Кеширование памятей

Основы информатики

Компьютерные основы программирования

u.to/DbCmFA

На основе **CMU** 15-213/18-243:
Introduction to Computer Systems

u.to/XoKmFA

Лекция 10, 14 апреля 2023

Лектор:

Дмитрий Северов, кафедра информатики 608 КПМ

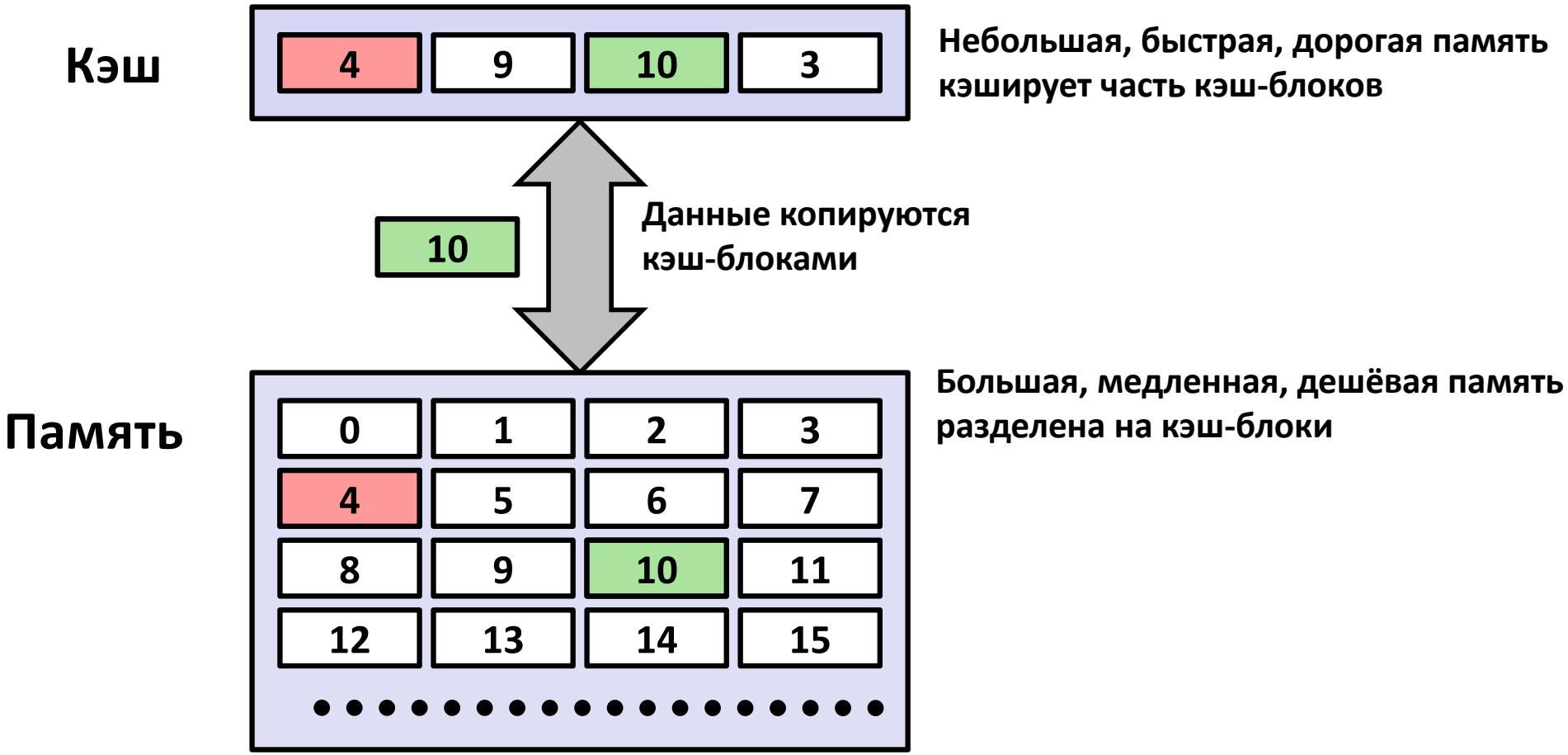
cs.mipt.ru/wp/?page_id=346

Кеширование памятей

- **Организация и работа кэша**
- **Влияние кэша на быстродействие памяти**
  - Диаграмма быстродействия памяти
  - Реорганизация циклов улучшает пространственную локальность
  - Блокирование улучшает временнУю локальность

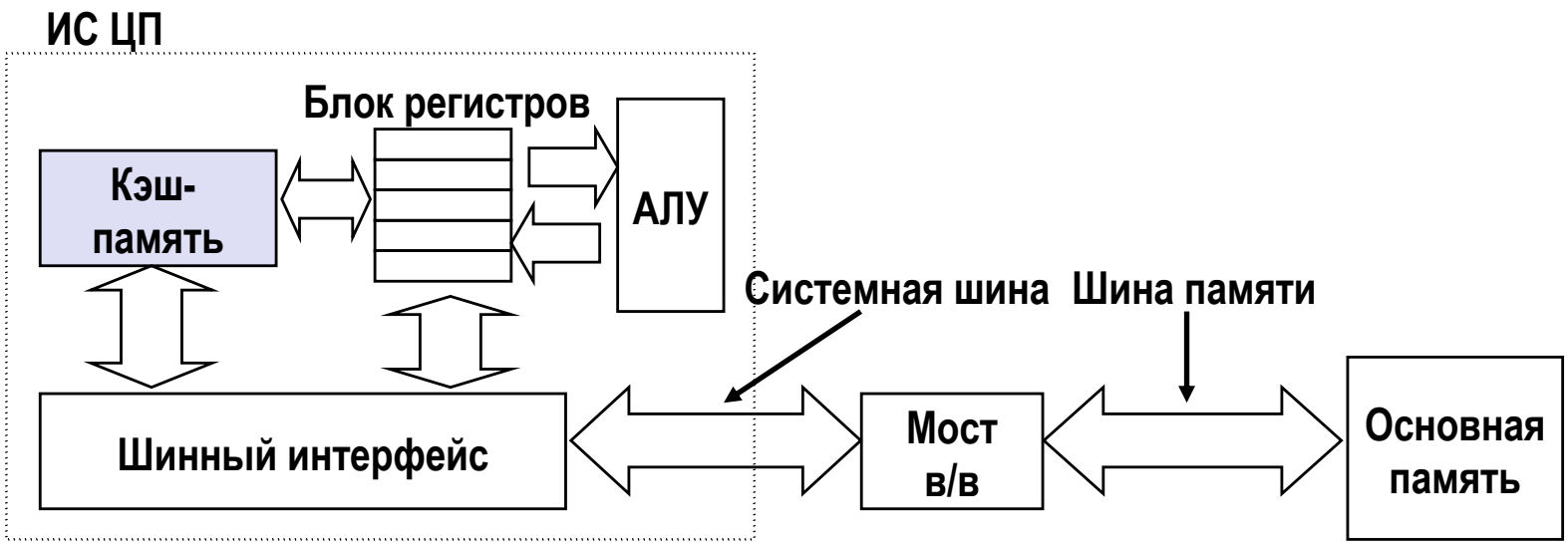
Пример иерархии памятей

Кэш в общем

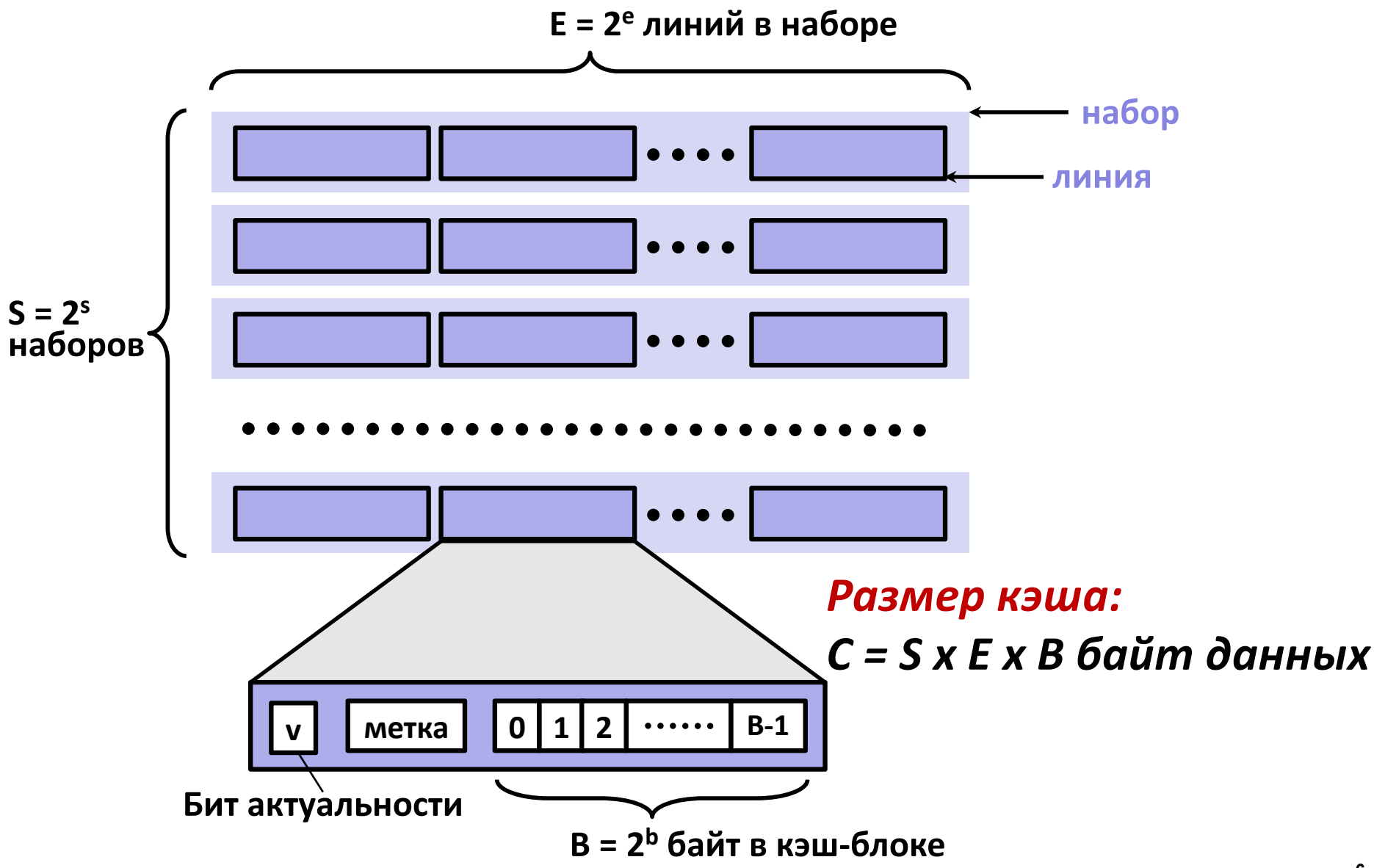

Кэш
4
9
10
3
Небольшая, быстрая, дорогая память кэширует часть кэш-блоков
10
Данные копируются кэш-блоками
Память
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Большая, медленная, дешёвая память разделена на кэш-блоки

Кэш-памяти

- **Кэш-памяти – небольшие, быстрые памяти на основе SRAM, автоматически управляемые аппаратурой.**
  - Хранит часто используемые блоки основной памяти
- **Ядро ЦП сначала обращается за данными в кэши, а затем в основную память. Если потребуется.**
- **Типичная структура системы:**

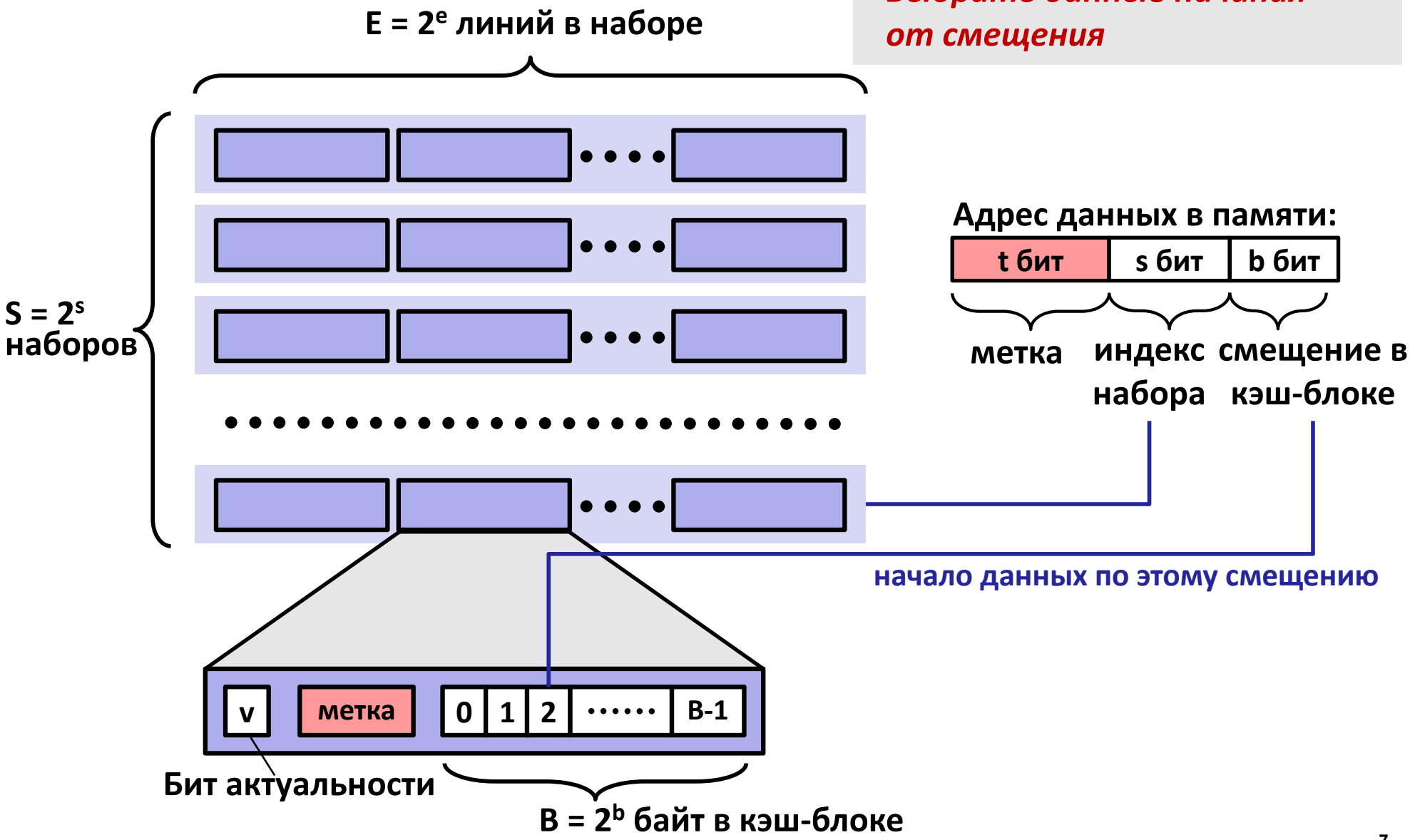


Общая организация кэша (S, E, B)



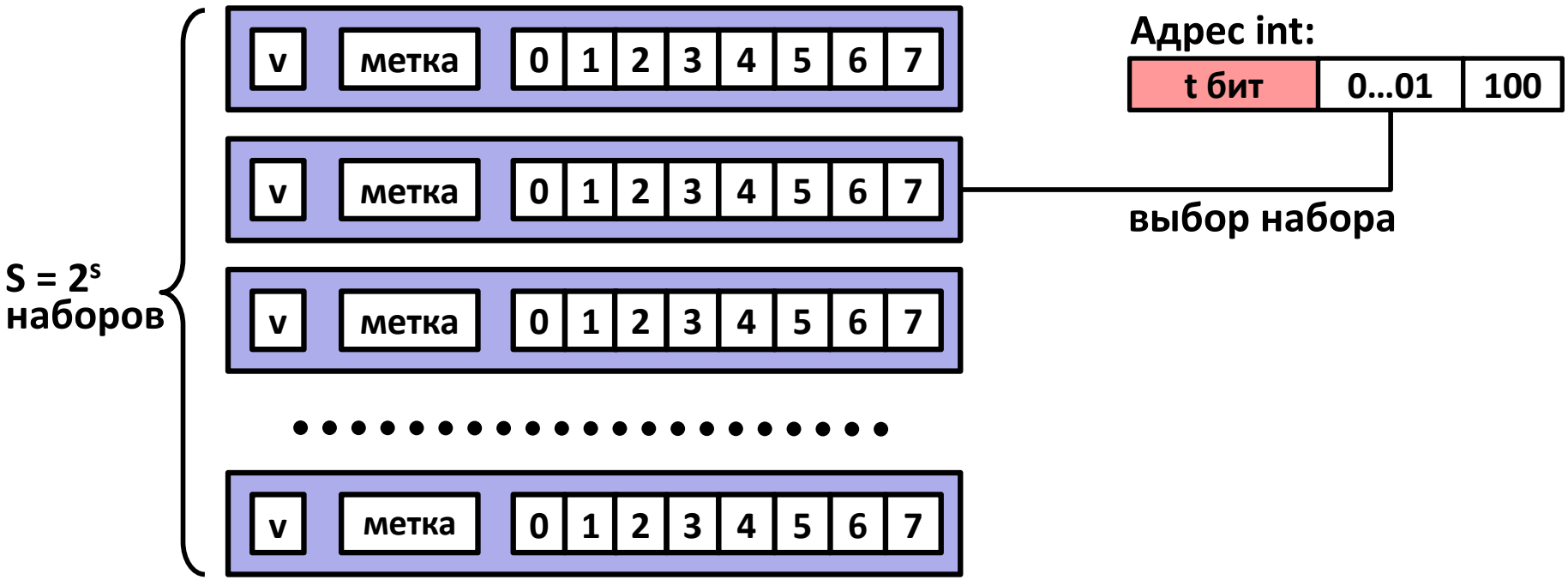
Чтение кэша

- *Выбрать набор*
- *Проверить на совпадение метки линий в наборе*
- *Есть + актуальна: попадание!*
- *Выбрать данные начиная от смещения*



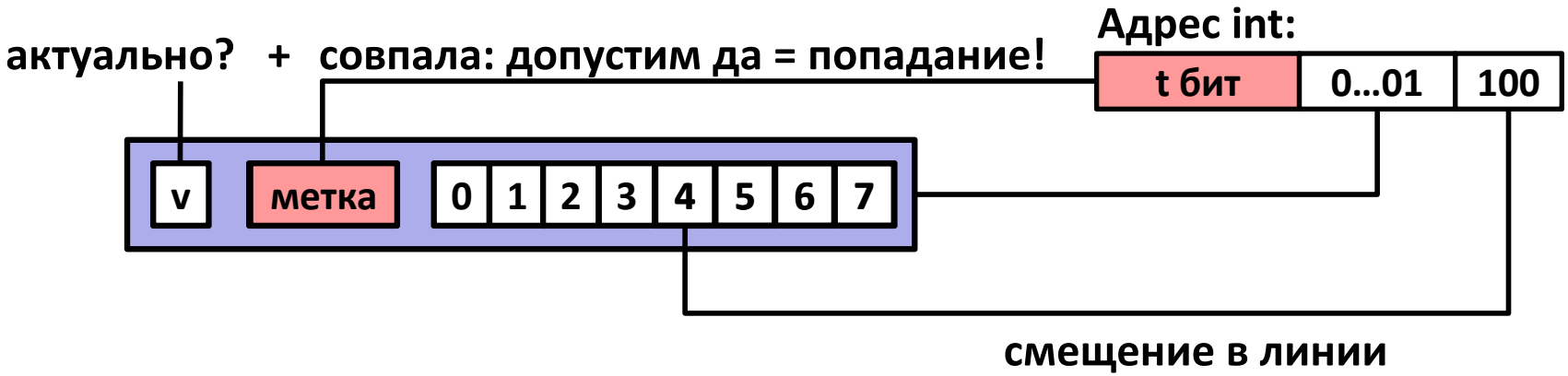
Пример: Кэш прямого отображения (E = 1)

Прямое отображение: одна линия в наборе
Допустим: размер кэш-блока - 8 байт



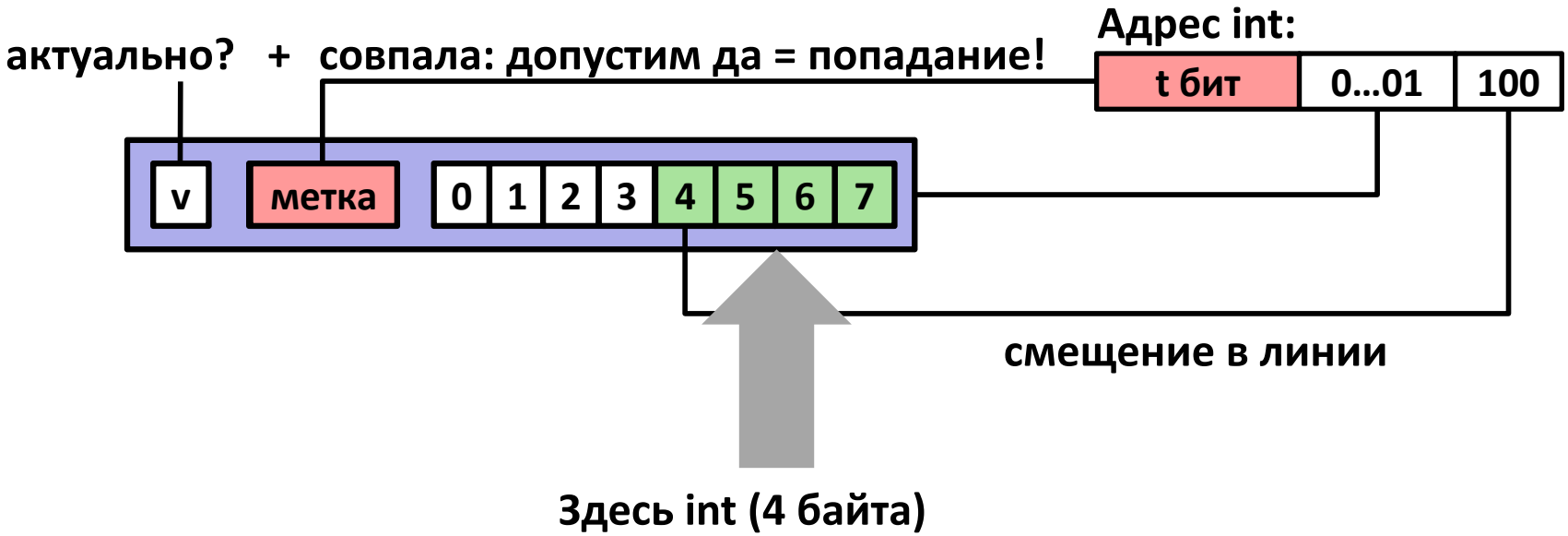
Пример: Кэш прямого отображения (E = 1)

Прямое отображение: одна линия в наборе
Допустим: размер кэш-блока - 8 байт



Пример: Кэш прямого отображения (E = 1)

Прямое отображение: одна линия в наборе
Допустим: размер кэш-блока - 8 байт



Если не совпала, то старая линия освобождается и замещается

Имитирование кэша прямого отображения

t=1	s=2	b=1
x	xx	x

M=16 адресов байтов, B=2 байта в кэш-блоке,
S=4 набора, E=1 линия в наборе

Трассировка адресов (чтения, по одному байту):

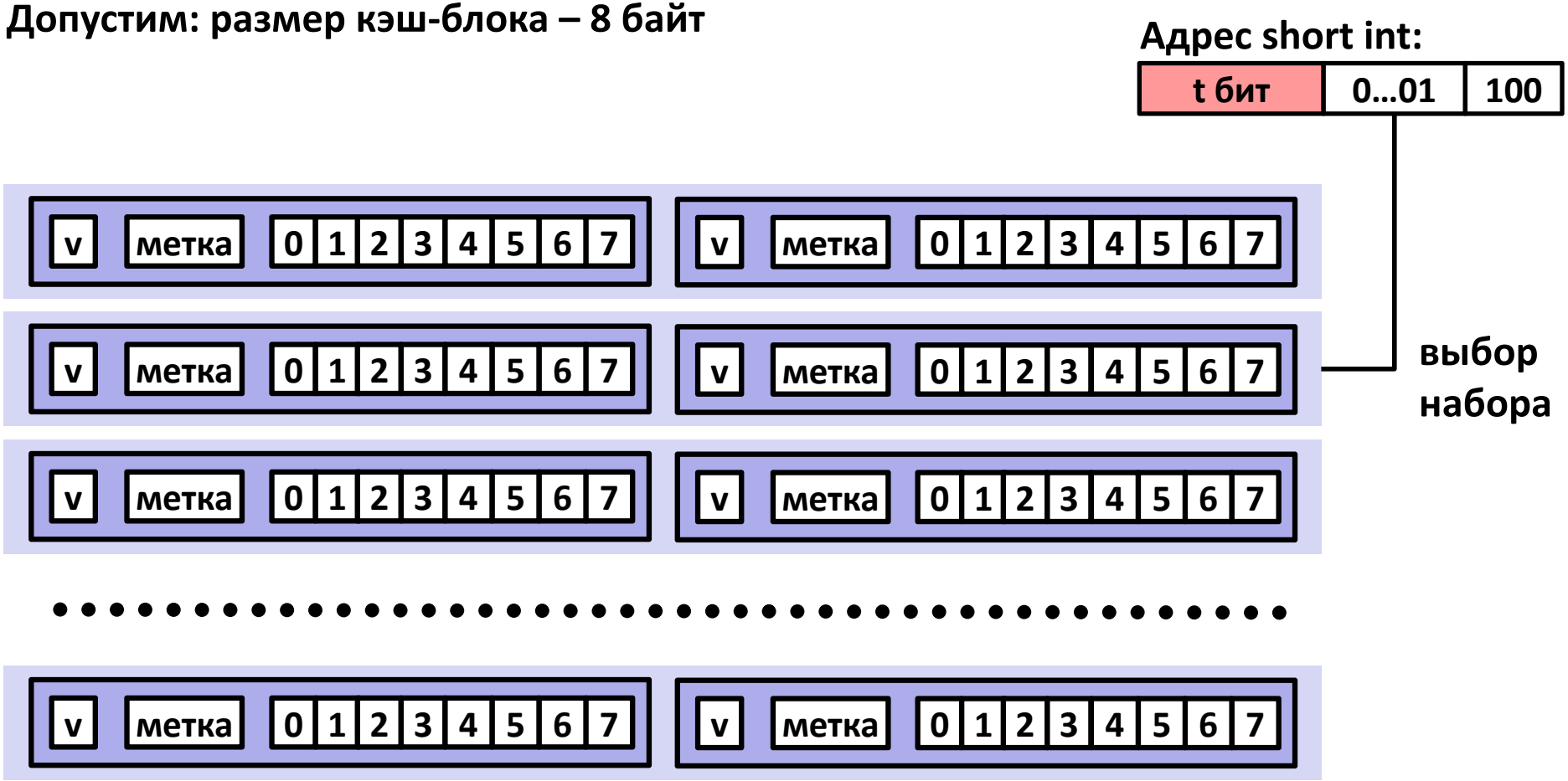
0	**[0$\underline{00}$0$_2$],**	промах
1	**[0$\underline{00}$1$_2$],**	попадание
7	**[0$\underline{11}$1$_2$],**	промах
8	**[1$\underline{00}$0$_2$],**	промах
0	**[0$\underline{00}$0$_2$]**	промах

	v	метка	кеш-блок
Набор 0	1	0	M[0-1]
Набор 1			
Набор 2			
Набор 3	1	0	M[6-7]

E-канальный наборно-ассоциативный кэш (здесь: E = 2)

E = 2: Две линии в наборе

Допустим: размер кэш-блока – 8 байт



E-канальный наборно-ассоциативный кэш (здесь: E = 2)

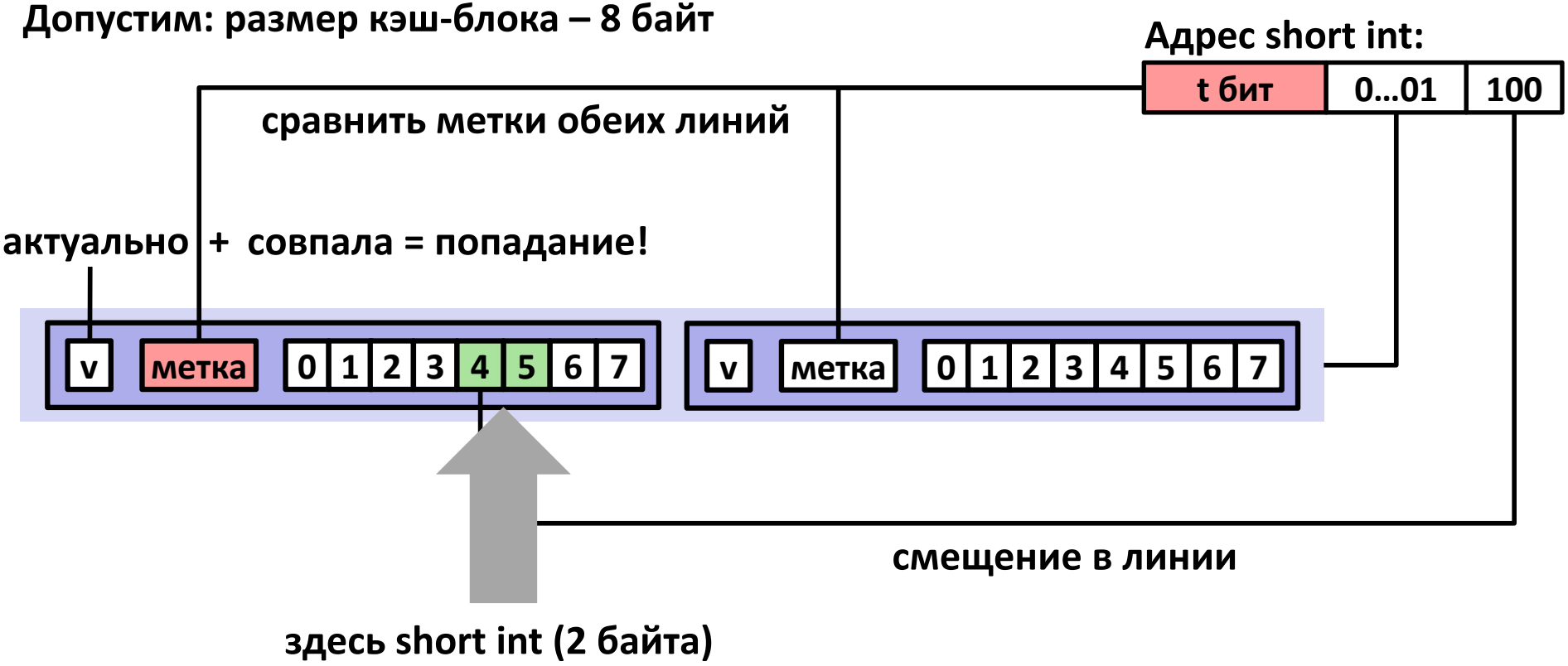
E = 2: Две линии в наборе

Допустим: размер кэш-блока – 8 байт

E-канальный наборно-ассоциативный кэш (здесь: E = 2)

E = 2: Две линии в наборе

Допустим: размер кэш-блока – 8 байт



Если не совпала, то…

- Одна линия в наборе освобождается и замещается
- Политики замещения: случайно, least recently used (LRU), …

Имитирование 2-канального наборно-ассоциативного кэша

t=2	s=1	b=1
xx	x	x

M=16 байтовых адресов, B=2 байта в кэш-блоке, S=2 набора, E=2 линии в набор

Трассировка адресов (чтения, по одному байту):

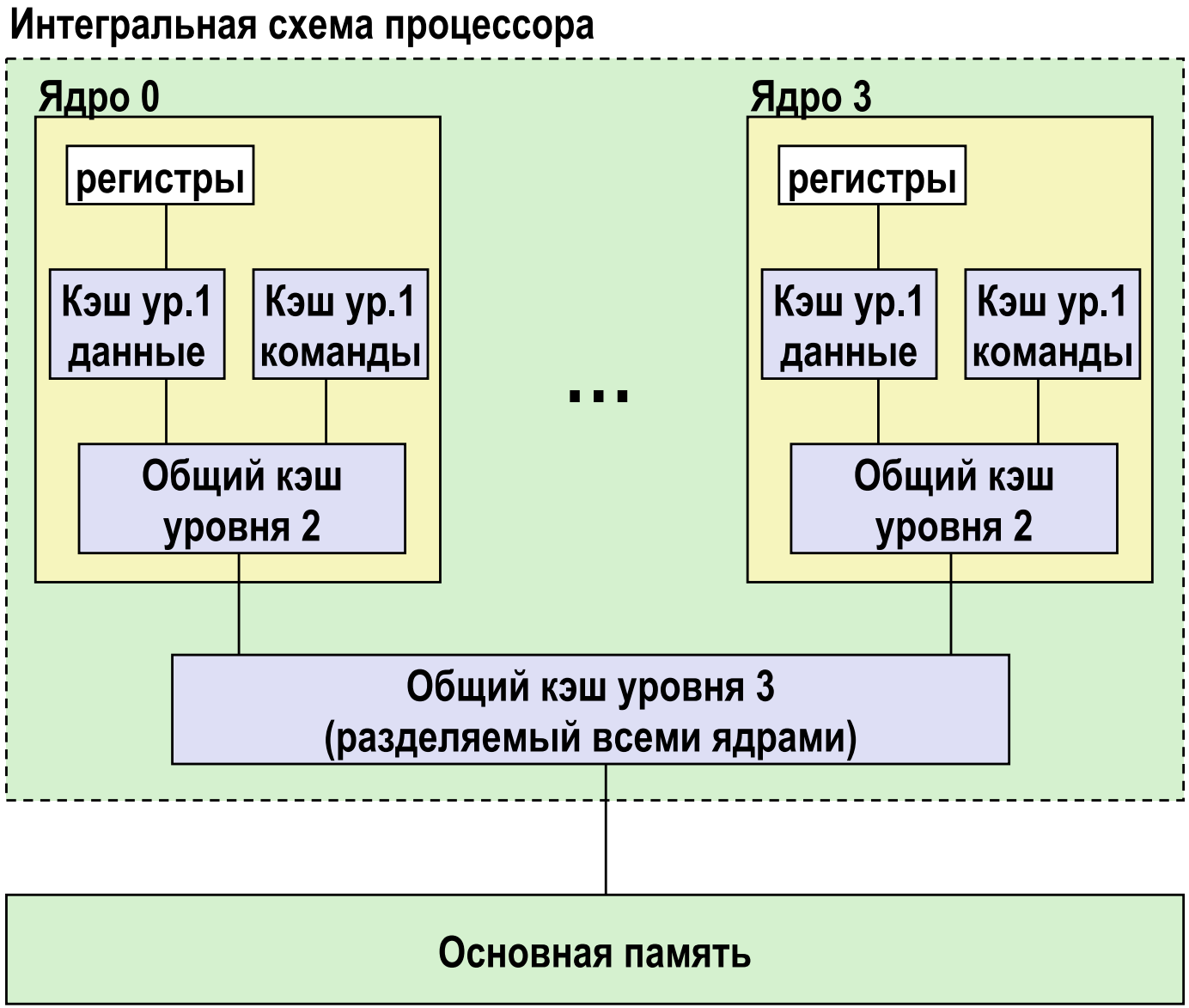
0	**[00$\underline{0}$0$_2$],**	промах
1	**[00$\underline{0}$1$_2$],**	попадание
7	**[01$\underline{1}$1$_2$],**	промах
8	**[10$\underline{0}$0$_2$],**	промах
0	**[00$\underline{0}$0$_2$]**	попадание

	v	**метка**	**кэш-блок**
Блок 0	1	00	M[0-1]
	1	10	M[8-9]
Блок 1	1	01	M[6-7]
	0		

Немного о записи

- **Присутствуют несколько копий данных:**
  - Кэши памяти, основная память, диск
- **Что делать при записи с попаданием?**
  - Write-through (запись непосредственно в память)
  - Write-back (запись в память задерживается до замены линии)
    - Нужен бит несоответствия (линия совпадает с памятью или нет)
- **Что делать при записи с промахом?**
  - Write-allocate (загрузка в кэш, изменение линии кэша)
    - Хорошо если ожидаются ещё записи
  - No-write-allocate (запись непосредственно в память)
- **Типичные политики**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

Иерархия кэшей Intel Core i7



Кэш команд и кэш данных уровня 1:
32 КБ, 8-каналов,
Доступ: 4 цикла

Общий кэш уровня 2:
256 КБ, 8-каналов,
Доступ: 11 циклов

Общий кэш уровня 3:
8 МБ, 16-каналов,
Доступ: 30-40 циклов

Размер кеш-блока: 64 байта для всех кэшей

Характеристики эффективности кэша

- **Вероятность промахов**
  - Доля обращений в память не обнаруженных к кэше (промахов / доступов) = 1 – вероятность попаданий
  - Типичные значения (в процентах):
    - 3-10% для кэша уровня 1
    - Может быть весьма малым (< 1%) для кэша уровня 2, в зависимости от размера
- **Продолжительность доступа в кэш**
  - Время доставки данных из кэша в процессор
    - Включая время определение наличия данных в кэше
  - Типичные величины:
    - 1-2 такта для кэша уровня 1
    - 5-20 тактов для кэша уровня 2
- **Продолжительность промаха**
  - Дополнительное время необходимое при промахе
    - обычно 50-200 тактов для основной памяти (и будет расти!)

Некоторые мысли о характеристиках эффективности кэша

- **Громадная разница между попаданиями и промахами**
  - До 100 раз, для кэша первого уровня и основной памяти

- **Верно ли что 99% попаданий в два раза лучше чем 97%?**
  - Допустим:
    продолжительность доступа в кэш – 1 такт
    продолжительность промаха – 100 тактов

  - Среднее время доступа:
    97% попаданий: 1 такт + 0.03 * 100 тактов = **4 такта**
    99% попаданий: 1 такт + 0.01 * 100 тактов = **2 такта**

- **Поэтому в основном используется термин “вероятность промаха”, и не “вероятность попадания”**

Создание программ дружелюбных к кэшу

- **Ускорение наиболее часто исполняемых участков**
  - Сосредоточение на внутренних циклах основных функций

- **Минимизация промахов во внутренних циклах**
  - Повторные обращение к переменным (временнАя локальность)
  - Доступ с единичным шагом (пространственная локальность)

Ключевая идея: благодаря пониманию кэш-памяти качественное теоретическое понятие локальности получает практическую количественную меру

Кеширование памятей

- **Организация и работа кэша**
- **Влияние кэша на быстродействие памяти**
  - Диаграмма быстродействия памяти
  - Реорганизация циклов улучшает пространственную локальность
  - Блокирование улучшает временнУю локальность

«Гора» (быстродействия) памяти

- **Скорость чтения (пропускная способность чтения)**
  - К-во байт считываемых из памяти за секунду (МБ/сек)

- **Диаграмма быстродействия памятей «Гора» : Измеренная пропускная способность чтения как функция временной и пространственной локальности.**
  - Компактный способ охарактеризовать быстродействие подсистемы памяти.

Функция измерения «горы»

```
long data[MAXELEMS]; /* Глобальный читаемый массив */

/* test – Читаем первые "elems" элементов
 *        массива “data” с шагом "stride",
 *        разворачивая цикл 4x4.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Накапливаем 4 4 элемента за итерацию */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Подбираем оставшиеся элементы */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Многократно вызывать test() варьируя elems и stride.

Для каждого elems и stride:

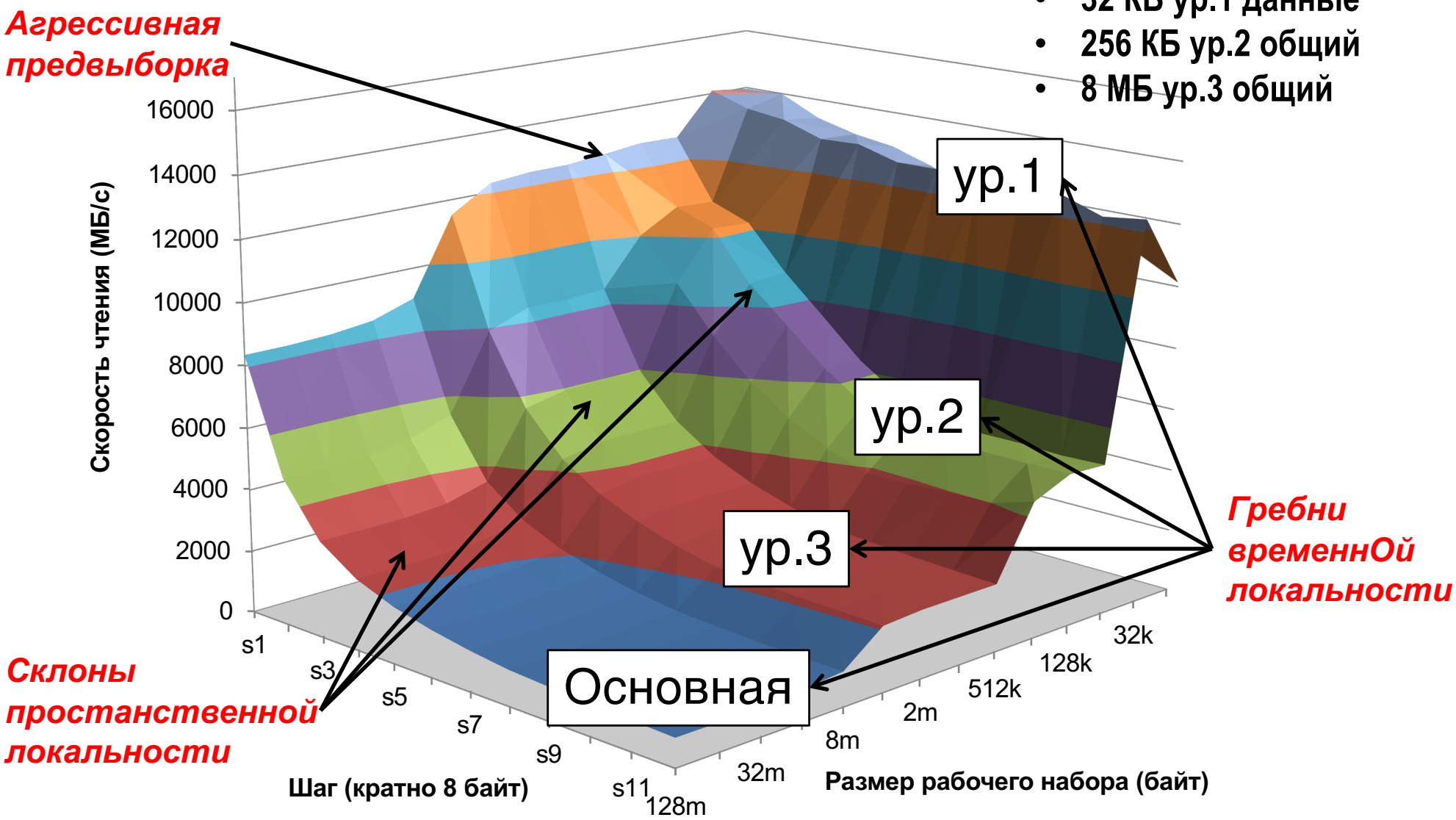
1. Для заполнения кеша вызвать test().

2. Вызвать test() снова и измерить again and скорость чтения (МБ/с)

«Гора» памяти

Core i7 Haswell 2.1 ГГц
Размеры кеш-памятей:

- 64 Б блок
- 32 КБ ур.1 данные
- 256 КБ ур.2 общий
- 8 МБ ур.3 общий



Кеширование памятей

- **Организация и работа кэша**
- **Влияние кэша на быстродействие памяти**
  - Диаграмма быстродействия памяти
  - Реорганизация циклов улучшает пространственную локальность
  - Блокирование улучшает временнУю локальность

Пример перемножения матриц

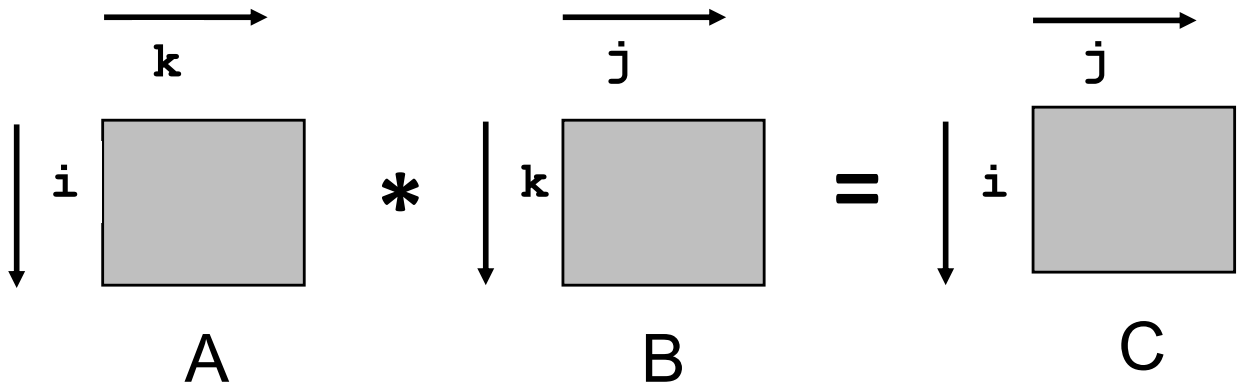
- **Описание:**
  - Перемножение матриц N x N
  - Всего $O(N^3)$ операций
  - N чтений каждого исходного элемента
  - Каждый результат - сумма N значений
    - может накапливаться в регистре

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Переменная **sum** *находится в регистре*

Анализ вероятности промаха для матричного умножения

- **Допустим:**
  - Размер линии = 32 байта (достаточно для 4-х 64-битных слов)
  - Размер матрицы (N) очень большой
    - 1/N приблизительно представляется 0.0
  - Кэш недостаточно велик, чтобы содержать несколько строк матрицы
- **Метод анализа:**
  - Посмотрим на схему доступа во внутреннем цикле



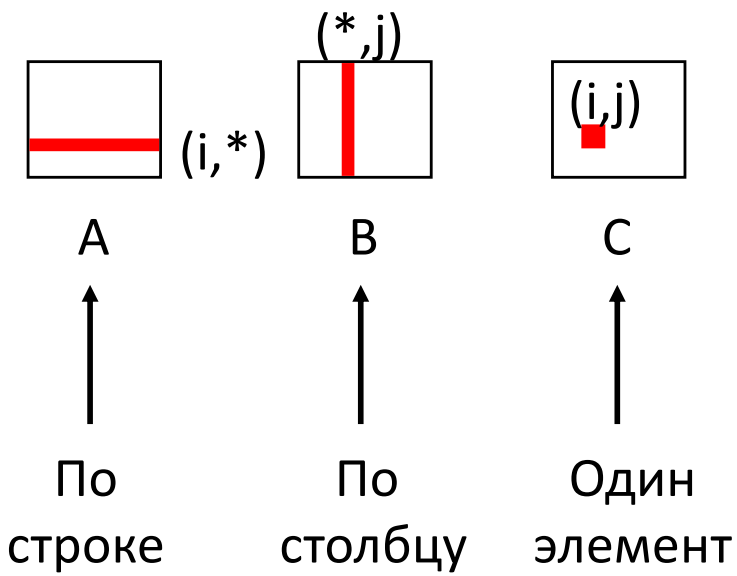
Расположение в памяти массивов Си

- **Массивы Си хранятся в памяти по строкам**
  - строки располагаются друг за другом
- **Проход по столбцам в одной строке:**
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i];
    ```
  - доступ к последовательно расположенным элементам
  - Если размер кэш-блока (B) > 4 байт, действует пространственная локальность
    - вероятность вынужденного промаха = 4 байта / B
- **Проход по строкам в одном столбце:**
  - ```
    for (i = 0; i < n; i++)
      sum += a[i][0];
    ```
  - Доступ к разнесённым в памяти элементам
  - Пространственная локальность отсутствует!
    - вероятность вынужденного промаха = 1 (т.е. 100%)

Перемножение матриц (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Внутренний цикл:



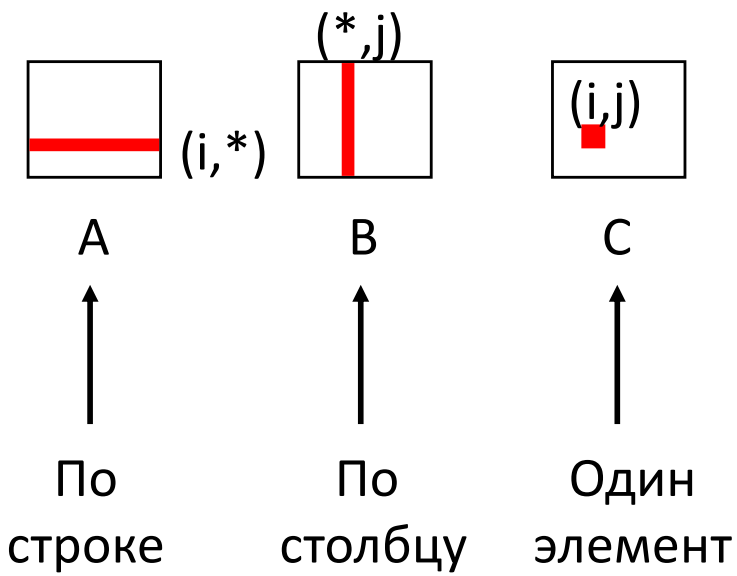
Промахов в итерации внутреннего цикла:

A	B	C
0.25	1.0	0.0

Перемножение матриц (jik)

```
/* jik */
for (j=0; i<n; i++)  {
  for (i=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Внутренний цикл:



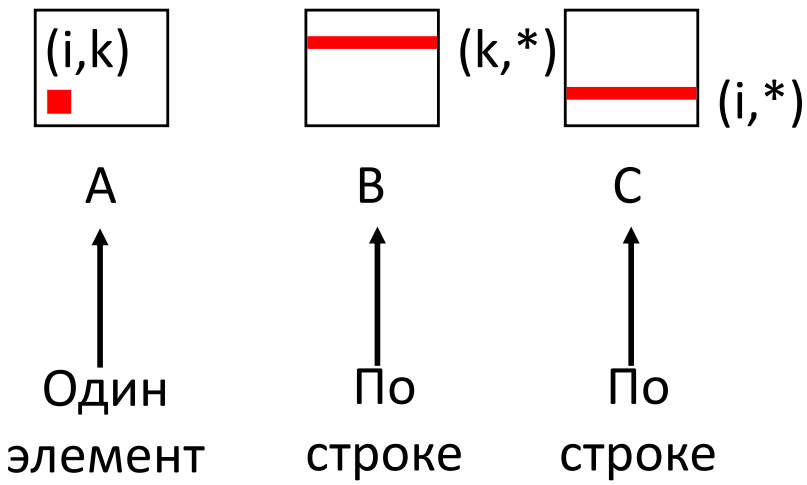
Промахов в итерации внутреннего цикла:

A	B	C
0.25	1.0	0.0

Перемножение матриц (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Внутренний цикл:



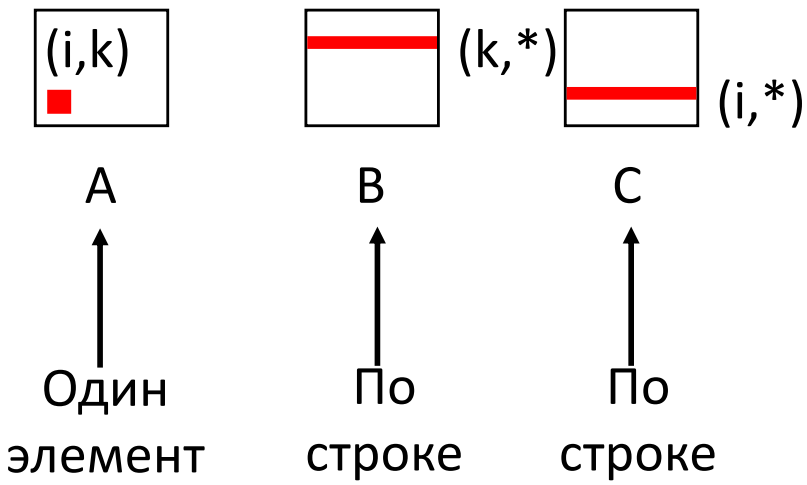
Промахов в итерации внутреннего цикла:

A	B	C
0.0	0.25	0.25

Перемножение матриц (ikj)

```
/* ikj */
for (i=0; k<n; k++) {
  for (k=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Внутренний цикл:



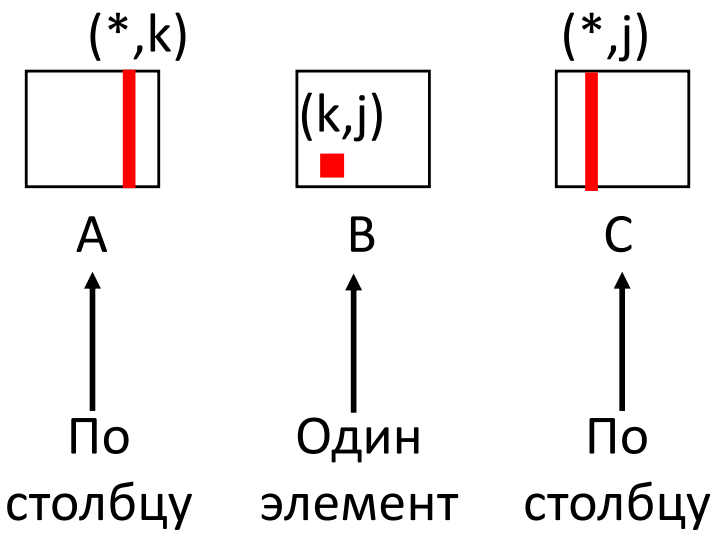
Промахов в итерации внутреннего цикла:

A	B	C
0.0	0.25	0.25

Перемножение матриц (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Внутренний цикл:



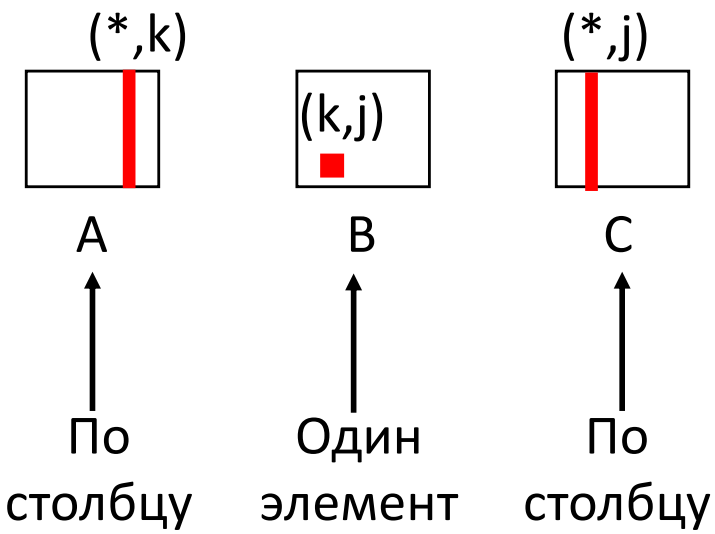
Промахов в итерации внутреннего цикла:

A	B	C
1.0	0.0	1.0

Перемножение матриц (kji)

```
/* kji */
for (k=0; j<n; j++) {
  for (j=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Внутренний цикл:



Промахов в итерации внутреннего цикла:

A	B	C
1.0	0.0	1.0

Сводка перемножений матриц

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

ijk и jik:

- 2 чтения, 0 записей
- промахов в итерации = **1.25**

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

kij и ikj:

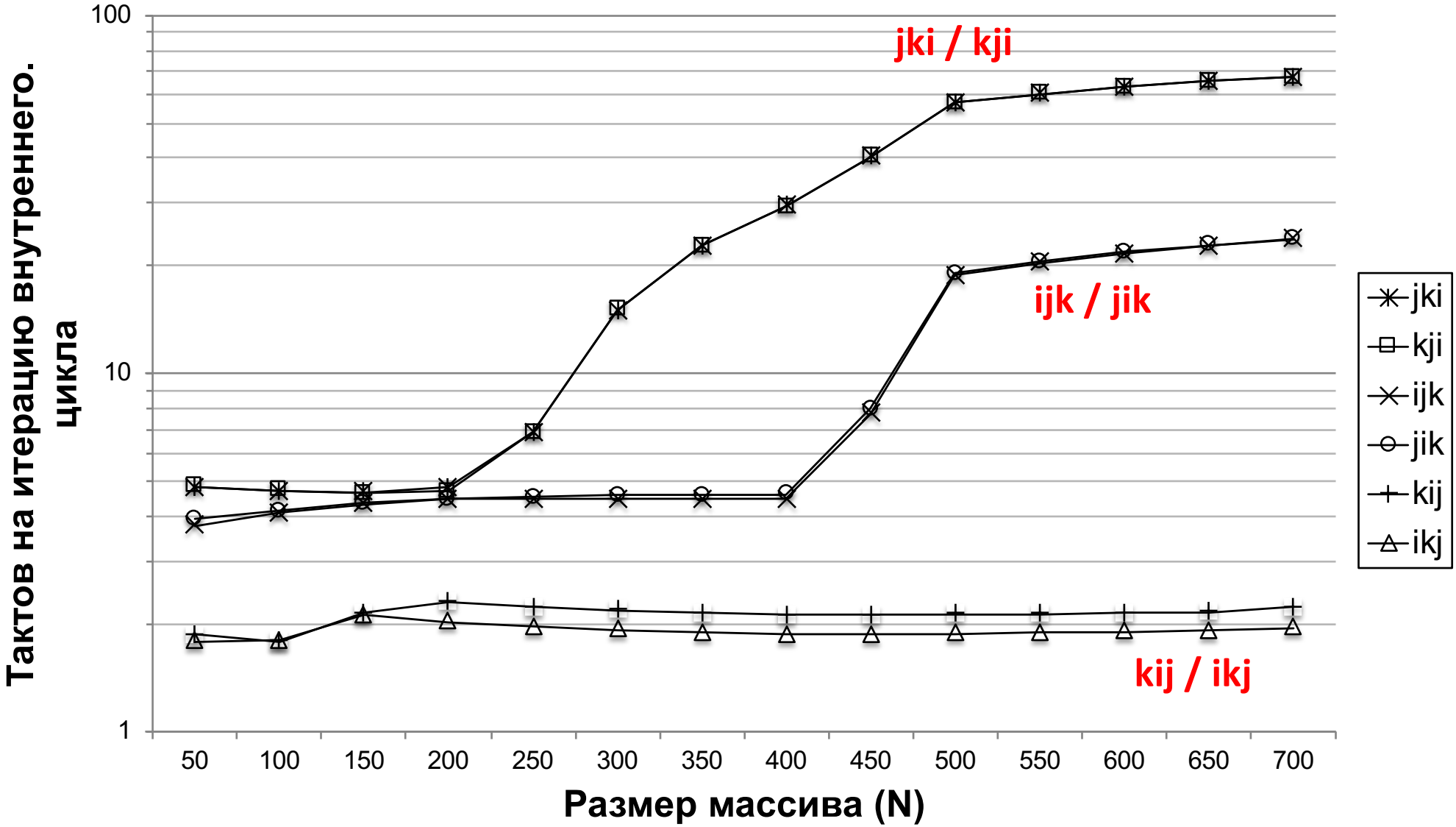
- 2 чтения, 1 запись
- промахов в итерации = **0.5**

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

jki и kji:

- 2 чтения, 1 запись
- промахов в итерации = **2.0**

Скорость перемножения матриц на Core i7

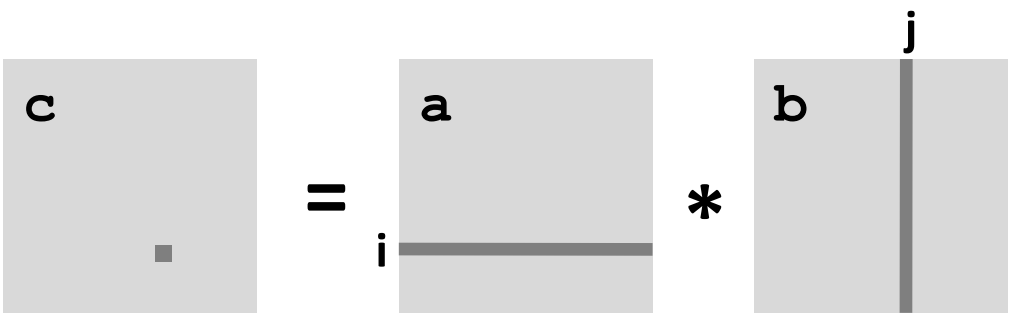


Кеширование памятей

- **Организация и работа кэша**
- **Влияние кэша на быстродействие памяти**
  - Диаграмма быстродействия памяти
  - Реорганизация циклов улучшает пространственную локальность
  - Блокирование улучшает временнУю локальность

Пример: Перемножение матриц

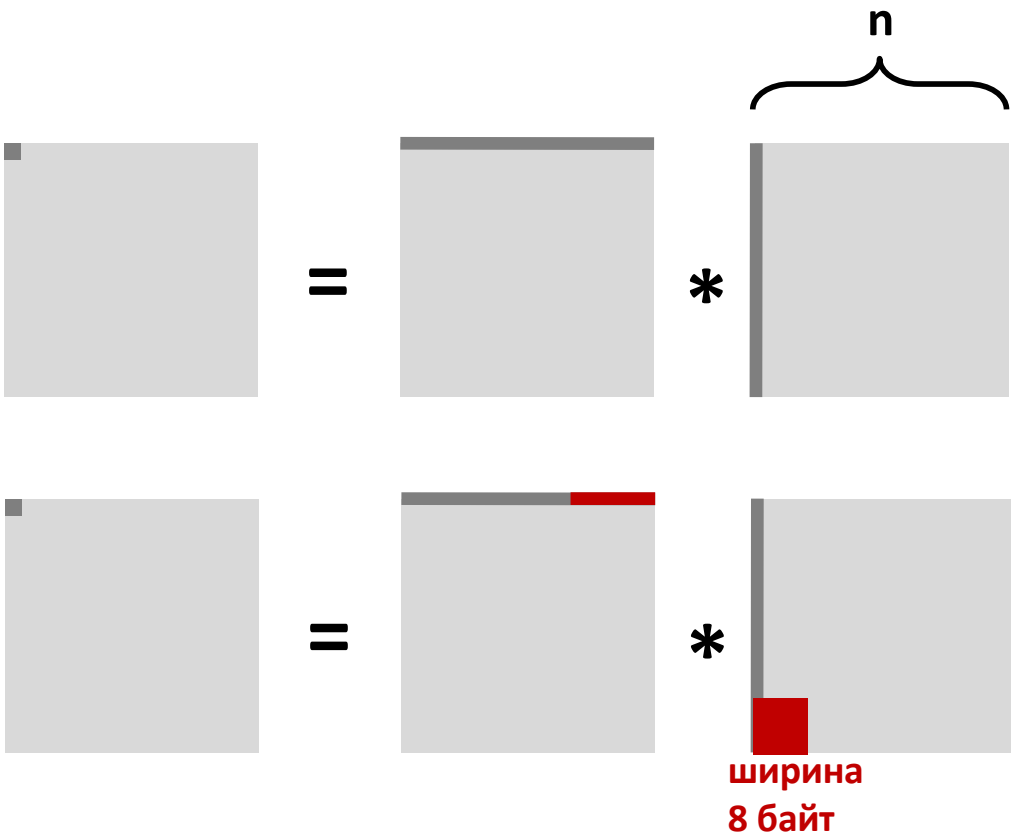
```
c = (double *) calloc(sizeof(double), n*n);

/* Перемножение a и b - матриц размерами n x n */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
       for (j = 0; j < n; j++)
             for (k = 0; k < n; k++)
          c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```



Анализ промахов кеша

- **Допустим:**
  - Элементы матриц – double
  - Блок кэша = 8 double (64 байта)
  - Размер кэша C << n (много меньше n)

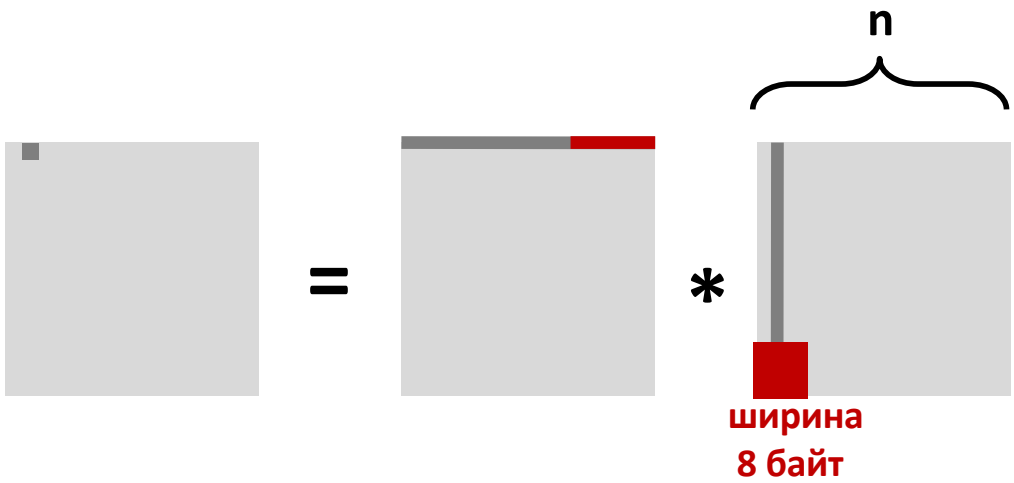
- **Первая итерация:**
  - n/8 + n = 9n/8 промахов
  - После итерации в кэше: (схематично)



Анализ промахов кеша

- **Допустим:**
  - Элементы матриц – double
  - Блок кэша = 8 double (64 байта)
  - Размер кэша C << n (много меньше n)

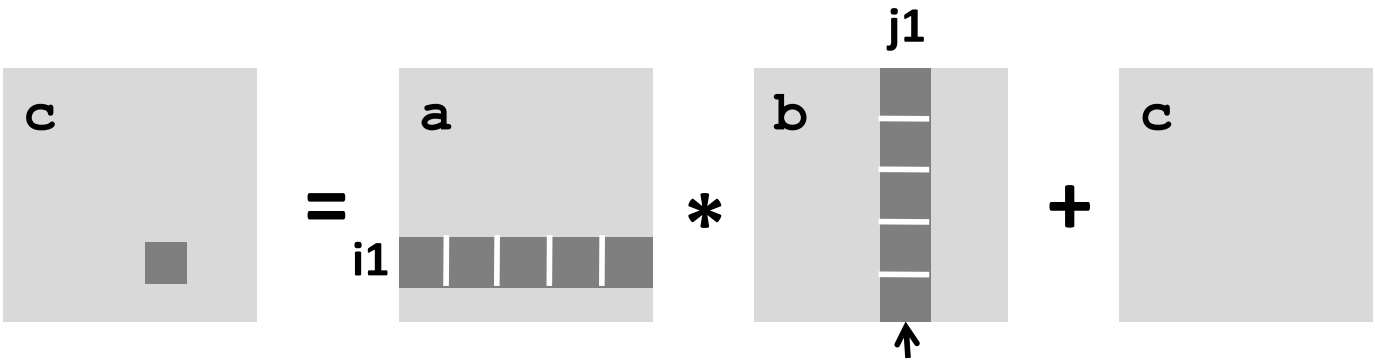
- **Вторая итерация:**
  - Опять:
    n/8 + n = 9n/8 промахов



- **Всего промахов:**
  - $9n/8 * n^2 = (9/8) * n^3$

Блочное перемножение матриц

```
c = (double *) calloc(sizeof(double), n*n);

/* Перемножение a и b - матриц размерами n x n */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* Перемножение мини-матриц размерами B x B */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

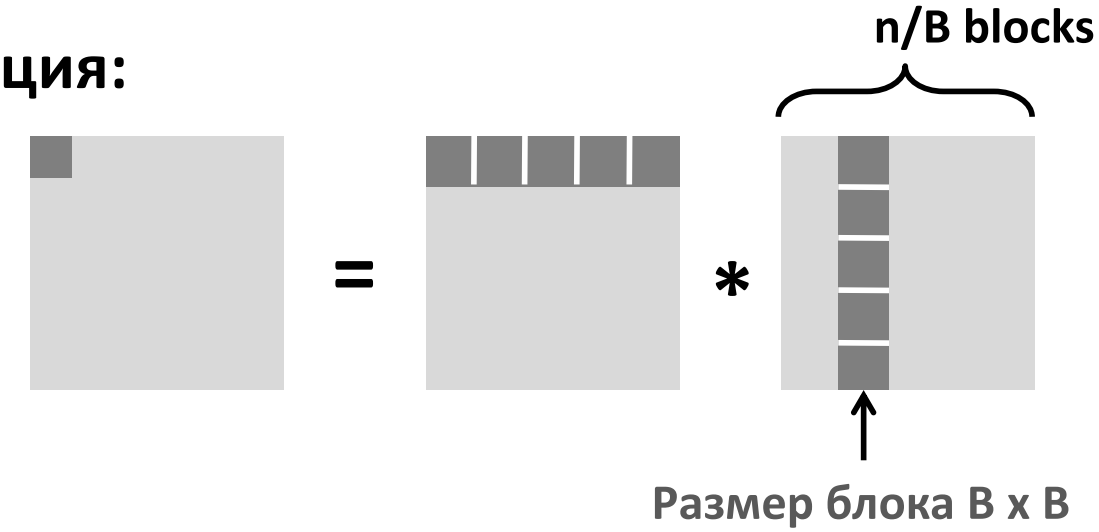


Размер блока B x B

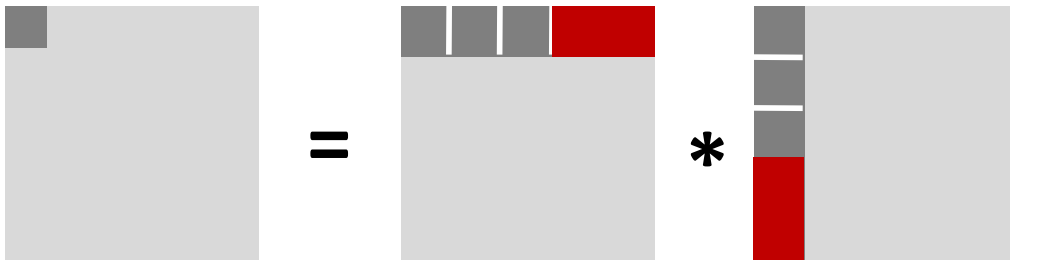
Анализ промахов кеша

- **Допустим:**
  - Блок кэша = 8 doubles
  - Размер кэша C << n (много меньше n)
  - Четыре блока ■ умещаются в кеш : $4B^2 < C$

- **Первая (блочная) итерация:**
  - $B^2/8$ промахов в блоке
  - $2n/B * B^2/8 = nB/4$
    (не считая матрицу c)



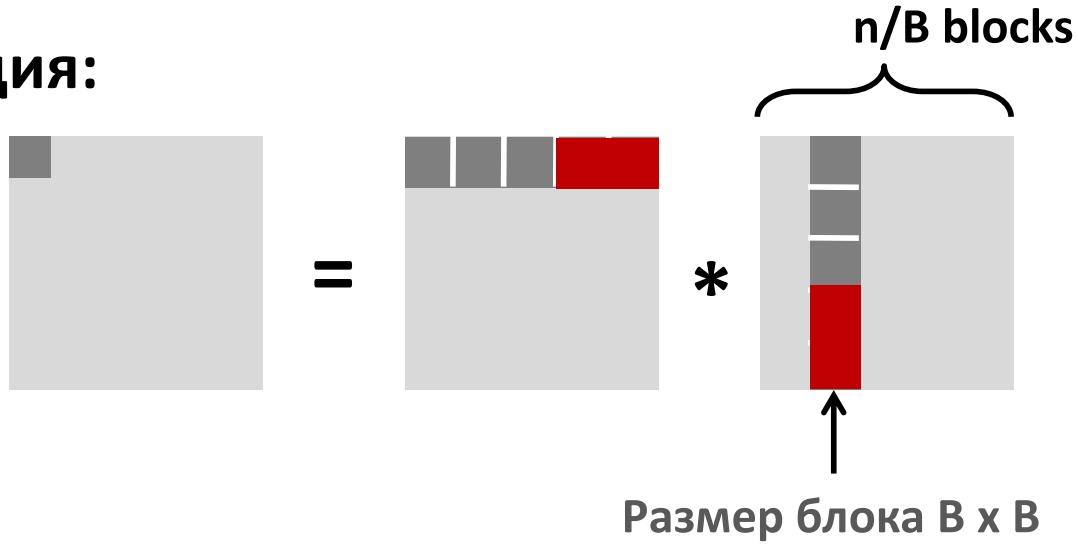
  - То, что осталось в кэше
    (схематично)



Анализ промахов кеша

- **Допустим:**
  - Блок кэша = 8 doubles
  - Размер кэша C << n (много меньше n)
  - Четыре блока ■ умещаются в кеш : $4B^2 < C$

- **Вторая (блочная) итерация:**
  - Как и на первой итерации
  - $2n/B * B^2/8 = nB/4$



- **Всего промахов:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

Сводка блокирования

- **Без блокирования: $(9/8) * n^3$**
- **С блокированием: $1/(4B) * n^3$**

- **Предполагается наибольший размер блока B, ограниченный как $4B^2 < C$!**

- **Причины существенной разницы:**
  - Матрице присуща временная локальность:
    - $3n^2$ входных данных, $2n^3$ операций
    - Каждый элемент массивов используется O(n) раз!
  - При условии, что программа написана правильно

Сводка кеширования

- **Кеш-памяти могут сильно влиять на быстродействие**

- **В можете использовать это в своих программах**

  - Фокусируйтесь на внутренних циклах, где возникает большая часть вычислений и обращений в память.

  - Старайтесь улучшать пространственную локальность, обращаясь к данным с минимальным шагом 1.

  - Старайтесь улучшать временнУю локальность, максимально используя повторно данные, однажды считанные из основной памяти.