

Иерархия памяти

Основы информатики.

Компьютерные основы программирования

goo.gl/X7evF

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

goo.gl/TDDVV

Лекция 10, 20 апреля, 2015

Лектор:

Дмитрий Северов, кафедра информатики 608 КПМ

dseverov@mail.mipt.ru



Иерархия памяти

- Кэширование в иерархии памяти
- Организация и работа кэша
- Влияние кэша на быстродействие памяти
 - Диаграмма быстродействия памяти
 - Реорганизация циклов улучшает пространственную локальность
 - Блокирование улучшает временную локальность

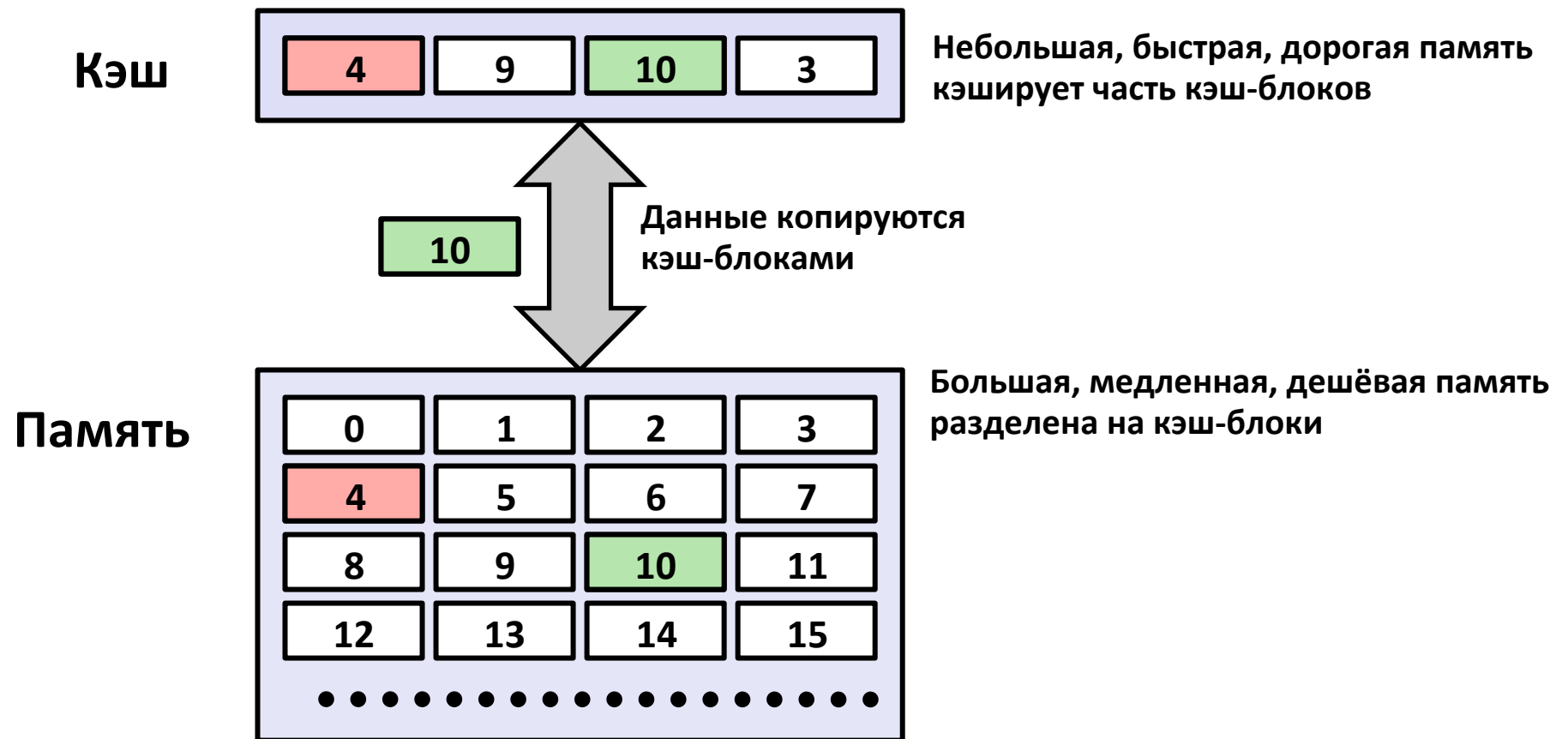
Пример иерархии хранения данных



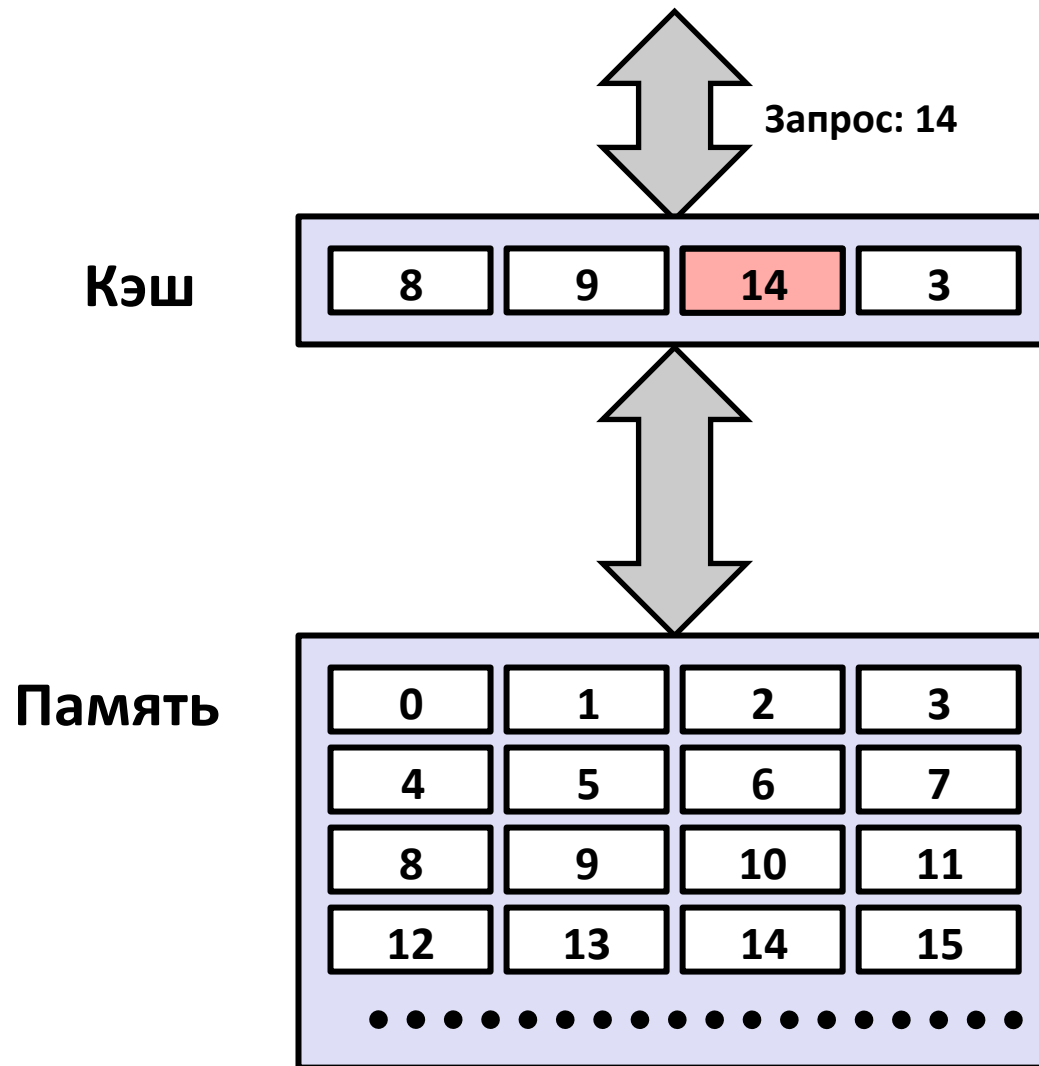
Кеш

- **Кеш(Cache):** Устройство хранения меньшей ёмкости и большего быстродействия, действующее как вспомогательное для доступа к части данных более крупного и медленного устройства.
- **Фундаментальная идея иерархии:**
 - Устройство уровня k служит кешем для устройства уровня $k+1$.
- **Почему работает иерархия хранения данных?**
 - Благодаря локальности программы к данным уровня k доступ происходит чаще, чем к данным уровня $k+1$.
 - Значит уровень хранения $k+1$ может быть медленнее, а значит больше и дешевле (на единицу хранения)
- **Идеально:** Иерархия предоставляет большой объём хранения по цене экономичного нижнего уровня с скоростью доступа верхнего уровня для программ.

Кэш в общем



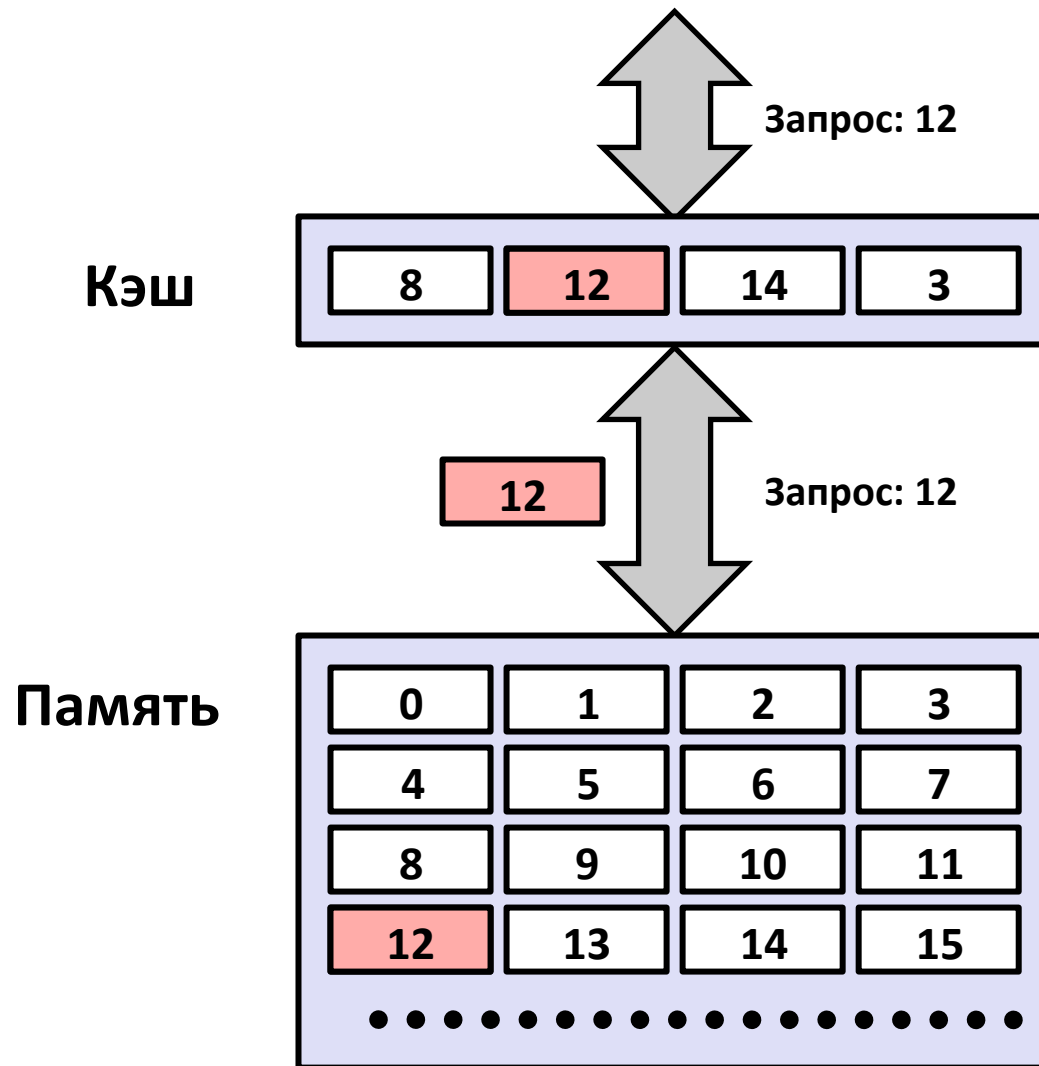
Кэш в общем: Попадание



Нужен блок данных b

*Блок b в кэше есть:
Попадание!*

Кэш в общем: Промех



Нужен блок данных b

Блока b в кэше нет:
Промех!

**Блок b поступает
из памяти**

Блок b помещается в кеше

- **Политика размещения:** определяет, где b в кэше

- **Политика замещения:**
определяет удаляемый блок (жертву)

Кэш в общем: типы промахов

■ Холодный промах

- Кэш пуст. Например при старте.

■ Конфликтный промах

- Большинство кэшей ограничивает (вплоть до 1) количество позиций размещения в кэше для каждого кэш-блока памяти.
 - Пример: блок i в памяти должен размещаться в $(i \bmod 4)$ позиции кэша.
- Конфликтный промах, если в кэше есть место, но несколько блоков памяти претендуют на одно место в кэше.
 - Пример: обращения к блокам 0, 8, 0, 8, 0, 8, ... промахируются всегда.

■ Промач ёмкости

- Множество используемых кэш-блоков, рабочий набор (**working set**) превышает размер кэша.

Пример кэширования в иерархии

Тип кэша	Что кэшируем?	Где кэшируем?	Задержка (циклов)	Кто управляет?
Регистры	Слова 4-8 байт	Ядро ЦП	0	Компилятор
TLB	Трансляция адресов	TLB на кристалле ЦП	0	Аппаратура
Кэш уровня 1	Блок 64 байта	На кристалле ЦП	1	Аппаратура
Кэш уровня 2	Блок 64 байта	На кристалле ЦП	10	Аппаратура
Виртуальная пам.	Страница 4-КБ	Основная память	100	Аппарат.+ОС
Кэш-буфер	Части файлов	Основная память	100	ОС
Кэш диска	Сектора дисков	Контроллер диска	100,000	ПО в диске
Кэш сети	Части файлов	Местный диск	10,000,000	AFS/NFS клиент
Кэш браузера	Веб-страницы	Местный диск	10,000,000	Веб браузер
Веб-кэш	Веб-страницы	Удалённый сервер	1,000,000,000	Веб прокси-сервер

Иерархия памяти

- Кэширование в иерархии памяти

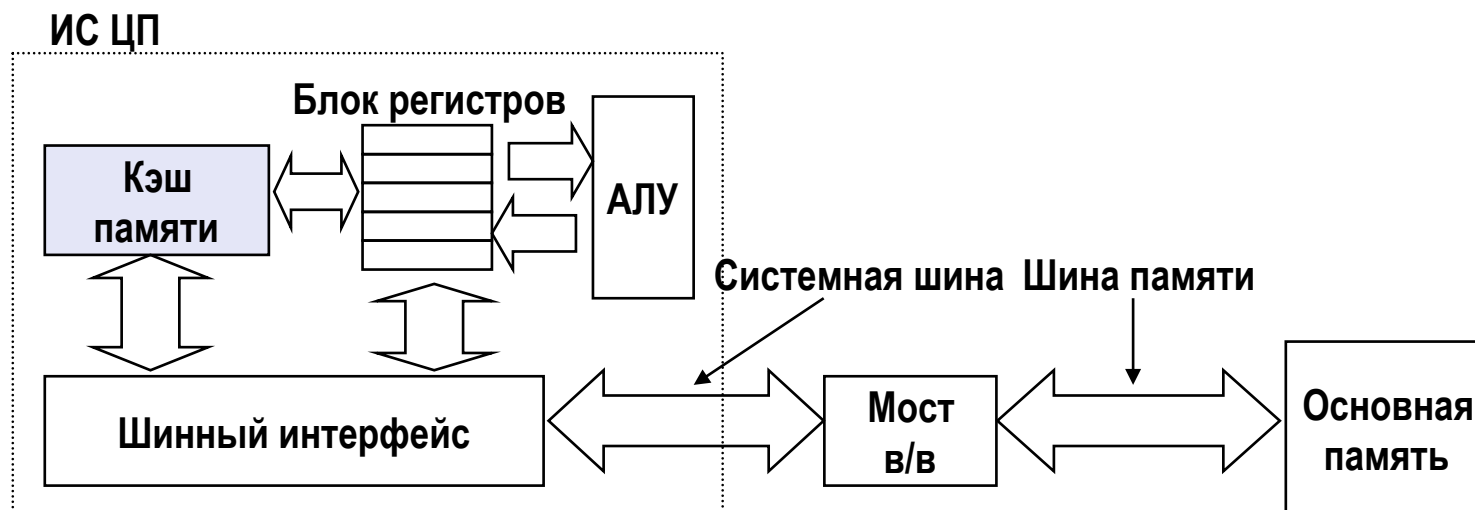
- Организация и работа кэша

- Влияние кэша на быстродействие памяти

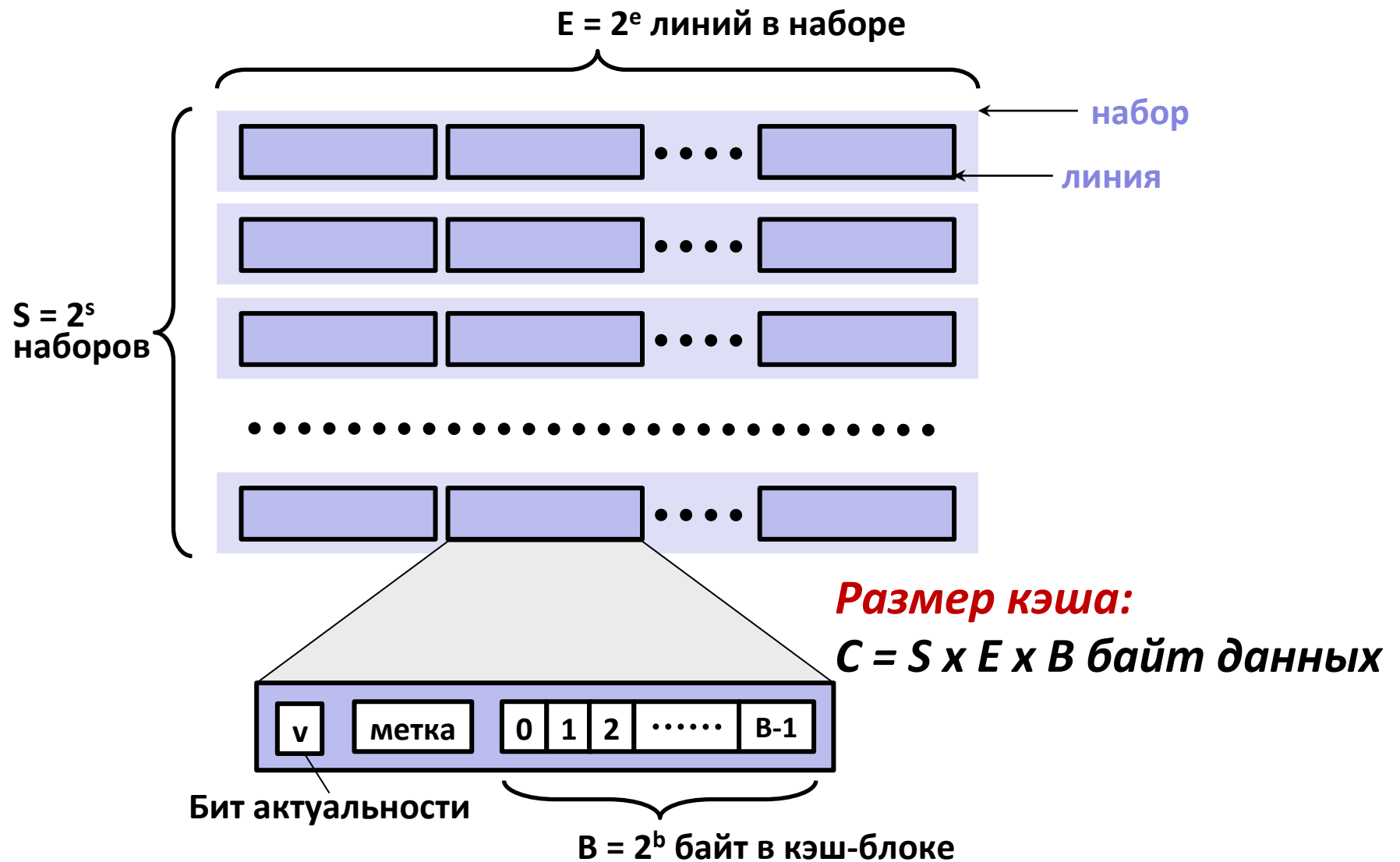
- Диаграмма быстродействия памяти
- Реорганизация циклов улучшает пространственную локальность
- Блокирование улучшает временную локальность

Кэш памяти

- **Кэш памяти** – небольшая, быстрая память на основе SRAM, автоматически управляемая аппаратурой.
 - Хранит часто используемые блоки основной памяти
- Ядро ЦП сначала обращается за данными в кэши, а затем в память. Если необходимо.
- Типичная структура системы:

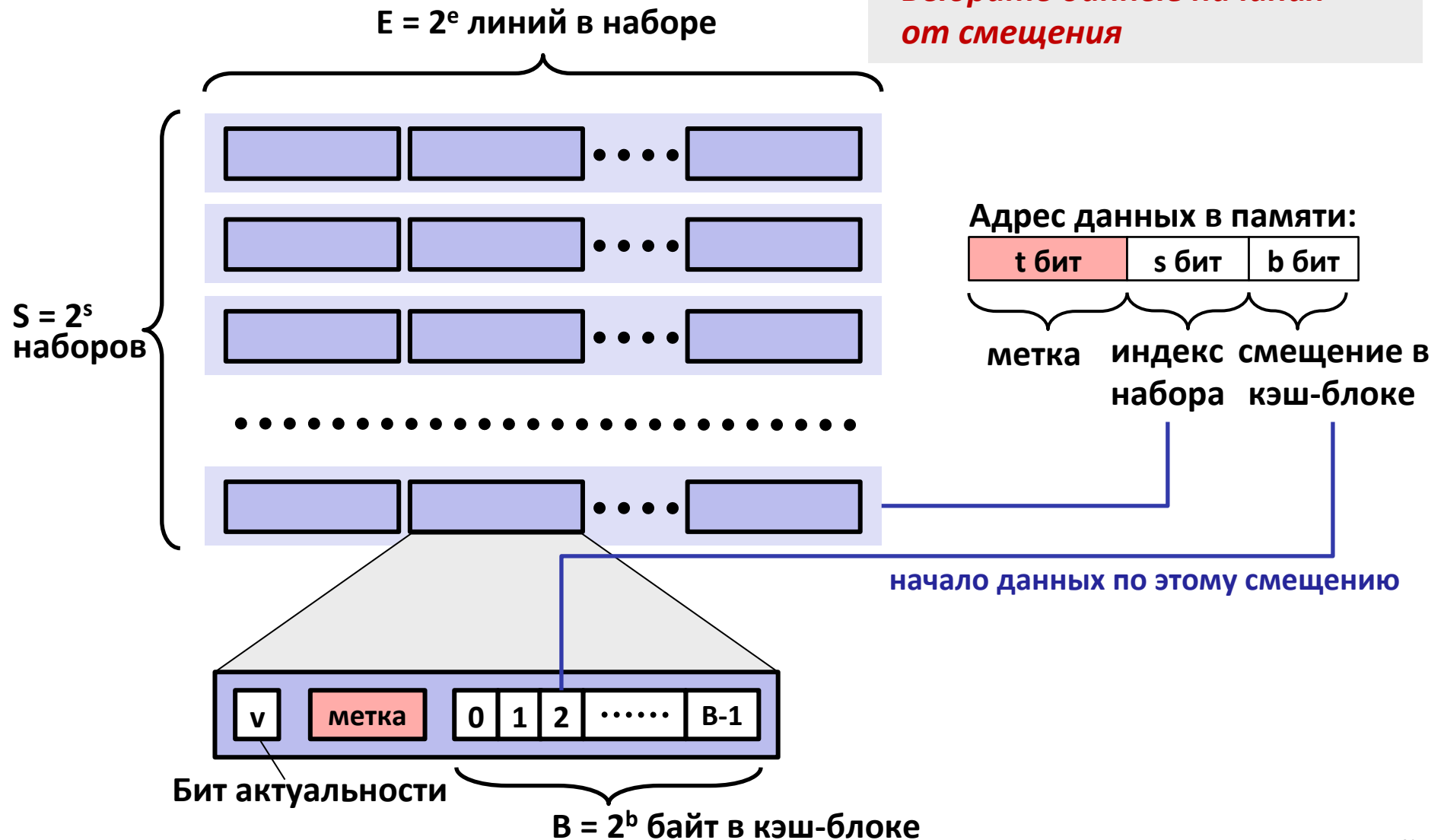


Общая организация кэша (S, E, V)



Чтение кэша

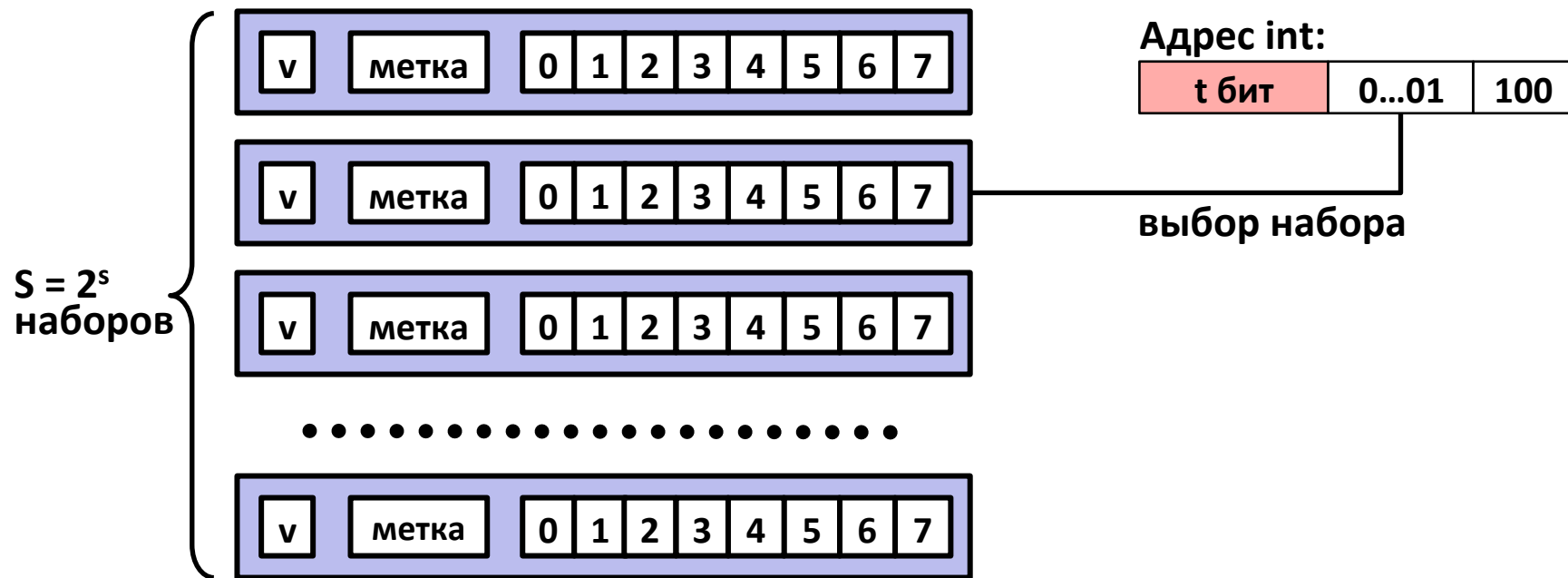
- **Выбрать набор**
- **Проверить на совпадение метки линий в наборе**
- **Есть + актуальна: попадание!**
- **Выбрать данные начиная от смещения**



Пример: Кэш прямого отображения ($E = 1$)

Прямое отображение: одна линия в наборе

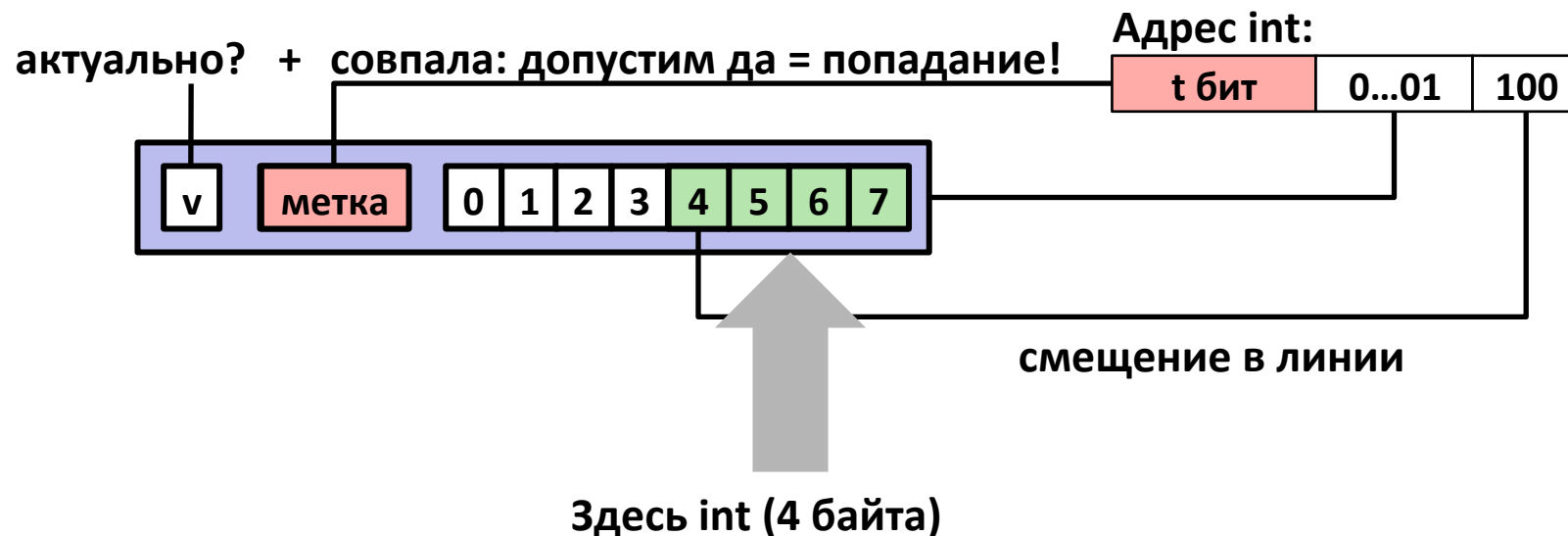
Допустим: размер кэш-блока - 8 байт



Пример: Кэш прямого отображения ($E = 1$)

Прямое отображение: одна линия в наборе

Допустим: размер кэш-блока - 8 байт



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, V=2 байта в кэш-блоке,
S=4 набора, E=1 линия в наборе

Трассировка адресов (чтения, по одному байту):

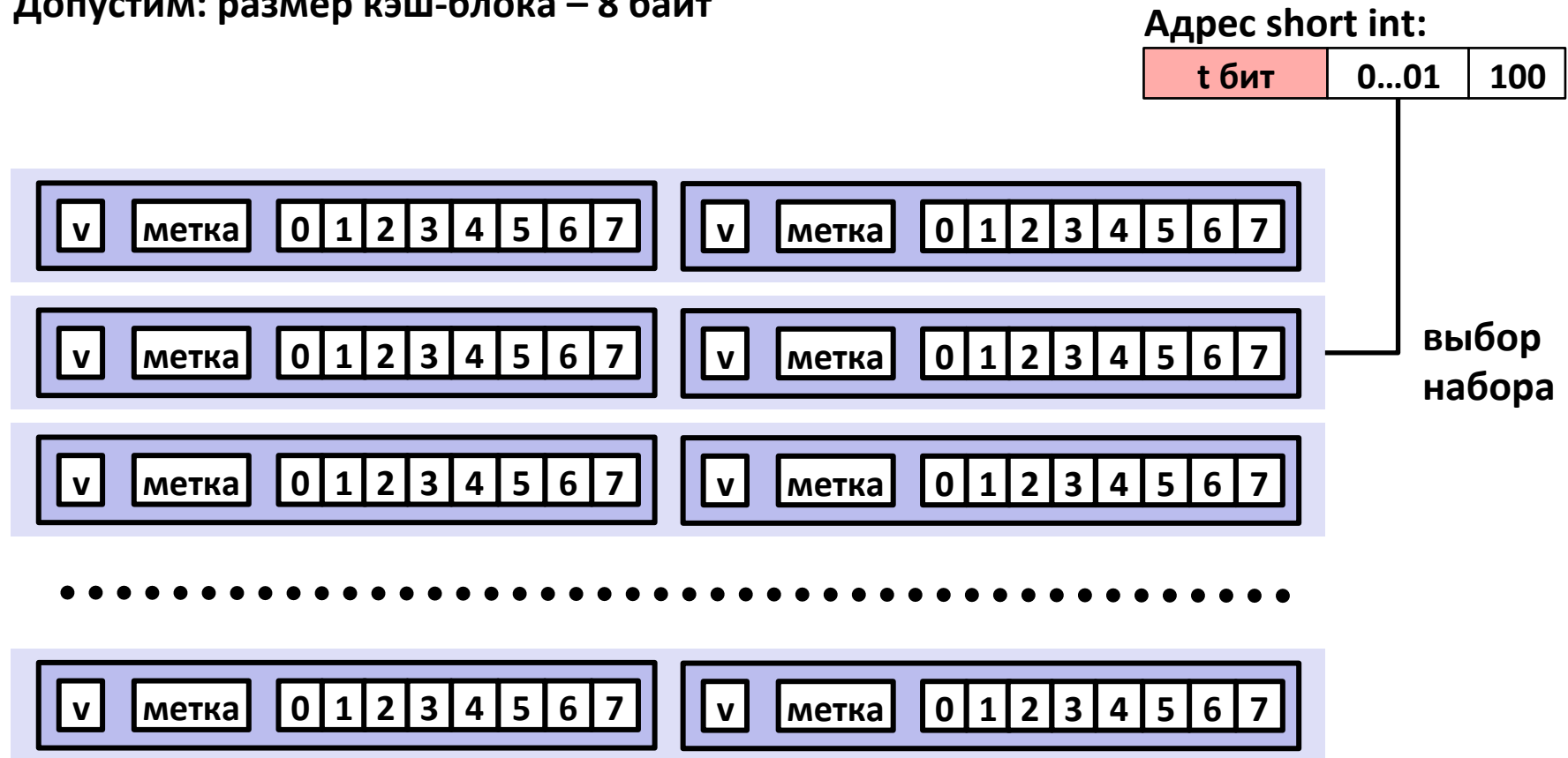
0	[0000] ₂ ,	промах
1	[0001] ₂ ,	попадание
7	[0111] ₂ ,	промах
8	[1000] ₂ ,	промах
0	[0000] ₂	промах

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

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

$E = 2$: Две линии в наборе

Допустим: размер кэш-блока – 8 байт



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

E = 2: Две линии в наборе

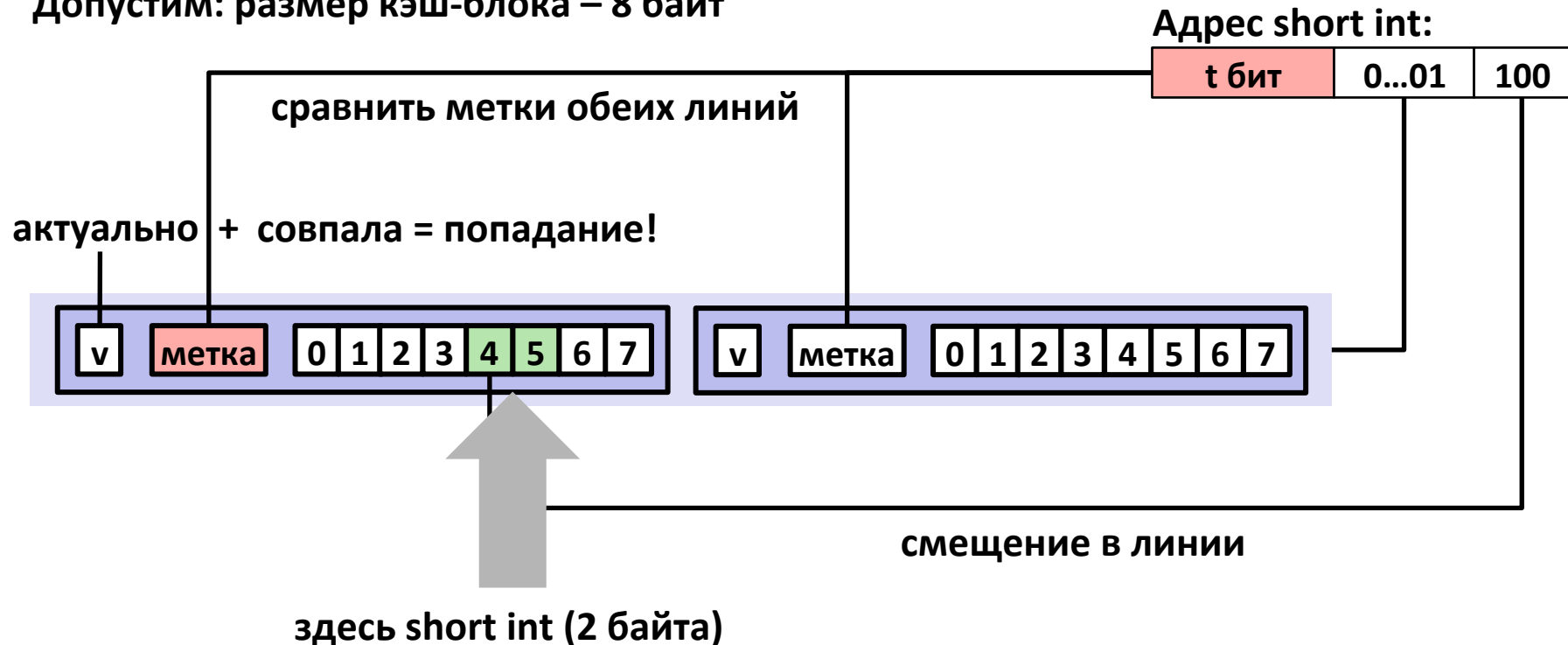
Допустим: размер кэш-блока – 8 байт



Е-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 <u>00</u> ₂],	промах
1	[00 <u>01</u> ₂],	попадание
7	[01 <u>11</u> ₂],	промах
8	[10 <u>00</u> ₂],	промах
0	[00 <u>00</u> 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?

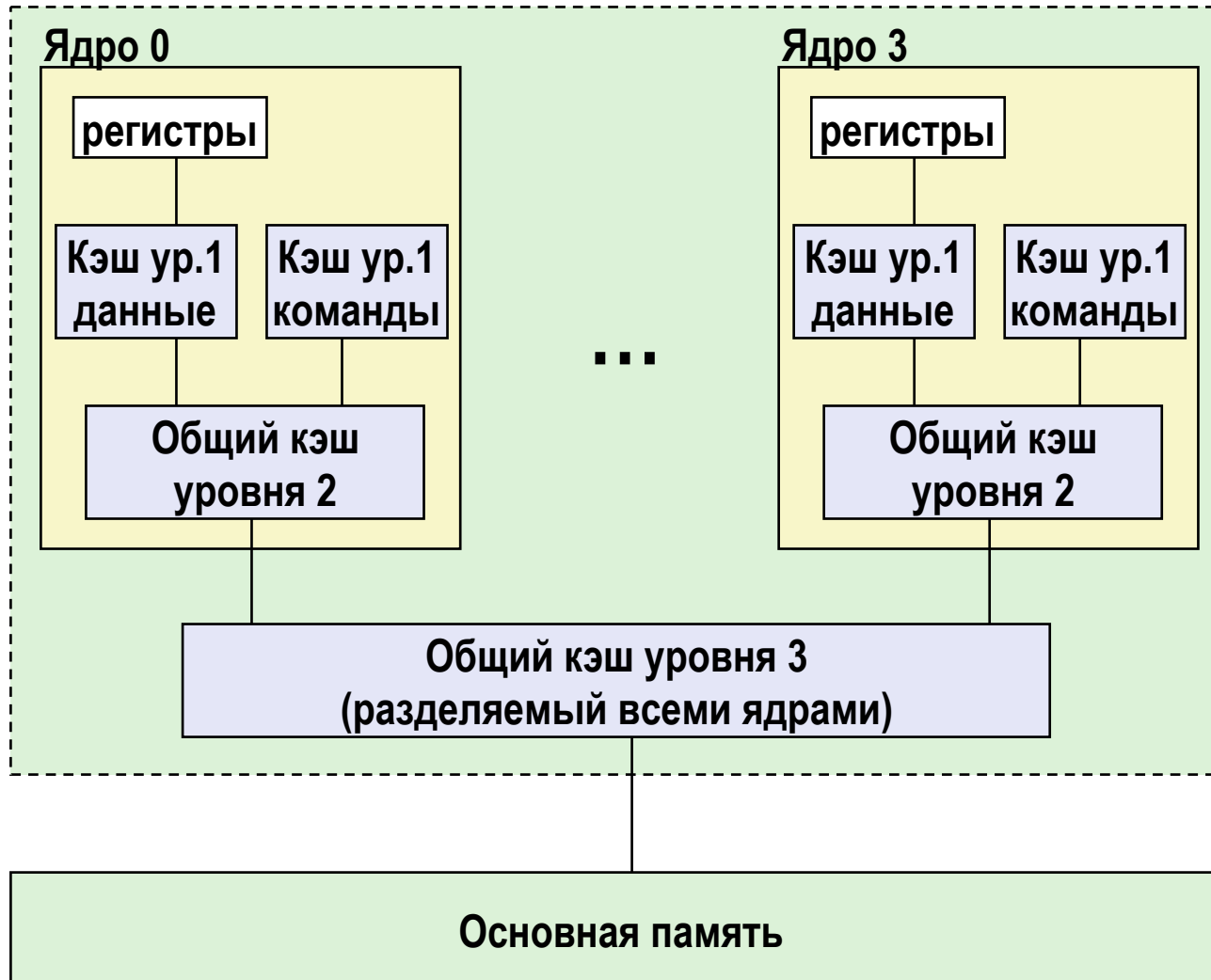
- **Write-allocate** (загрузка в кэш, изменение линии кэша)
 - Хорошо если ожидаются ещё записи
- **No-write-allocate** (запись непосредственно в память)

■ Типичные политики

- Write-through + No-write-allocate
- Write-back + Write-allocate

Иерархия кэшей Intel Core i7

Интегральная схема процессора



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) = $1 - \text{вероятность попаданий}$
- 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: $1 \text{ такт} + 0.03 * 100 \text{ тактов} = 4 \text{ такта}$
99% попаданий: $1 \text{ такт} + 0.01 * 100 \text{ тактов} = 2 \text{ такта}$
- **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

Иерархия памяти

- Кэширование в иерархии памяти
- Организация и работа кэша
- Влияние кэша на быстродействие памяти
 - Диаграмма быстродействия памяти
 - Реорганизация циклов улучшает пространственную локальность
 - Блокирование улучшает временную локальность

Диаграмма быстродействия памяти

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

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

- **Ссылки**
 - <http://www.cs.inf.ethz.ch/cops/ECT/>
 - <http://www.cs.cmu.edu/afs/cs/academic/class/15213-f05/code/mem/mountain/>

Измерительная функция для диаграммы

```
/* Измерительная функция */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* Блокировка оптимизации */
}

/* Запуск test(elems, stride) и возврат пропускной способности (МБ/сек) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride); /* предзаполнение кэша */
    cycles = fcyc2(test, elems, stride, 0); /* вызов test(elems, stride) */
    return (size / stride) / (cycles / Mhz); /* преобраз. циклы в МБ/сек */
}
```

Диаграмма быстродействия памяти

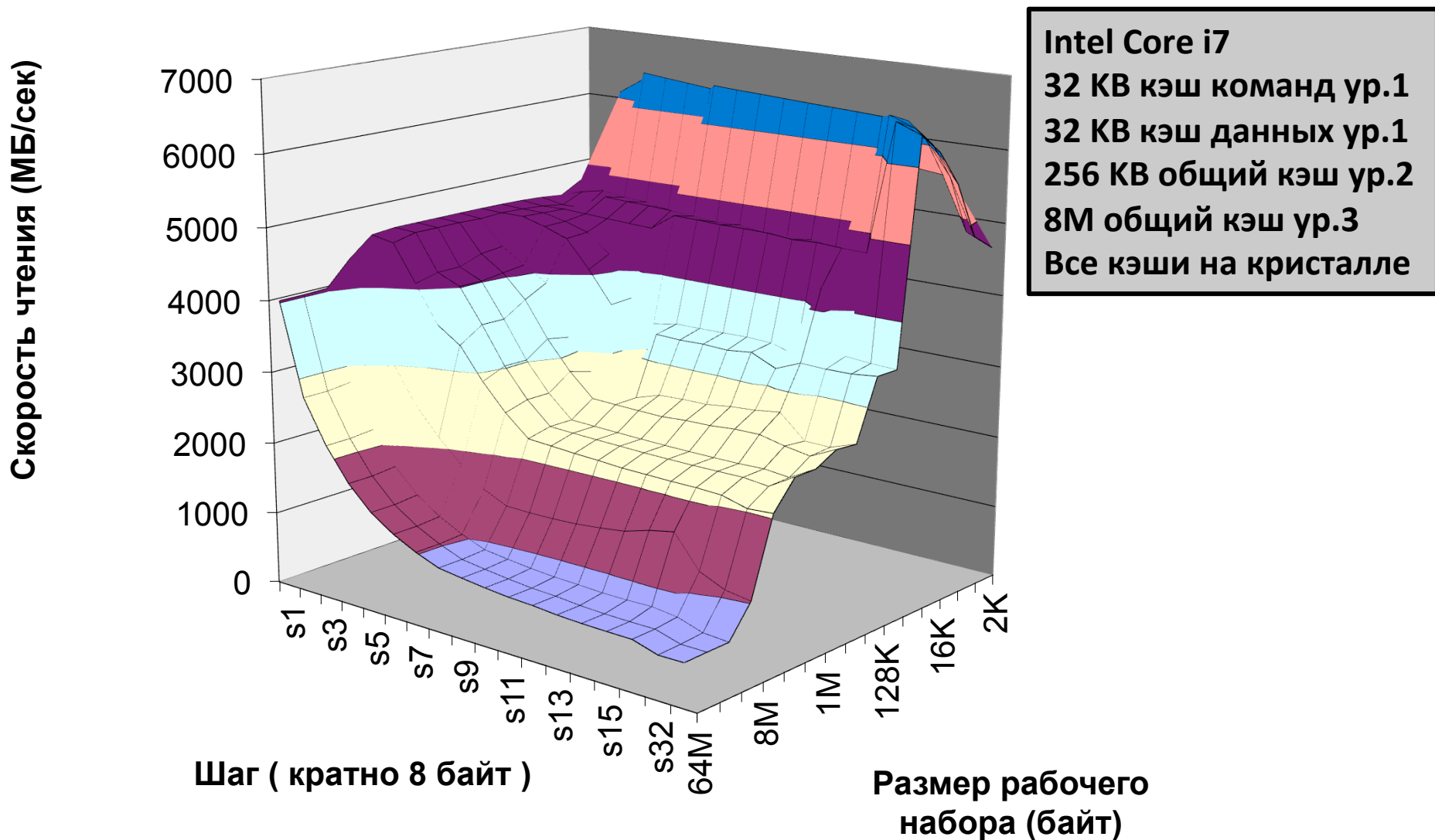


Диаграмма быстродействия памяти

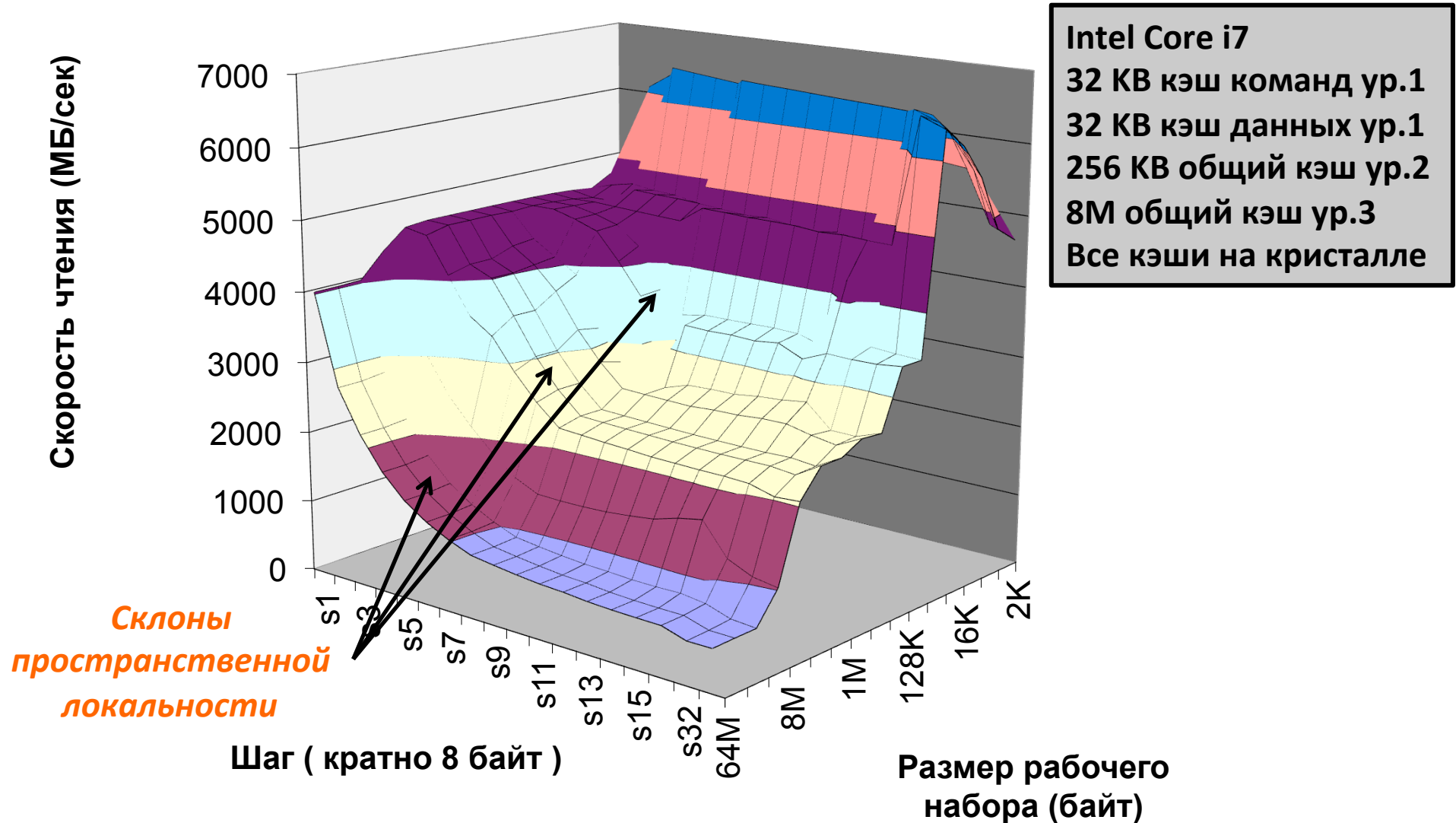
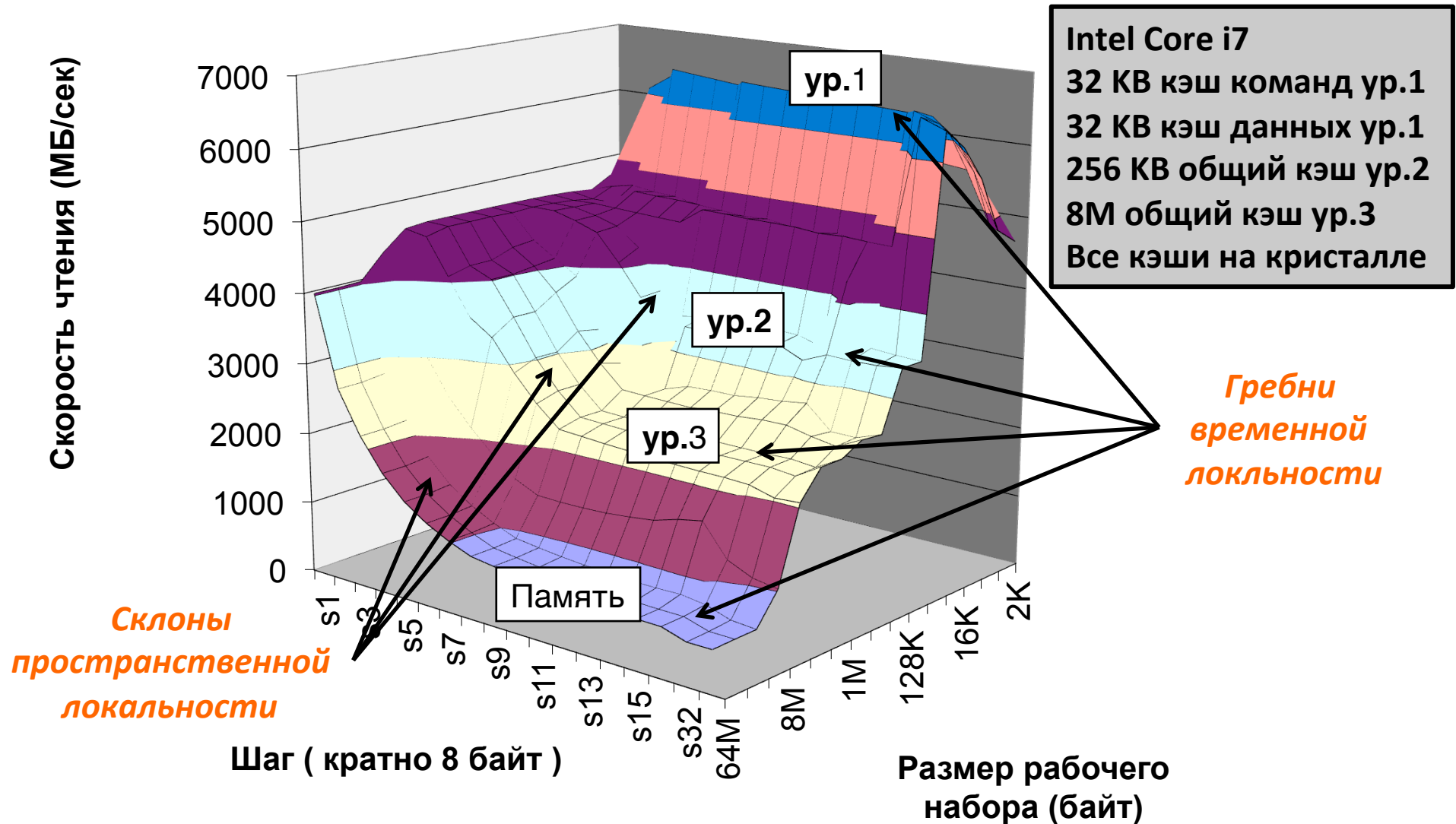


Диаграмма быстродействия памяти



Иерархия памяти

- Кэширование в иерархии памяти
- Организация и работа кэша
- Влияние кэша на быстродействие памяти
 - Диаграмма быстродействия памяти
 - Реорганизация циклов улучшает пространственную локальность
 - Блокирование улучшает временную локальность

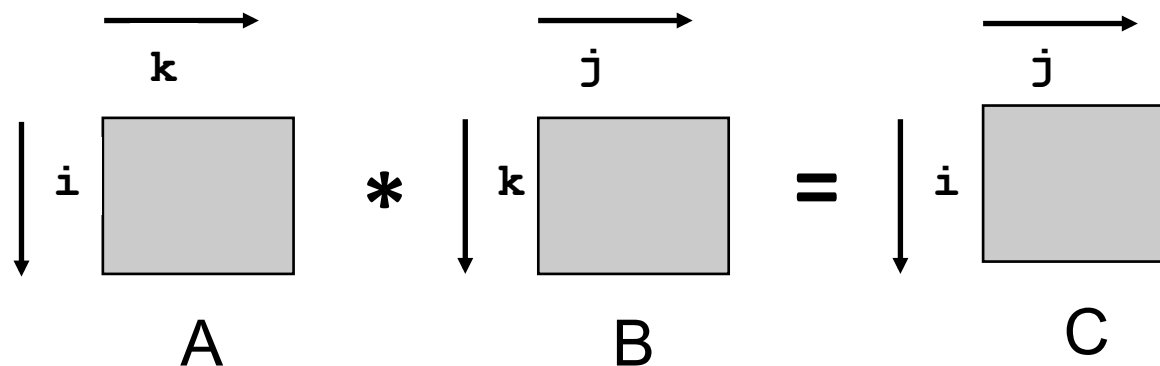
Анализ вероятности промаха для матричного умножения

■ 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



Пример перемножения матриц

■ Описание:

- Перемножение матриц $N \times N$
- 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*

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 =  $4 \text{ байта} / B$

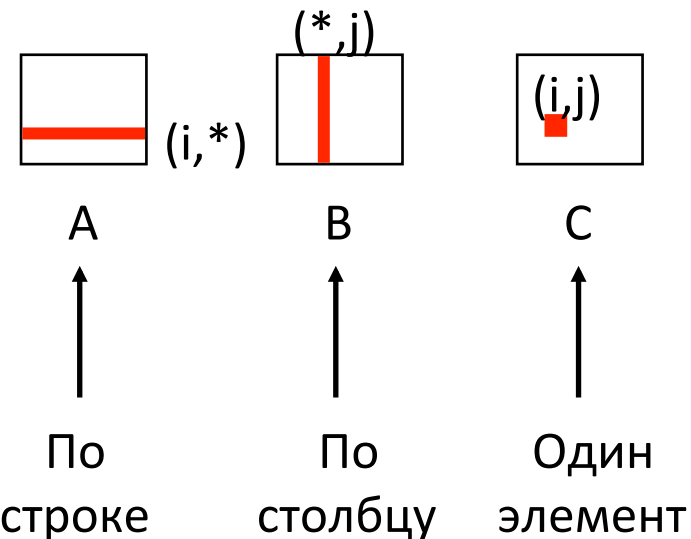
- **Проход по строкам в одном столбце:**

- ```
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:



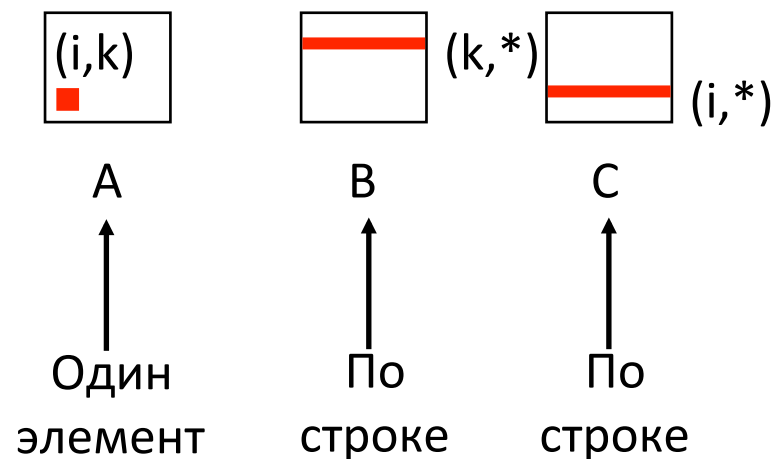
Промехов в итерации внутреннего цикла:

<u>A</u>	<u>B</u>	<u>C</u>
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:



Прوماхов в итерации внутреннего цикла:

A
0.0

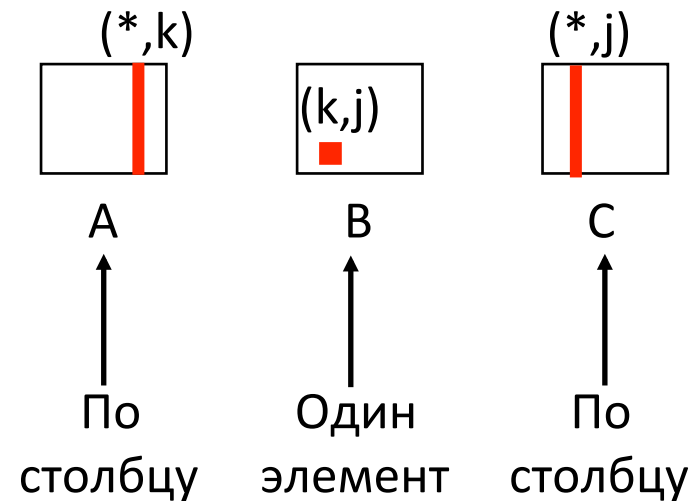
B
0.25

C
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:



Промехов в итерации внутреннего цикла:

A
1.0

B
0.0

C
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:

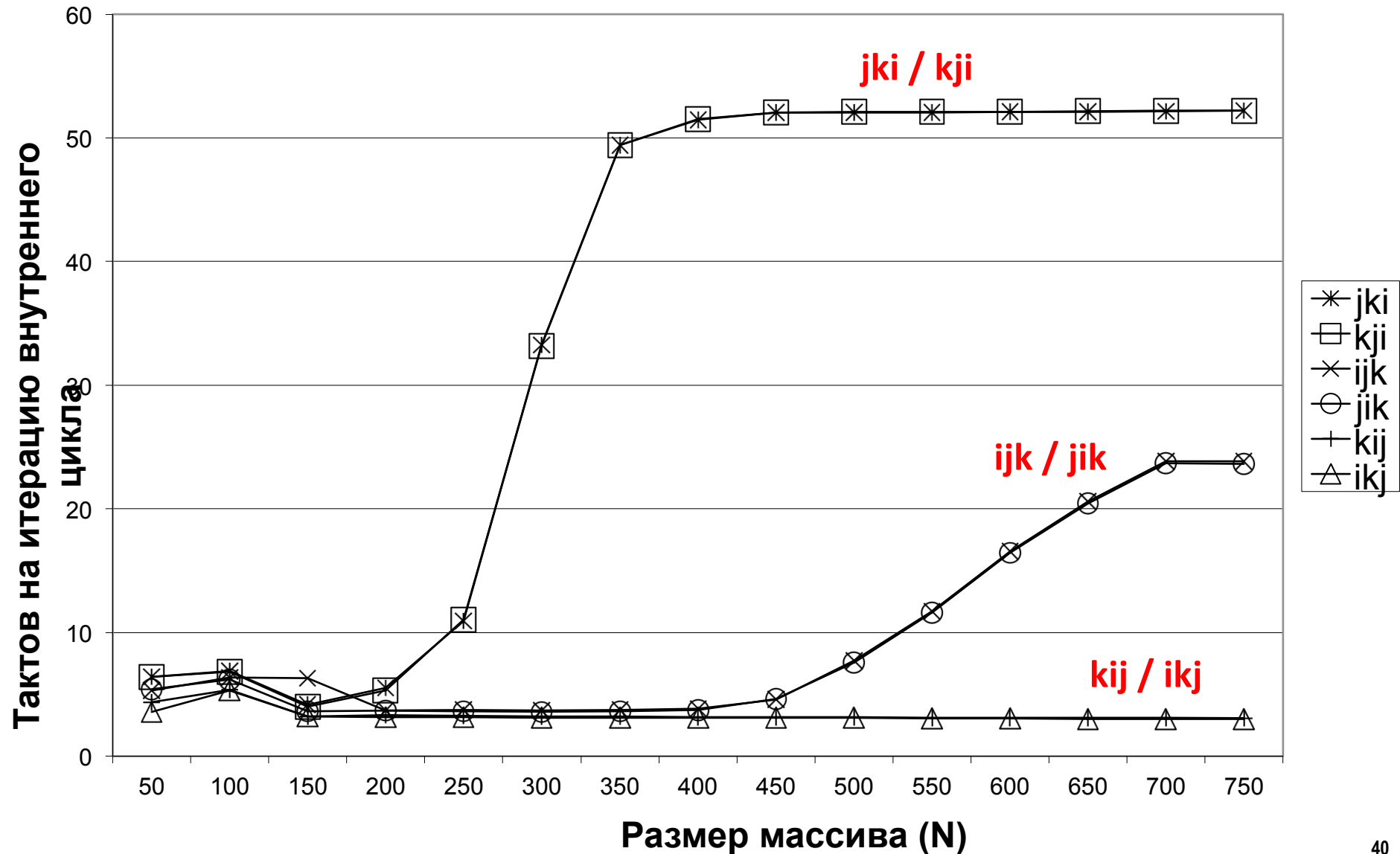
- 2 чтения, 1 запись
- промахов в итерации = **0.5**

```
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

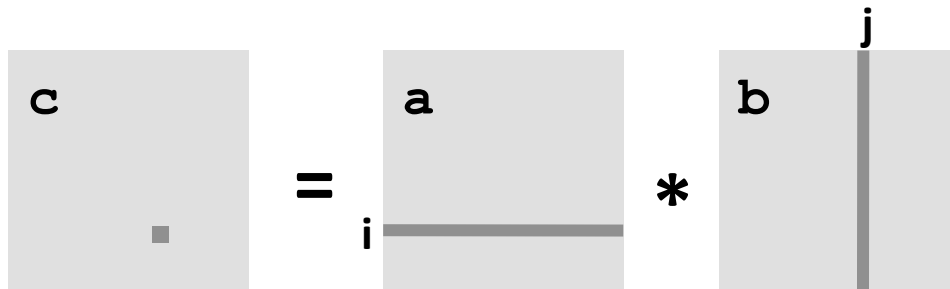


Иерархия памяти

- Кэширование в иерархии памяти
- 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

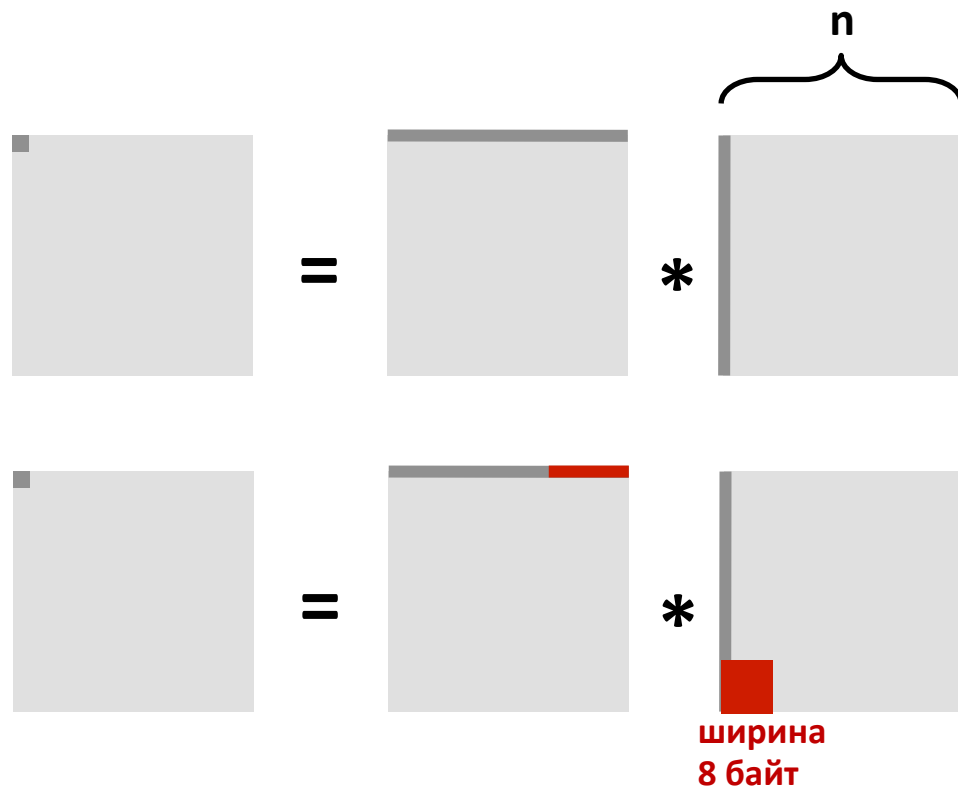
■ Допустим:

- Элементы матриц – double
- Блок кэша = 8 double (64 байта)
- Размер кэша $C \ll n$ (много меньше n)

■ Первая итерация:

- $n/8 + n = 9n/8$ промахов

- То, что **в кэше**:
(схематично)



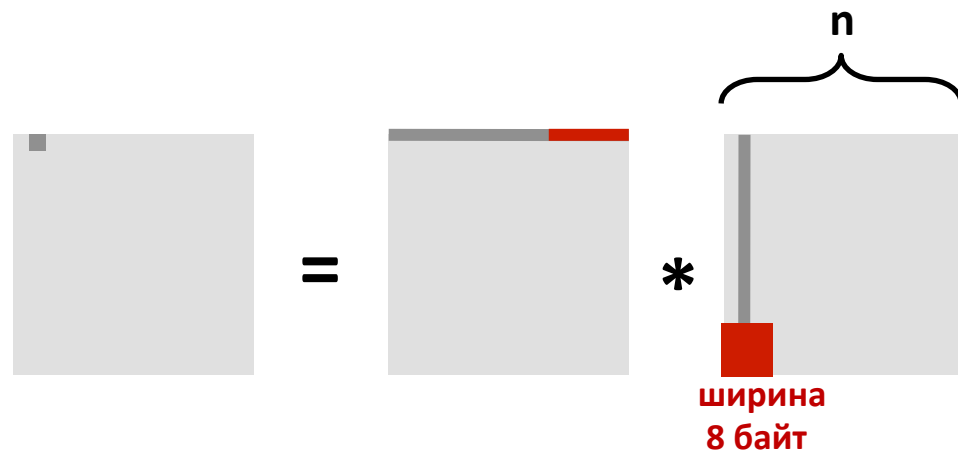
Анализ промахов

■ Допустим:

- Элементы матриц – double
- Блок кэша = 8 double (64 байта)
- Размер кэша $C \ll n$ (много меньше n)

■ Вторая итерация:

- Опять:
 $n/8 + n = 9n/8$ промахов



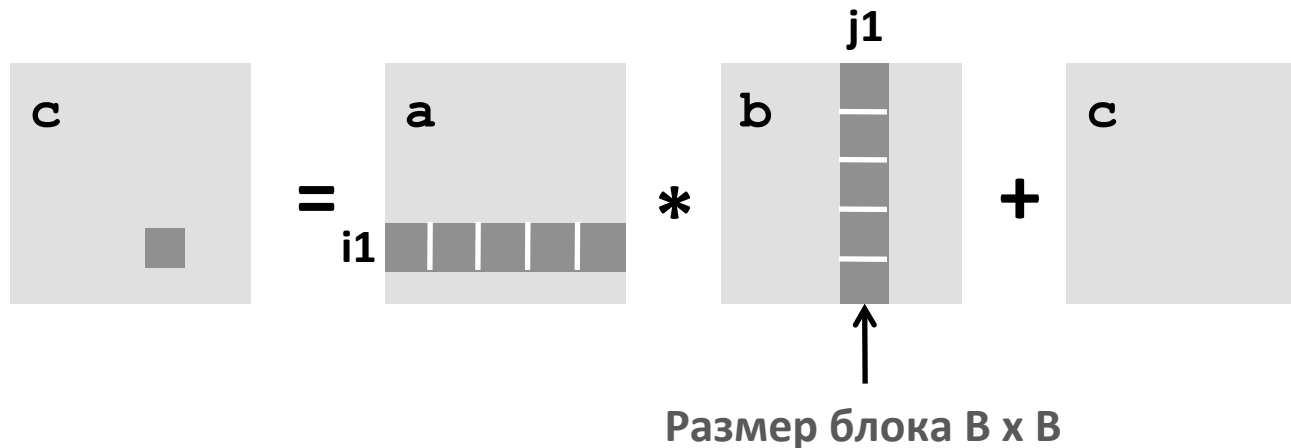
■ Всего промахов:

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

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];
}
```



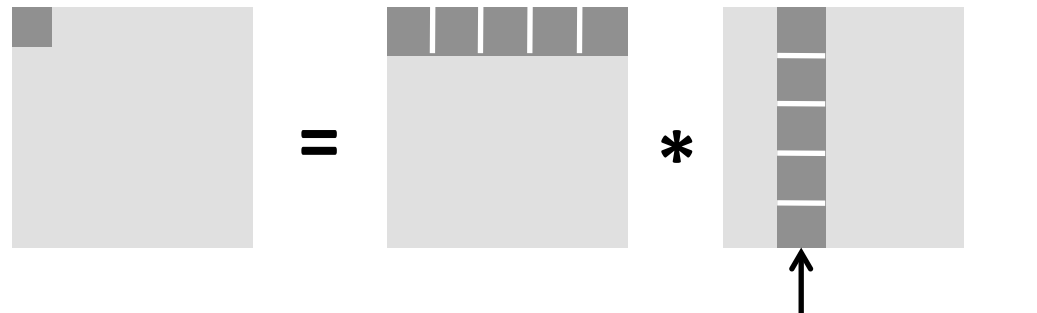
Анализ промахов

■ Допустим:

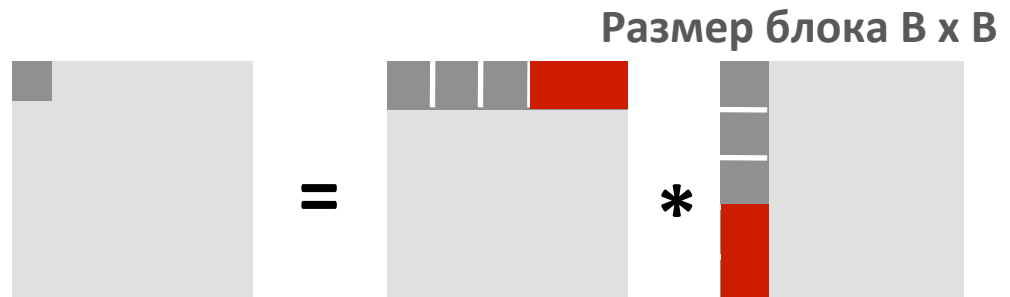
- Блок кэша = 8 doubles
- Размер кэша $C \ll n$ (много меньше n)
- Четыре блока  умещаются в кеш : $4B^2 < C$

■ 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)




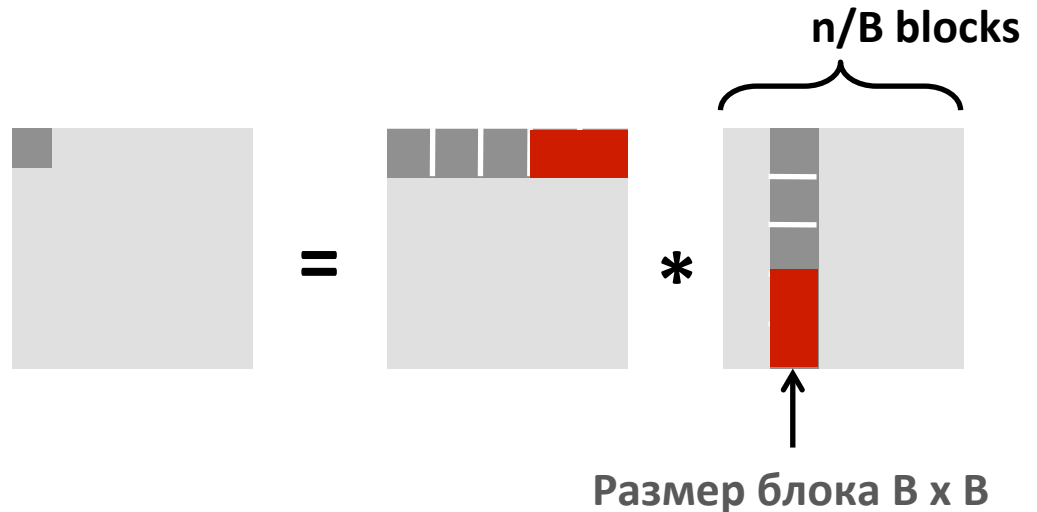
Анализ промахов

■ Допустим:

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

■ Second (block) iteration:

- Как и на первой итерации 
- $2n/B * B^2/8 = nB/4$



■ Всего промахов:

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

Итого

- 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

Заключение

- **Программист может оптимизировать производительность кэша**
 - Организация структур данных
 - Организация доступа к данным
 - Структура вложенных циклов
 - Объединение в блоки общеупотребительная техника
- **Для всех систем полезен “дружелюбный к кэшу код”**
 - Получение абсолютно оптимального кода очень зависит от платформы
 - Размер кэша, размер линии, канальность, и т.п.
 - Основные улучшения можно выполнить в общем случае
 - Минимизация рабочего набора данных (временная локальность)
 - Использование малых шагов (пространственная локальность)