Средства и системы параллельного программирования

сентябрь – декабрь 2016 г.

Лектор доцент Н.Н.Попова

Лекция 1
12 сентября 2016 г.

Тема

- О курсе
- Исследование производительности матричного умножения

Содержание курса

- Модели параллельного программирования и их реализация.
- Базовые параллельные алгоритмы
- Методы разработки параллельных программ.
- Методы и средства анализа и настройки эффективности параллельных программ.

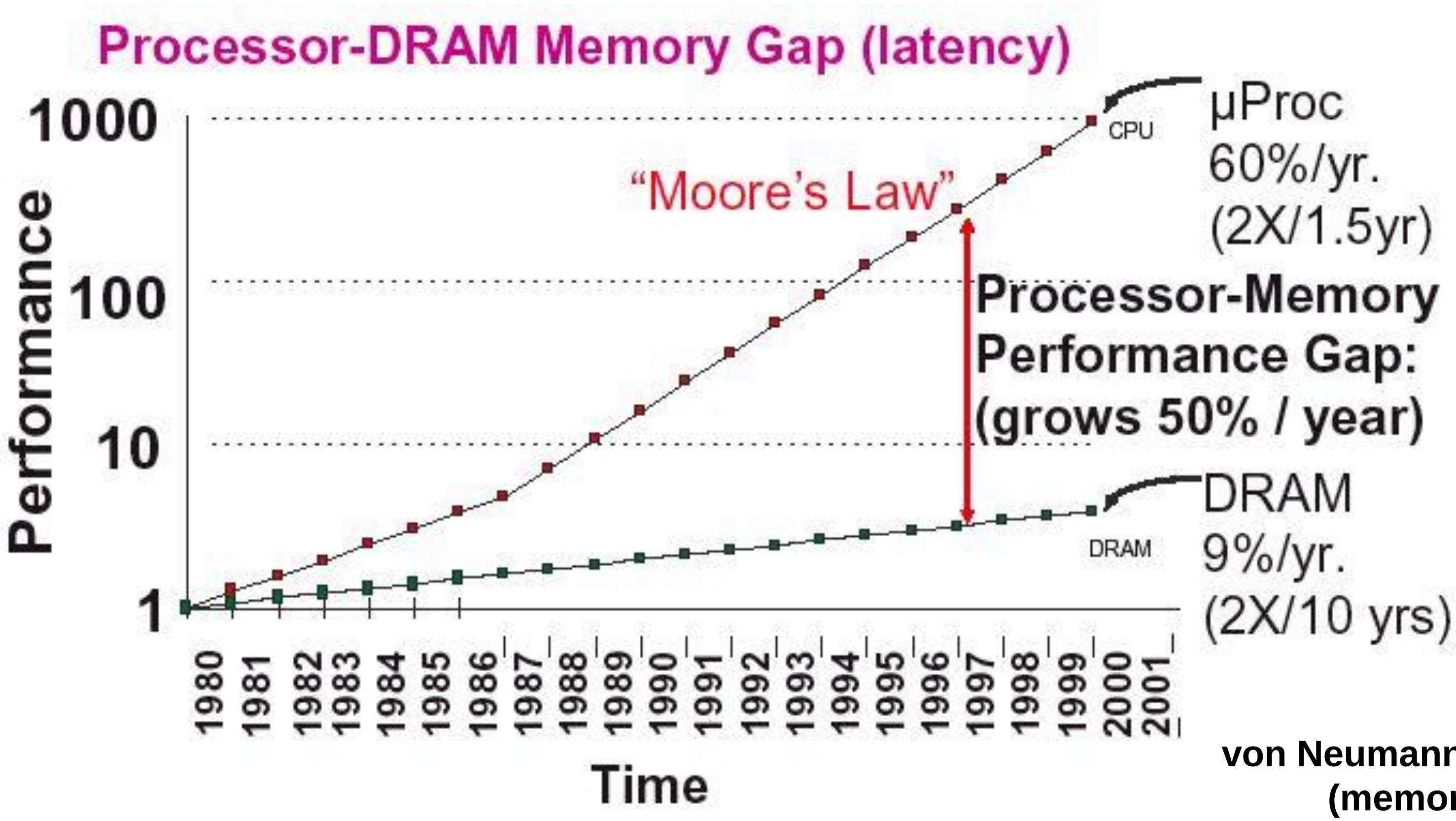
Критерии сдачи курса

- Своевременное выполнение заданий
- Экзамен
- Посещение лекций

Вычислительные системы для выполнения заданий

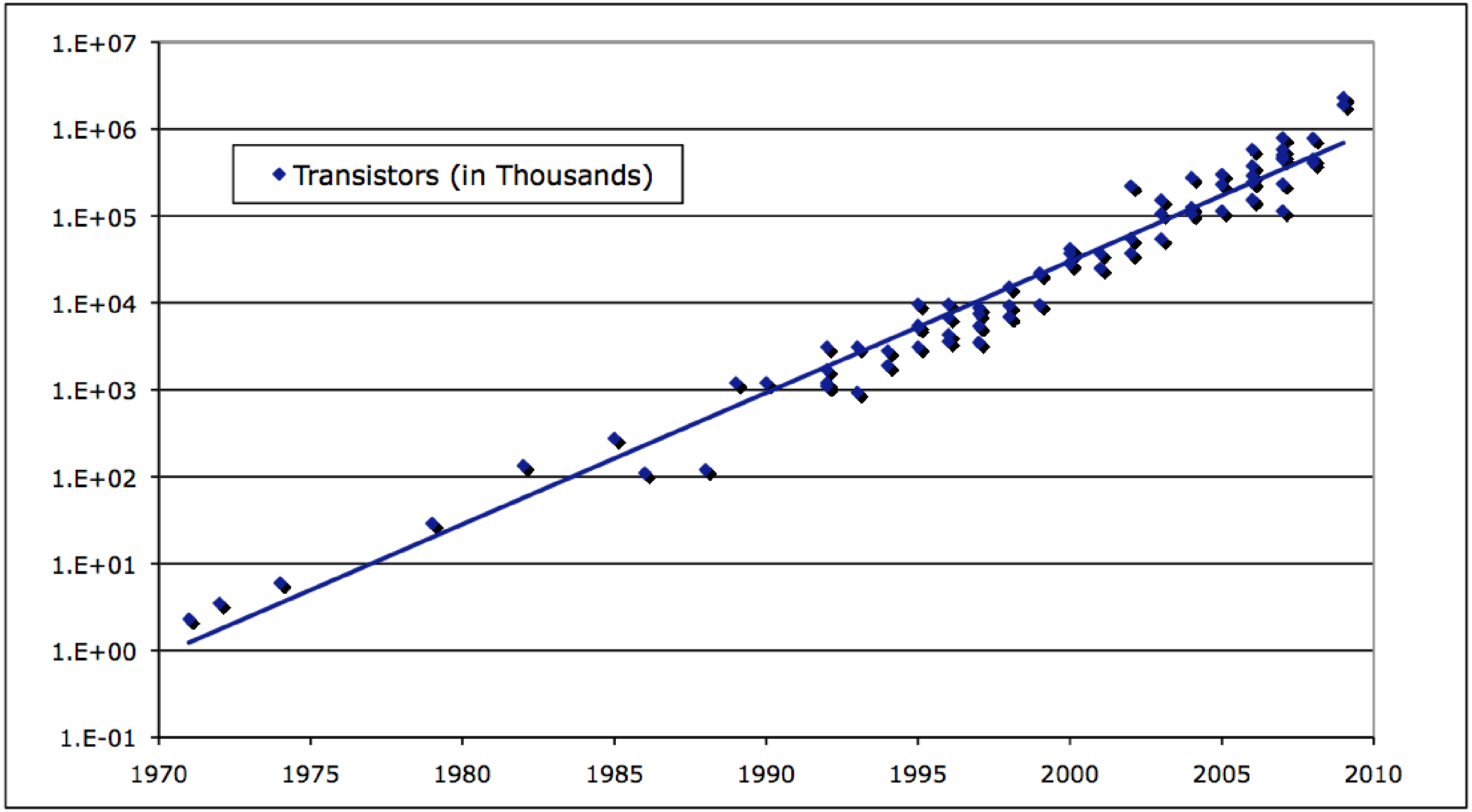
- Доступные (домашние) системы
- Регатта – regatta.cs.msu.su
- Blue Gene/P
- Ломоносов-1

Основные тенденции развития микропроцессоров



von Neumann bottleneck!!
(memory wall)

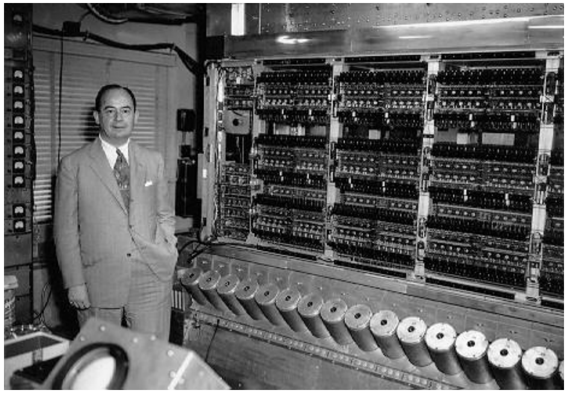
Число транзисторов в микропроцессорах (1971-2011)



Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick

Метрики производительности

- FLOPS или FLOP/S: FLoating-point Operations Per Second
  - MFLOPS: MegaFLOPS, 10^6 flops
  - GFLOPS: GigaFLOPS, 10^9 flops, home PC
  - TFLOPS: TeraGLOPS, 10^12 flops,
  - PFLOPS: PetaFLOPS, 10^15 flops, с 2011
  - EFLOPS: ExaFLOPS, 10^18 flops, ожидается 2020
  - MIPS=Mega Instructions per Second



von Neumann компьютер -- 0.00083 MIPS

Метрики производительности

- Теоретическая пиковая производительность R_theor: maximum FLOPS, которые могут быть достигнуты теоретически.
  - Clock_rate*#cpus*#FPU/CPU
  - 3GHz, 2 cpus, 1 FPU/CPU → R_theor=3x10^9 * 2 = 6 GFLOPS

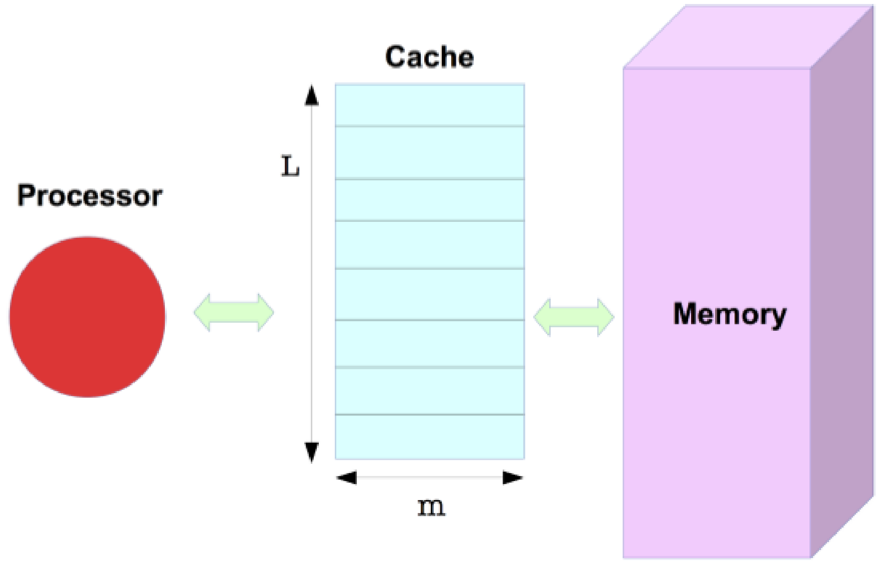
- Реальная производительность R_real: FLOPS на определенных операциях, например, векторном умножении

- Sustained performance R_sustained: производительность, полученная для конкретных приложений

R_sustained << R_real << R_theor

Типично: R_sustained < 10%R_theor

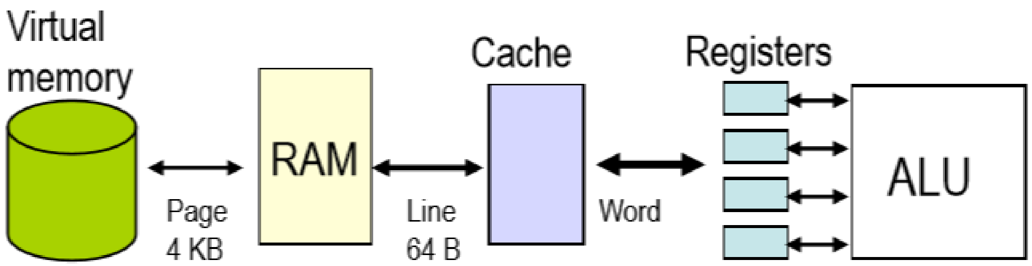
Схема последовательного компьютера



- L lines of capacity m double precision numbers each
- Tall Cache assumption : L > m

Иерархия памяти

Иерархия памяти:
1. Регистры
2. Кэш
3. ОЗУ
4. Виртуальная память



Типичное время доступа (Intel Nehalem)
- register immediately (0 clock cycles)
  - L1 cache 3 clock cycles
  - L2 cache 13 clock cycles
  - L3 cache 30 clock cycles
- memory 100 clock cycles
  - disk 100 000 – 1 000 000 clock cycles

Информация о процессоре

- /proc/cpuinfo summarizes the processor
  - vendor_id : GenuineIntel
  - model name : Intel®Xeon®CPU E5-2680 0 @ 2.70GHz
  - cache size : 20480 KB
  - cpu cores : 8
- processor : 0 through processor : 16
- Detailed information at /sys/devices/system/cpu/cpu*/cache/index*/*

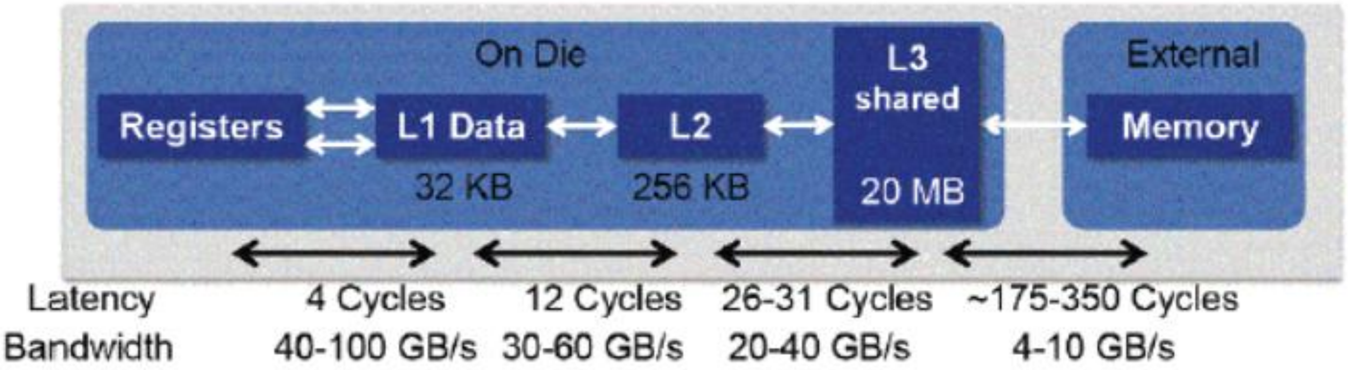
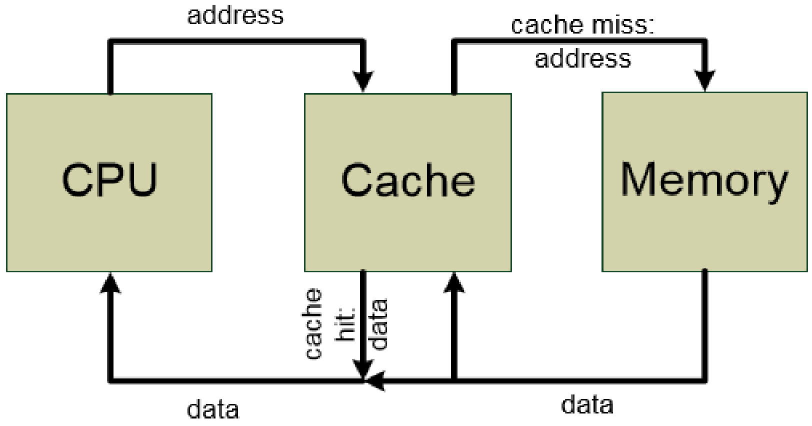


Схема обращения к памяти



Cache

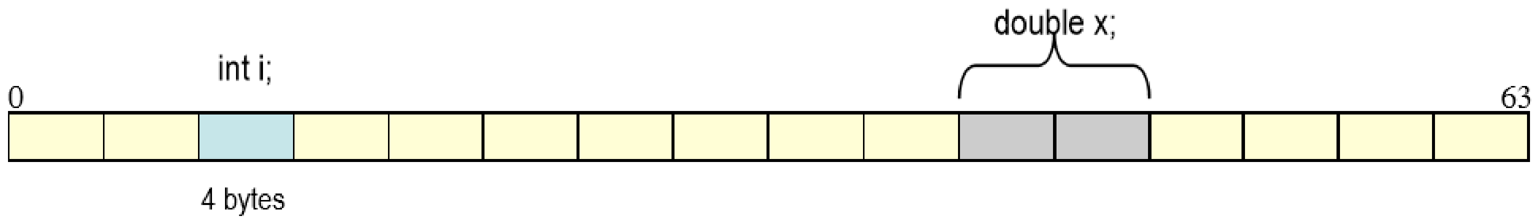
- Small, fast memory located between the processor and main memory
  - implemented by fast SRAM
  - can only store a small subset of the main memory
- Separate L1 caches for instructions and data
  - can simultaneously fetch instructions and operands
  - if the cache stores both instructions and data it is called unified
- Data in a higher memory level may also be stored in the lower levels
  - inclusive cache: data in L1 cache is also in L2 cache
  - exclusive cache: data is stored in at most one cache level
- Strategies to maintain coherence between cache and memory:
  - write-through: data is immediately written back to memory when it is updated
  - write-back: data is written to memory when a modified value is replaced in cache

Cache line

- The unit of data transferred between main memory and cache is called a cache line
  - consists of a number of consecutive memory locations
  - typical cache line size is 64 bytes
- When a memory location is accessed, the whole cache line containing the address is copied from memory to the cache
  - a cache replacement policy defines how old data in the cache is replaced with new data
  - tries to keep frequently used data in the cache
  - Least Recently Used algorithm
- For each memory access, the computer first checks if the cache line containing the memory location already is in the cache
  - if it is in cache, we have a cache hit, and the copy in cache is used
  - if not, a cache miss occurs and the cache line is read from main memory
  - reading from main memory takes a longer time than accessing data in cache

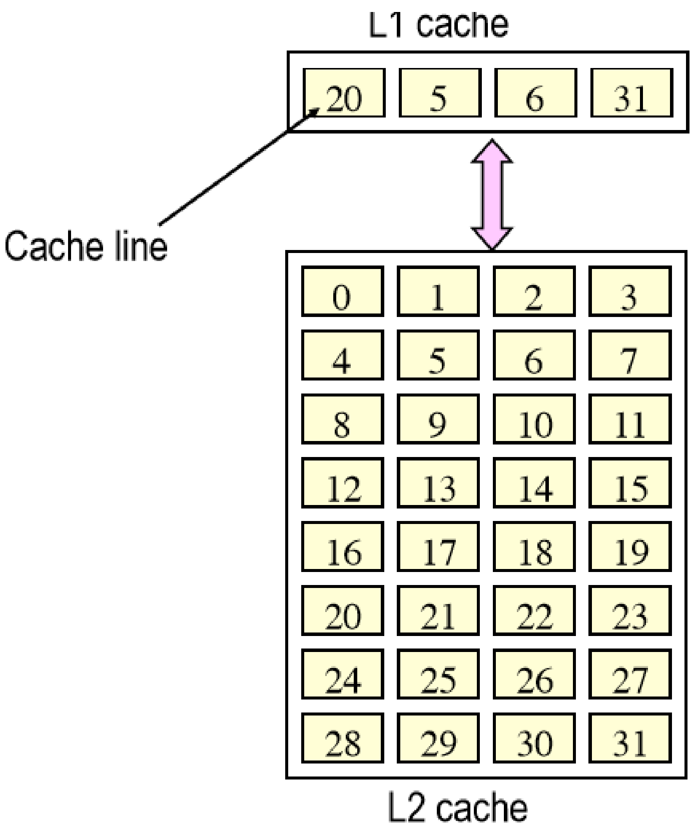
Cache line

- For each memory access, the computer first checks if the cache line containing the memory location already is in the cache
  - if it is in cache, we have a cache hit, and the copy in cache is used
  - if not, a cache miss occurs and the cache line is read from main memory
  - reading from main memory takes a longer time than accessing data in cache



L1 & L2

- Most processors have a hierarchical cache organization
  - two or three levels of cache memory
- Typical cache sizes
  - L1: 32 KB data + 32 KB instruction cache private for each core
  - L2: 1 MB, unified, private or shared
  - L3: 8 MB, unified, often shared between all cores
- Level 1 cache may contain a subset of the data cache
  - level 2 cache contains a subset of the data main memory



Промахи кэша

- When we try to locate a memory address in the cache but can not find it, a cache miss occurs
  - the corresponding cache line has to be brought in from lower levels in the memory hierarchy
  - if the cache is full, some cache line has to be evicted and replaced by the new cache line that is brought in
- There are three different reasons for cache misses (3C):
  - **C**ompulsory cache misses
  - **C**apacity cache misses
  - **C**onflict cache misses

Оптимизация программ для улучшения производительности памяти

- Efficient memory access is crucial for good performance
  - should try to design programs so memory accesses can be served from cache memory
  - if data has to be fetched from main memory, the instruction may have to stall for many clock cycles
- Data accesses in time-critical parts of a program should take advantage of the principle of locality
- Spatial locality
  - when a value is brought in to cache, a whole cache line (normally 64 bytes) is brought in at the same time
  - the program should use all the values in the cache line
- Temporal locality
  - a value that is already in cache should be reused multiple times

Доступ к памяти

- Arrange loops so that memory is accessed with unit stride

  access consecutive words in memory
- In C and C++, matrices are stored in memory in row-major order
  - the innermost loop should iterate over the last index
  - in Fortran, matrices are stored in column-major order
- Accessing consecutive memory locations uses all the data in a cache line
  - improves locality
  - can use automatic prefetching
- Accessing non-consecutive memory locations may generate a cache miss for each access

Организация доступа в память

```
for (i=0; i<rows; i++)
  for (j=0; j<cols; j++)
    X[i][j] = 0;
```

0	1	2	3	4	5	6	7
8	9	.	.				

```
for (j=0; j<cols; j++)
  for (i=0; i<rows; i++)
    X[i][j] = 0;
```

0	6						
1	7						
2	8						
3	9						
4	.						
5	.						

Cache trashing

- Watch out for large data structures with a size that is a power of 2
  - two elements with the same index in different data structures may map to the same cache set
  - in the loop all accesses to element i in the 6 arrays may be mapped to the same cache set
  - especially problematic in systems with a low cache associativity
- Causes data that is brought in to cache to be evicted immediately in the same iteration
- Can pad the structures with the cache line size
  - add 64 bytes to the size of arrays

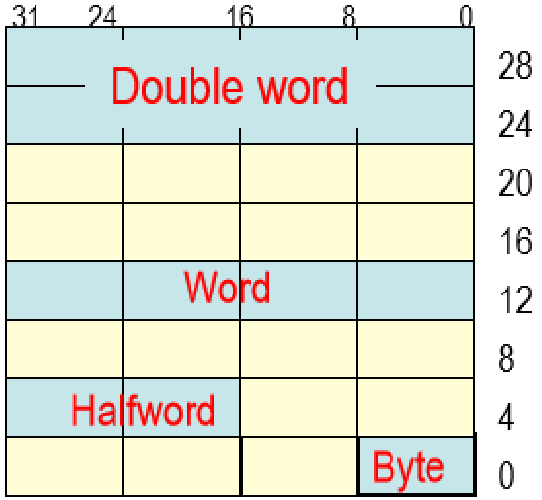
Пример предотвращения Cache trashing

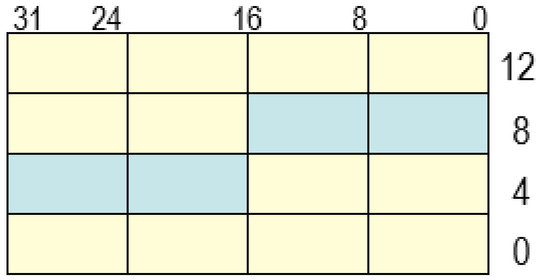
```
const int N=8*1024;
...
 double X[N], Y[N], Z[N];
  int    a[N], b[N], c[N];
 ...
for (i=0; i<N; i++) {
   X[i] = Y[i] + Z[i];  a[i] = b[i] +
c[i];
 }
```

```
const int N=8*1024 ;
...
 double X[N+8], Y[N+8], Z[N+8];
  int    a[N+16], b[N+16], c[N+16];
 ...
for (i=0; i<N; i++) {
   X[i] = Y[i] + Z[i];  a[i] = b[i] + c[i];
 }
```

Выравнивание данных в памяти (Memory alignment)

- An object of a primitive data type of size S bytes at address A is memory aligned if A mod S = 0
  - the address must be a multiple of some value k (often 2, 4, 8 or 16)
- Misaligned data
  - Example: a word located at byte offset 6
- Misaligned data causes performance degradation
  - fetching an unaligned value from memory may require two memory accesses instead of one
- Compilers automatically align data structures
  - alignment rules differ among operating systems and compilers





Объявление переменных с учетом размера представления в памяти

- Variables that are used together should be stored together
  – variables that are declared close to each other in the source code will be placed close to each other in memory
  – most likely in the same cache line
- Local variables should be declared in order of type size
  – declare variables of a largest types first and variables of smallest size last
  – the compiler allocates local variables on the stack in the order they are declared in the program
  – reduces the amount of padding the compiler has to insert in order to align the variables
  – uses the cache more efficiently

```
char c1; int i, j, k;
double cost;
```

```
double cost;
char c1; int i, j, k;
char c1;
```

Использование динамической памяти

- In C, memory is dynamically allocated with the malloc (or calloc) system function

  void * malloc(size_t SIZE)

  void * calloc(size_t COUNT, size_t ELTSIZE)

  - calloc also initializes the allocated elements to zero
  - in 64-bit systems malloc returns a 16-byte aligned block of memory
  - dynamically allocated memory is freed with free()
- Can allocate arrays where the size is not known at compile time
  - the size of the array can depend on the run-time behaviour
- Large data structures have to be allocated dynamically
  - there is a upper limit on the size of static allocations (the size of the stack)
- Dynamic memory allocation can be slow
  - often more efficient to allocate a large block for all objects then to repeatedly allocate small blocks for each object

Allocating aligned memory

- On a 64-bit system malloc (in gcc) returns a 16-byte aligned block of memory
- On a 32-bit systems, it returns a 8-byte aligned memory block
- Can also explicitly dynamically allocate aligned memory blocks with memalign
  - void *memalign(size_t boundary, size_t size);
  - does the same thing as malloc, but you can specify the alignment, which must be a power of two
  - memalign is not portable to all compilers and operating systems
- An alternative is to declare the variable with an attribute for alignment
  - int myArray[1024] __attribute__((aligned(64)));
  - however, this is not portable to all operating systems and compilers

Allocating multi-dimensional array

- Multi-dimensional arrays can be allocated statically or dynamically
  - matrices with dimension 2, 3 or higher
- Static allocation: int X[rows][cols];
  - allocates a fixed size block at compile time on the stack
- The stack has a limited size, so large matrices can not be allocated this way
  - can increase the stack size with the command % limit stacksize unlimited
  - however, the size of the stack still has an upper limit and can not be used for very large data structure
- Have to use dynamic memory allocation for large data structures

Allocating a 2-dimensional array

- Allocation as a linear one-dimensional array

```
double *M;
M = (double *) malloc(rows*cols*sizeof(double));
/* Set matrix M to zero */
for (i=0; i<rows; i++) {
   for (j=0; j<cols; j++) {
          M[i*cols+j] = 0.0;     /* Element i,j */
} }
```

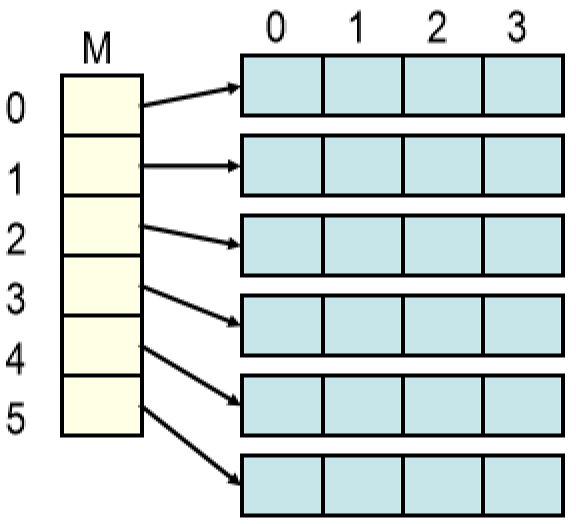
- If we don't want to calculate the address expressions explicitly in the code, we can define a macro for this

```
#define MAT(i,j) (M[i*cols+j])
. . .
for (i=0; i<rows; i++) {  for (j=0; j<cols; j++) {    MAT(i,j) = 0.0;  } }
```

2-D строчный массив

```
double **M;
M = (double **) malloc(rows*sizeof(double *));
for (i=0; i<rows; i++) {
     M[i] = (double *) malloc(cols*sizeof(double));
}
 . . .
for (i=0; i<rows; i++) {
      for (j=0; j<cols; j++) {
         M[i][j] = 0.0;  } }
```

Compulsory промах кэша при доступе к элементу строки



Альтернативный блочный метод

```
double **M; double *Mb;
M = (double **) malloc(rows*sizeof(double *));
Mb = (double *) malloc(rows*cols*sizeof(double));
 /* Set pointers to rows in the matrix */
for (i=0; i<rows; i++) {
M[i] = Mb + i*cols; }
 . . .
for (i=0; i<rows; i++) {
  for (j=0; j<cols; j++) {
      M[i][j] = 0.0;  } }
```