

Семинар 5

Стандарт OpenMP (часть 5)

Михаил Курносов

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Цикл семинаров «Основы параллельного программирования»
Институт физики полупроводников им. А. В. Ржанова СО РАН
Новосибирск, 2015

Локальность ссылок в программах

- **Локальность ссылок (locality of reference)** – свойство программ повторно (часто) обращаться к одним и тем же адресам в памяти
- **Временная локализация (temporal locality)** – повторное обращение к одному и тому же адресу через короткий промежуток времени
- **Пространственная локализация ссылок (spatial locality)** – свойство программ повторно обращаться через короткий промежуток времени к адресам близко расположенным в памяти друг к другу

```
for (i = 0; i < n; i += 32)
    v[i] += f(i);

for (i = 0; i < n; i += 32)
    s += v[i];
```

```
for (i = 0; i < n; i++)
    c[i] = x[i] + y[i];
```

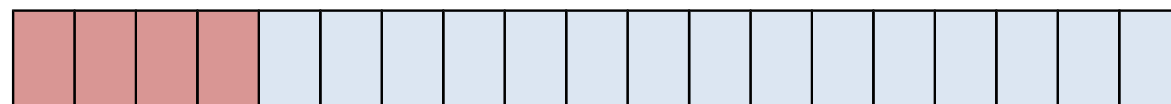
Локальность ссылок в программах

Структура (шаблон) доступа к массиву (Reference pattern)

```
int sumvec1(int v[N])
{
    int i, sum = 0;
    for (i = 0; i < N; i++)
        sum += v[i];
    return sum;
}
```

Address	0	4	8	12	16	20
Value	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]
Step	1	2	3	4	5	6

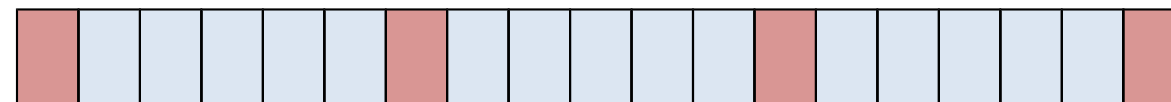
stride-1 reference pattern (good locality)



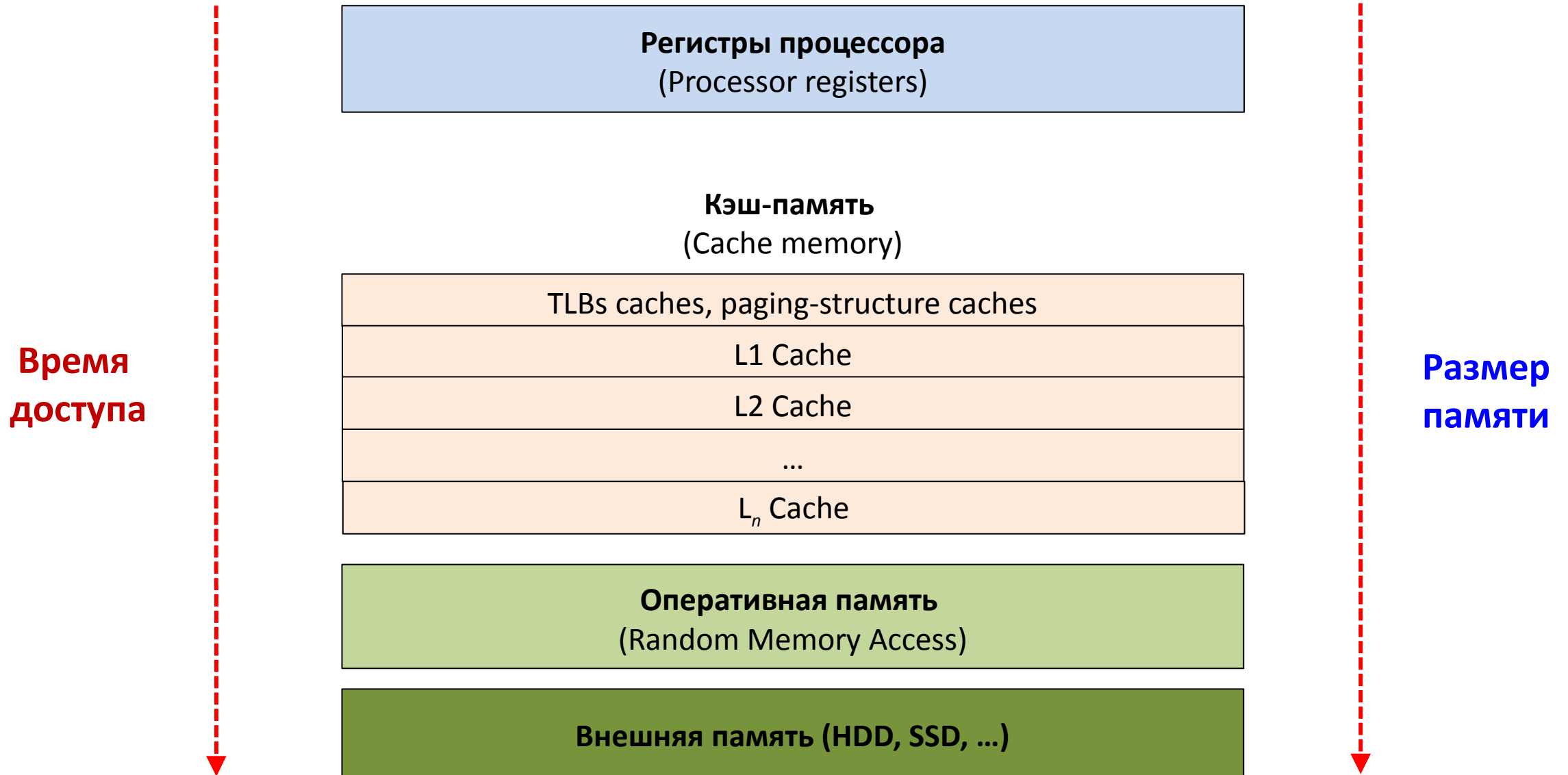
```
int sumvec2(int v[N])
{
    int i, sum = 0;
    for (i = 0; i < N; i += 6)
        sum += v[i];
    return sum;
}
```

Address	0	24	48	72	96	120
Value	v[0]	v[6]	v[12]	v[18]	v[24]	v[30]
Step	1	2	3	4	5	6

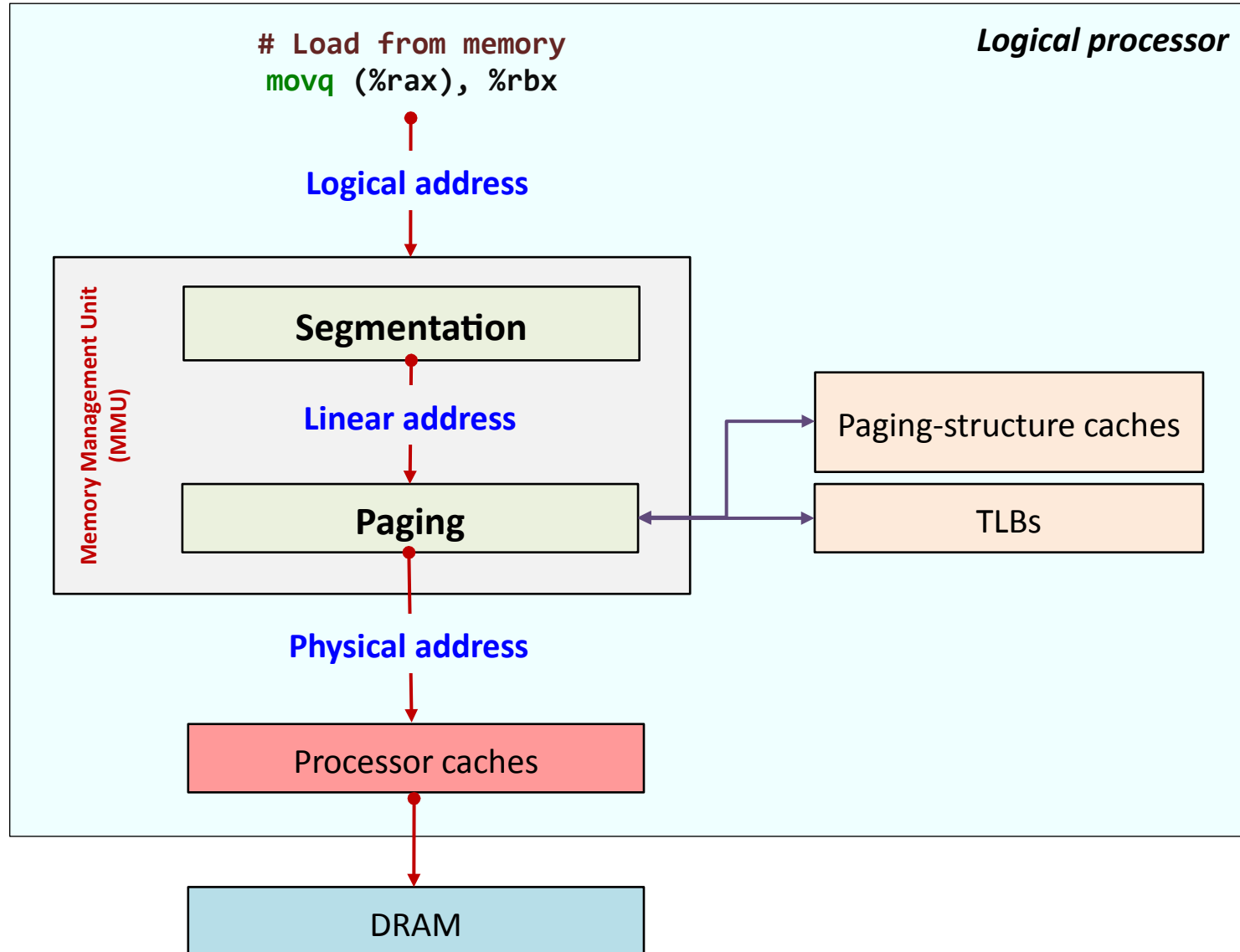
stride-6 reference pattern



Иерархическая организация памяти

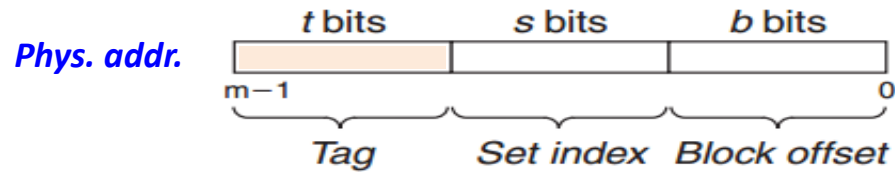


Страничная организация памяти



- Адресное пространство процесса разбито на страницы (pages)
- За каждой используемой страницей виртуальной памяти процесса закреплена страница физической памяти (page frame)
- Размер страницы 4 KiB (зависит от архитектуры CPU и ОС)
- Программы оперируют линейными адресами (linear address)
- Процессор работает с физическими адресами (physical address)

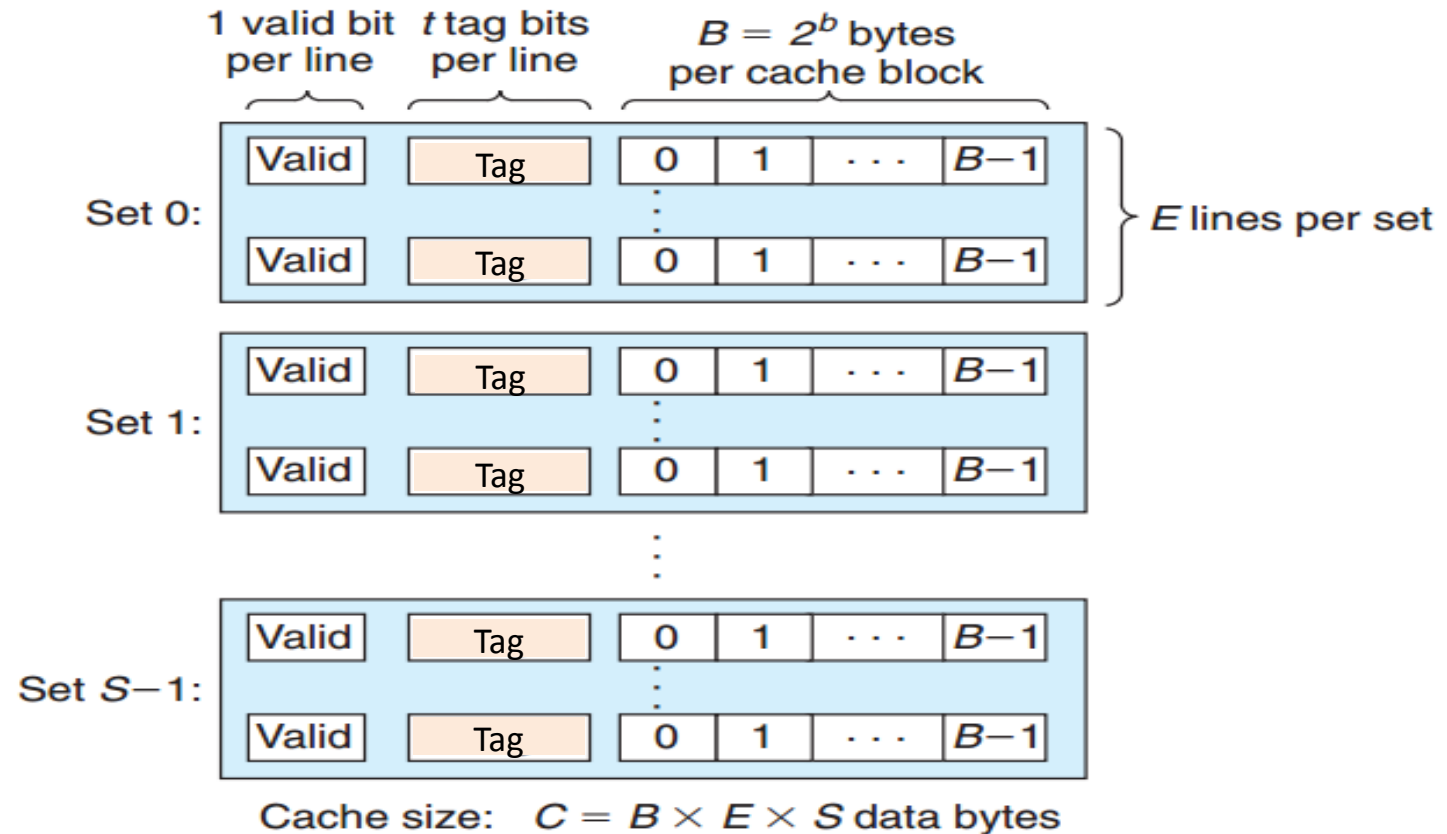
Структурная организация кеш-памяти



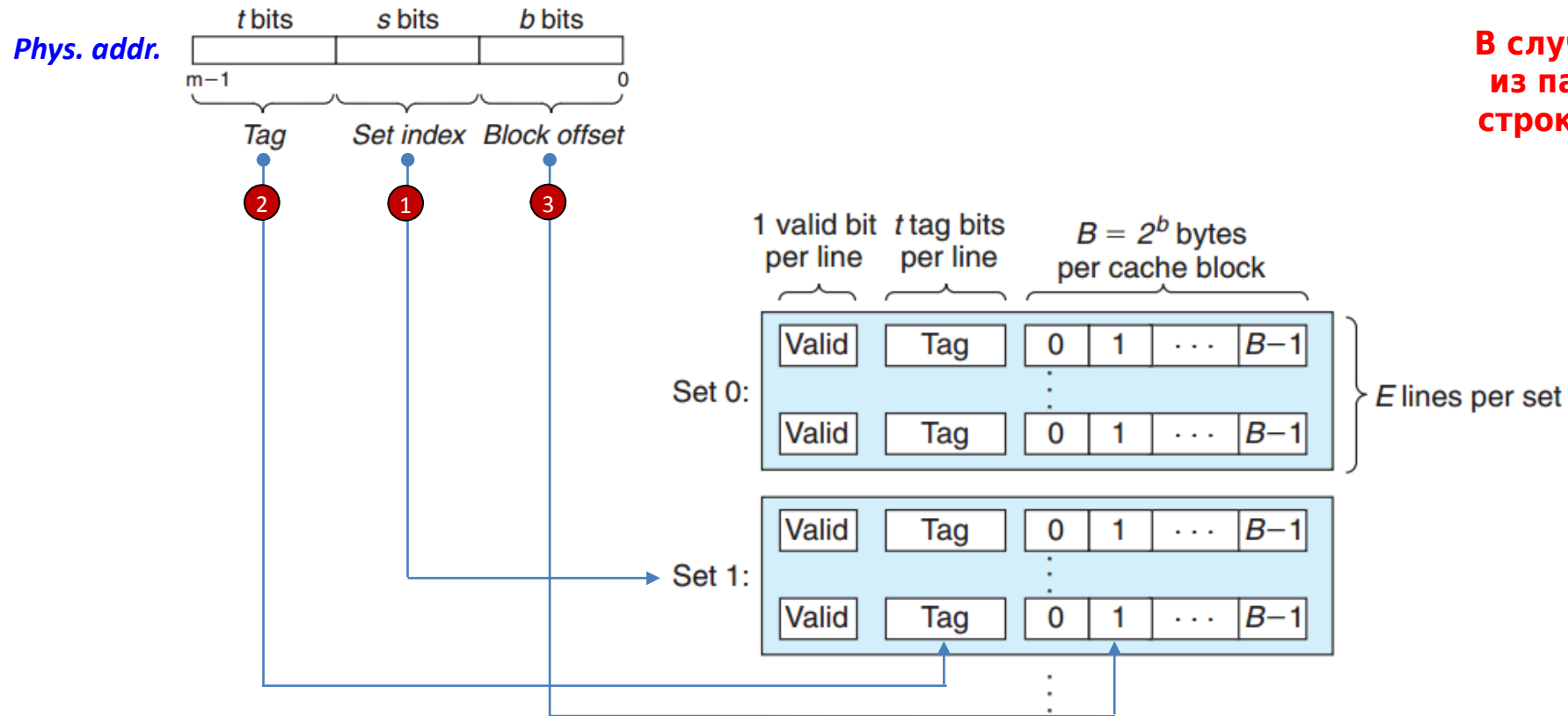
- Кеш содержит $S = 2^s$ множеств (**sets**)
- Каждое множество содержит E строк/записей (**cache lines**)
- Каждая строка содержит поля *valid bit*, *tag* (t бит) и блок данных ($B = 2^b$ байт)
- Данные между кеш-памятью и оперативной памятью передаются блоками по B байт (cache lines)
- «Полезный» размер кеш-памяти

$$C = S * E * B$$

Множественно-ассоциативная кеш-память



Загрузка данных из памяти (load/read)



В случае промаха (cache miss) из памяти в кеш загружается строка (даже если требовался всего один байт)

1. Выбирается одно из S множеств (по полю *Set index*)
2. Среди E записей множества отыскивается строка с требуемым полем *Tag* и установленным битом *Valid* (нашли – *cache hit*, не нашли – *cache miss*)
3. Данные из блока считываются с заданным смещением *Block offset*

Замещение записей кеш-памяти

2-way set associative cache:

Set0	V	Tag	Word0	Word1	Word2	Word3
Set1						
Set2	1	0001	15	20	35	40
	1	0011	1234	1222	3434	896
Set3						

Промех при загрузке данных

```
// Load from memory to register  
movl (40), %eax
```

- В какую строку (way) множества 2 загрузить блок с адресом 40?
- Какую запись вытеснить (evict) из кеш-памяти в DRAM?

Memory:

0-3	Word0	Word1	Word2	Word3
4-7				
8-11				
12-15				
16-19				
20-23				
24-27	15	20	35	40
28-31				
32-35				
36-39				
40-43	12	2312	342	7717
44-47				
48-51				
52-55				
56-59	1234	1222	3434	896
60-63				
64-67				
68-71				
...				

Размер кеш-памяти
значительно меньше
объема оперативной
памяти

Некоторые адреса
памяти отображаются
на одни и те же строки
кеш-памяти

Address 40:

Tag: 000010 ₂	Index: 10 ₂	Offset: 00 ₂
--------------------------	------------------------	-------------------------

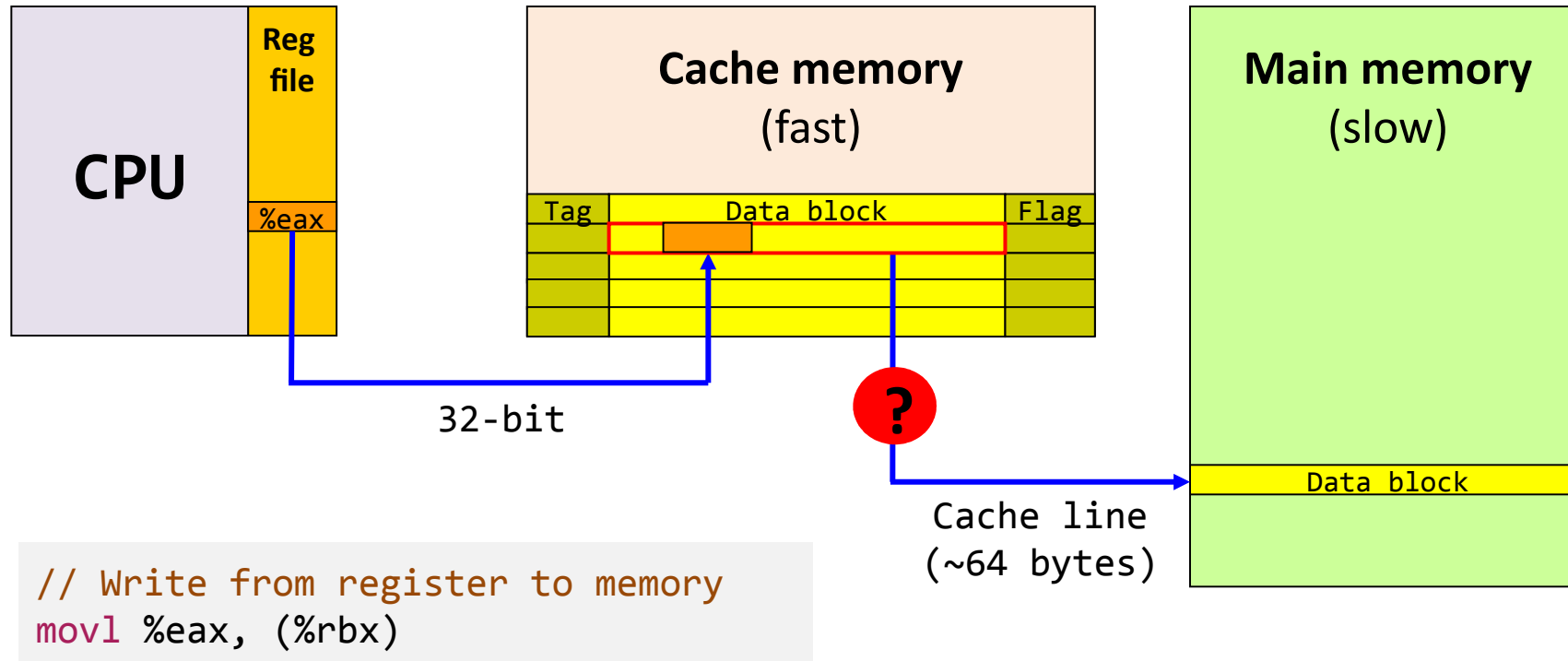
Address 24:

Tag: 000001 ₂	Index: 10 ₂	Offset: 00 ₂
--------------------------	------------------------	-------------------------

Алгоритмы замещения записей кеш-памяти

- **Алгоритмы замещения** (Replacement policy) требуют хранения вместе с каждой строкой кеш-памяти специализированного поля флагов/истории
- **LRU** (Least Recently Used) – вытесняется строку, неиспользованную дольше всех
- **MRU** (Most Recently Used) – вытесняет последнюю использованную строку
- **Алгоритм L. Belady** – вытесняет запись, которая с большой вероятностью не понадобится в будущем
- **RR** (Random Replacement) – вытесняет случайную строку
- ...

Запись данных в кеш-память



- Политики записи (write policy) определяют:
 - ☐ Когда данные должны быть переданы из кеш-памяти в оперативную память
 - ☐ Как должна вести себя кеш-память при событии «write miss» – запись отсутствует в кеш-памяти

Алгоритмы записи в кеш-память (write policy)

Политика поведения кеш-памяти в ситуации “write hit” (запись имеется в кеш-памяти)

- **Политика write-through** (сквозная запись) – каждая запись в кеш-память влечет за собой обновление данных в кеш-памяти и оперативной памяти (кеш “отключается” для записи)
- **Политика write-back** (отложенная запись, copy-back) – первоначально данные записываются только в кеш-память
- Все измененные строки кеш-памяти помечаются как “грязные” (dirty)
- Запись в память “грязных” строк осуществляется при их замещении или специальному событию (lazy write)
- **Внимание:** чтение может повлечь за собой запись в память
 - ☐ При чтении возник cache miss, данные загружаются из кеш-памяти верхнего уровня (либо DRAM)
 - ☐ Нашли строку для замещения, если флаг dirty = 1, записываем её данные в память
 - ☐ Записываем в строку новые данные

GNU/Linux CPU Cache Information (/proc)

```
$ cat /proc/cpuinfo
processor      : 0
...
model name    : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
stepping: 7
microcode     : 0x29
cpu MHz       : 2975.000
cache size    : 3072 KB
physical id   : 0
siblings: 4
core id       : 0
cpu cores     : 2
apicid        : 0
initial apicid : 0
...
bogomips: 4983.45
clflush size  : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
```

Вычислительный узел кластера Oak

```
$ cat /proc/cpuinfo
model name      : Intel(R) Xeon(R) CPU E5620  @ 2.40GHz
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual

$ cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
```

Анализ /sys/devices/system/cpu/cpuX/cache/

- L1 Data cache: 32K, set associative cache — sets 64, ways — 8
- L1 Instruction cache: 32K, set associative cache — sets 128, ways — 4
- L2 Unified cache: 256K, set associative cache — sets 512, ways — 8
- L3 Unified cache: 12288K, set associative cache — sets 12288, ways — 16

Умножение матриц (DGEMM)

```
int main(int argc, char **argv)
{
    double gflop = 2.0 * N * Q * M * 1E-9;
    double *a, *b, *c1, *c2;

    // Launch naive version
    a = malloc(sizeof(*a) * N * M);
    b = malloc(sizeof(*b) * M * Q);
    c1 = malloc(sizeof(*c1) * N * Q);

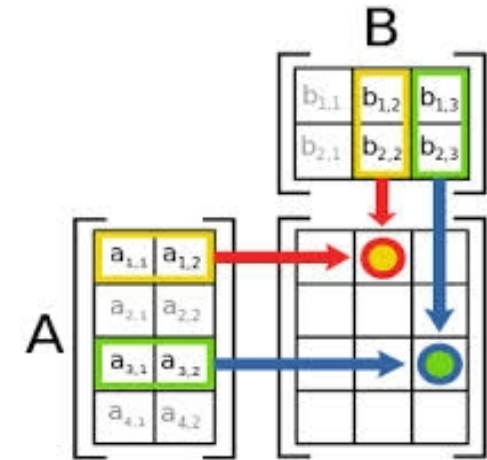
    srand(0);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++)
            a[i * M + j] = rand() % 100;
    }
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < Q; j++)
            b[i * Q + j] = rand() % 100;
    }

    double tdef = wtime();
    dgemv_def(a, b, c1, N, M, Q);
    tdef = wtime() - tdef;
    printf("Execution time (naive): %.6f\n", tdef);
    printf("Performance (naive): %.2f GFLOPS\n", gflop / tdef);
    free(b);
    free(a);

    return 0;
}
```

Умножение матриц (DGEMM Naive)

```
enum {  
    N = 100, M = 200, Q = 300  
};  
  
/*  
 * dgemm_def: Naive matrix multiplication C[n, q] = A[n, m] * B[m, q].  
 * FP ops = 2*n*q*m.  
 */  
void dgemm_def(double *a, double *b, double *c, int n, int m, int q)  
{  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < q; j++) {  
            double s = 0.0;  
            for (int k = 0; k < m; k++)  
                s += a[i * m + k] * b[k * q + j];  
            c[i * q + j] = s;  
        }  
    }  
}
```



Число операций над числами
с плавающей запятой (FLOP)

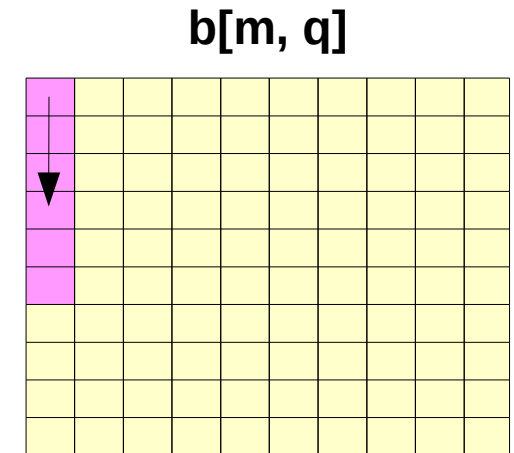
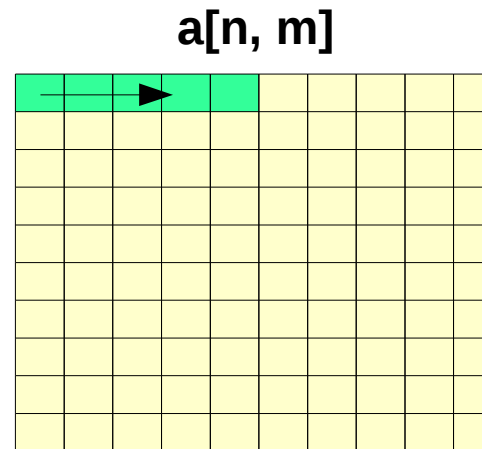
$$2 * N * Q * M$$

Умножение матриц (DGEMM Naive)

```
enum {
    N = 100, M = 200, Q = 300
};

/*
 * dgemm_def: Naive matrix multiplication C[n, q] = A[n, m] * B[m, q].
 *           FP ops = 2*n*q*m.
 */
void dgemm_def(double *a, double *b, double *c, int n, int m, int q)
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < q; j++) {
            double s = 0.0;
            for (int k = 0; k < m; k++)
                s += a[i * m + k] * b[k * q + j];
            c[i * q + j] = s;
        }
    }
}
```

Load a[0][0] — cache miss
Load b[0][0] — cache miss
Load a[0][1] — **cache hit**
Load b[1][0] — cache miss
Load a[0][1] — **cache hit**
Load b[2][0] — cache miss
Load a[0][2] — **cache hit**
Load b[2][0] — cache miss
...



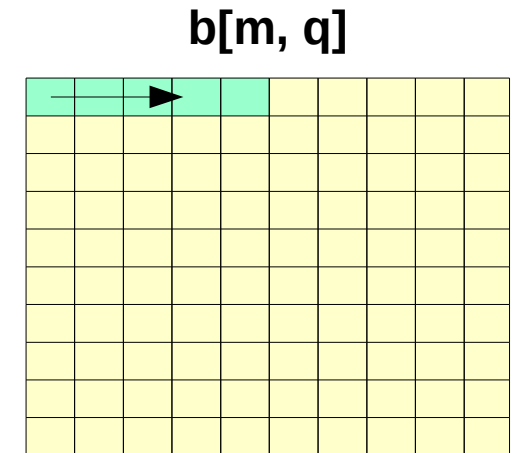
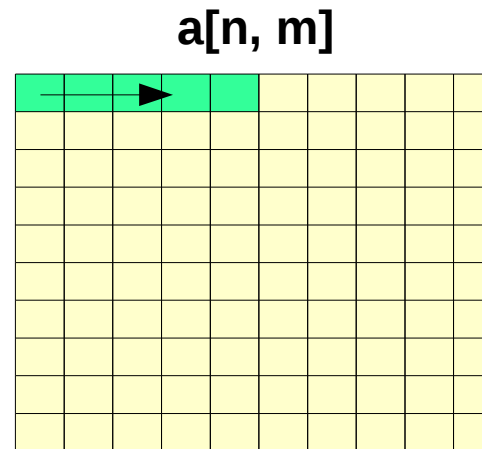
Прыжки через строку
Cache miss!

Умножение матриц (DGEMM Opt.)

```
/*
 * dgemm_opt: Matrix multiplication  $C[n, q] = A[n, m] * B[m, q]$ .
 *
 */
void dgemm_opt(double *a, double *b, double *c, int n, int m, int q)
{
    for (int i = 0; i < n * q; i++)
        c[i] = 0;

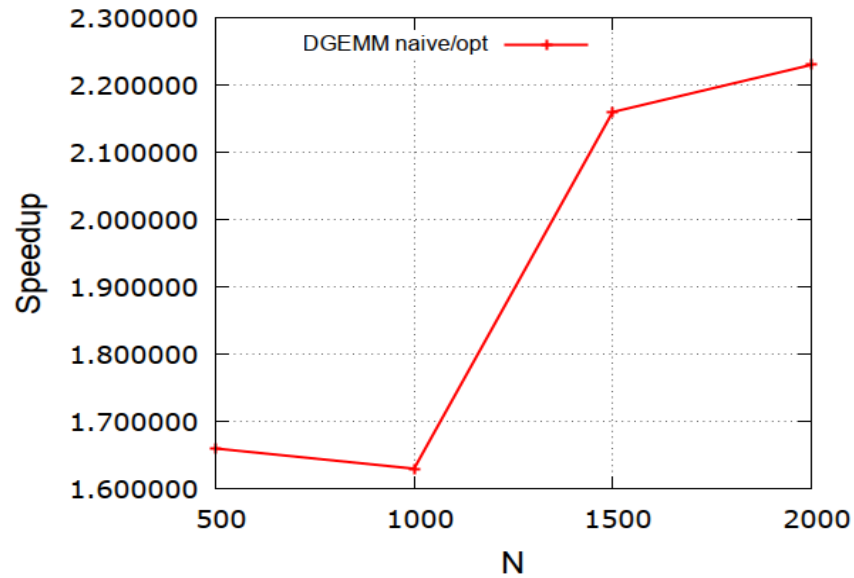
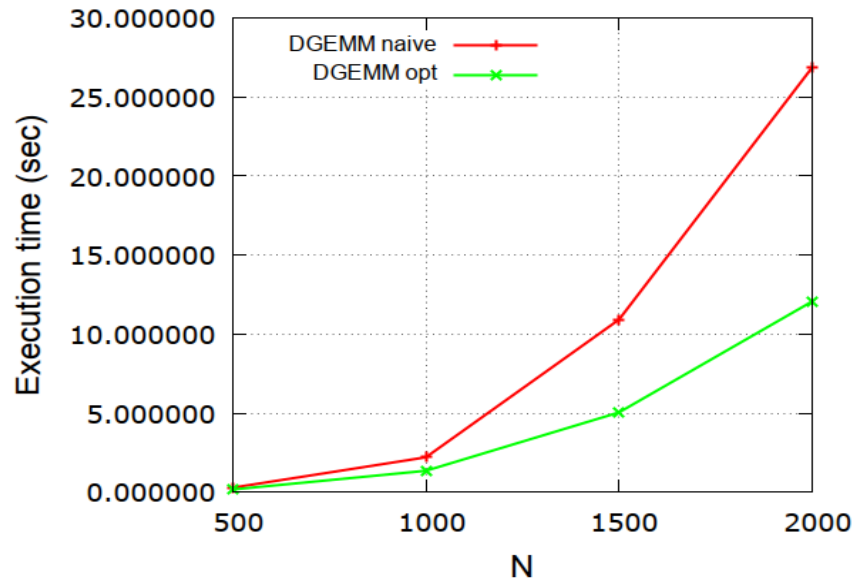
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < m; k++) {
            for (int j = 0; j < q; j++)
                c[i * q + j] += a[i * m + k] * b[k * q + j];
        }
    }
}
```

Load a[0][0] — cache miss
Load b[0][0] — cache miss
Load a[0][1] — cache hit
Load b[0][1] — cache hit
Load a[0][1] — cache hit
Load b[0][1] — cache hit
Load a[0][2] — cache hit
Load b[0][2] — cache hit
...



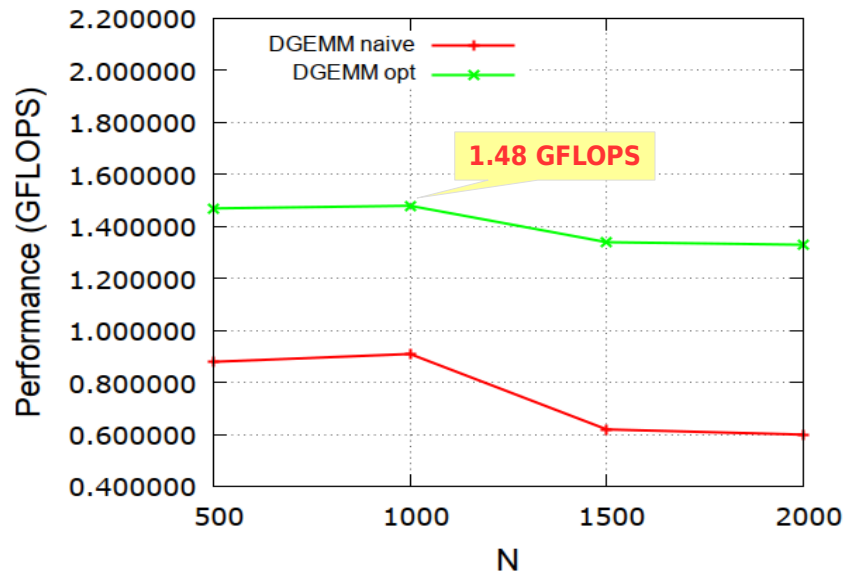
Обращение
по смежным адресам

Анализ производительности DGEMM (Naive vs Opt.)



Вычислительный узел кластера Oak

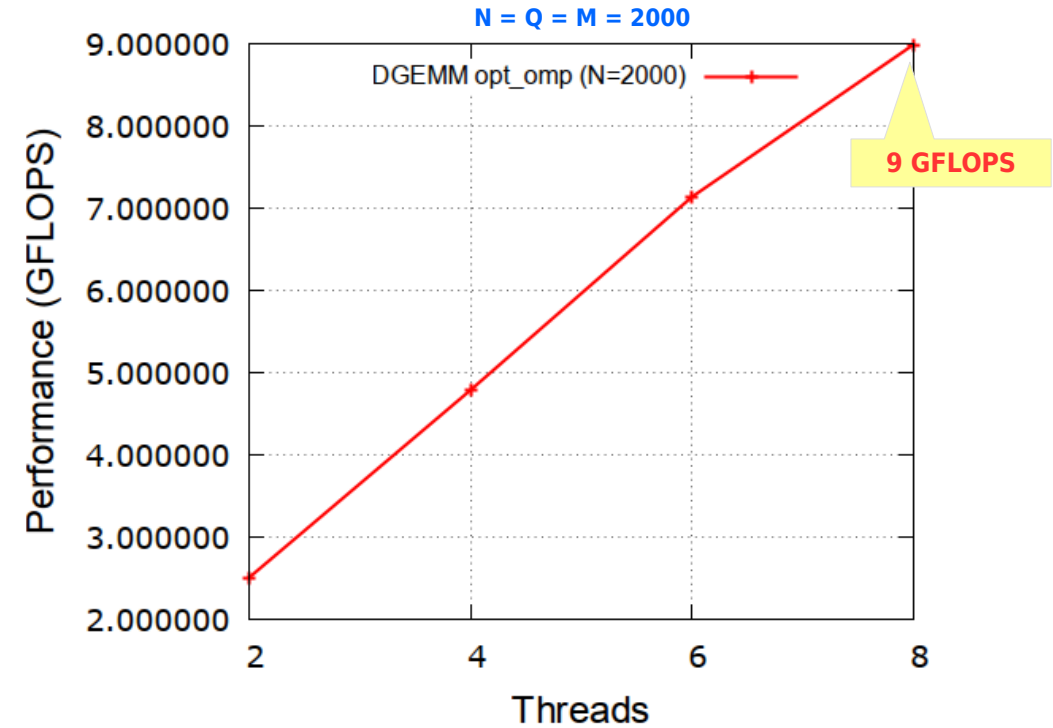
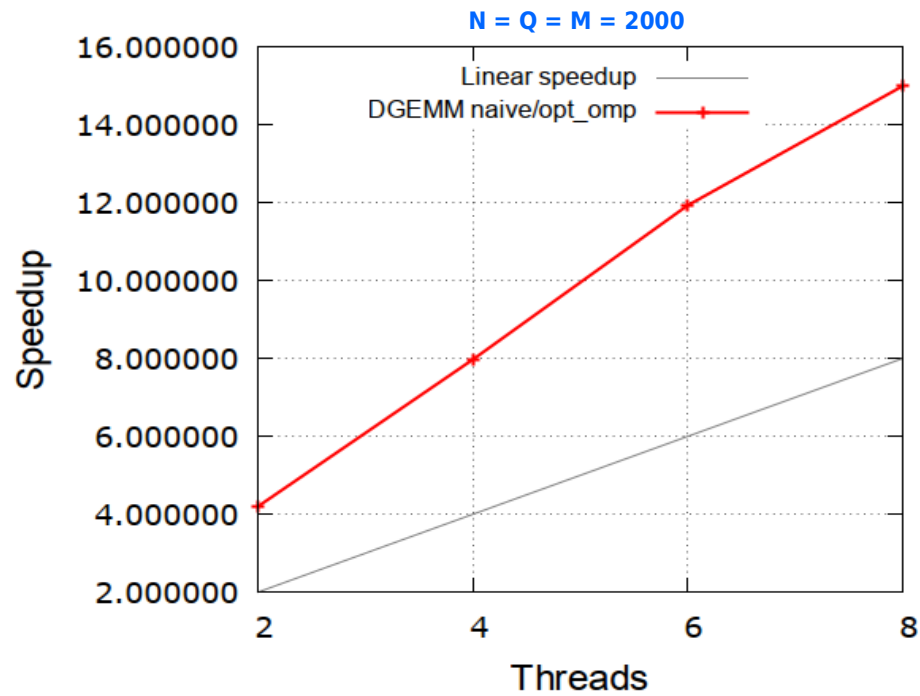
- 8 ядер (два Intel Quad Xeon E5620)
- 24 GiB RAM (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64



Дальнейшие оптимизации:

- Блочное умножение матриц
- TLB-оптимизация
- Векторизация (SSE/AVX)
- Распараллеливание (OpenMP)
- ...

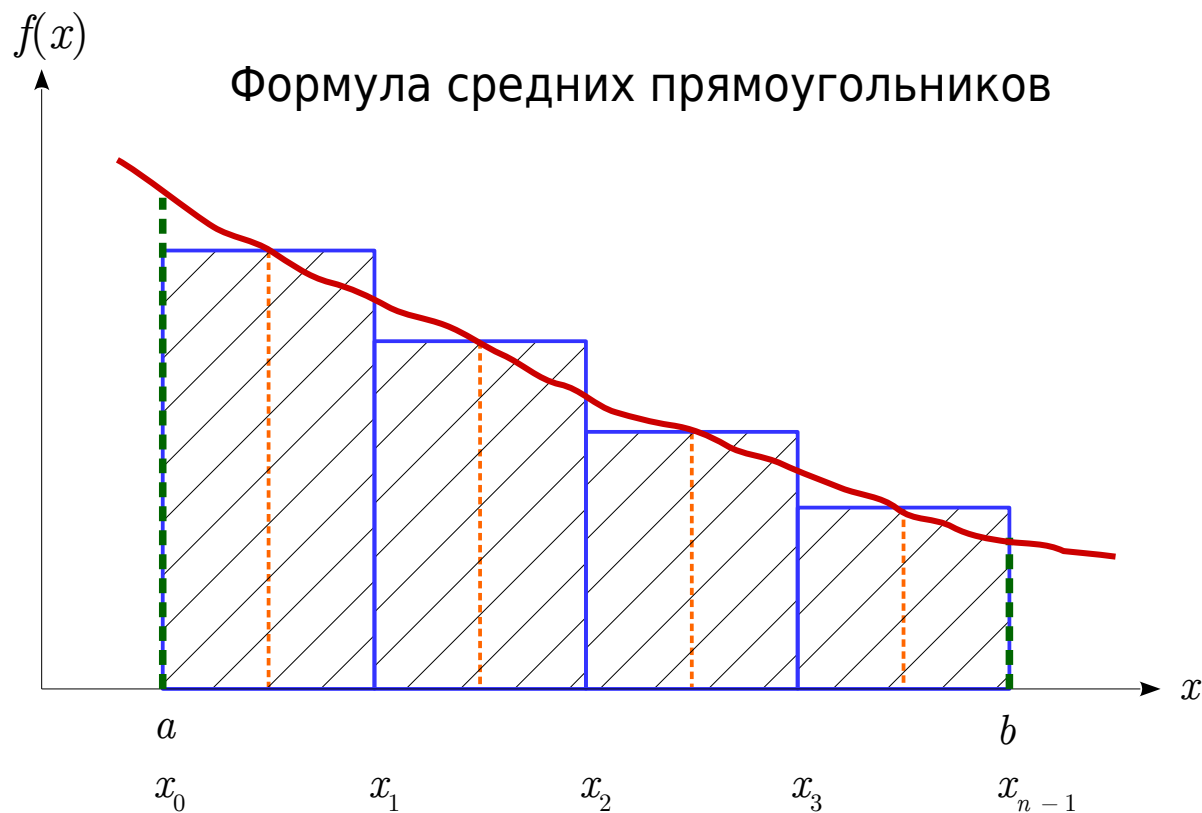
Анализ производительности (OpenMP-версия)



Ускорение больше линейного:

За ускорение принято отношение времени
выполнения двух разных алгоритмов
— `dgemt_naive` и OpenMP-версии с эффективным
доступом к элементам матрицы B

Численное интегрирование (метод прямоугольников)



```
const double a = -4.0;           /* [a, b] */
const double b = 4.0;
const int nsteps = 40000000;     /* n */

double func(double x)
{
    return exp(-x * x);
}

double integrate(double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    for (int i = 0; i < n; i++)
        sum += func(a + h * (i + 0.5));

    sum *= h;
    return sum;
}
```

$$\int_a^b f(x) dx \approx h \sum_{i=1}^n f\left(x_{i-1} + \frac{h}{2}\right) = h \sum_{i=1}^n f\left(x_i - \frac{h}{2}\right), \quad h = \frac{b-a}{n}$$

Численное интегрирование (метод прямоугольников)

```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    double thread_sum[32];

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        thread_sum[tid] = 0;

        #pragma omp for
        for (int i = 0; i < n; i++)
            thread_sum[tid] += func(a + h * (i + 0.5));

        #pragma omp atomic
        sum += thread_sum[tid];
    }
    sum *= h;
    return sum;
}
```

Локальное хранилище
для каждого потока

Численное интегрирование (метод прямоугольников)

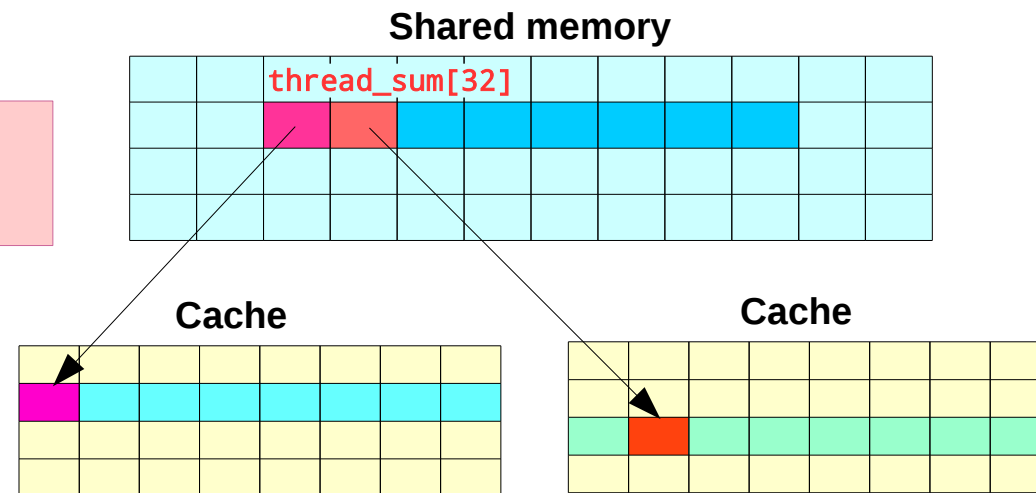
```
double integrate_omp(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    double thread_sum[32];

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        thread_sum[tid] = 0;

        #pragma omp for
        for (int i = 0; i < n; i++)
            thread_sum[tid] +=
                func(a + h * (i + 0.5));
        #pragma omp atomic
        sum += thread_sum[tid];
    }
    sum *= h;
    return sum;
}
```

Ложное разделение
(false sharing)



Протокол обеспечения когерентности кешей MESI (Intel MESIF)

- Поток обновляет свою строку в кеш-памяти и аннулирует строку в других кешах
- Другие кеши загружают актуальную версию строки из оперативной памяти
- Происходит постоянное обновление строк из памяти (кеширование «отключается»)

Численное интегрирование (метод прямоугольников)

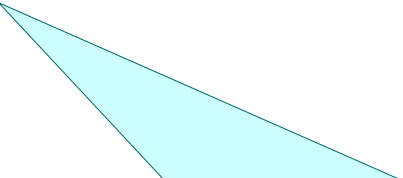
```
double integrate_omp_opt(double (*func)(double), double a, double b, int n)
{
    double h = (b - a) / n;
    double sum = 0.0;

    /* Each struct occupied one cache line (64 bytes) */
    struct thread_param {
        double sum;           /* 8 bytes */
        double padding[7];    /* 56 bytes */
    };
    struct thread_param thread_sum[32] __attribute__((aligned(64)));

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        thread_sum[tid].sum = 0;

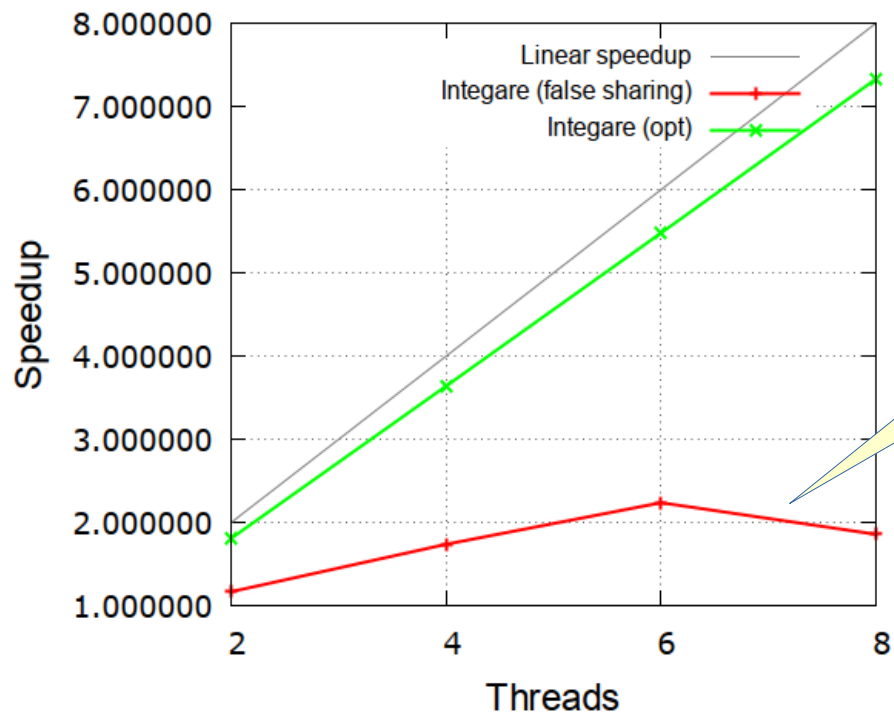
        #pragma omp for
        for (int i = 0; i < n; i++)
            thread_sum[tid].sum += func(a + h * (i + 0.5));

        #pragma omp atomic
        sum += thread_sum[tid].sum;
    }
    sum *= h;
    return sum;
}
```

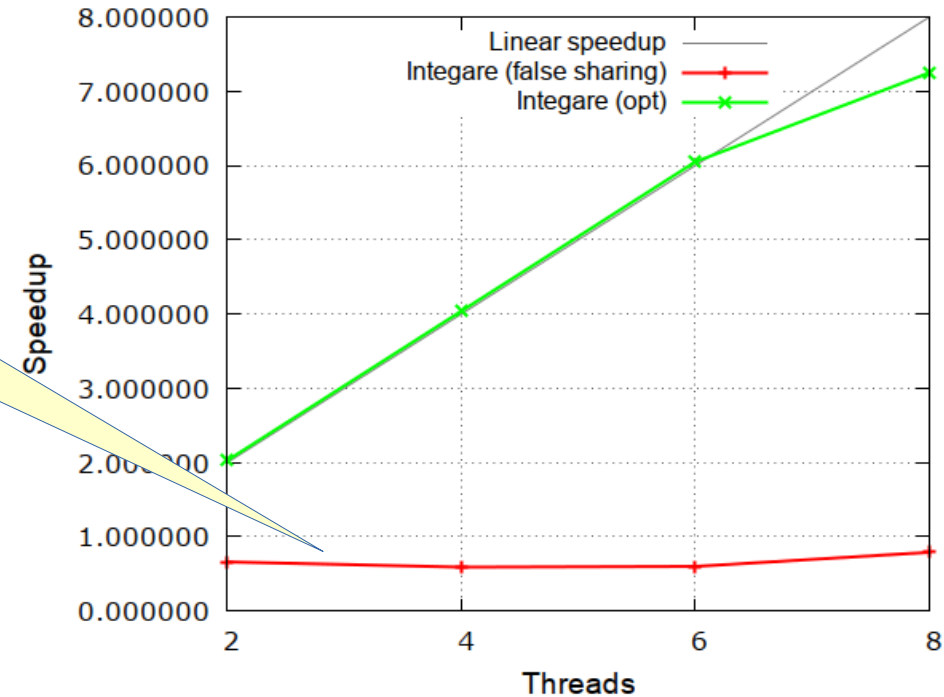


Каждый элемент массива будет занимать одну строку кеш-памяти

Анализ производительности (integrate)



OpenMP-версия
хуже последовательной
программы использует
кеш-память



Вычислительный узел кластера Oak (NUMA)

- **8 ядер** (два Intel Quad Xeon E5620)
- **24 GiB RAM** (6 x 4GB DDR3 1067 MHz)
- CentOS 6.5 x86_64

Вычислительный узел кластера Jet (SMP)

- **8 ядер** (два Intel Quad Xeon E5420)
- **8 GiB RAM**
- Fedora 20 x86_64

Задание

- Разработать на OpenMP параллельную версию функции `dgemm_opt` — написать код функции `dgemm_opt_omp`
- Провести анализ масштабируемости параллельной программы
- Шаблон программы находится в каталоге `_task`
- Как реализовать параллельную инициализацию матриц?

Суммирование элементов массива

```
int sumarray3d_def(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

Массив `a[3][3][3]` хранится в памяти строка за строкой (row-major order)

*Reference pattern: stride-(N*N)*

Address 0	a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[0][2][0]	a[0][2][1]	a[0][2][2]
36	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]	a[1][2][0]	a[1][2][1]	a[1][2][2]
72	a[2][0][0]	a[2][0][1]	a[2][0][2]	a[2][1][0]	a[2][1][1]	a[2][1][2]	a[2][2][0]	a[2][2][1]	a[2][2][2]

Суммирование элементов массива

```
int sumarray3d_def(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                sum += a[i][j][k];
            }
        }
    }
    return sum;
}
```

Массив `a[3][3][3]` хранится в памяти строка за строкой (row-major order)

Reference pattern: stride-1

Address 0	a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[0][2][0]	a[0][2][1]	a[0][2][2]
36	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]	a[1][2][0]	a[1][2][1]	a[1][2][2]
72	a[2][0][0]	a[2][0][1]	a[2][0][2]	a[2][1][0]	a[2][1][1]	a[2][1][2]	a[2][2][0]	a[2][2][1]	a[2][2][2]