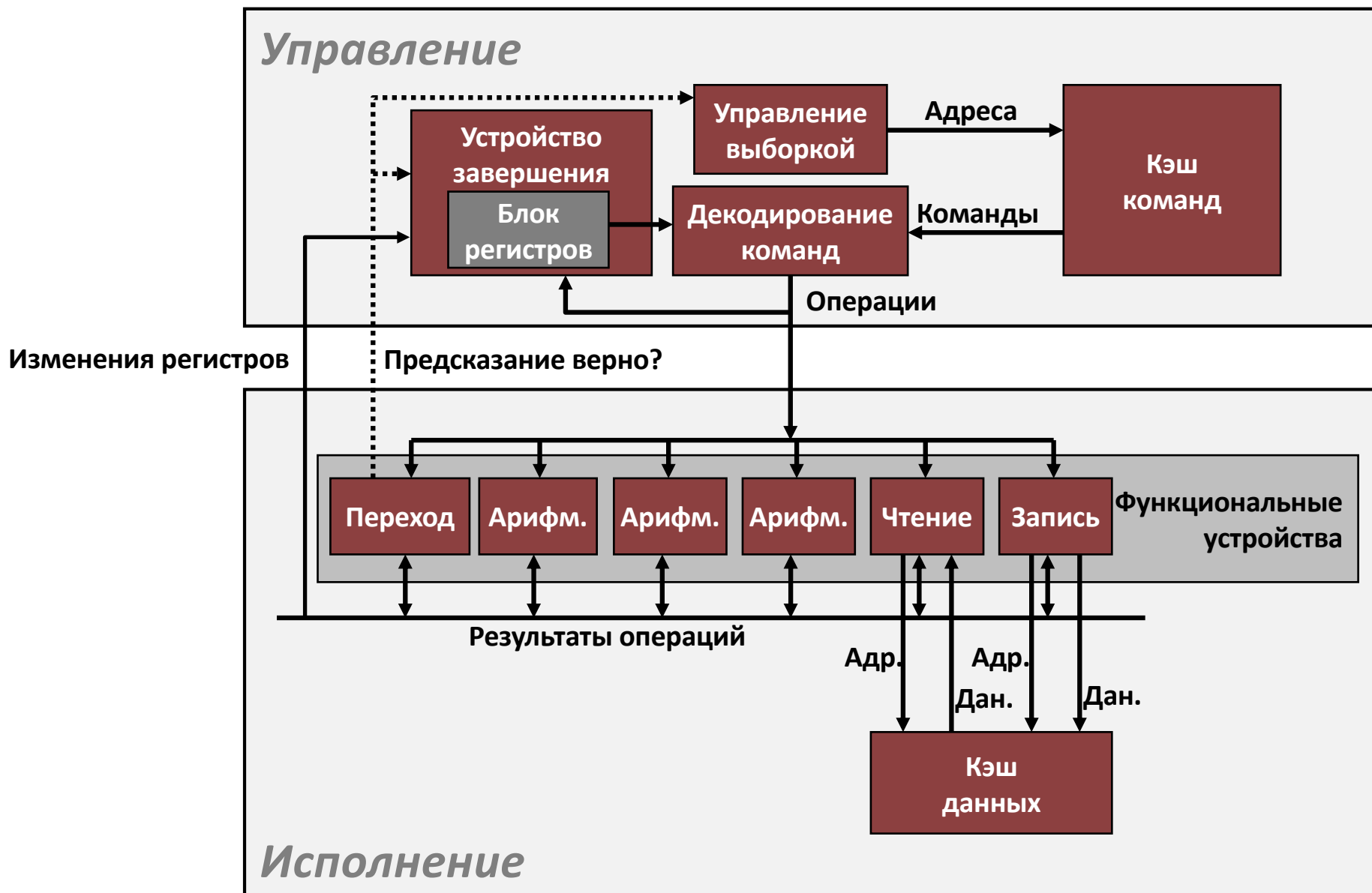
- Устройство Управления (командами) должно работать опережая Исполнительное Устройство и порождать операции так, чтобы последнее было эффективно загружено и не простаивало.

404663:	mov	\$0x0, %eax
404668:	cmp	(%rdi), %rsi
40466b:	jge	404685 ←
40466d:	mov	0x8(%rdi), %rax
. . .		
404685:	repz retq	

} Исполнение
Куда дальше?

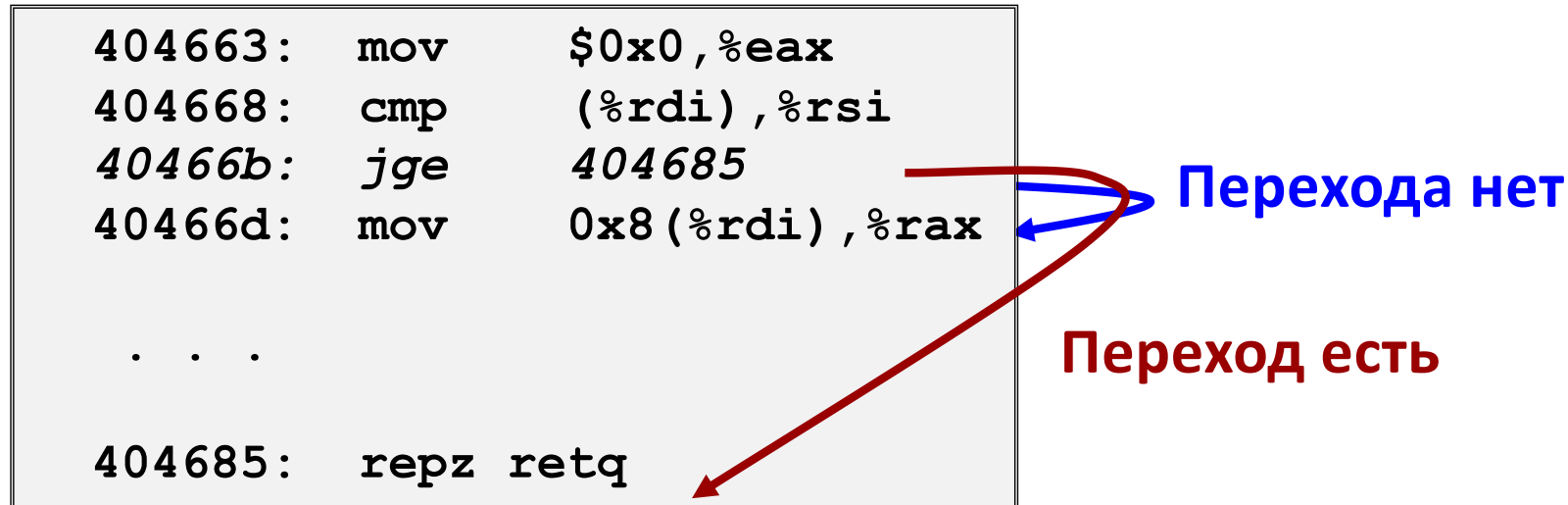
- Когда попадаете условный переход, нет возможности надёжно определить, откуда продолжать выборку команд

Конструкция современного ЦП



Результат условного перехода

- Когда попадаете условный переход, нет возможности надёжно определить откуда продолжать выборку команд
 - Переход произошёл: передача управления на цель перехода
 - Переход не произошёл: продолжение со следующей по коду командой
- Невозможно решить пока результат не будет выдан функциональным устройством «Переход»



Предсказание переходов

■ Идея

- Допустим условный переход произойдёт
- Начнём исполнять команды по предсказанному адресу
 - Но не будем изменять значения регистров и памяти

```
404663:  mov    $0x0,%eax
404668:  cmp    (%rdi),%rsi
40466b:  jge    404685
40466d:  mov    0x8(%rdi),%rax

. . .

404685:  repz   retq
```

Переход есть

} Продолжение
исполнения

Предсказание перехода в цикле

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

$i = 98$

Положим
длину вектора = **100**

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

$i = 99$

Предсказание успешно

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

$i = 100$

Предсказание ошибочно

Чтение из
ошибочного
места

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

$i = 101$

Исполнение

Выборка

Отмена ошибки предсказания

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 98

Положим
длину вектора = **100**

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 99

Предсказание успешно

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 100

Предсказание ошибочно

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
```

i = 101

Отмена

Исправление ошибки предсказания

```
401029: vmulsd (%rdx), %xmm0, %xmm0
40102d: add    $0x8, %rdx
401031: cmp    %rax, %rdx
401034: jne    401029
401036: jmp    401040
. . .
401040: vmovsd %xmm0, (%r12)
```

i = 99

перехода нет совсем

перезагрузка
конвейера

■ Потери быстродействия

- Много тактов на современных процессорах
- Может быть основным ограничителем быстродействия

Достижение высокого быстродействия

- **Хороший компилятор и флаги**
- **Не делайте глупостей**
 - Следите за скрытыми алгоритмическими неэффективностями
 - Пишите дружелюбный к компилятору код
 - Следите за помехами оптимизации:
обращениями к процедурам и в память
 - Тщательно анализируйте самые внутренние циклы (где делается большая часть работы)
- **Подстраивайте код под машину**
 - Используйте параллелизм команд
 - Избегайте непредсказуемых переходов
 - Пишите код дружелюбный к кэшу (продолжение следует...)