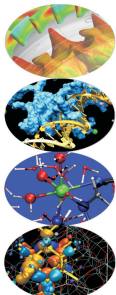


Programmazione Avanzata

Vittorio Ruggiero

(v.ruggiero@cineca.it)

Roma, Marzo 2017



- ▶ Architetture
- ▶ La cache ed il sistema di memoria
- ▶ Pipeline
- ▶ Compilatori
- ▶ Librerie Scientifiche
- ▶ Makefile
- ▶ Profiling seriale
- ▶ Profiling parallelo
- ▶ Debugging seriale
- ▶ Debugging parallelo
- ▶ Esercitazioni

► Quale linguaggio di programmazione conoscete?

Bene

Abbastanza

Per niente

Fortran

C

C++

Altro

► In quale ambiente lavorate abitualmente?

Windows

Unix/Linux

Mac

► Quali architetture conoscete?

Bene

Abbastanza

Poco

IBM

Intel

AMD

Altra

► Mai sentito parlare di...

Si Poco No

Cache

Memoria virtuale

TLB

Pipeline

Registri

► Esperienze di calcolo ad alte prestazioni

Introduction

Architectures

Cache and memory system

Matrix multiplication (time in seconds)

Precision	single	double
Incorrect Loop	7500	7300
Without optimization	206	246
With optimization (-fast)	84	181
Optimized code	23	44
Using ACML Library (serial)	6.7	13.2
Using ACML Library (2 threads)	3.3	6.7
Using ACML Library (4 threads)	1.7	3.5
Using ACML Library (8 threads)	0.9	1.8
Using Pgi accelerator	3	5
Using CUBLAS	1.6	3.2

- ▶ Write the main loop of the code and verify the obtained performances.
- ▶ Use the Fortran and/or the C code.
 - ▶ What performances have been obtained?
 - ▶ There are differences between Fortran and C codes?
 - ▶ How change the performances using different compilers?
 - ▶ And using different compilers' options?
 - ▶ Do you have a different performances changing the order of the loops?
 - ▶ Can I rewrite the loop in a more efficient mode?

- ▶ Fortran and C code
- ▶ Columns rows product $C_{i,j} = A_{i,k}B_{k,j}$
- ▶ Time:
 - ▶ Fortran: date_and_time (> 0.001")
 - ▶ C: clock (>0.05")
- ▶ Square matrices of size n
 - ▶ Required memory (double precision) $\approx (3 * n * n) * 8$
 - ▶ Number of total operations $\approx 2 * n * n * n$
 - ▶ We must access n elements of the two original matrices for each element of the destination matrix.
 - ▶ n products and n sums for each element of the destination matrix.
 - ▶ Total Flops = $2 * n^3 / Time$
- ▶ Always verify the results :-)

- ▶ Estimate the number of computational operations at execution

N_{Flop}

- ▶ 1 FLOP= 1 Floating point OPERATION (addition or multiplication).
- ▶ Division, square root, trigonometric functions require much more work and hence take more time.

- ▶ Estimate execution time T_{es}

- ▶ The number of floating operation per second is the most widely unit used to estimate the computer performances:

$$Perf = \frac{N_{Flop}}{T_{es}}$$

- ▶ The minimum count is 1 Floating-pointing Operation per second (FLOPS)
- ▶ We usually use the multiples:
 - ▶ 1 MFLOPS= 10^6 FLOPS
 - ▶ 1 GFLOPS= 10^9 FLOPS
 - ▶ 1 TFLOPS= 10^{12} FLOPS

- ▶ Directory tree
 - ▶ `src/eser_*/fortran`
 - ▶ `src/eser_*/c`

- ▶ Go to src/esor_1
- ▶ Write the main loop (columns rows product) to Fortran(mm.f90) or/and C(mm.c) code.
- ▶ Run the matrix multiplication code
- ▶ N=1024

Language	time	Mflops
Fortran		
C		

- ▶ To compile
 - ▶ make
- ▶ To clean
 - ▶ make clean
- ▶ To change the compiler options
 - ▶ make "FC=ifort"
 - ▶ make "CC=icc"
 - ▶ make "OPT=fast"
- ▶ To compile using single precision arithmetic
 - ▶ make "FC=ifort -DSINGLEPRECISION"
- ▶ To compile using double precision arithmetic
 - ▶ make "FC=ifort"

```
processor           : 0
vendor_id          : GenuineIntel
cpu family         : 6
model              : 37
model name         : Intel(R) Core(TM) i3 CPU           M 330    @ 2.13GHz
stepping           : 2
cpu MHz            : 933.000
cache size         : 3072 KB
physical id        : 0
siblings           : 4
core id            : 0
cpu cores          : 2
apicid             : 0
initial apicid     : 0
fpu                : yes
fpu_exception      : yes
cpuid level        : 11
wp                 : yes
wp                 : yes
flags              : fpu vme de pse tsc msr pae mce cx8
bogomips           : 4256.27
clflush size      : 64
cache_alignment    : 64
address sizes      : 36 bits physical, 48 bits virtual
...
```

```
Architecture:          x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
CPU(s):               4
On-line CPU(s) list:  0-3
Thread(s) per core:   2
Core(s) per socket:   2
CPU socket(s):        1
NUMA node(s):         1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 37
Stepping:              2
CPU MHz:               933.000
BogoMIPS:              4255.78
Virtualization:       VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
NUMA node0 CPU(s):    0-3
```

What do you think about the obtained results?

Problem



Algorithms



Source code

```
#include <stdio.h>
#define SIZE 1000
main(int argc, char** argv) {
  int A[SIZE], B[SIZE], C[SIZE];
  int i;

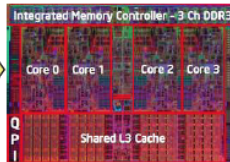
  for (i=0; i<SIZE; i++) {
    B[i] = i;
    C[i] = SIZE-i;
  }

  /* Add B and C */
  for (i=0; i<SIZE; i++) {
    A[i] = B[i]+C[i];
  }
}
```



Compiled and optimized code

```
.globl main
.type main,@function
main:
  pushl %ebp
  movl %esp,%ebp
  subl $12004,%esp
  pushl %edi
  pushl %eax
  pushl %ebx
  movl $0,-12004(%ebp)
.L2:
  cmpl $999,-12004(%ebp)
  jle .L4
  jmp .L3
.L4:
  movl -12004(%ebp),%eax
  movl %eax,%edx
  leal 0f,%edx,4,%eax
  leal -4000(%ebp),%ecx
  movl -12004(%ebp),%ecx
  movl %ecx,%ebx
  leal 0f,%ebx,4,%ecx
  leal -8000(%ebp),%ebx
  movl -12004(%ebp),%eax
  movl %eax,%edx
  leal 0f,%edx,4,%eax
  leal -12000(%ebp),%eax
  movl (%ecx,%ebx),%ecx
  imull (%eax,%edx),%ecx
  movl %ecx,(%eax,%ebx)
.L4:
  incl -12004(%ebp)
  jmp .L2
.L3:
  leal -12016(%ebp),%esp
  popl %ebx
.L5:
  .size main,.L5-.main
.ident "GCC: (GNU) 2.8.1"
```



Output

Hierarchical code: Flummor model

tbody	dtime	qps	thats	usage	stout	tstop
1024	0.0315	0.050	1.00	5.350	0.350	2.000
tnow	T=U	Z/U	ntot	nbgv	ncvrg	cpuime
0.000	-0.1617	-0.4943	203185	84	114	0.00
cm pos	0.0000	-0.0000	0.0000			
cm val	-0.0000	0.0000	0.0000			
an vec	0.0037	0.0135	-0.0222			
tnow	T=U	Z/U	ntot	nbgv	ncvrg	cpuime
0.031	-0.1617	-0.4940	202260	81	116	0.01
cm pos	0.0000	-0.0000	0.0000			
cm val	0.0000	-0.0000	0.0000			
an vec	0.0037	0.0135	-0.0222			

Time is: 9.4 seconds

- ▶ A problem can typically be solved in many different ways
 - ▶ we have to choose a correct and efficient solution method
- ▶ A solution may include many different stages of computation using different algorithms
 - ▶ Example : first solve a linear equation system, then do a matrix multiplication and after that a FFT
- ▶ Each stage in the solution may operate on the same data
 - ▶ the data representation should be well suited for all the stages of the computation
 - ▶ different stages in the solution may have conflicting requirements on how data is represented

- ▶ A specific problem can typically be solved using a number of different algorithms
- ▶ The algorithm has to
 - ▶ be correct
 - ▶ give the required numerical accuracy
 - ▶ be efficient, both with respect to execution time and use of memory
- ▶ The choice of the numerical algorithm significantly affects the performances.
 - ▶ efficient algorithm → good performances
 - ▶ inefficient algorithm → bad performances
- ▶ Good performances are related to the choice of the algorithm.
- ▶ Golden rule
 - ▶ Before writing code choose an efficient algorithm: otherwise, the code must be rewritten!!!!

- ▶ We program in high level languages
 - ▶ C,C++,Fortran,Java Python
- ▶ To achieve best performances, languages which are compiled to executable machine code are preferred (C,C++,Fortran,..)
 - ▶ the differences in efficiency between these depend mainly on how well developed the compiler is, not so much on the languages themselves
- ▶ Interpreted languages, and languages based on byte code are in general less efficient (Python, Java, JavaScript, PHP, ...)
 - ▶ the program code is not directly executed by the processor, but goes instead through a second step of interpretation
- ▶ High-level code is translated into machine code by a compiler
- ▶ the compiler transforms the program into an equivalent but more efficient program
- ▶ the compiler must ensure that the optimized program implements exactly the same computation as the original program

- ▶ The compiler analyzes the code and tries to apply a number of optimization techniques to improve the performance
 - ▶ it tries to recognize code which can be replaced with equivalent, but more efficient code
- ▶ Modern compilers are very good at low-level optimization
 - ▶ register allocation, instruction reordering, dead code removal, common subexpression elimination, function inlining , loop unrolling, vectorization, ...
- ▶ To enable the compiler to analyze and optimize the code the programmer should:
 - ▶ avoid using programming language constructs which are known to be inefficient
 - ▶ avoid programming constructs that are difficult for the compiler to optimize (optimization blockers)
 - ▶ avoid unnecessarily complicated constructs or tricky code, which makes the compiler analysis difficult
 - ▶ write simple and well-structured code, which is easy for the compiler to analyze and optimize

- ▶ Modern processors are very complex systems
 - ▶ superscalar, superpipelined architecture
 - ▶ out of order instruction execution
 - ▶ multi-level cache with pre-fetching and write-combining
 - ▶ branch prediction and speculative instruction execution
 - ▶ vector operations
- ▶ It is very difficult to understand exactly how instructions are executed by the processor
- ▶ Difficult to understand how different alternative program solutions will affect performance
 - ▶ programmers often have a weak understanding of what happens when a program is executed

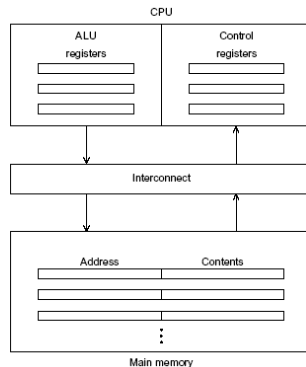
- ▶ Find out where the program spends most of its time
 - ▶ it is unnecessary to optimize code that is seldom executed
- ▶ The 90/10 rule
 - ▶ a program spends 90% of its time in 10% of the code
 - ▶ look for optimizations in this 10% of the code
- ▶ Use tools to find out where a program spends its time
 - ▶ profilers
 - ▶ hardware counters

Introduction

Architectures

Cache and memory system

- ▶ Central processing unit (CPU)
 - ▶ Arithmetic Logic Unit (performs all arithmetic computations)
 - ▶ Control Unit
 - ▶ Registers (fast memory)
- ▶ Interconnection (Bus)
- ▶ Random Access Memory (RAM)
 - ▶ Address to access the memory locations
 - ▶ Data contents (instructions, data)



- ▶ Data are transferred from memory to CPU (fetch or read instruction)
- ▶ Data are transferred from CPU to memory (written to memory or stored)
- ▶ Von Neumann Architectures carry out instructions one after another, in a single linear sequence, and they spend a lot of time moving data to and from memory. This slows the computer
- ▶ The difficulty of overcoming the disparity between processor speeds and data access speeds is called Von Neumann bottleneck.
- ▶ The modern CPU are able to perform the instructions at least one hundred times faster than the time required to recover data from the RAM (fetch instruction).

The solution for the von Neumann bottleneck are:

- ▶ Caching

Very fast memories that are physically located on the chip of the processor.

There are multi levels of cache (first, second and third).

- ▶ Virtual memory

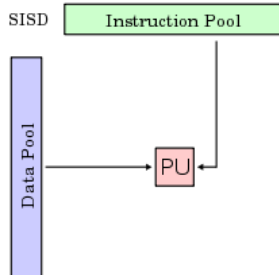
The RAM works as a cache to store large amounts of data.

- ▶ Instruction level parallelism

A family of processor and compiler design techniques that speed up execution by causing individual machine operations to execute in parallel (pipelining ,multiple issue).

- ▶ A classification of computer architectures based on instructions and data streams.
 - ▶ **SISD**: single instruction, single data. Traditional Von Neumann architecture, scalar uniprocessor system.
 - ▶ **SIMD**: single instruction, multiple data. Vector architectures, Vector processors, GPU.
 - ▶ **MISD**: multiple instruction, single data. Does not exist.
 - ▶ **MIMD**: multiple instruction, multiple data. Different processors/cores may be executing different instructions on different pieces of data or run multiple independent programs at the same time.
- ▶ The modern architectures are a mixed of these classes.

- ▶ Computers with only one executive unit and one memory. At one time, one instruction operates on one data.
- ▶ Good performances can be achieved increasing the bus data and the levels of memory or by pipelining and multiple issues.

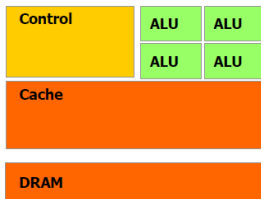


- ▶ The same instruction is executed synchronously on different sets of data.
 - ▶ Model for synchronous computation.
- ▶ Vector processors.
 - ▶ Many ALUs
 - ▶ vector registers
 - ▶ vector Load/Store Units
 - ▶ vector instructions
 - ▶ interleaved memory
 - ▶ OpenMP, MPI
- ▶ Graphical Processing Unit
 - ▶ GPU fully programmable
 - ▶ many ALUs
 - ▶ many Load/Store units
 - ▶ many SFUs
 - ▶ many thousands of parallel threads
 - ▶ CUDA

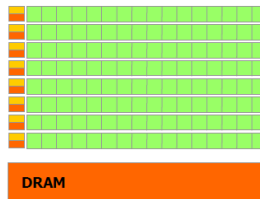
- ▶ Multiple autonomous processors simultaneously executing different instructions on different data
 - ▶ Model for asynchronous computation
- ▶ Cluster
 - ▶ a large number of compute nodes (hundreds, thousands)
 - ▶ Many nodes with many multicore processors.
 - ▶ Shared memory on a node
 - ▶ Distributed memory between nodes
 - ▶ Multi-level memory hierarchies
 - ▶ OpenMP, MPI, MPI+OpenMP

- ▶ Hybrid solution CPU multi-core + GPU many-core:
 - ▶ Nodes with multicore processors and dedicated graphics cards for GPU computing.
 - ▶ High theoretical peak performance on single node
 - ▶ Additional memory provided from the GPU.
 - ▶ OpenMP, MPI, CUDA and hybrid solution MPI+OpenMP, MPI+CUDA, OpenMP+CUDA, OpenMP+MPI+CUDA

- ▶ The CPU is a general purpose processor that is able to execute all the algorithms.
 - ▶ 1 thread per computational node
- ▶ The GPU is a processor dedicated to intense data-parallel computations
 - ▶ Many light parallel threads.

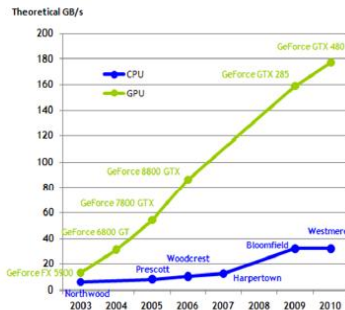
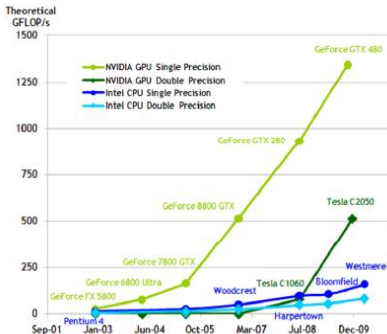


CPU



GPU

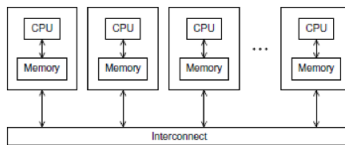
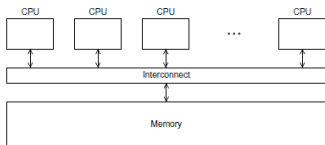
- ▶ The increasing number of transistors on a chip can be used to place more than one processor core on a chip
- ▶ The new era for the architecture of the microprocessors:
 - ▶ Increasing the computing power by increasing the number of units more than their power.
 - ▶ The computing power and bandwidth of the GPU have overtaken that of CPU by a factor 10.



- ▶ It is the rate at which data can be read from or stored into the memory by a processor.
- ▶ It is expressed in units of bytes/second (Mb/s, Gb/s, etc..)
- ▶ $A = B * C$
 - ▶ Read B data from the memory
 - ▶ Read C data from the memory
 - ▶ Calculate $B * C$ product
 - ▶ Save the result in memory (A variable)
- ▶ 1 floating-point operation \rightarrow 3 memory accesses

- ▶ It is a simple, synthetic benchmark designed to measure sustainable memory bandwidth (in MB/s) and a corresponding computation rate for four simple vector kernels
 - ▶ Copy $a \rightarrow c$ (copy)
 - ▶ Copy $a*b \rightarrow c$ (scale)
 - ▶ Sum $a+b \rightarrow c$ (add)
 - ▶ Sum $a+b*c \rightarrow d$ (triad)
- ▶ <http://www.cs.virginia.edu/stream/ref.html>

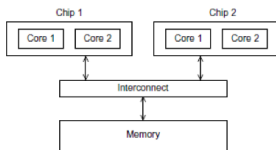
- ▶ The MIMD architectures and hybrid CPU GPU architectures can be divided in two classes.
 - ▶ Shared memory systems where every single core access the full memory
 - ▶ Distributed memory systems where different CPU units have their own memory systems and are able to communicate with each other by exchanging explicit messages



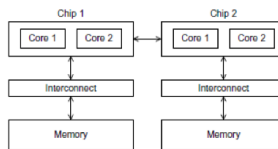
- ▶ Modern multicore systems have shared memory on node and distributed memory between nodes.

The shared memory architectures with multicore processors has two kinds of access to the main memory.

- ▶ **Uniform Memory Access** all the processors in the UMA model share the physical memory uniformly
- ▶ **Non Uniform Memory Access** a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors)



UMA



NUMA

- ▶ Main disadvantages:
 - ▶ UMA machines: Thread synchronization and accessing shared resources can cause the code to execute serially, and possibly produce bottlenecks. For example, when multiple processors use the same bus to access the memory, the bus can become saturated.
 - ▶ NUMA machines: the increased cost of accessing remote memory over local memory can affect performances.
- ▶ Solutions:
- ▶ Software can maximize performance by increasing usage of local memory
 - ▶ binding to keep processes , or threads on a particular processor.
 - ▶ memory affinity
 - ▶ on AIX architecture, set MEMORY_AFFINITY environment variable.
 - ▶ where is supported use numactl command.

- ▶ All high performance computer systems are clusters of nodes with shared memory on node and distributed memory between nodes
 - ▶ A cluster must have multiple network connections between nodes, forming cluster interconnect.
- ▶ The more commonly used network communications protocols:
 - ▶ Gigabit Ethernet : the more common, cheap, low performances.
 - ▶ Infiniband : very common, high performances, very expansive.
- ▶ Others:
 - ▶ Myrinet
 - ▶ Quadrics
 - ▶ Cray

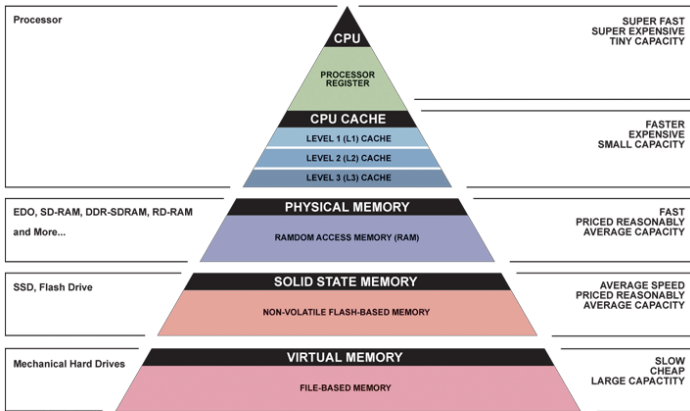
Introduction

Architectures

Cache and memory system

- ▶ CPU power computing doubles every 18 months
- ▶ Access rate to RAM doubles every 120 months
- ▶ Reducing the cost of the operations is useless if the loading data is slow

- ▶ Solution: intermediate fast memory layers
- ▶ A Hierarchical Memory System
- ▶ The hierarchy is transparent to the application but the performances are strongly enhanced



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

- ▶ The speed of a computer processor, or CPU, is determined by the clock cycle, which is the amount of time between two pulses of an oscillator.
- ▶ Generally speaking, the higher number of pulses per second, the faster the computer processor will be able to process information
- ▶ The clock speed is measured in Hz, typically either megahertz (MHz) or gigahertz (GHz). For example, a 4GHz processor performs 4,000,000,000 clock cycles per second.
- ▶ Computer processors can execute one or more instructions per clock cycle, depending on the type of processor.
- ▶ Early computer processors and slower CPUs can only execute one instruction per clock cycle, but modern processors can execute multiple instructions per clock cycle.

- ▶ From small, fast and expensive to large, slow and cheap
- ▶ Access times increase as we go down in the memory hierarchy
- ▶ Typical access times (Intel Nehalem)
 - ▶ register immediately (0 clock cycles)
 - ▶ L1 3 clock cycles
 - ▶ L2 13 clock cycles
 - ▶ L3 30 clock cycles
 - ▶ memory 100 clock cycles
 - ▶ disk 100000 - 1000000 clock cycles

Why this hierarchy?

Why this hierarchy?

It is not necessary that all data are available at the same time.

What is the solution?

Why this hierarchy?

It is not necessary that all data are available at the same time.

What is the solution?

- ▶ The cache is divided in one (or more) levels of intermediate memory, rather fast but small sized (kB ÷ MB)
- ▶ Basic principle: we always work with a subset of data.
 - ▶ data needed → fast memory access
 - ▶ data not needed (for now) → slower memory levels
- ▶ Limitations
 - ▶ Random access without reusing
 - ▶ Never large enough . . .
 - ▶ faster, hotter and . . . expensive → intermediate levels hierarchy.

- ▶ CPU accesses higher level cache:
- ▶ The cache controller finds if the required element is present in cache:
 - ▶ **Yes**: data is transferred from cache to CPU registers
 - ▶ **No**: new data is loaded in cache; if cache is full, a replacement policy is used to replace (a subset of) the current data with the new data
- ▶ The data replacement between main memory and cache is performed in data chunks, called **cache lines**
- ▶ **block** = The smallest unit of information that can be transferred between two memory levels (between two cache levels or between RAM and cache)



- ▶ Spatial locality
 - ▶ High probability to access memory cell with contiguous address within a short period of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)



- ▶ Spatial locality
 - ▶ High probability to access memory cell with contiguous address within a short period of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)
 - ▶ Possible advantage: we read more data than we need (complete block) in hopes of next request



- ▶ Spatial locality
 - ▶ High probability to access memory cell with contiguous address within a short period of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)
 - ▶ Possible advantage: we read more data than we need (complete block) in hopes of next request

- ▶ Temporal locality
 - ▶ High probability to access memory cell that was recently accessed within a period space of time (instructions within body of cycle frequently and sequentially accessed, etc.)

- ▶ Spatial locality
 - ▶ High probability to access memory cell with contiguous address within a short period of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)
 - ▶ Possible advantage: we read more data than we need (complete block) in hopes of next request

- ▶ Temporal locality
 - ▶ High probability to access memory cell that was recently accessed within a period space of time (instructions within body of cycle frequently and sequentially accessed, etc.)
 - ▶ We take advantage replacing the least recently used blocks

- ▶ Spatial locality
 - ▶ High probability to access memory cell with contiguous address within a short period of time (sequential instructions; data arranged in matrix and vectors sequentially accessed, etc.)
 - ▶ Possible advantage: we read more data than we need (complete block) in hopes of next request
- ▶ Temporal locality
 - ▶ High probability to access memory cell that was recently accessed within a period space of time (instructions within body of cycle frequently and sequentially accessed, etc.)
 - ▶ We take advantage replacing the least recently used blocks

Data required from CPU are stored in the cache with contiguous memory cells as long as possible

- ▶ **Hit**: The requested data from CPU is stored in cache
- ▶ **Miss**: The requested data from CPU is not stored in cache
- ▶ **Hit rate**: The percentage of all accesses that are satisfied by the data in the cache.
- ▶ **Miss rate**:The number of misses stated as a fraction of attempted accesses (miss rate = 1-hit rate).
- ▶ **Hit time**: Memory access time for cache hit (including time to determine if hit or miss)
- ▶ **Miss penalty**: Time to replace a block from lower level, including time to replace in CPU (mean value is used)
- ▶ **Miss time**: = miss penalty + hit time, time needed to retrieve the data from a lower level if cache miss is occurred.

Level	access cost
L1	1 clock cycle
L2	7 clock cycles
RAM	36 clock cycles

- ▶ 100 accesses with 100% cache hit: $\rightarrow t=100$
- ▶ 100 accesses with 5% cache miss in L1: $\rightarrow t=130$
- ▶ 100 accesses with 10% cache miss in L1 $\rightarrow t=160$
- ▶ 100 accesses with 10% cache miss in L2 $\rightarrow t=450$
- ▶ 100 accesses with 100% cache miss in L2 $\rightarrow t=3600$

1. search two data, A and B
2. search A in the first level cache (L1) $O(1)$ cycles
3. search A in the second level cache (L2) $O(10)$ cycles
4. copy A from RAM to L2 to L1 to registers $O(10)$ cycles
5. search B in the first level cache (L1) $O(1)$ cycles
6. search B in the second level cache (L2) $O(10)$ cycles
7. copy B from RAM to L2 to L1 to registers $O(10)$ cycles
8. run command
 $O(100)$ overhead cycles !!!

- ▶ search two data, A and B
- ▶ search A in the first level cache(L1) O(1) cycles

- ▶ search B in the first level cache(L1) O(1) cycles

- ▶ run command

O(1) overhead cycles

- ▶ Dynamic RAM (DRAM) main memory
 - ▶ one transistor cell
 - ▶ cheap
 - ▶ it needs to be periodically refreshed
 - ▶ data are not available during refreshing
- ▶ Static RAM (SRAM) cache memory
 - ▶ cell requires 6-7 transistor
 - ▶ expensive
 - ▶ it does not need to be refreshed
 - ▶ data are always available.
- ▶ DRAM has better price/performance than SRAM
 - ▶ also higher densities, need less power and dissipate less heat
- ▶ SRAM provides higher speed
 - ▶ used for high-performance memories (registers, cache memory)

```
float sum = 0.0f;
for (i = 0; i < n; i++)
    sum = sum + x[i]*y[i];
```

- ▶ At each iteration, one sum and one multiplication floating-point are performed
- ▶ The number of the operations performed is $2 \times n$

- ▶ $T_{es} = N_{flop} * t_{flop}$
- ▶ $N_{flop} \rightarrow$ Algorithm

- ▶ $T_{es} = N_{flop} * t_{flop}$
- ▶ $N_{flop} \rightarrow$ Algorithm
- ▶ $t_{flop} \rightarrow$ Hardware

- ▶ $T_{es} = N_{flop} * t_{flop}$
- ▶ $N_{flop} \rightarrow$ Algorithm
- ▶ $t_{flop} \rightarrow$ Hardware
- ▶ consider only execution time
- ▶ What are we neglecting?

- ▶ $T_{es} = N_{flop} * t_{flop}$
- ▶ $N_{flop} \rightarrow$ Algorithm
- ▶ $t_{flop} \rightarrow$ Hardware
- ▶ consider only execution time
- ▶ What are we neglecting?
- ▶ t_{mem} The required time to access data in memory.

- ▶ $T_{es} = N_{flop} * t_{flop} + N_{mem} * t_{mem}$
- ▶ $t_{mem} \rightarrow$ Hardware
- ▶ How N_{mem} affects the performances?

- ▶ $Perf = \frac{N_{Flop}}{T_{es}}$
- ▶ for $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- ▶ for $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- ▶ Performance decay factor
- ▶ $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- ▶ how to achieve the peak performance?

- ▶ $Perf = \frac{N_{Flop}}{T_{es}}$
- ▶ for $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- ▶ for $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- ▶ Performance decay factor
- ▶ $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- ▶ how to achieve the peak performance?
- ▶ **Minimize the memory accesses.**

- ▶ Fortran and C code
- ▶ Columns rows product $C_{i,j} = A_{i,k} B_{k,j}$
- ▶ Time:
 - ▶ Fortran: date_and_time (> 0.001")
 - ▶ C: clock (>0.05")
- ▶ Square matrices of size n
 - ▶ Required memory (double precision) $\approx (3 * n * n) * 8$
 - ▶ Number of total operations $\approx 2 * n * n * n$
 - ▶ We must access n elements of the two original matrices for each element of the destination matrix.
 - ▶ n products and n sums for each element of the destination matrix.
 - ▶ Total Flops = $2 * n^3 / Time$

- ▶ Estimate the number of computational operations at execution

N_{Flop}

- ▶ 1 FLOP= 1 Floating point OPERATION (addition or multiplication).
- ▶ Division, square root, trigonometric functions require much more work and hence take more time.

- ▶ Estimate execution time T_{es}

- ▶ The number of floating operation per second is the most widely unit used to estimate the computer performances:

$$Perf = \frac{N_{Flop}}{T_{es}}$$

- ▶ The minimum count is 1 Floating-pointing Operation per second (FLOPS)
- ▶ We usually use the multiples:
 - ▶ 1 MFLOPS= 10^6 FLOPS
 - ▶ 1 GFLOPS= 10^9 FLOPS
 - ▶ 1 TFLOPS= 10^{12} FLOPS

$$c_{ij} = c_{ij} + a_{ik} * b_{kj}$$

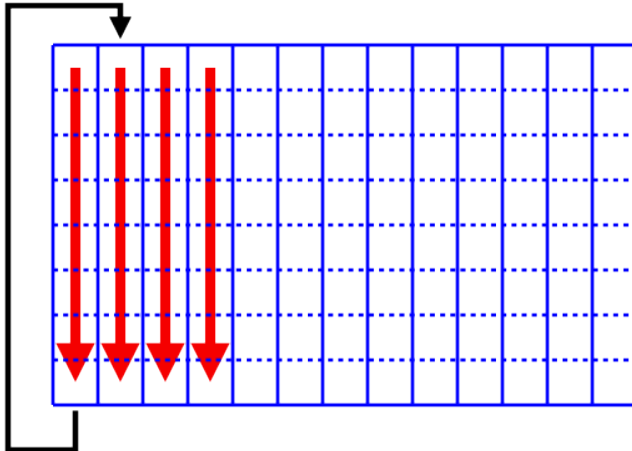
- ▶ Matrix multiplication in double precision 512X512
- ▶ Measured MFlops on Jazz (Intel(R) Xeon(R) CPU X5660 2.80GHz)
- ▶ gfortran compiler with -O0 optimization

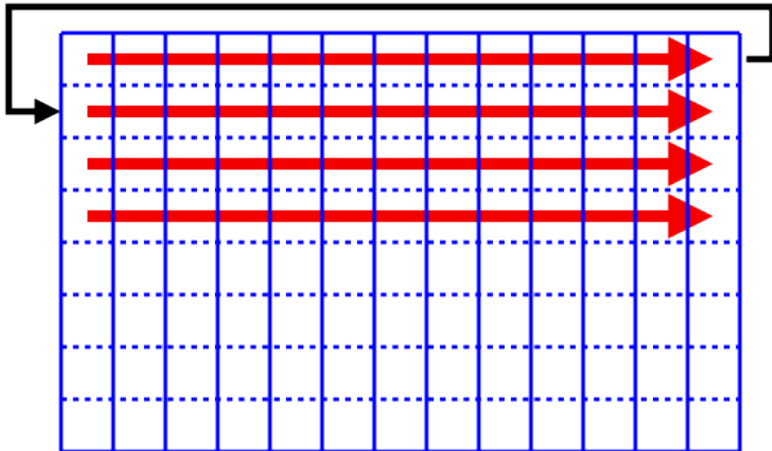
index order	Fortran	C
i,j,k	109	128
i,k,j	90	177
j,k,i	173	96
j,i,k	110	127
k,j,i	172	96
k,i,j	90	177

The efficiency of the access order depends more on the data location in memory, rather than on the language.

- ▶ Memory → elementary locations sequentially aligned
- ▶ A matrix, a_{ij} element : i row index, j column index
- ▶ Matrix representation is by arrays
- ▶ How are the array elements stored in memory?
- ▶ **C**: sequentially access starting from the last index, then the previous index ...
 $a[1][1]$ $a[1][2]$ $a[1][3]$ $a[1][4]$...
 $a[1][n]$ $a[2][1]$... $a[n][n]$
- ▶ **Fortran**: sequentially access starting from the first index, then the second index ...
 $a(1,1)$ $a(2,1)$ $a(3,1)$ $a(4,1)$...
 $a(n,1)$ $a(1,2)$... $a(n,n)$

- ▶ The distance between successively accessed data
 - ▶ stride=1 → I take advantage of the spatial locality
 - ▶ stride \gg 1 → I don't take advantage of the spatial locality
- ▶ Golden rule
 - ▶ Always access arrays, if possible, with unit stride.





► Calculate multiplication matrix-vector:

- Fortran: $d(i) = a(i) + b(i,j)*c(j)$
- C: $d[i] = a[i] + b [i][j]*c[j];$

► Fortran

- **do j=1,n**
 do i=1,n
 $d(i) = a(i) + b(i,j)*c(j)$
 end do
end do

► C

- **for(i=0;i<n,i++)**
 for(j=0;i<n,j++)
 $d[i] = a[i] + b [i][j]*c[j];$

Solving triangular system

- ▶ $Lx = b$
- ▶ Where:
 - ▶ L $n \times n$ lower triangular matrix
 - ▶ x n unknowns vector
 - ▶ b n right hand side vector
- ▶ we can solve this system by:
 - ▶ forward substitution
 - ▶ partitioning matrix

Solving triangular system

- ▶ $Lx = b$
- ▶ Where:
 - ▶ L $n \times n$ lower triangular matrix
 - ▶ x n unknowns vector
 - ▶ b n right hand side vector
- ▶ we can solve this system by:
 - ▶ forward substitution
 - ▶ partitioning matrix

What is faster?

Why?

Solution:

...

do i = 1, n

do j = 1, i-1

$b(i) = b(i) - L(i,j) b(j)$

enddo

$b(i) = b(i)/L(i,i)$

enddo

...

Solution:

```
...  
do i = 1, n  
  do j = 1, i-1  
    b(i) = b(i) - L(i,j) b(j)  
  enddo  
  b(i) = b(i)/L(i,i)  
enddo  
...
```

```
[vruggiel@fen07 TRI]$ ./a.out
```

```
time for solution    8.0586
```

Solution:

```
...  
do j = 1, n  
  b(j) = b(j)/L(j,j)  
  do i = j+1,n  
    b(i) = b(i) - L(i,j)*b(j)  
  enddo  
enddo  
...
```


Solution:

```
...  
do j = 1, n  
  b(j) = b(j)/L(j,j)  
  do i = j+1,n  
    b(i) = b(i) - L(i,j)*b(j)  
  enddo  
enddo  
...
```

```
[vruggiel@fen07 TRI]$ ./a.out
```

```
time for solution    2.5586
```

► Forward substitution

```
do i = 1, n
  do j = 1, i-1
    b(i) = b(i) - L(i,j) b(j)
  enddo
  b(i) = b(i)/L(i,i)
enddo
```

► Matrix partitioning

```
do j = 1, n
  b(j) = b(j)/L(j,j)
  do i = j+1,n
    b(i) = b(i) - L(i,j)*b(j)
  enddo
enddo
```

► Forward substitution

```
do i = 1, n
  do j = 1, i-1
    b(i) = b(i) - L(i,j) b(j)
  enddo
  b(i) = b(i)/L(i,i)
enddo
```

► Matrix partitioning

```
do j = 1, n
  b(j) = b(j)/L(j,j)
  do i = j+1,n
    b(i) = b(i) - L(i,j)*b(j)
  enddo
enddo
```

► Same number of operations, but very different elapsed times
the difference is a factor of 3

► Why?

This matrix is stored:

A	D	G	L
B	E	H	M
C	F	I	N

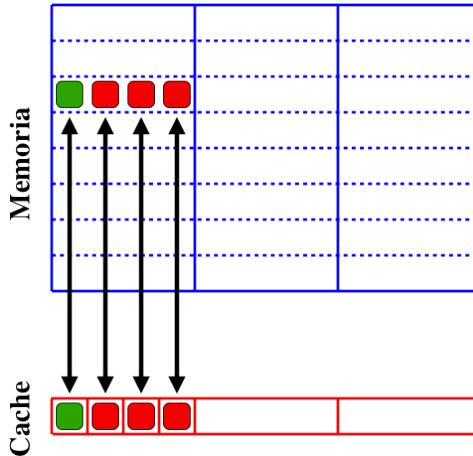
In C:

A	D	G	L	B	E	H	M	C	F	I	N
---	---	---	---	---	---	---	---	---	---	---	---

In Fortran:

A	B	C	D	E	F	G	H	I	L	M	N
---	---	---	---	---	---	---	---	---	---	---	---

- ▶ The cache is structured as a sequence of blocks (lines)
- ▶ The memory is divided in blocks with the same size of the cache line
- ▶ When data are required the system loads from memory the entire cache line that contains the data.



- ▶ Multiplication matrix-matrix in double precision
- ▶ Versions with different calls to BLAS library
- ▶ Performance in MFlops on Intel(R) Xeon(R) CPU X5660 2.80GHz

Dimension	1 DGEMM	N DGEMV	N^2 DDOT
500	5820	3400	217
1000	8420	5330	227
2000	12150	2960	136
3000	12160	2930	186

Same number of operations but the use of cache memory is changed!!!

```
...  
d=0.0  
do I=1,n  
j=index(I)  
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))  
...
```

```
...  
d=0.0  
do I=1,n  
  j=index(I)  
  d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))  
...  

```

Can I change the code to obtain best performances?


```
...  
d=0.0  
do I=1,n  
j=index(I)  
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))  
...
```

Can I change the code to obtain best performances?

```
...  
d=0.0  
do I=1,n  
j=index(I)  
d = d + sqrt(r(1,j)*r(1,j) + r(2,j)*r(2,j) + r(3,j)*r(3,j))  
...
```

- ▶ Registers are memory locations inside CPUs
- ▶ small amount of them (typically, less than 128), but with zero latency
- ▶ All the operations performed by computing units
 - ▶ take the operands from registers
 - ▶ return results into registers
- ▶ transfers memory \leftrightarrow registers are different operations
- ▶ Compiler uses registers
 - ▶ to store intermediate values when computing expressions
 - ▶ too complex expressions or too large loop bodies force the so called “register spilling”
 - ▶ to keep close to CPU values to be reused
 - ▶ but only for scalar variables, not for array elements

```

do 3000 z=1,nz
  k3=beta(z)
  do 3000 y=1,ny
    k2=eta(y)
    do 3000 x=1,nx/2
      hr(x,y,z,1)=hr(x,y,z,1)*norm
      hi(x,y,z,1)=hi(x,y,z,1)*norm
      hr(x,y,z,2)=hr(x,y,z,2)*norm
      hi(x,y,z,2)=hi(x,y,z,2)*norm
      hr(x,y,z,3)=hr(x,y,z,3)*norm
      hi(x,y,z,3)=hi(x,y,z,3)*norm
      .....
      k1=alfa(x,1)
      k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
      k_quad=1./k_quad
      sr=k1*hr(x,y,z,1)+k2*hr(x,y,z,2)+k3*hr(x,y,z,3)
      si=k1*hi(x,y,z,1)+k2*hi(x,y,z,2)+k3*hi(x,y,z,3)
      hr(x,y,z,1)=hr(x,y,z,1)-sr*k1*k_quad
      hr(x,y,z,2)=hr(x,y,z,2)-sr*k2*k_quad
      hr(x,y,z,3)=hr(x,y,z,3)-sr*k3*k_quad
      hi(x,y,z,1)=hi(x,y,z,1)-si*k1*k_quad
      hi(x,y,z,2)=hi(x,y,z,2)-si*k2*k_quad
      hi(x,y,z,3)=hi(x,y,z,3)-si*k3*k_quad
      k_quad_cfr=0.
3000 continue

```

```
do 3000 z=1,nz
  k3=beta(z)
  do 3000 y=1,ny
    k2=eta(y)
    do 3000 x=1,nx/2
      br1=hr(x,y,z,1)*norm
      bi1=hi(x,y,z,1)*norm
      br2=hr(x,y,z,2)*norm
      bi2=hi(x,y,z,2)*norm
      br3=hr(x,y,z,3)*norm
      bi3=hi(x,y,z,3)*norm
      .....
      k1=alfa(x,1)
      k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
      k_quad=1./k_quad
      sr=k1*br1+k2*br2+k3*br3
      si=k1*bi1+k2*bi2+k3*bi3
      hr(x,y,z,1)=br1-sr*k1*k_quad
      hr(x,y,z,2)=br2-sr*k2*k_quad
      hr(x,y,z,3)=br3-sr*k3*k_quad
      hi(x,y,z,1)=bi1-si*k1*k_quad
      hi(x,y,z,2)=bi2-si*k2*k_quad
      hi(x,y,z,3)=bi3-si*k3*k_quad
      k_quad_cfr=0.
3000 Continue
```

```

do 3000 z=1,nz
  k3=beta(z)
  do 3000 y=1,ny
    k2=eta(y)
    do 3000 x=1,nx/2
      br1=hr(x,y,z,1)*norm
      bi1=hi(x,y,z,1)*norm
      br2=hr(x,y,z,2)*norm
      bi2=hi(x,y,z,2)*norm
      br3=hr(x,y,z,3)*norm
      bi3=hi(x,y,z,3)*norm
      .....
      k1=alfa(x,1)
      k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
      k_quad=1./k_quad
      sr=k1*br1+k2*br2+k3*br3
      si=k1*bi1+k2*bi2+k3*bi3
      hr(x,y,z,1)=br1-sr*k1*k_quad
      hr(x,y,z,2)=br2-sr*k2*k_quad
      hr(x,y,z,3)=br3-sr*k3*k_quad
      hi(x,y,z,1)=bi1-si*k1*k_quad
      hi(x,y,z,2)=bi2-si*k2*k_quad
      hi(x,y,z,3)=bi3-si*k3*k_quad
      k_quad_cfr=0.
3000 Continue

```

- ▶ Matrix transpose
 - do $j = 1, n$
 - do $i = 1, n$
 - $a(i,j) = b(j,i)$
 - end do
 - end do
- ▶ Which is the best loop ordering to minimize the stride?
 - ▶ For data residing in cache there is no dependency on the stride
 - ▶ idea: split computations in blocks fitting into the cache
 - ▶ task: balancing between spatial and temporal locality

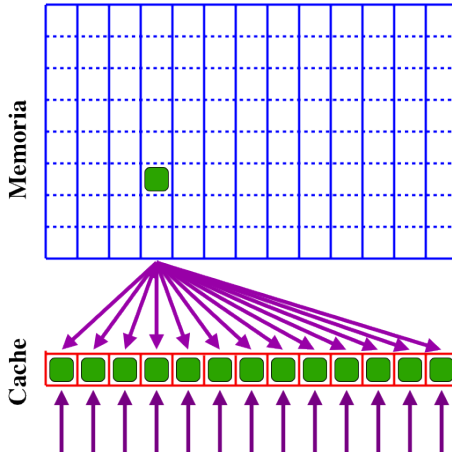
- ▶ Data are processed in chunks fitting into the cache memory
- ▶ Cache data are reused when working for the single block
- ▶ Compiler can do it for simple loops, but only at high optimization levels
- ▶ Example: matrix transpose

```
do jj = 1, n, step
  do ii = 1, n, step
    do j= jj,jj+step-1,1
      do i=ii,ii+step-1,1
        a(i,j)=b(j,i)
      end do
    end do
  end do
end do
```

- ▶ Cache may be affected by capacity miss:
 - ▶ only a few lines are really used (reduced effective cache size)
 - ▶ processing rate is reduced
- ▶ Another problem is the trashing:
 - ▶ a cache line is thrown away even when data need to be reused because new data are loaded
 - ▶ slower than not having cache at all!
- ▶ It may occur when different instruction/data flows refer to the same cache lines
- ▶ It depends on how the memory is mapped to the cache
 - ▶ fully associative cache
 - ▶ direct mapped cache
 - ▶ N-way set associative cache

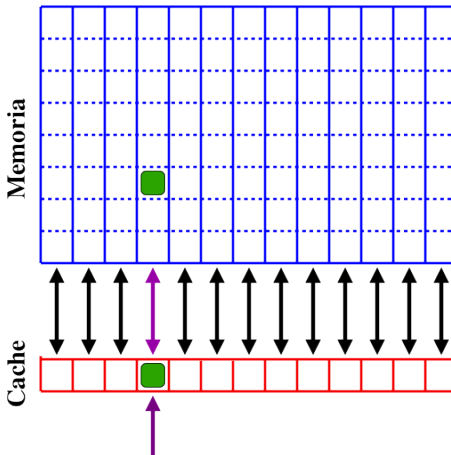
- ▶ A cache mapping defines where memory locations will be placed in cache
 - ▶ in which cache line a memory addresses will be placed
 - ▶ we can think of the memory as being divided into blocks of the size of a cache line
 - ▶ the cache mapping is a simple hash function from addresses to cache sets
- ▶ Cache is much smaller than main memory
 - ▶ more than one of the memory blocks can be mapped to the same cache line
- ▶ Each cache line is identified by a tag
 - ▶ determines which memory addresses the cache line holds
 - ▶ based on the tag and the valid bit, we can find out if a particular address is in the cache (hit) or not (miss)

- ▶ A cache where data from any address can be stored in any cache location.



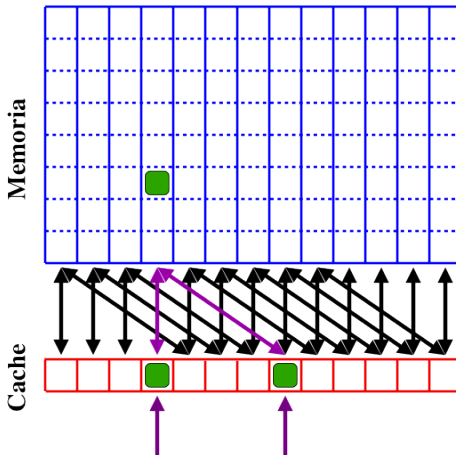
- ▶ Pros:
 - ▶ full cache exploitation
 - ▶ independent of the patterns of memory access
- ▶ Cons:
 - ▶ complex circuits to get a fast identify of hits
 - ▶ substitution algorithm: demanding, Least Recently Used (LRU) or not very efficient First In First Out (FIFO)
 - ▶ costly and small sized

- ▶ Each main memory block can be mapped to only one slot.
(linear congruence)



- ▶ Pros:
 - ▶ easy check of hit (a few bit of address identify the checked line)
 - ▶ substitution algorithm is straightforward
 - ▶ arbitrarily sized cache
- ▶ Cons:
 - ▶ strongly dependent on memory access patterns
 - ▶ affected by capacity miss
 - ▶ affected by cache trashing

- ▶ Each memory block may be mapped to any line among the N possible cache lines



► Pros:

- is an intermediate choice
 - $N=1 \rightarrow$ direct mapped
 - $N =$ number of cache lines \rightarrow fully associative
- allows for compromising between circuitual complexity and performances (cost and programmability)
- allows for achieving cache with reasonable sizes

► Cons:

- strongly conditioned by the memory pattern access
- partially affected by capacity miss
- partially affected by cache trashing

- ▶ Cache L1: 4÷8 way set associative
- ▶ Cache L2÷3: 2÷4 way set associative o direct mapped
- ▶ Capacity miss and trashing must be considered
 - ▶ strategies are the same
 - ▶ optimization of placement of data in memory
 - ▶ optimization of pattern of memory accesses
- ▶ L1 cache works with virtual addresses
 - ▶ programmer has the full control
- ▶ L2÷3 caches work with physical addresses
 - ▶ performances depend on physical allocated memory
 - ▶ performances may vary when repeating the execution
 - ▶ control at operating system level

- ▶ Problems when accessing data in memory
- ▶ A cache line is replaced even if its content is needed after a short time
- ▶ It occurs when two or more data flows need a same small subset of cache lines
- ▶ The number of load and store is unchanged
- ▶ Transaction on memory bus gets increased
- ▶ A typical case is given by flows requiring data with relative strides of 2 power

► Iteration $i=1$

1. Search for $A(1)$ in L1 cache → **cache miss**
2. Get $A(1)$ from RAM memory
3. Copy from $A(1)$ to $A(8)$ into L1
4. Copy $A(1)$ into a register
5. Search for $B(1)$ in L1 cache → **cache miss**
6. Get $B(1)$ from RAM memory
7. Copy from $B(1)$ to $B(8)$ in L1
8. Copy $B(1)$ into a register
9. Execute summation

► Iteration $i=2$

1. Search for $A(2)$ into L1 cache → **cache hit**
2. Copy $A(2)$ into a register
3. Search for $B(2)$ in L1 cache → **cache hit**
4. Copy $B(2)$ into a register
5. Execute summation

► Iteration $i=3$

► Iteration $i=1$

1. Search for $A(1)$ in the L1 cache → **cache miss**
2. Get $A(1)$ from RAM memory
3. Copy from $A(1)$ to $A(8)$ into L1
4. Copy $A(1)$ into a register
5. Search for $B(1)$ in L1 cache → **cache miss**
6. Get $B(1)$ from RAM memory
7. **Throw away cache line $A(1)$ - $A(8)$**
8. Copy from $B(1)$ to $B(8)$ into L1
9. Copy $B(1)$ into a register
10. Execute summation

► Iteration $i=2$

1. Search for $A(2)$ in the L1 cache → **cache miss**
2. Get $A(2)$ from RAM memory
3. **Throw away cache line $B(1)-B(8)$**
4. Copy from $A(1)$ to $A(8)$ into L1 cache
5. Copy $A(2)$ into a register
6. Search for $B(2)$ in L1 cache → **cache miss**
7. Get $B(2)$ from RAM memory
8. **Throw away cache line $A(1)-A(8)$**
9. Copy from $B(1)$ to $B(8)$ into L1
10. Copy $B(2)$ into a register
11. Execute summation

► Iteration $i=3$

► Effects depending on the size of data set

```

...
integer , parameter  :: offset=..
integer , parameter  :: N1=6400
integer , parameter  :: N=N1+offset

....
real (8)             :: x (N,N) , y (N,N) , z (N,N)

...
do j=1,N1
  do i=1,N1
    z (i, j)=x (i, j)+y (i, j)
  end do
end do

...

```

offset	time
0	0.361
3	0.250
400	0.252
403	0.253

Solution is padding

```
real, dimension=1024 :: a,b
common/my_comm /a,b
do i=1, 1024
a(i)=b(i) + 1.0
enddo
```

- ▶ If cache size = $4 * 1024$, direct mapped, a,b contiguous data (for example): we have cache thrashing (load and unload a cache block repeatedly)
- ▶ size of array = multiple of cache size \rightarrow cache thrashing
- ▶ Set Associative cache reduces the problem

a(1)	b(1)
1	1025

↑
 $1 \bmod 1024 = 1$ $1025 \bmod 1024 = 1$

In the cache:

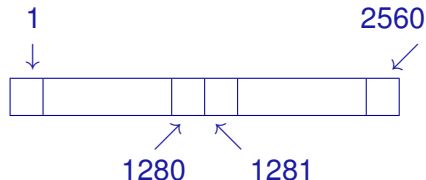
a(1)	1024
------	------

trashing

b(1)	1024
------	------

```
integer offset=
(linea di cache)/SIZE (REAL)
real, dimension=
(1024+offset) :: a,b
common/my_comm /a,b
do i=1, 1024
a(i)=b(i) + 1.0
enddo
```

offset → staggered matrixes
cache → no more problems



$$1 \bmod 1024 = 1$$

$$1281 \bmod 1024 = 257$$

Don't use matrix dimension that are powers of two:

- ▶ Bus transactions get doubled
- ▶ On some architectures:
 - ▶ may cause run-time errors
 - ▶ emulated in software
- ▶ A problem when dealing with
 - ▶ structured types (TYPE and struct)
 - ▶ local variables
 - ▶ “common”
- ▶ Solutions
 - ▶ order variables with decreasing order
 - ▶ compiler options (if available. . .)
 - ▶ different common
 - ▶ insert dummy variables into common

```
parameter (nd=1000000)
real*8 a(1:nd), b(1:nd)
integer c
common /data1/ a,c,b
....
do j = 1, 300
  do i = 1, nd
    sommal = sommal + (a(i)-b(i))
  enddo
enddo
```

Different performances for:

```
common /data1/ a,c,b
common /data1/ b,c,a
common /data1/ a,b,c
```

It depends on the architecture and on the compiler which usually warns and tries to fix the problem (align common)

In order to optimize cache using memory alignment is important. When we read memory data in word 4 bytes chunk at time (32 bit systems) The memory addresses must be powers of 4 to be aligned in memory.

```

struct MixedData{
char Data1;
short Data2;
int Data3
char Data4
}
  
```

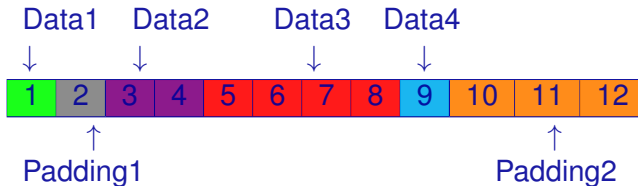


To have Data3 value two reading from memory need.

With alignment:

```

struct MixedData{
char Data1;
char Padding1[1];
short Data2;
int Data3
char Data4
char Padding2[3];
}
  
```

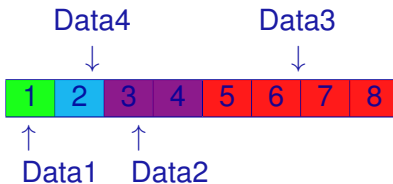


Old struct costs 8 bytes, new struct (with padding) costs 12 bytes.

We can align data exchanging their order.

```

struct MixedData{
char Data1;
char Data4
short Data2;
int Data3
}
  
```



Generally:

$\text{padding} = \text{align} - (\text{offset} \bmod \text{align})$

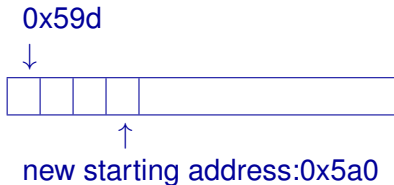
$\text{newoffset} = \text{offset} + \text{padding} = \text{offset} + \text{align} - (\text{offset} \bmod \text{align})$

For example:

$\text{offset} = 0x59d$, $\text{align}=4$

$\text{padding} = 4 - (0x59d\%4) = 4 - 1 = 3$

$\text{newoffset} = 0x59d + 3 = 0x5a0$



- ▶ Processors have hardware counters
- ▶ Devised for high clock CPUs
 - ▶ necessary to debug processors
 - ▶ useful to measure performances
 - ▶ crucial to ascertain unexpected behaviors
- ▶ Each architecture measures different events
- ▶ Of course, vendor dependent
 - ▶ IBM: HPCT
 - ▶ INTEL: Vtune
- ▶ Multi-platform measuring tools exist
 - ▶ Valgrind, Oprofile
 - ▶ PAPI
 - ▶ Likwid
 - ▶ ...

- ▶ Its state is persistent until a cache-miss requires a change
- ▶ Its state is hidden for the programmer:
 - ▶ does not affect code semantics (i.e., the results)
 - ▶ affects the performances
- ▶ The same routine called under different code sections may show completely different performances because of the cache state at the moment
- ▶ Code modularity tends to make the programmer forget it
- ▶ It may be important to study the issue in a context larger than the single routine

- ▶ Software Open Source useful for Debugging/Profiling of programs running under Linux OS, sources not required (black-box analysis), and different tools available:
 - ▶ Memcheck (detect memory leaks, ...)
 - ▶ Cachegrind (cache profiler)
 - ▶ Callgrind (callgraph)
 - ▶ Massif (heap profiler)
 - ▶ Etc.
- ▶ <http://valgrind.org>

```
valgrind --tool=cachegrind <nome_eseguibile>
```

- ▶ Simulation of program-cache hierarchy interaction
 - ▶ two independent first level cache (L1)
 - ▶ instruction (I1)
 - ▶ data cache (D1)
 - ▶ a last level cache, L2 or L3(LL)
- ▶ Provides statistics
 - ▶ I cache reads (Ir executed instructions), I1 cache read misses (I1mr), LL cache instruction read misses (ILmr)
 - ▶ D cache reads, Dr,D1mr,DLImr
 - ▶ D cache writes, Dw,D1mw,DLImw
- ▶ Optionally provides branches and mispredicted branches

```

==14924== I   refs:          7,562,066,817
==14924== I1  misses:           2,288
==14924== LLi misses:         1,913
==14924== I1  miss rate:           0.00%
==14924== LLi miss rate:        0.00%
==14924==
==14924== D   refs:          2,027,086,734 (1,752,826,448 rd + 274,260,286 wr)
==14924== D1  misses:           16,946,127 ( 16,846,652 rd + 99,475 wr)
==14924== LLd misses:           101,362 ( 2,116 rd + 99,246 wr)
==14924== D1  miss rate:           0.8% ( 0.9% + 0.0% )
==14924== LLd miss rate:           0.0% ( 0.0% + 0.0% )
==14924==
==14924== LL refs:           16,948,415 ( 16,848,940 rd + 99,475 wr)
==14924== LL misses:           103,275 ( 4,029 rd + 99,246 wr)
==14924== LL miss rate:           0.0% ( 0.0% + 0.0% )

```

```

==15572== I   refs:          7,562,066,871
==15572== I1  misses:           2,288
==15572== LLi misses:          1,913
==15572== I1  miss rate:           0.00%
==15572== LLi miss rate:          0.00%
==15572==
==15572== D   refs:          2,027,086,744 (1,752,826,453 rd + 274,260,291 wr)
==15572== D1  misses:          151,360,463 ( 151,260,988 rd +    99,475 wr)
==15572== LLd misses:           101,362 (    2,116 rd +    99,246 wr)
==15572== D1  miss rate:           7.4% (    8.6% +    0.0% )
==15572== LLd miss rate:           0.0% (    0.0% +    0.0% )
==15572==
==15572== LL refs:          151,362,751 ( 151,263,276 rd +    99,475 wr)
==15572== LL misses:           103,275 (    4,029 rd +    99,246 wr)
==15572== LL miss rate:           0.0% (    0.0% +    0.0% )

```

- ▶ Cachegrind automatically produces the file `cachegrind.out.<pid>`
- ▶ In addition to the previous information, more detailed statistics for each function is made available

```
cg_annotate cachegrind.out.<pid>
```

- ▶ `—l1=<size>,<associativity>,<line size>`
- ▶ `—D1=<size>,<associativity>,<line size>`
- ▶ `—LL=<size>,<associativity>,<line size>`
- ▶ `—cache-sim=no|yes [yes]`
- ▶ `—branch-sim=no|yes [no]`
- ▶ `—cachegrind-out-file=<file>`

- ▶ Performance Application Programming Interface
- ▶ Standard Interface to access hardware counters
- ▶ 2 interfaces available (C and Fortran):
 - ▶ High-level interface: easy
 - ▶ Low-level interface: complex but more programmable
- ▶ Pros:
 - ▶ administrative permission not required for measuring
 - ▶ standard event names for monitored events (if available) with the high-level interface
 - ▶ easy monitoring of user-defined code sections
- ▶ Cons:
 - ▶ Manual instrumenting is mandatory
 - ▶ OS kernel makes the event available

Associated to hardware counters depending on the machine

Example:

PAPI_TOT_CYC: clock cycles

PAPI_TOT_INS: completed instructions

PAPI_FP_INS: floating-point instructions

PAPI_L1_DCA: L1 data cache accesses

PAPI_L1_DCM: L1 data cache misses

PAPI_SR_INS: store instructions

PAPI_BR_MSP: branch instructions mispredicted

- ▶ High-level library calls are intuitive
- ▶ It is possible to simultaneously monitor different events
- ▶ In Fortran:

```
#include "fpapi_test.h"
integer events(2), retval ; integer*8 values(2)
.....
events(1) = PAPI_FP_INS ; events(2) = PAPI_L1_DCM
.....
call PAPIf_start_counters(events, 2, retval)
call PAPIf_read_counters(values, 2, retval) ! Clear values
< sezione di codice da monitorare >
call PAPIf_stop_counters(values, 2, retval)
print*, 'Floating point instructions', values(1)
print*, 'L1 Data Cache Misses: ', values(2)
.....
```

- ▶ It is possible to handle errors analyzing a variable returned by the subroutine
- ▶ It is possible to perform queries to check the availability of hardware counters, etc.
- ▶ It is recommended to call a dummy subroutine after the monitored section to ensure that the optimization has not altered the instruction flow