



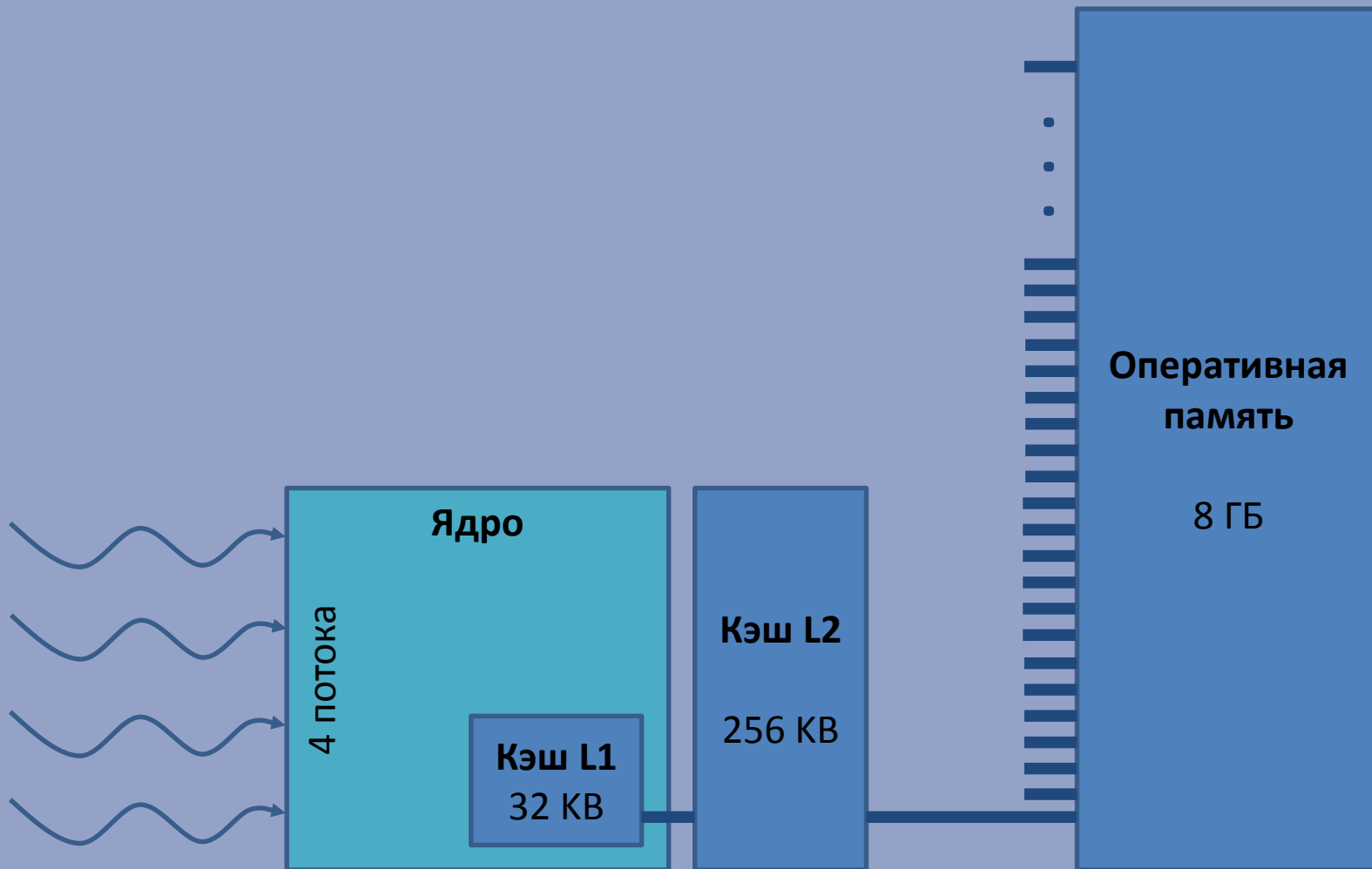
Новосибирский государственный университет  
Факультет информационных технологий  
Кафедра параллельных вычислений

# Ускоритель Intel Xeon Phi

## Оптимизация работы с памятью

Преподаватель:  
Киреев С.Е.

# Иерархия памяти Xeon Phi



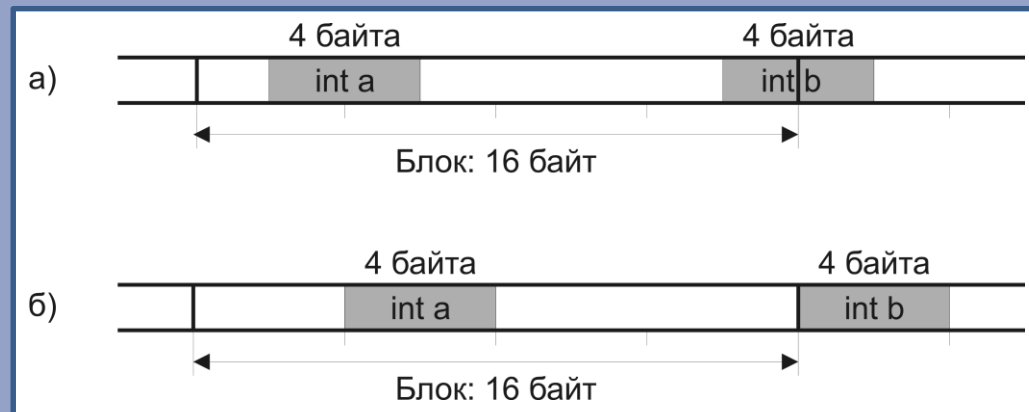
# На что следует обращать внимание при работе с памятью

- Выравнивание данных
- Плотность размещения данных
- Объём данных
- Порядок обхода данных
- Доступ к памяти нескольких потоков
- Дополнительные средства

# **ВЫРАВНИВАНИЕ ДАННЫХ**

# Выравнивание данных

- Память делится на блоки
  - в соответствии с размером страницы, кэш-строки, сектора кэша, шины данных, ...
- Элементы данных могут пересекать границы блоков
  - Это приводит к нескольким операциям чтения/записи вместо одной.
- Пример:



# Выравнивание данных

- Естественное выравнивание
  - По размеру элемента данных (до машинного слова)
  - Применяется компилятором автоматически
- Выравнивание вручную

```
int x[N] __attribute__((aligned(64)));
```

```
ptr = malloc(size + 64);
```

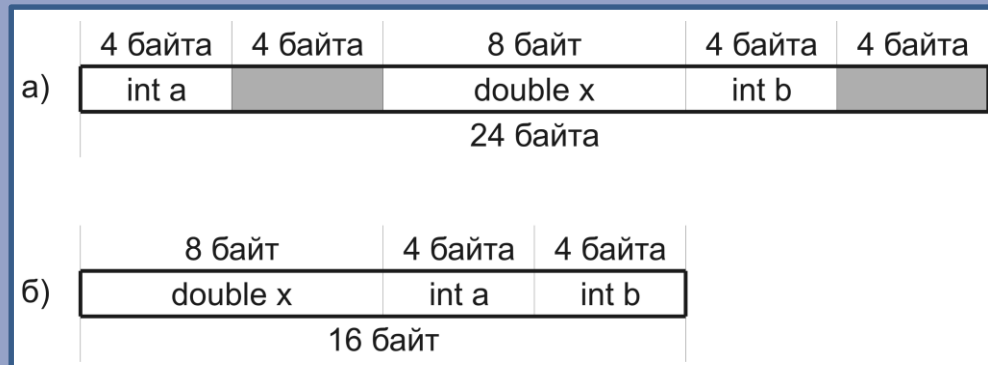
```
ptr = (ptr % 64) ? (ptr & 0xfffffc0) + 64 : ptr;
```

```
err = posix_memalign(&ptr, 64, size);
```

```
ptr = _mm_malloc(size, 64);
```

# Выравнивание данных

- Выравнивание элементов структур:
  - a) `struct S1 { int a; double x; int b; };`
  - б) `struct S2 { double x; int a; int b; };`

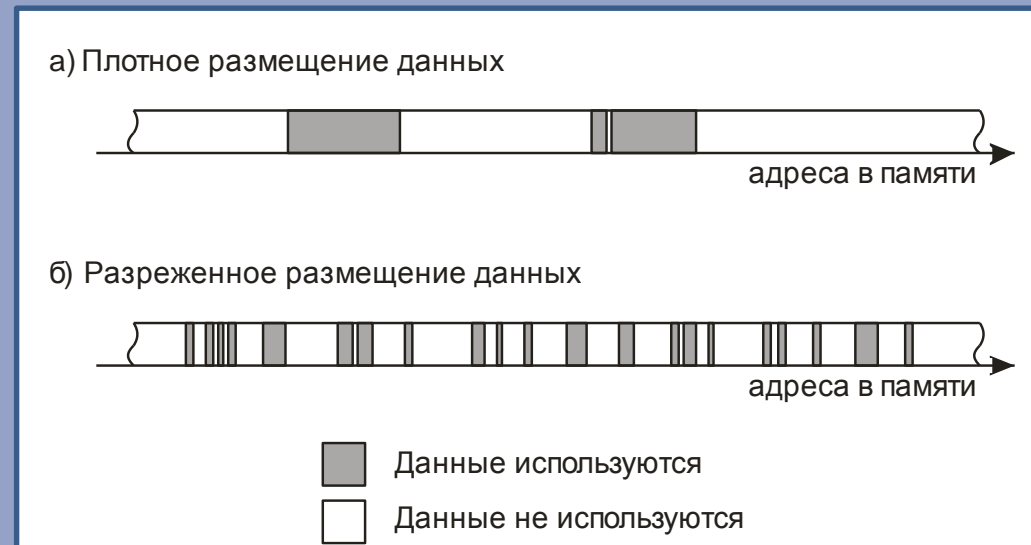


- Управление выравниванием в компиляторе  
`#pragma pack(...)`

# **ПЛОТНОЕ И РАЗРЕЖЕННОЕ РАЗМЕЩЕНИЕ ДАННЫХ**

# Плотное и разреженное размещение данных

- Память делится на блоки
  - в соответствии с размером страницы, кэш-строки, сектора кэша, шины данных, ...
- Загрузка блока занимает время
- Плотно размещенные данные используют меньше блоков



# Плотное и разреженное размещение данных

- Пример:

Вариант А	Вариант В
<pre>struct point { float x, y; }; struct point points[N];</pre>	<pre>float x[N]; float y[N];</pre>

а)



б)



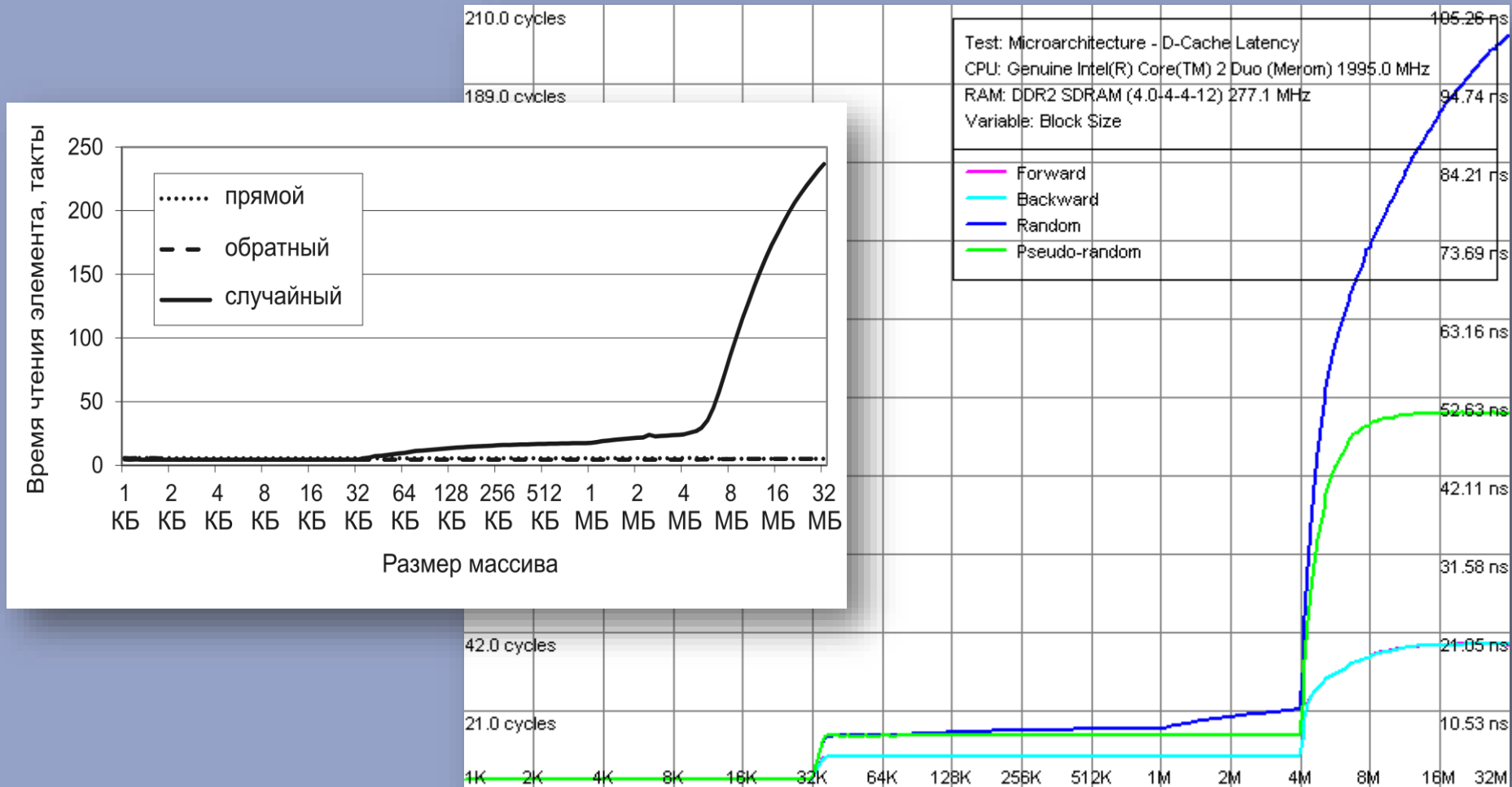
```
for (i=0;i<N;i++)  
    process(x[i],y[i])
```

```
for (i=0;i<N;i++) process(x[i]);  
for (i=0;i<N;i++) process(y[i])
```

**ОБЪЁМ ДАННЫХ**

# Объём данных

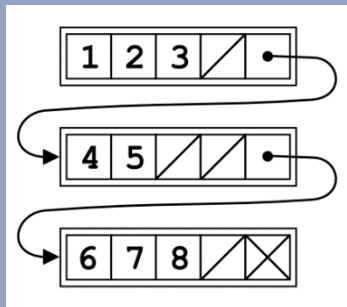
- Среднее время обращения к данным зависит от объема обрабатываемых данных:



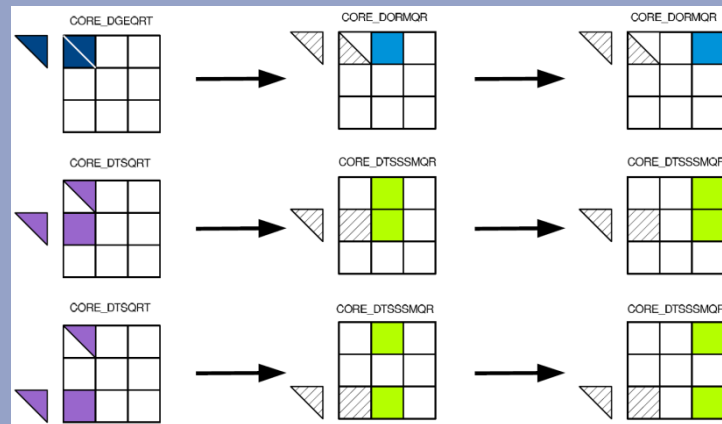
# Объём данных

- Для увеличения скорости работы с памятью используются специальные алгоритмы
  - Cache-oblivious algorithms, Блочные (Tiled), ...
  - Параметр – размер блока, зависит от размера кэша
- Идея: то, что загрузили, использовать по максимуму

Развернутый связный список

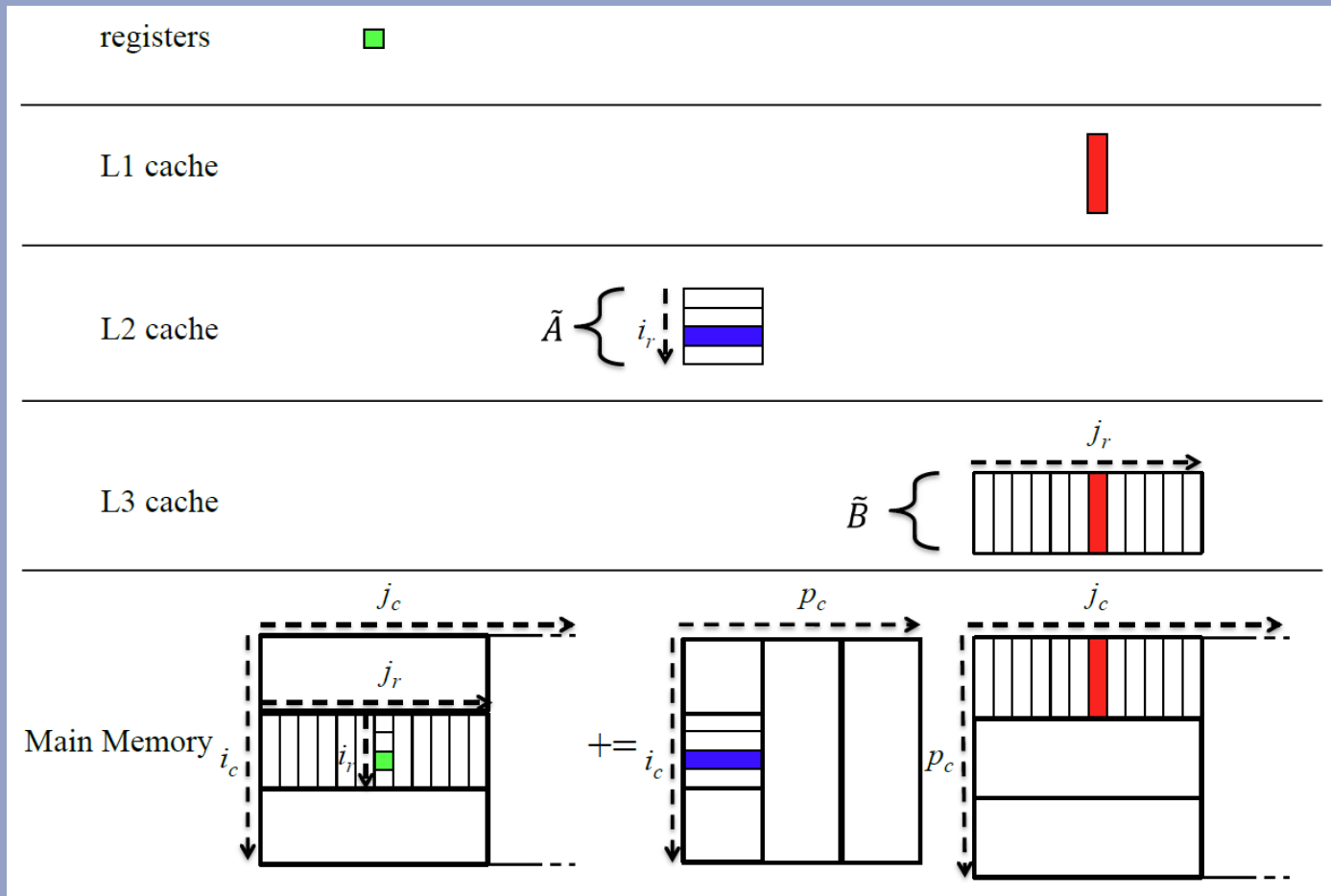


Блочные матричные алгоритмы



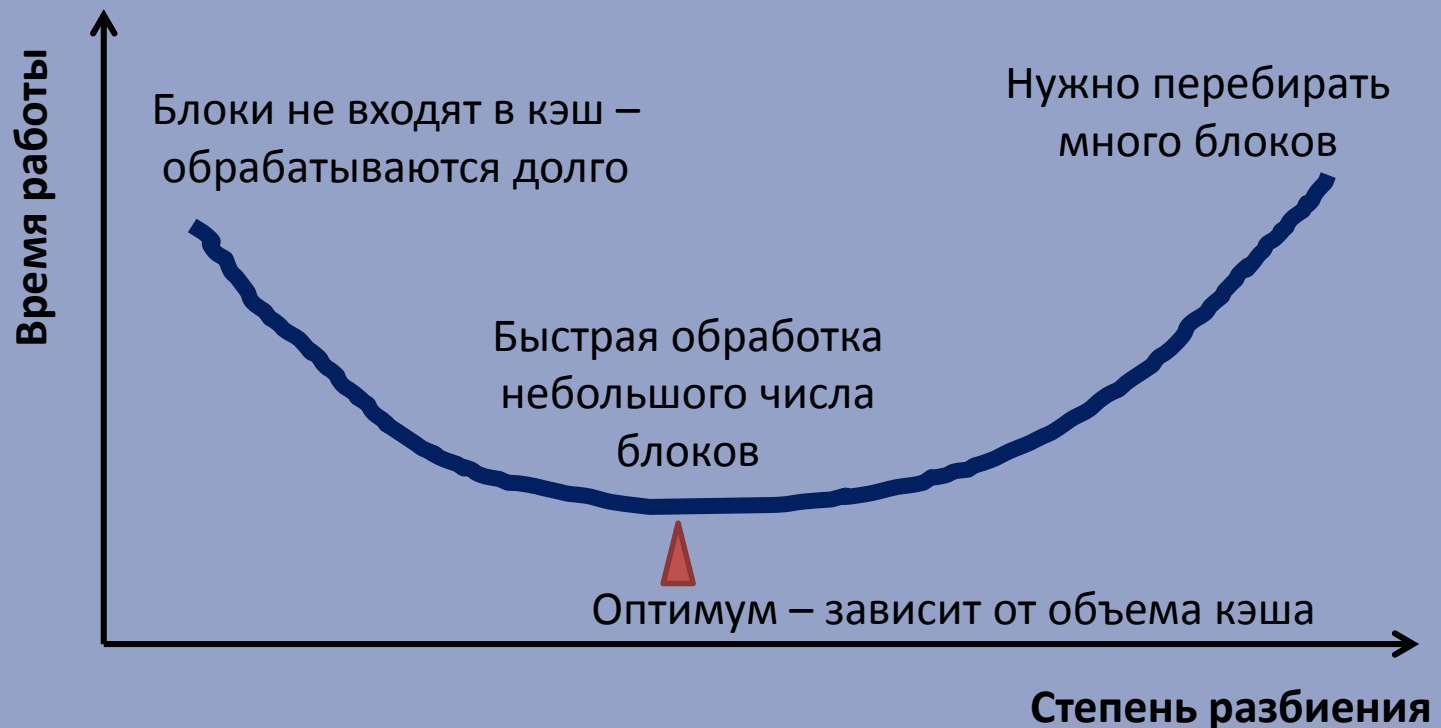
# Объём данных

Планирование размещения данных в иерархии памяти для умножения матриц



# Объём данных

- Типичная зависимость времени работы от степени разбиения задачи на блоки:



# Объём данных

- Особенности виртуальной памяти
  - Память разделена на страницы (стандарт x86: 4 КБ)
  - Есть таблица страниц (в оперативной памяти)
  - Есть TLB – кэш таблицы страниц
    - множественно-ассоциативный – есть буксование
    - многоуровневый (L1/L2, данных/команд)
  - Обращение к новой странице памяти очень долгое
- Как учитывать виртуальную память
  - Обращаться к данным компактно (в пределах небольшого числа страниц)
  - Использовать большие страницы (huge pages)

# Объём данных

## Проверка поддержки больших страниц

- `hudeadm --pool-list`
- `cat /proc/meminfo | grep Huge`
- `cat /proc/sys/vm/nr_hugepages`

## Использование больших страниц памяти (huge pages)

- Выделение памяти

```
#include<sys/mman.h>
void *data = mmap(0, size, PROT_READ | PROT_WRITE,
MAP_ANONYMOUS | MAP_PRIVATE | MAP_HUGETLB, -1, 0);
// (size должно быть кратно размеру huge страницы)
if (data == MAP_FAILED) { exit(1); }
```

- Освобождение памяти

```
munmap(data, size);
```

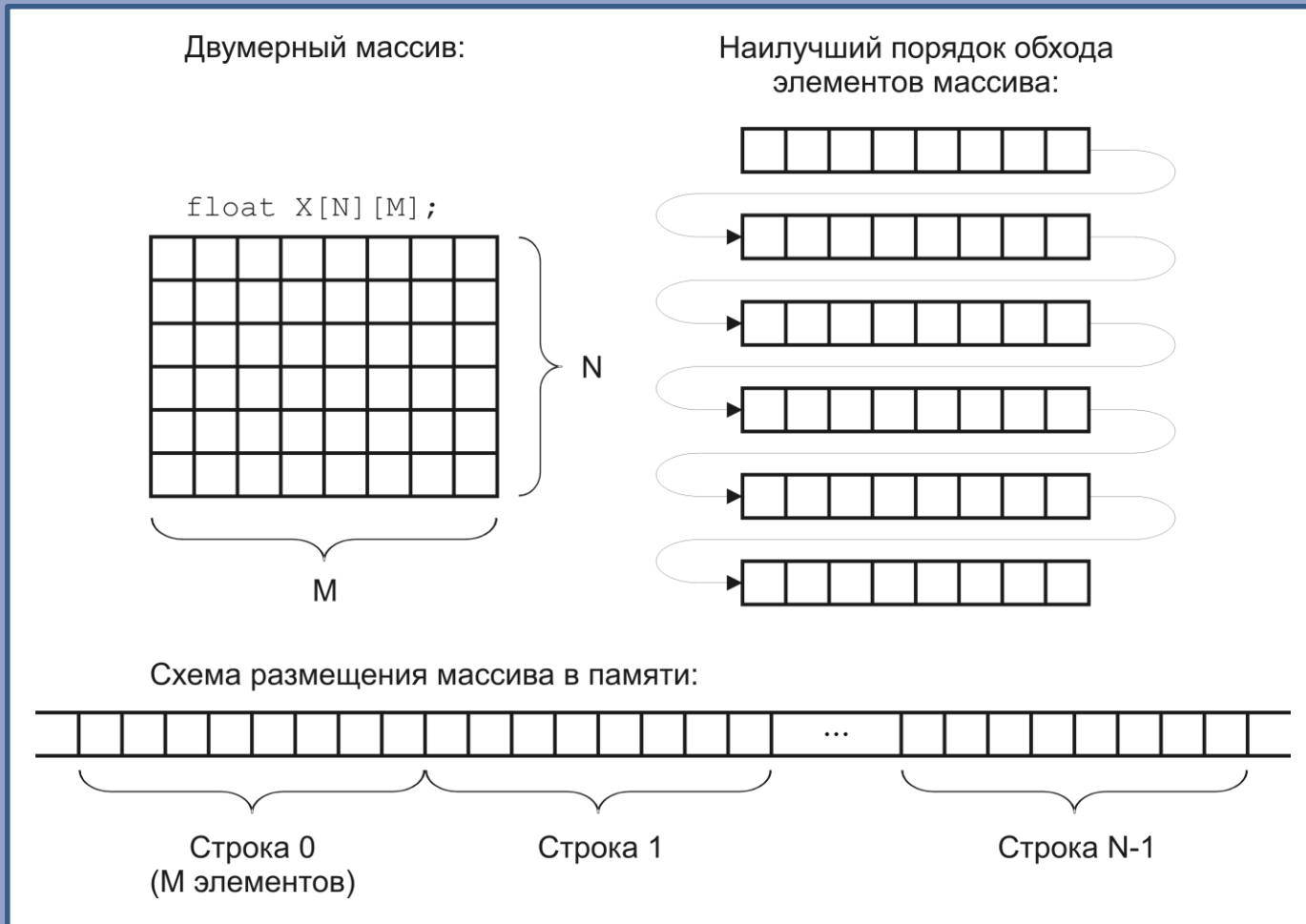
**ПОРЯДОК ОБХОДА ДАННЫХ**

# Порядок обхода данных

- Подсистема памяти оптимизирована для последовательного обхода
  - Блочное хранение и передача данных
  - Аппаратная предвыборка данных

# Порядок обхода данных

- Расположение данных в памяти



# Порядок обхода данных

## Пример: умножение матриц

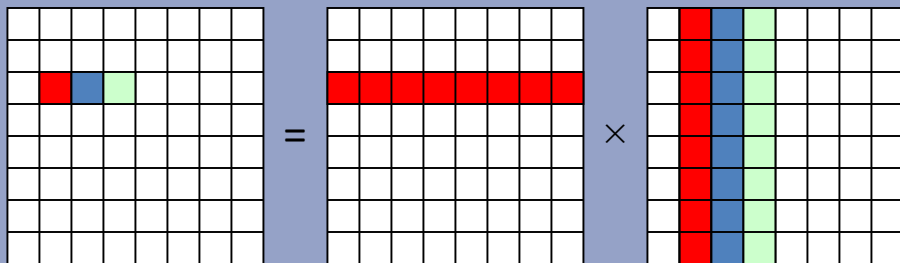
по формуле:  $C_{ik} = \sum_{j=0}^{Ny-1} A_{ij} B_{jk}$

```
for (i=0; i<Nx-1; i++)  
  for (k=0; k<Nz-1; k++)  
    for (j=0; j<Ny-1; j++)  
      C[i][k] += A[i][j] * B[j][k];
```

$C_{ik}$

$A_{ij}$

$B_{jk}$



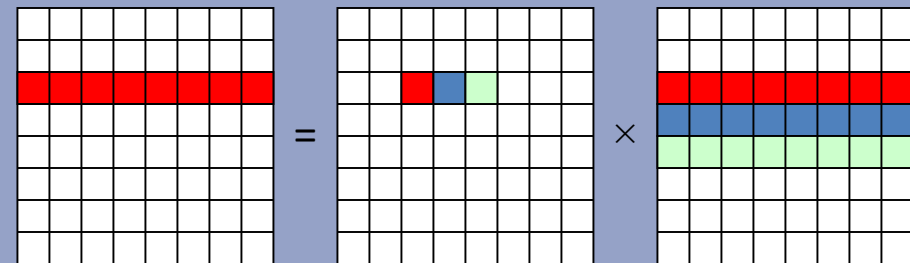
переставим циклы

```
for (i=0; i<Nx-1; i++)  
  for (j=0; j<Ny-1; j++)  
    for (k=0; k<Nz-1; k++)  
      C[i][k] += A[i][j] * B[j][k];
```

$C_{ik}$

$A_{ij}$

$B_{jk}$



Время перемножения матриц 1000×1000

120.67 с

**Alpha**

6.24 с

16.67 с

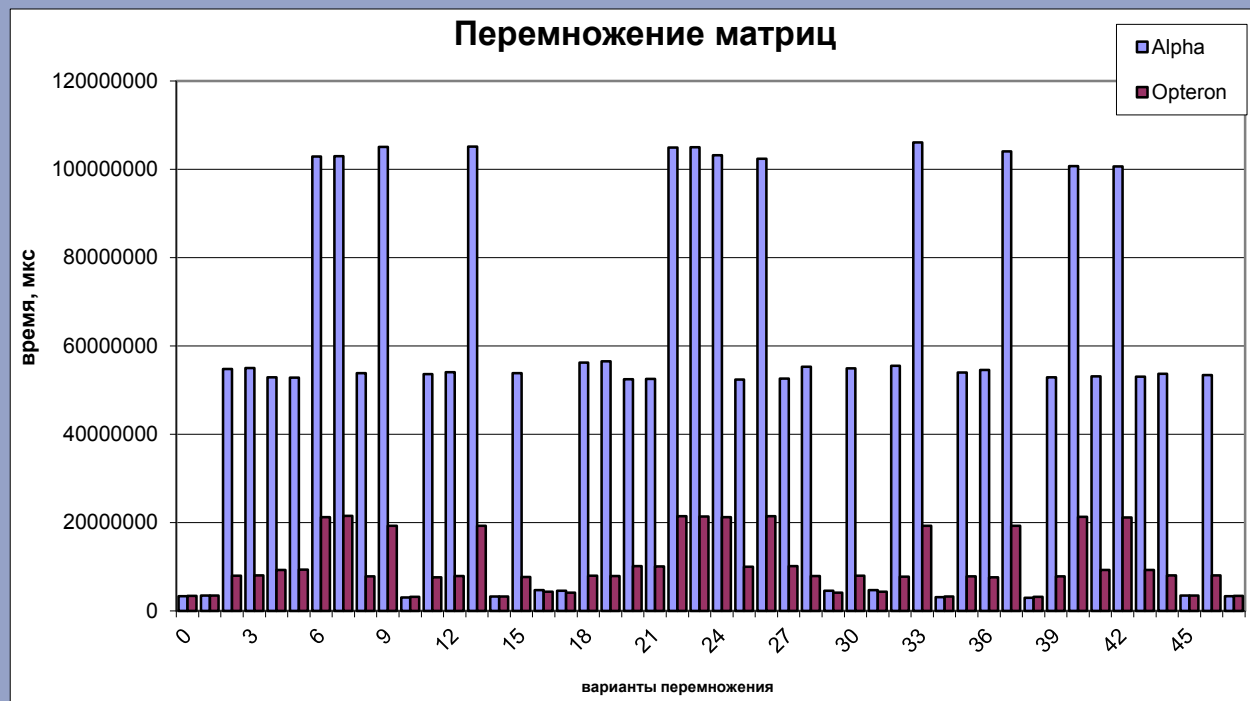
**Opteron**

6.3 с

# Порядок обхода данных

- Пример: умножение матриц

```
for (i=0; i<N; i++)  
  for (j=0; j<N; j++)  
    for (k=0; k<N; k++)  
      C[i][j] += A[i][k] * B[k][j];
```



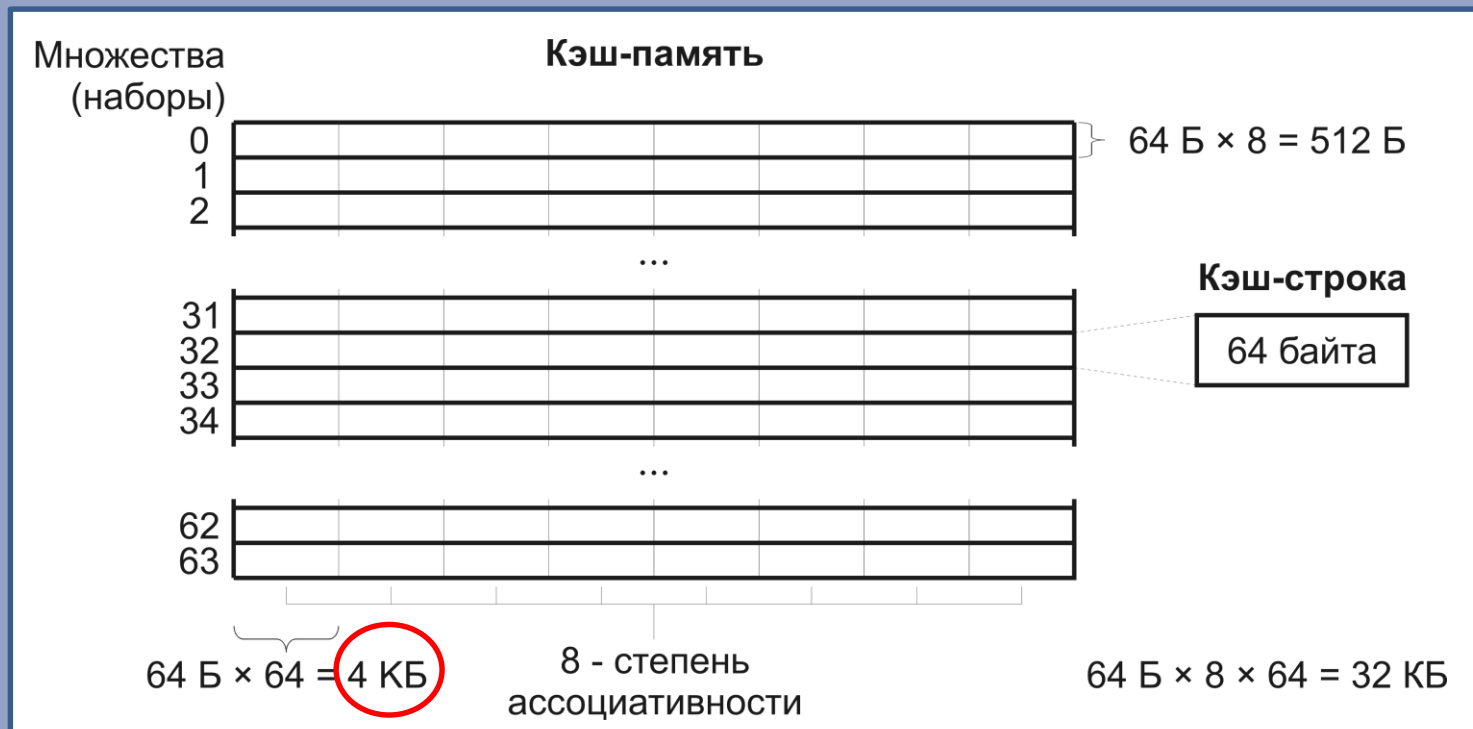
# Порядок обхода данных

- Специальный случай обхода:

кэш-буксование (cache-thrashing)

Обход элементов с шагом кратным

размеру банка = размер / степень ассоциативности



# Порядок обхода данных

- Простой пример 1 (один массив):

```
double a[4096000], sum[4096];
int i, j;
for (i=0; i<4096; i++) {
    sum[i]=0;
    for(j=0; j<1000; j++) sum[i] += a[i+j*4096];
}
```

- Простой пример 2 (несколько массивов):

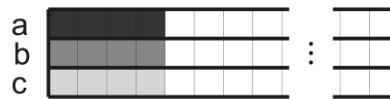
```
double a[4096], b[4096], c[4096];
int i;
for (i=0; i<4096; i++) c[i]=a[i]+b[i];
```

# Порядок обхода данных

- Стандартные способы избежать кэш-букования
  - Изменить порядок обхода
  - Изменить расстояние между элементами
- Пример:

```
double a[4096], b[4096], c[4096];  
int i;  
for (i=0; i<4096; i++) c[i]=a[i]+b[i];
```

Три массива размером 4096 элементов типа double

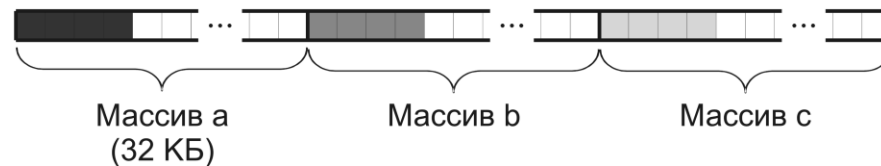


Размещение элементов массивов в кэш-памяти



Степень ассоциативности: 2  
Размер кэш-строки: 32 Б

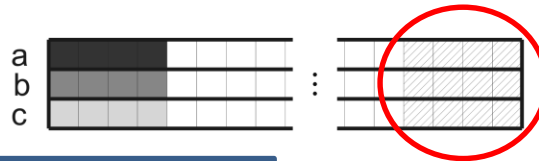
Расположение элементов массивов в памяти



# Порядок обхода данных

- Стандартные способы избежать кэш-букования
  - Изменить порядок обхода
  - Изменить расстояние между элементами
- Пример:

Три массива размером 4096 элементов типа double

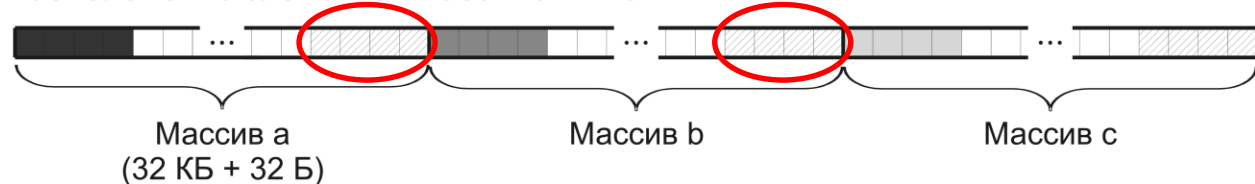


Размещение элементов массивов в кэш-памяти



Степень ассоциативности: 2  
Размер кэш-строки: 32 Б

Расположение элементов массивов в памяти

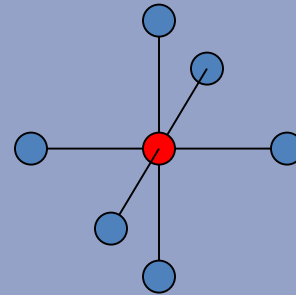
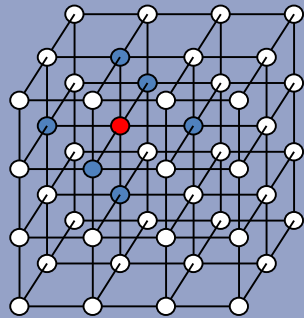


```
double a[4096 +4], b[4096 +4], c[4096 +4];  
int i;  
for (i=0; i<4096; i++) c[i]=a[i]+b[i];
```

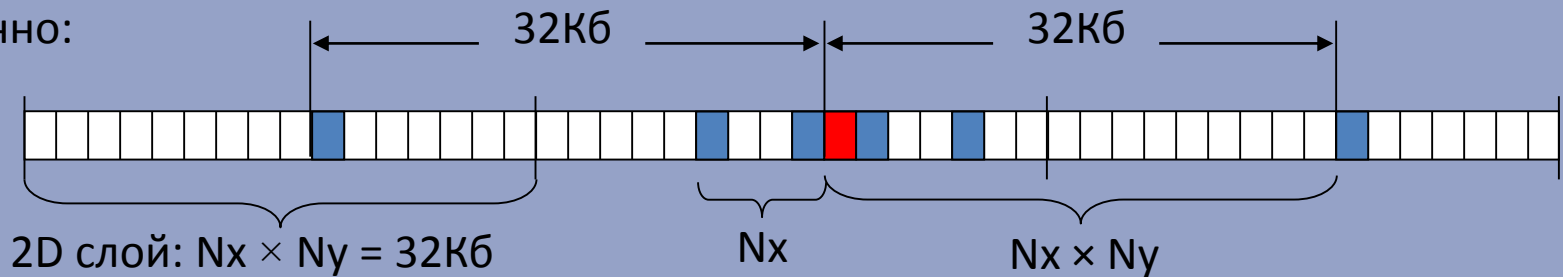
# Порядок обхода данных

Пример: «буксование» кэша – семиточечная схема

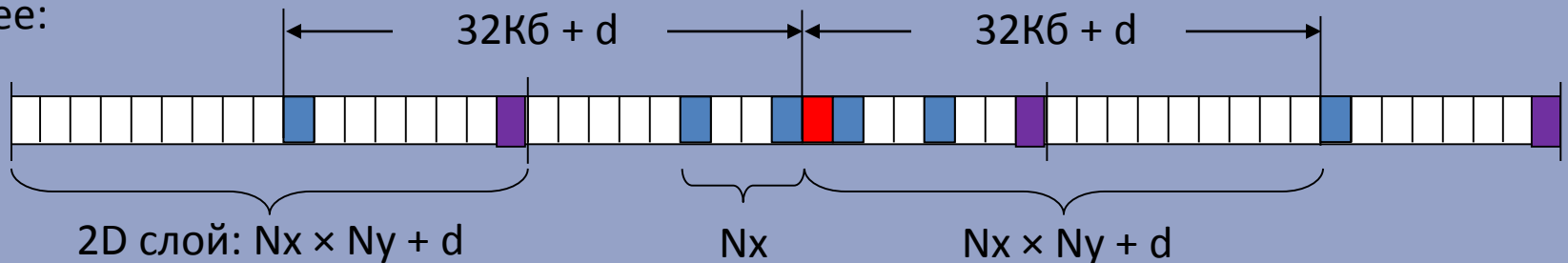
$X[Nz][Ny][Nx]$  ;



Медленно:



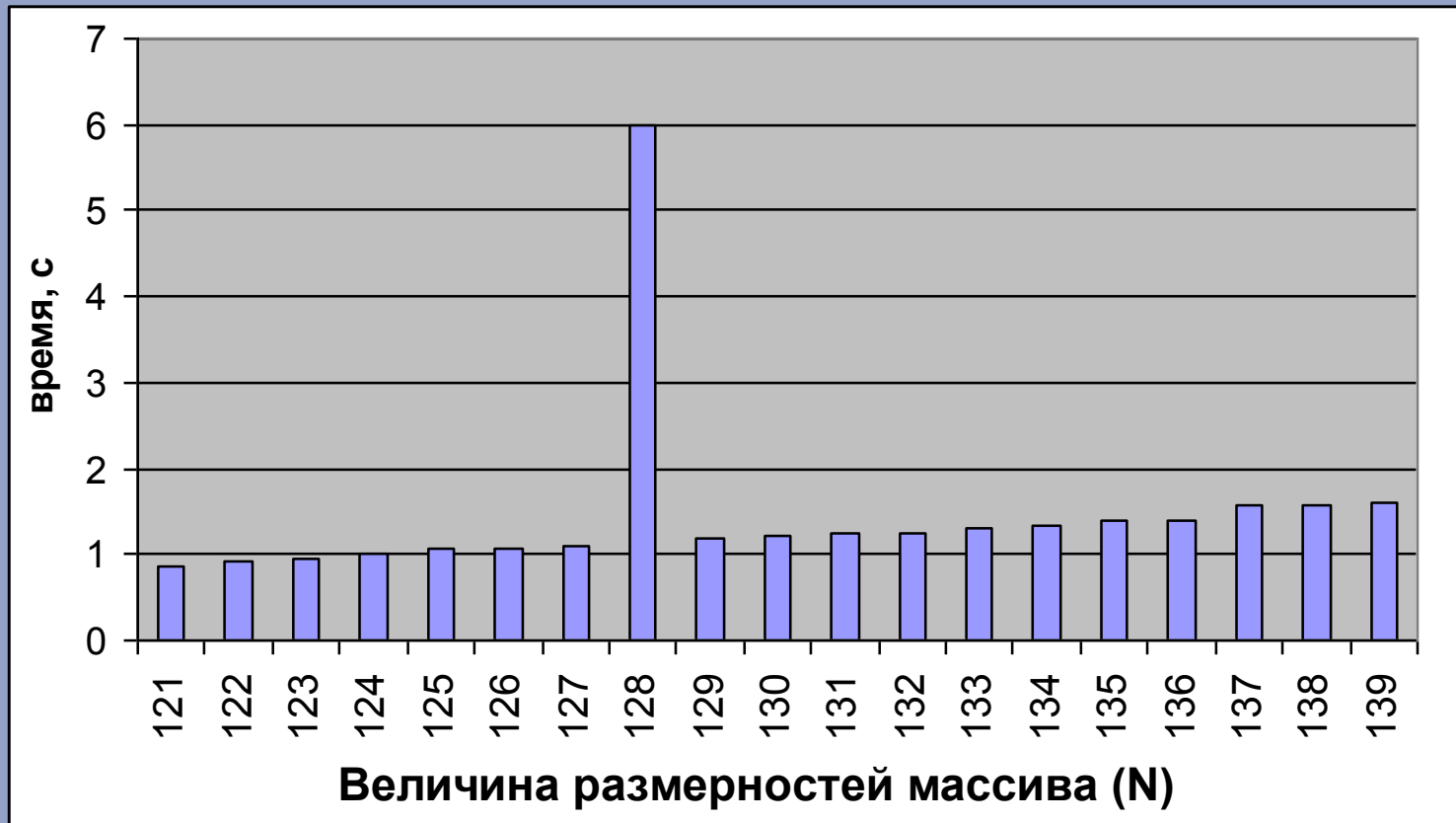
Быстрее:



# Порядок обхода данных

## Пример: «буксование» кэша – семиточечная схема

Размер 3D массива:  $N \times N \times N$



Размер 2D слоя для  $N=128$ :  $128 \times 128 = 16384$  элементов = 64 Кб

# **ДОСТУП К ПАМЯТИ НЕСКОЛЬКИХ ПОТОКОВ**

# Доступ к памяти нескольких потоков

- Совместное использование ядрами кэш-памяти
  - Доступный каждому потоку объём кэш-памяти меньше
- Совместное использование каналов доступа к памяти
  - Доступ потока к данным в памяти медленнее
- Поддержка когерентности кэш-памяти
  - Доступ потока к совместным данным медленнее
  - Возможно ложное разделение кэш-строк

# Доступ к памяти нескольких потоков

- Пример: Совместное использование канала доступа к памяти



Хеон X5365 3.0 GHz



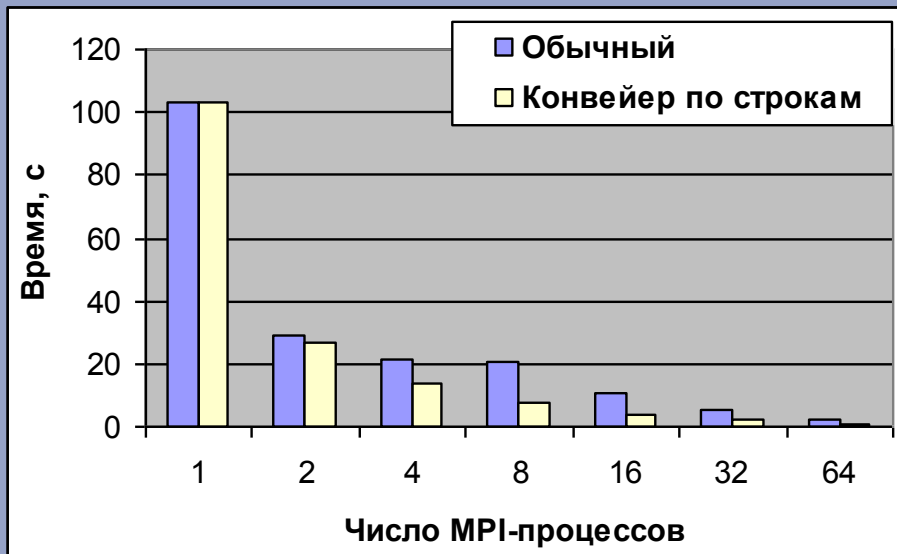
Itanium2 1.6 GHz

# Доступ к памяти нескольких потоков

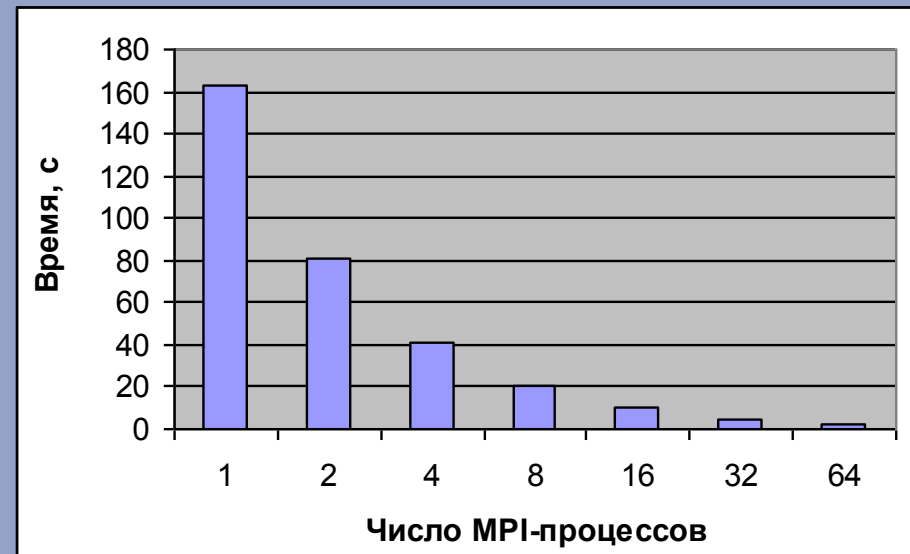
- Совместное использование канала доступа к памяти
  - Доступ потока к данным в памяти медленнее
- Решение
  - Реже обращаться в оперативную память – использовать кэш-память более эффективно (использовать блочные алгоритмы...)

# Доступ к памяти нескольких потоков

- Пример: Совместное использование канала доступа к памяти



Xeon X5365 3.0 GHz



Itanium2 1.6 GHz

# Доступ к памяти нескольких потоков

- Поддержка когерентности кэш-памяти
  - требует постоянных обновлений данных в памяти при обращении нескольких потоков к одним и тем же данным – кэш работает неэффективно

- Пример:

```
int i, s;  
#pragma omp parallel for  
for (i=0; i<N; i++)  
#pragma omp atomic  
s += i;
```

# Доступ к памяти нескольких потоков

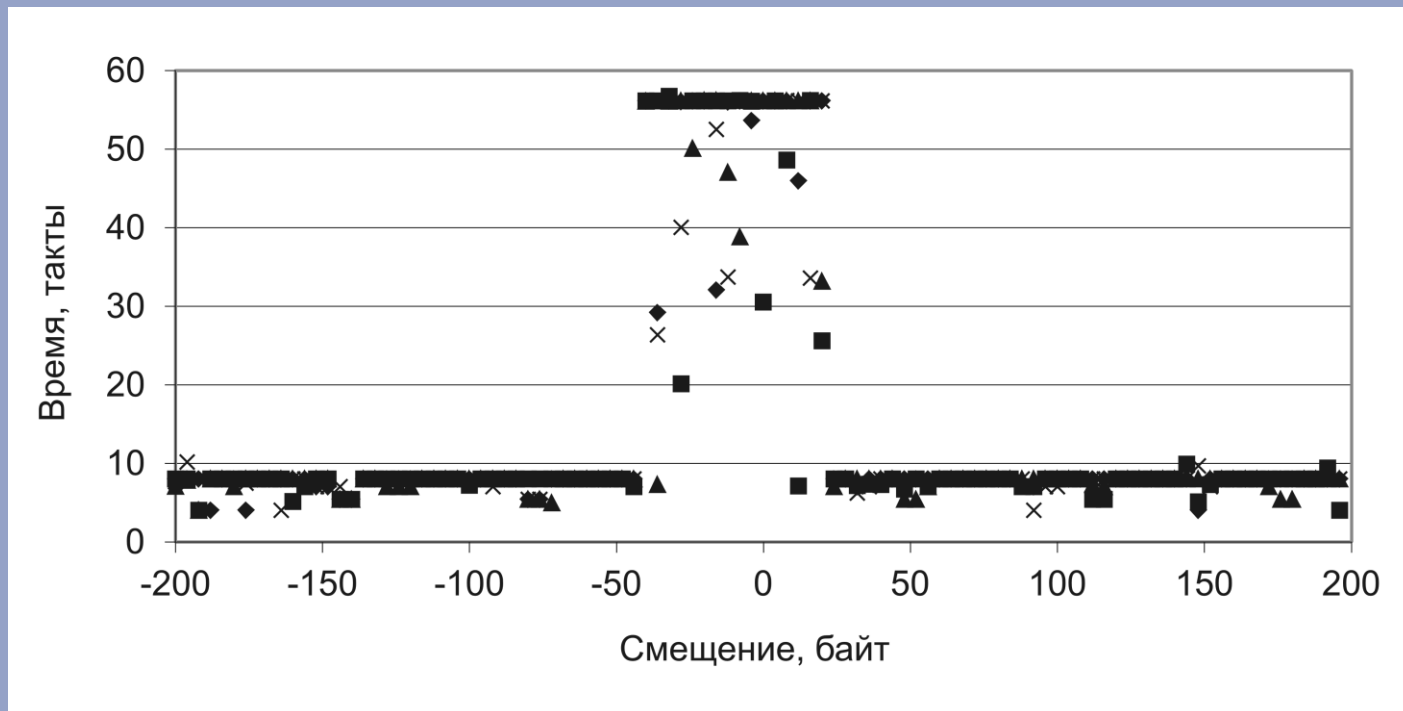
- Ложное разделение кэш-строк
  - Синхронизация данных происходит кэш-строками
  - Различные данные, размещенные в одной кэш-строке, будут тоже синхронизироваться

- Пример:

```
int i, s[nth];  
#pragma omp parallel for num_threads(nth)  
for (i=0; i<N; i++)  
#pragma omp atomic  
s[ omp_get_thread_num() ] += i;
```

# Доступ к памяти нескольких потоков

- Тест: доступ двух потоков к данным
  - Время работы одного из потоков в зависимости от расстояния между данными потоков:



# **ДОПОЛНИТЕЛЬНЫЕ СРЕДСТВА ОПТИМИЗАЦИИ ДОСТУПА К ПАМЯТИ**

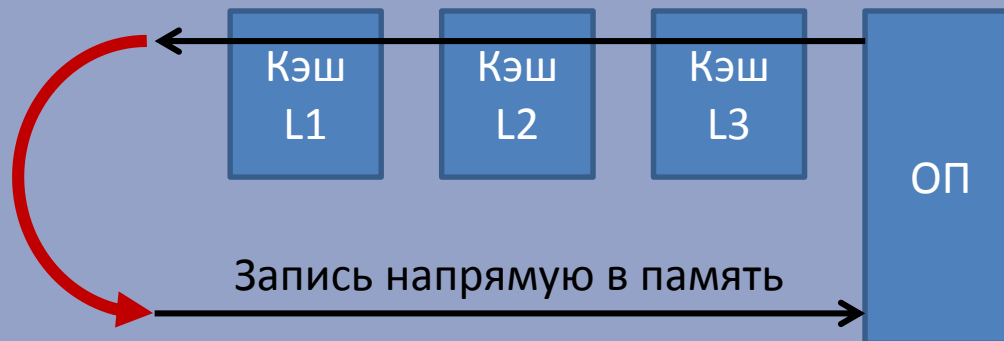


# Дополнительные средства оптимизации доступа к памяти

- Запись в память без записи в кэш  
(поточковая запись)
- Пример: копирование массива

```
for (i=0; i<N; i++)  
{  
  _mm_prefetch(&x[i+d],0);  
  _mm_stream_ps(&y[i], x[i]);  
}
```

Загрузка с предвыборкой в кэши: все каналы работают



# **ОБЗОР СРЕДСТВ АВТОМАТИЧЕСКОЙ ОПТИМИЗАЦИИ ДОСТУПА К ПАМЯТИ В INTEL C/C++ COMPILER**

# Выровненный доступ к данным средствами Intel C/C++ Compiler

## Выделение памяти с выровненным началом

- `void *_mm_malloc(int size, int base);`

## Освобождение памяти с выровненным началом

- `void _mm_free(void *addr);`

## Создание статического объекта с выровненным началом

- `__attribute__((align(64))) float x[N];`

## Указание компилятору: считать адрес выровненным

- `__assume_aligned(ptr,64);`

## Указание компилятору: считать данные в цикле [не]выровненными

- `#pragma vector [un]aligned`

# Предвыборка данных средствами Intel C/C++ Compiler

## Ключи компилятора

- `-opt-prefetch=#` – использовать оптимизацию предвыборки уровня # = 0...4
- `-no-opt-prefetch` – не использовать программную предвыборку
- `-opt-prefetch-distance=n1[,n2]` – дистанция предвыборки (число итераций) в кэш L1 [, L2] (только на Xeon Phi)

## Директивы компилятора

- `#pragma prefetch addr:hint:distance` – предвыборка с параметрами:
  - `addr`: базовый адрес массива
  - `hint`: 0 – в L1, 1 – в L2
  - `distance`: число итераций
- `#pragma noprefetch addr` – не делать предвыборку элементов данного массива

## Встроенные функции компилятора (intrinsics)

- `vprefetch0(addr); / _mm_prefetch(addr, _MM_HINT_T0);` – в L1 для чтения
- `vprefetchnta(addr); / _mm_prefetch(addr, _MM_HINT_NTA);` – в L1 для чтения (non-temp.)
- `vprefetch1(addr); / _mm_prefetch(addr, _MM_HINT_T1);` – в L2 для чтения
- `vprefetch2(addr); / _mm_prefetch(addr, _MM_HINT_T2);` – в L2 для чтения (non-temp.)
- `vprefetche0(addr); / _mm_prefetch(addr, _MM_HINT_ET0);` – в L1 для записи
- `vprefetchenta(addr); / _mm_prefetch(addr, _MM_HINT_ENTA);` – в L1 для записи (non-temp.)
- `vprefetche1(addr); / _mm_prefetch(addr, _MM_HINT_ET1);` – в L2 для записи
- `vprefetche2(addr); / _mm_prefetch(addr, _MM_HINT_ET2);` – в L2 для записи (non-temp.)

# Потоковая запись средствами Intel C/C++ Compiler

## Ключи компилятора

- `-opt-streaming-stores` `always/never/auto` – контроль потоковой записи
- `-opt-streaming-cache-evict=#` – управление вытеснением кэш-строк после потокового чтения и записи (только на Xeon Phi)
  - 0 – не вытеснять кэш-строки
  - 1 – вытеснять кэш-строки из L1
  - 2 – вытеснять кэш-строки из L2
  - 3 – вытеснять кэш-строки из L1 и L2

## Директивы компилятора

- `#pragma vector nontemporal` – выполнять потоковую запись в цикле
- `#pragma vector temporal` – не выполнять потоковую запись в цикле

# Литература

- **User and Reference Guide for the Intel® C++ Compiler 14.0**
  - Google: «[intel compiler 14 reference manual](#)» 1-я ссылка
  - [https://software.intel.com/en-us/compiler\\_14.0\\_ug\\_c](https://software.intel.com/en-us/compiler_14.0_ug_c)

