Summer School on PARALLEL COMPUTING

Code Optimization

V. Ruggiero (v.ruggiero@cineca.it)
Roma, 5 July 2019
HPC Department

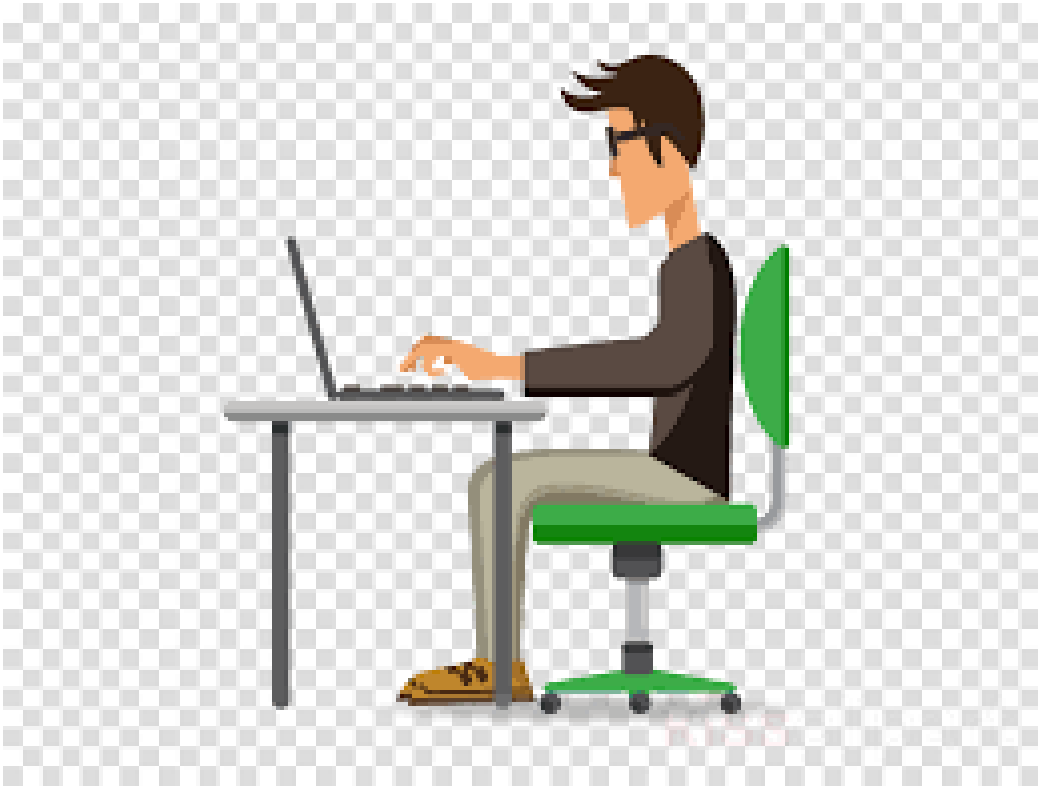
CINECA

Exercises

```
[user@home]$ git clone https://gitlab.hpc.cineca.it/training/summer-school-2019-rome.git
[user@home]$ cd summer-school-2019-rome/CODE-OPTIMIZATION/
```

- README file
- Directory tree
  - name_directory/fortran
  - name_directory/c

Hands-on 1



Matrix multiplication

- Write the main loop (columns rows product) to Fortran(mm.f90) or/and C(mm.c) code.
- Run the matrix multiplication code
- N=1024
- Verify the obtained performances

Language	time	Mflops
Fortran		
C		

Matrix multiplication

- 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 :-)

Measure of performances

- 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

Makefile

- 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"

Let's start!

- What performances have been obtained?
- There are differences between Fortran and C codes?
- How change the perfomances 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?

Let's start!

- What performances have been obtained?
- There are differences between Fortran and C codes?
- How change the perfomances 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?

What do you think about the obtained results?

Outline

What is Optimization?

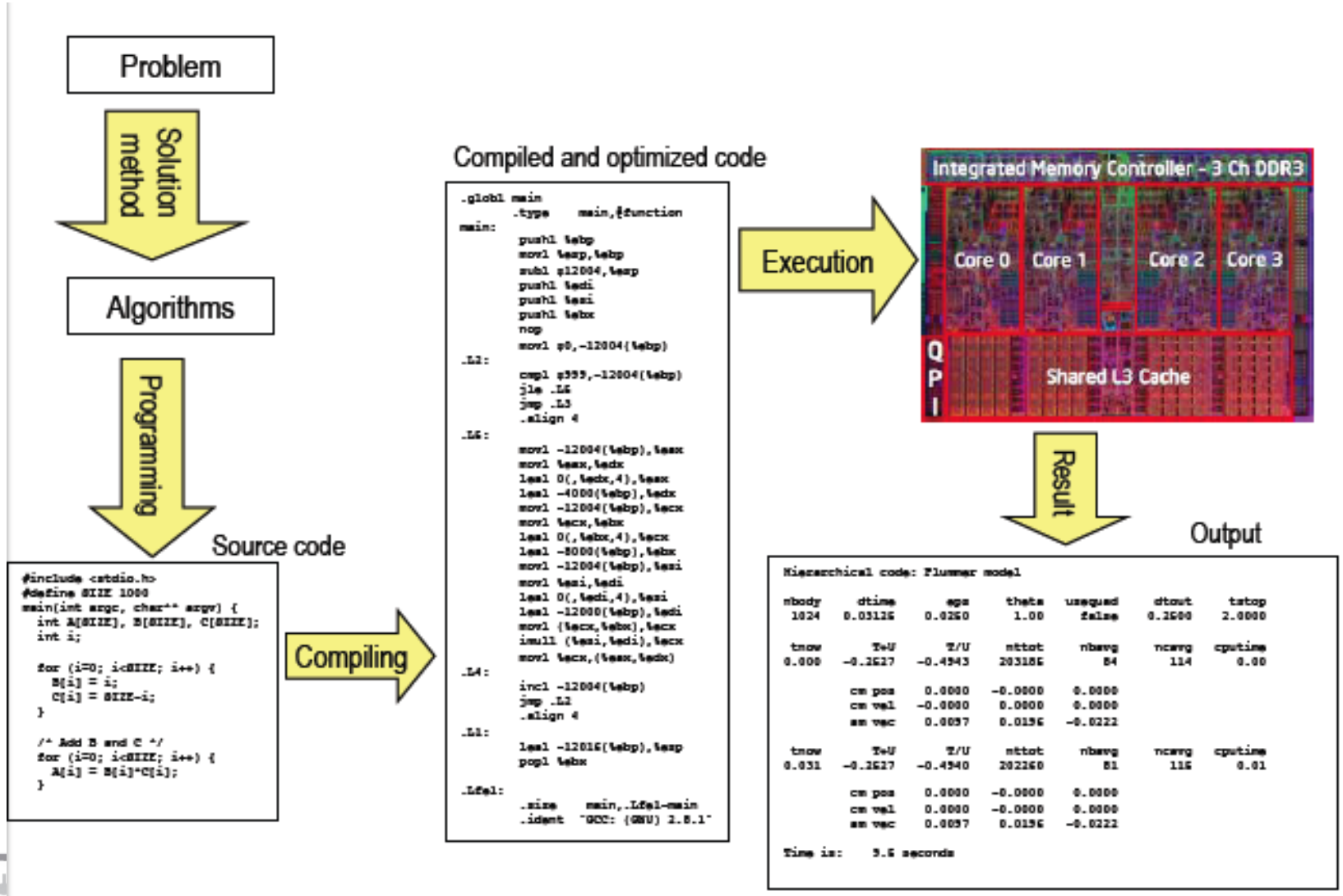
- Finding hotspots and bottlenecks (profiling)
  - Code in the program that uses a disproporitional amount of time
  - Code in the program that uses system resources inefficiently
- Reducing wall clock time
- Reducing resource requirements

Efficient Program

In general two main factors determine the speed of computations

- The algorithm efficiency, i.e. the number of steps it needs to complete a computation for a given input
- How the executable exploits processor architecture

Software development




```
#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];
  }
```

```
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $12004,%esp
        pushl %edi
        pushl %esi
        pushl %ebx
        nop
        movl $0,-12004(%ebp)
.L2:
        cmpl $999,-12004(%ebp)
        jle .L6
        jmp .L3
        .align 4
.L6:
        movl -12004(%ebp),%eax
        movl %eax,%edx
        leal 0(,%edx,4),%eax
        leal -4000(%ebp),%edx
        movl -12004(%ebp),%ecx
        movl %ecx,%ebx
        leal 0(,%ebx,4),%ecx
        leal -8000(%ebp),%ebx
        movl -12004(%ebp),%esi
        movl %esi,%edi
        leal 0(,%edi,4),%esi
        leal -12000(%ebp),%edi
        movl (%ecx,%ebx),%ecx
        imull (%esi,%edi),%ecx
        movl %ecx,(%eax,%edx)
.L4:
        incl -12004(%ebp)
        jmp .L2
        .align 4
.L3:
        leal -12016(%ebp),%esp
        popl %ebx

.Lfe1:
        .size    main,.Lfe1-main
        .ident   "GCC: (GNU) 2.8.1"
```

```
Hierarchical code: Plummer model

nbody      dtime      eps        theta      usequad    dtout      tstop
 1024    0.03125    0.0250     1.00       false      0.2500     2.0000

tnow       Te+U       T/U        nttot      nbavg      ncavg      cputime
0.000    -0.2627    -0.4943     203185       84        114        0.00

           cm pos     0.0000    -0.0000     0.0000
           cm vel    -0.0000     0.0000     0.0000
           am vec     0.0057     0.0196    -0.0222

tnow       Te+U       T/U        nttot      nbavg      ncavg      cputime
0.031    -0.2627    -0.4940     202260       81        115        0.01

           cm pos     0.0000    -0.0000     0.0000
           cm vel     0.0000    -0.0000     0.0000
           am vec     0.0057     0.0196    -0.0222

Time is:    3.6 seconds
```

Steps of optimization

- Profile
- Integrate libraries
- Optimize compiler switches
- Optimize blocks of code that dominate execution time
- Always examine corrcetness at every stage.

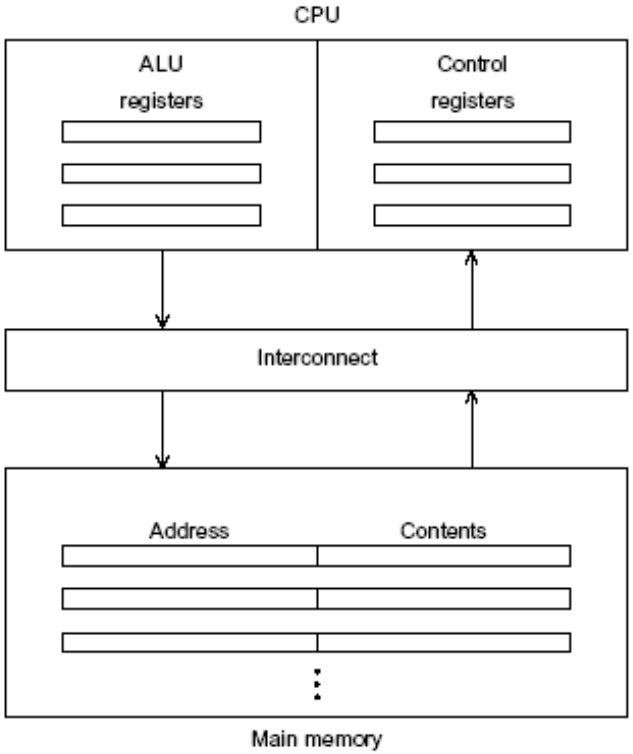
Perfomance Strategies

- Always use optimal or near optimal algorithms.
  - Be careful of resource requirements and problen sizes.
- Maintain realistic and consistent input data sets/sizes during optimization.
- Know when to stop.

Outline

Von Neumann Architecture

- Central Processing Unit (CPU)
  - Arithmetic logic unit (executes instructions)
  - Control unit
  - Registers (fast memory)
- Interconnection CPU RAM (Bus)
- Random Access Memory (RAM)
  - Address to access memory locations
  - Memory content (instructions, data)



Von Neumann Architecture

- Data are sent from memory to CPU (fetch or read)
- Data are sent from CPU to memory (written to memory or stored)
- The separation between the CPU and memory leads to what is known as the «von Neumann bottleneck»: the limited throughput (data transfer rate) between the CPU and memory compared to the amount of memory
- In most modern computers, the throughput is one hundred smaller compared to the rate the CPU can process

Solutions to the von Neumann Bottleneck

- Caching
  Very fast memories integrated directly into the processor chip.
  There are first, second or third level caches
- Virtual memory
  The RAM is used as a cache for big data storage.
- Instruction level parallelism
  single CPU core, but multiple functional units to execute multiple nstructions in parallel (pipelining, multiple issues)

Bandwidth

- Defined as the amount of data transferred per second between memory and processor
- Measured in number of bytes per second (Mb/s, Gb /s, etc..)
- A = B * C
  - B: data which is read from memory
  - C: data which is read from memory
  - Multiplication B * C is calculated
  - The result is saved to memory using the same place of the A variable
- 1 floating-point operation $\rightarrow$ 3 memory accesses

Stream

- The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels
- Time elapsed to:
  - a $\rightarrow$ c (copy)
  - a*b $\rightarrow$ c (scale)
  - a+b $\rightarrow$ c (add)
  - a+b*c $\rightarrow$ d (triad)
- It measures the maximum bandwidth
- http://www.cs.virginia.edu/stream/ref.html

cat /proc/cpuinfo

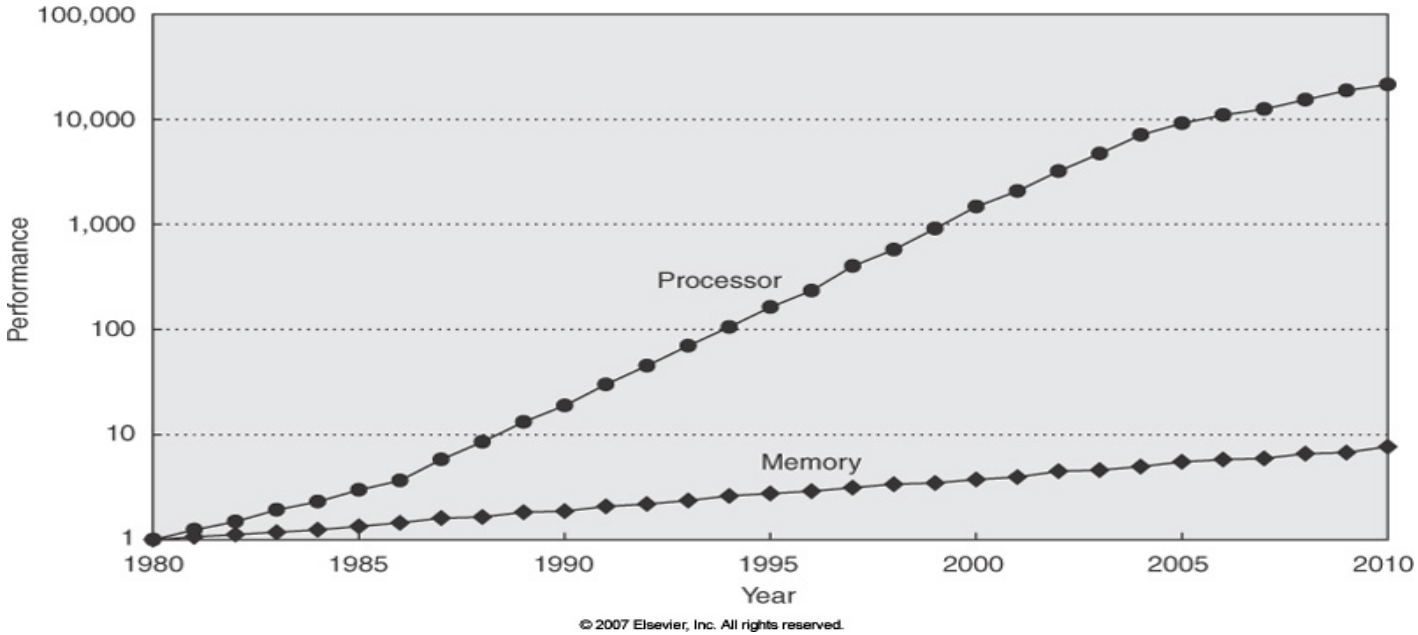
```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 37
model name      : Intel(R) Core(TM) i3 CPU       M 330  @ 2.13GHz
stepping        : 2
cpu MHz         : 933.000
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 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
...
```

lscpu

```
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
```

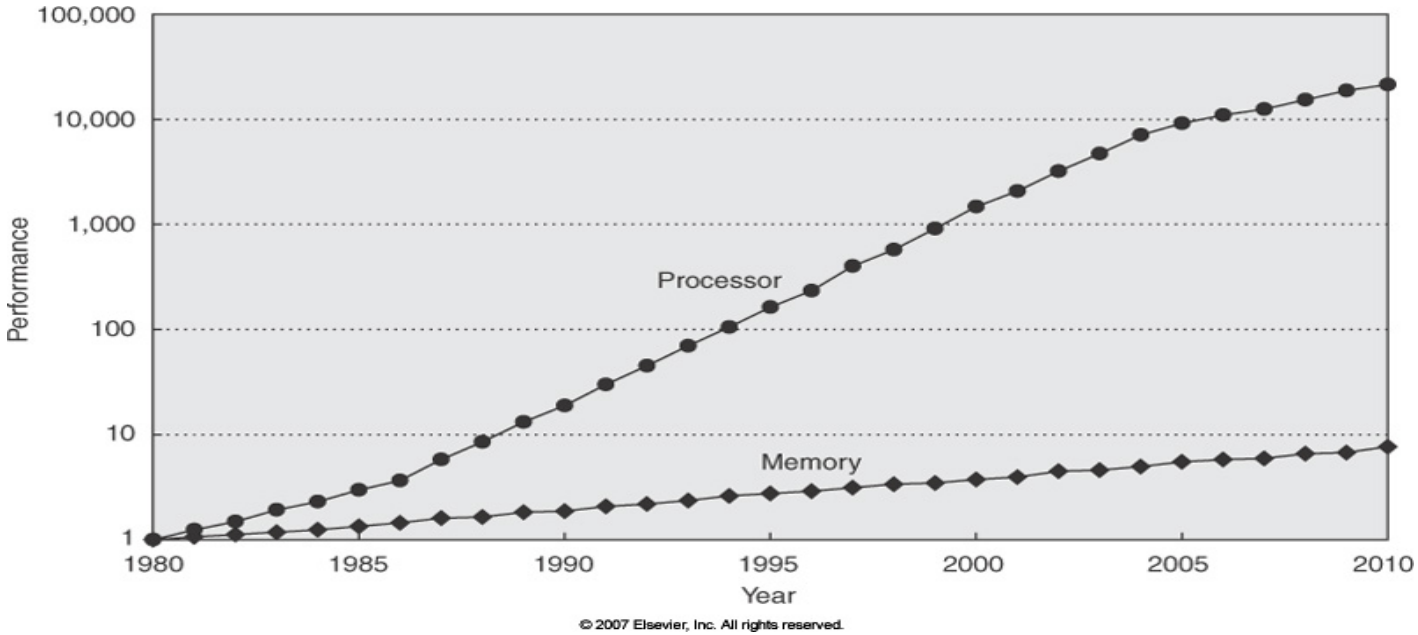
Outline

Memory system



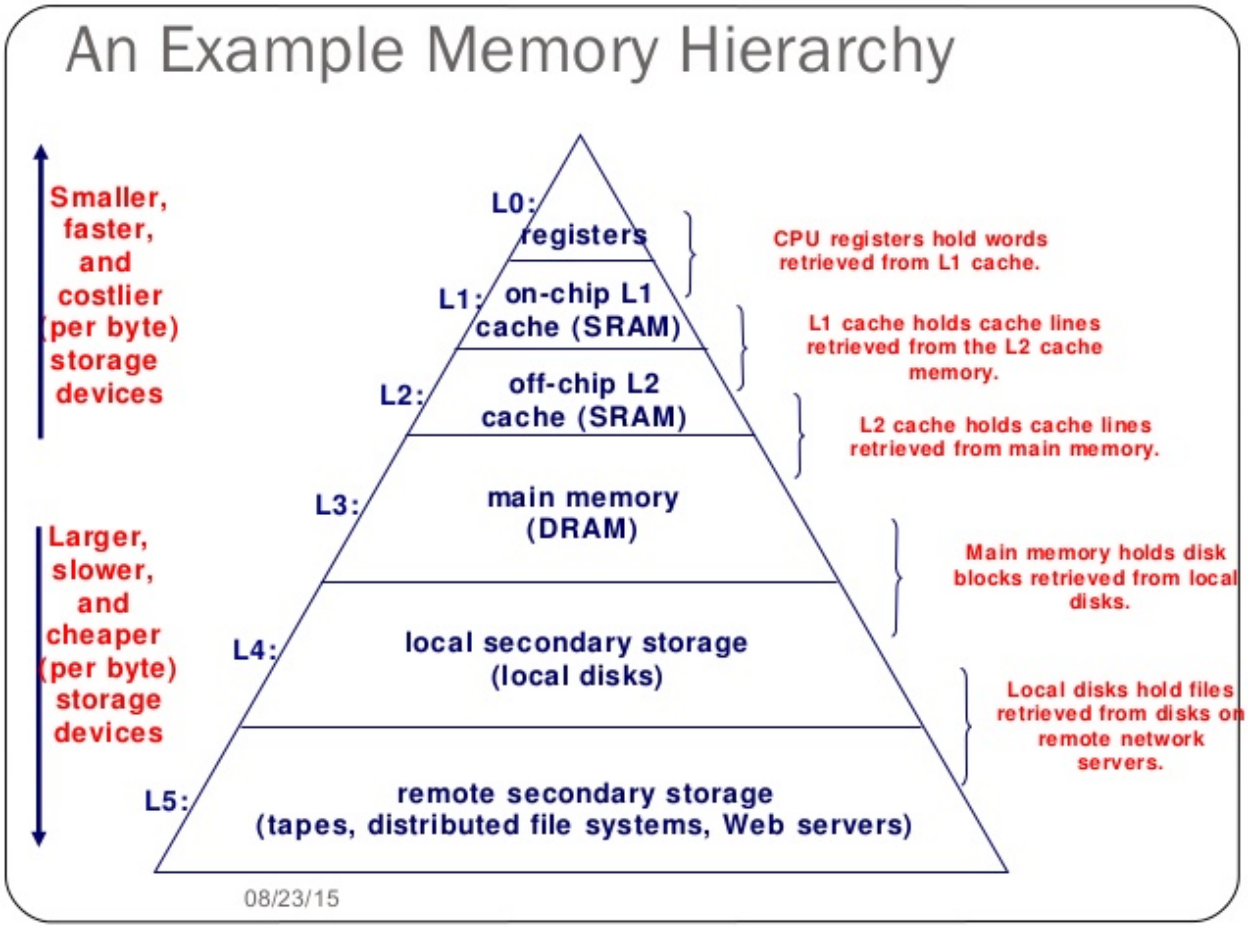
- CPU power computing doubles every 18 months
- Access rate to RAM doubles every 120 months
- Reducing the cost of the operations is useless if the loading data is slow

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

The Memory Hierarchy



Clock cycle

- 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.

The Memory Hierarchy

- 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

The Cache

Why this hierarchy?

The Cache

Why this hierarchy?
It is not necessary that all data are available at the same time. What is the solution?

The Cache

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 $\div$ MB)
- Basic principle: we always work with a subset of data.
  - data needed $\rightarrow$ fast memory access
  - data not needed (for now) $\rightarrow$ slower memory levels
- Limitations
  - Random access without reusing
  - Never large enough . . .
  - faster, hotter and . . . expensive $\rightarrow$ intermediate levels hierarchy.

The cache

- CPU accesses higher level cache:
- The cache controller finds if the required element is present in cache:
  - Yes: data is transferred from cache ti 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)

Replacement: locality principles

- 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.)

Replacement: locality principles

- 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(hardware prefetcher)

Replacement: locality principles

- 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(hardware prefetcher)
- 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.)

Replacement: locality principles

- 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(hardware prefetcher)
- 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

Replacement: locality principles

- 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(hardware prefetcher)
- 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

Cache:Some definition

- 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.

Cache: access cost

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

SRAM vs. DRAM

- 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)

Performance estimate: an example

```
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 $2 \times n$

Execution time T_{es}

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

Execution time T_{es}

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

Execution time T_{es}

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

Execution time T_{es}

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

Therefore ...

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

N_{mem} Effect

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

N_{mem} Effect

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

Performance Metrics

- Wall Clock time= Time from start to finish of our program
  - Possibly ignore set-up cost.
- MFLOPS - Millions of floating point operations per second.
- MIPS - Millions of instructions per second.

Performance Metrics

- Wall Clock time= Time from start to finish of our program
  - Possibly ignore set-up cost.
- MFLOPS - Millions of floating point operations per second.
- MIPS - Millions of instructions per second.
- For purposes of optimization, we are interested in:
  - Execution time of our code
  - MFLOPS of our kernel code vs. peak in order to determine EFFICIENCY

Performance Metrics

- Fallacies
  - MIPS is an accurate measure for comparing performance among computers .
  - MFLOPS is a consistent and useful measure of performance.
  - Synthetic benchmarks predict performance for real programs.
  - Peak performance tracks observed performance.

Measure of performances

- 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

Memory Access Behavior: Cache Misses

Classification:

- cold / compulsory miss
  - first time a memory block was accessed
- capacity miss
  - recent copy was evicted because of too small cache size
- conflict miss
  - recent copy was evicted because of too low associativity

Bad Memory Access Behavior (1)

Lot cold misses

- each memory block only accessed once, and
- prefetching not effective because accesses are not predictable or bandwidth is fully used
- usually not important, as programs access data multiple times
- can become relevant if there are lots of context switches (when multiple processes synchronize very often)

Bad Memory Access Behavior (2)

Lot capacity misses

- blocks are only accessed again after eviction due to limited size
  - number of other blocks accessed in-between (= reuse distance) > number of cache lines
  - example: sequential access to data structure larger than cache size

Countermeasures

- reduce reuse distance of accesses = increase temporal locality
- improve utilization inside cache lines = increase spatial locality
- increase predictability of memory accesses

Bad Memory Access Behavior (3)

Lots conflict misses

- blocks are only accessed again after eviction due to limited set size

Countermeasures

- set sizes are similar to cache sizes

Spatial locality: access order

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.

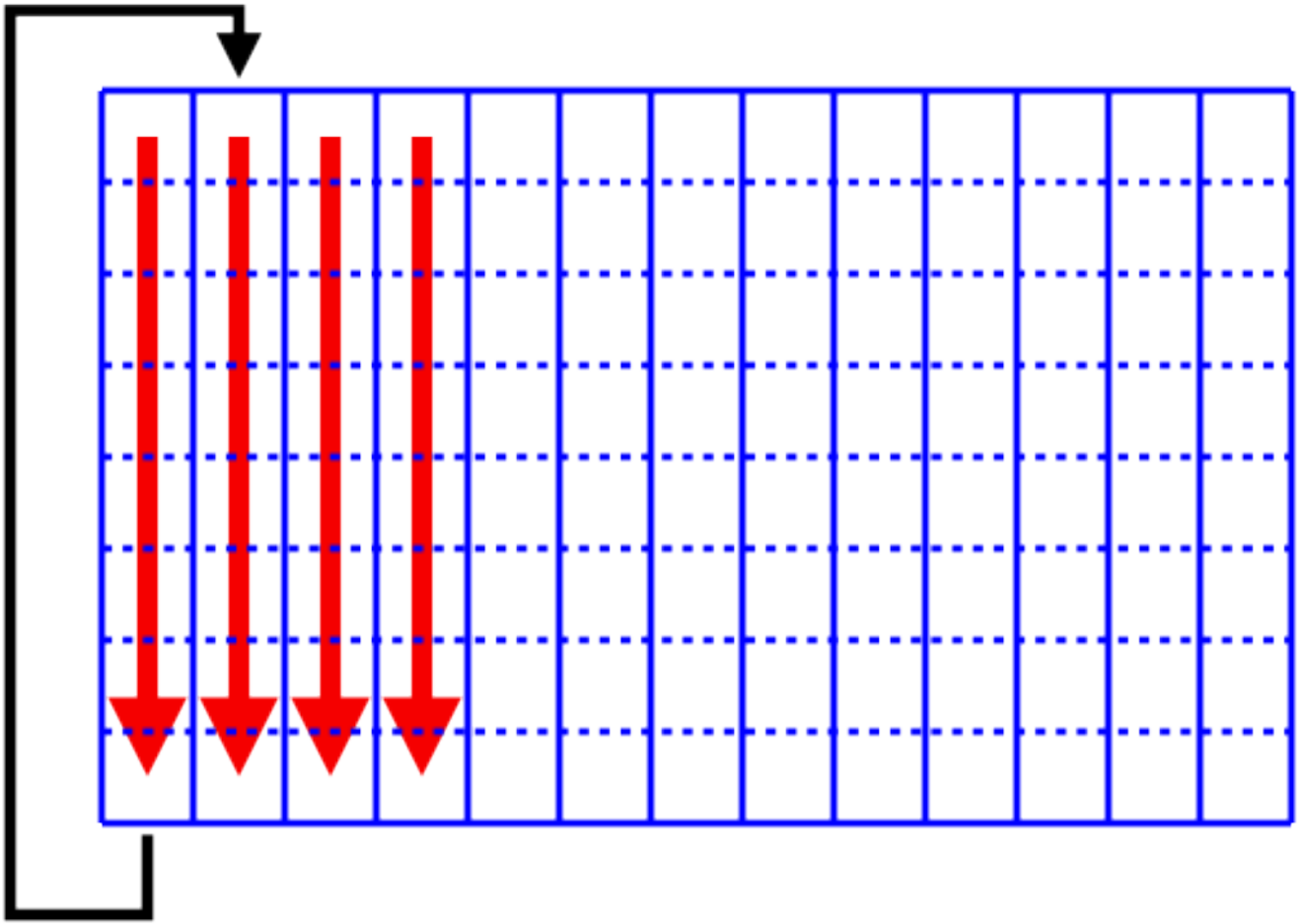
Array in memory

- Memory $\rightarrow$ 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)

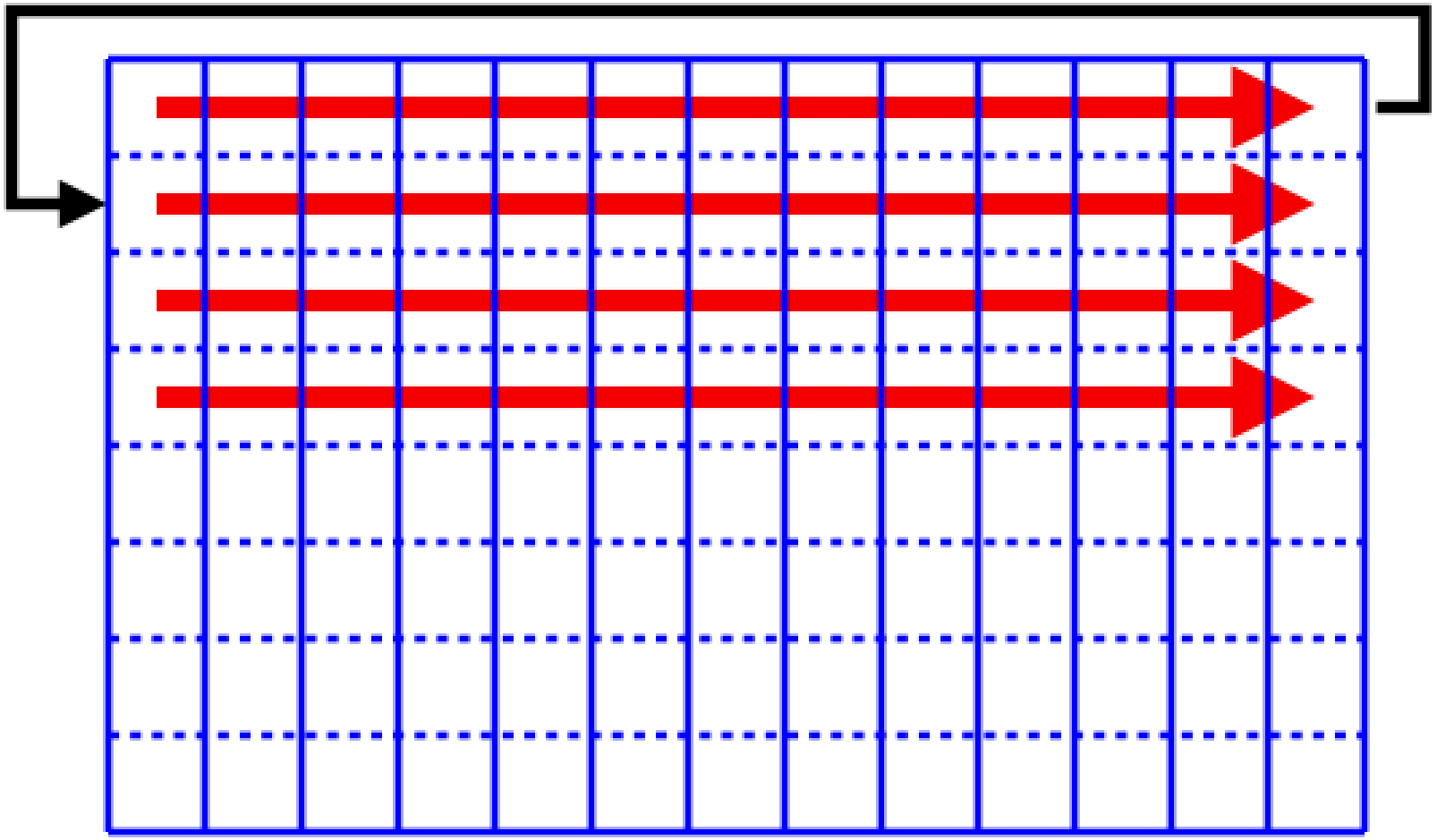
The stride

- The distance between successively accessed data
  - stride=1 $\rightarrow$ I take advantage of the spatial locality
    - using entire cache lines
  - stride » 1 $\rightarrow$ I don't take advantage of the spatial locality
- Golden rule
  - Always access arrays, if possible, with unit stride.

Fortran memory ordering



C memory ordering



Best access order

- 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];

Spatial locality:linear system

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

Spatial locality:linear system

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?

Forward substitution

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
. . .
```

Forward substitution

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
...
```

```
[vruggie1@fen07 TRI]$ ./a.out
```

```
time for solution    8.0586
```

Matrix partitioning

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
. . .
```

Matrix partitioning

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
...
```

```
[vruggie1@fen07 TRI]$ ./a.out
```

```
time for solution    2.5586
```

What is the difference?

- 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
```

What is the difference?

- 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?

Let us clarify...

This matrix is stored:

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

In C:

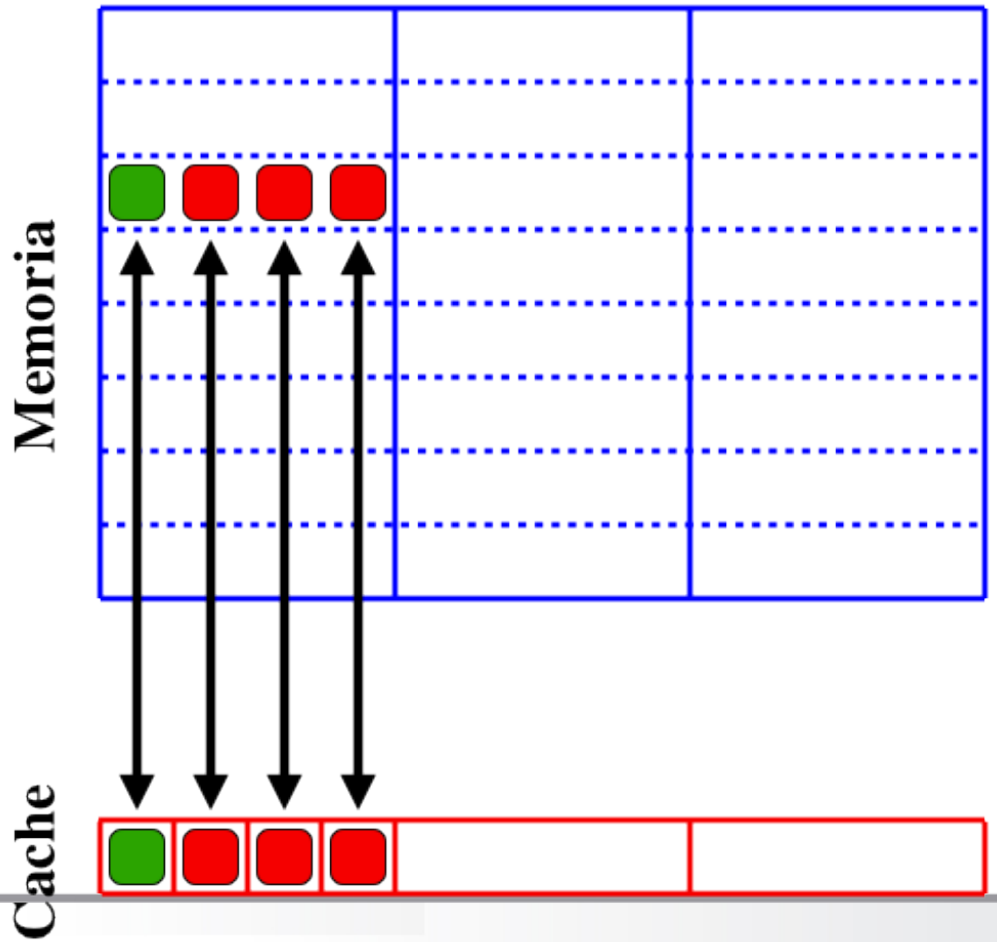
A	D	G	L	B	E	H	M	C	F	I	N

In Fortran:

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

Spatial locality: cache lines

- 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.



Cache reuse

```
...
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))
...
```

Cache reuse

```
...
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?

Cache reuse

```
...
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

- 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

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
```

Temporary scalars

```
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
```

Temporary scalars(time -25%)

```
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
```

Spatial and temporal locality

- 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

Cache blocking

- 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: capacity miss and trashing

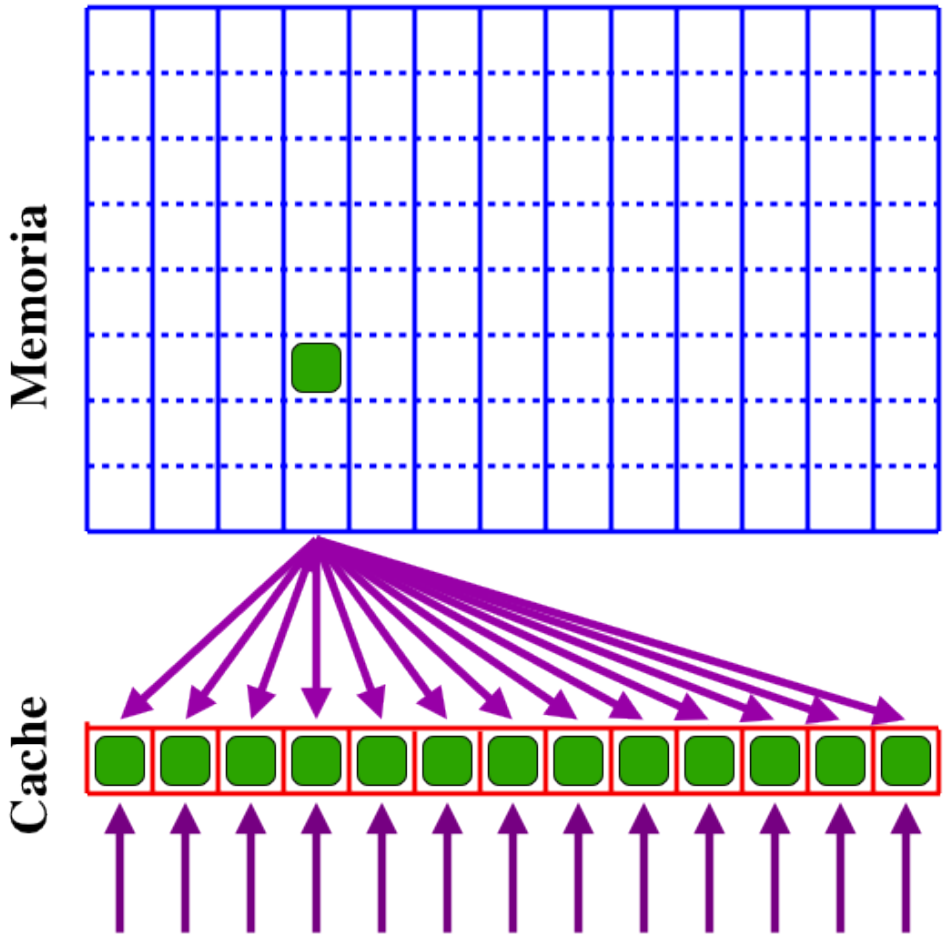
- Cache may be affected by capacity miss:
  - only a few lines are really used (reduced effective cache size)
  - processing rate is reduced
- Another problem is the trashing:
  - a cache line is thrown away even when data need to be reused because new data are loaded
  - slower than not having cache 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

Cache mapping

- 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)

Fully associative cache

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

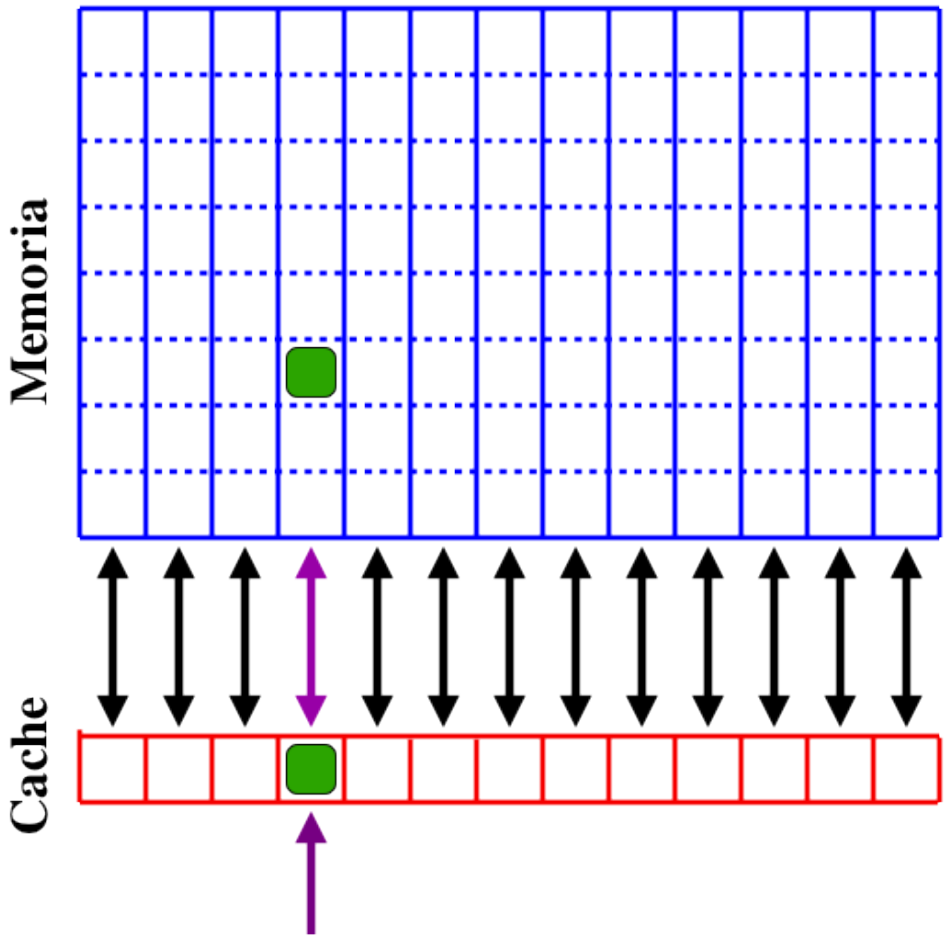


Fully associative cache

- 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

Direct mapped cache

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

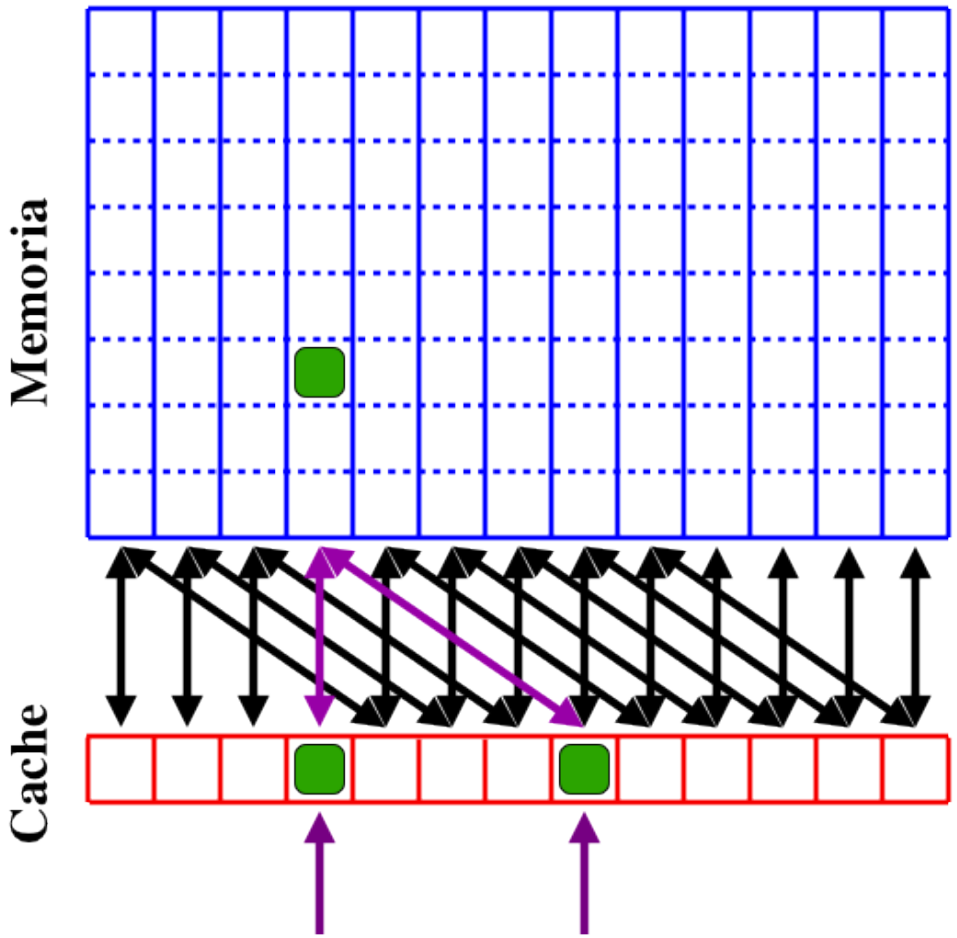


Direct mapped cache

- 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

N-way set associative cache

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



N-way set associative cache

- Pros:
  - is an intermediate choice
    - N=1 $\rightarrow$ direct mapped
    - N= number of cache lines $\rightarrow$ fully associative
  - allows for compromising between circuital 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: typical situation

- 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

Cache Trashing

- 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

No trashing: C(i)= A(i)+B(i)

- 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

Trashing: C(i)= A(i)+B(i)

- 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

Trashing: C(i)= A(i)+B(i)

- 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

How to identify it?

- 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

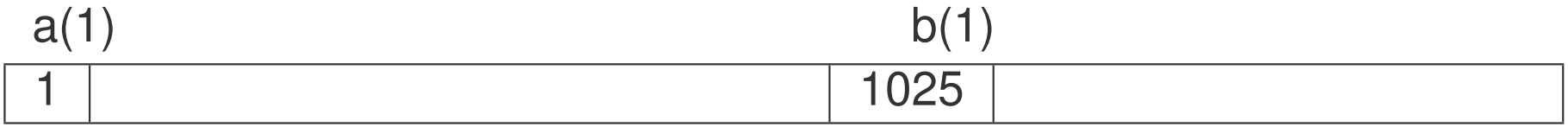
- Solution is padding

Cache 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 trashing (load and unload a cache block repeatedly)
- size of array = multiple of cache size $\rightarrow$ cache trashing
- Set Associative cache reduces the problem

Cache padding



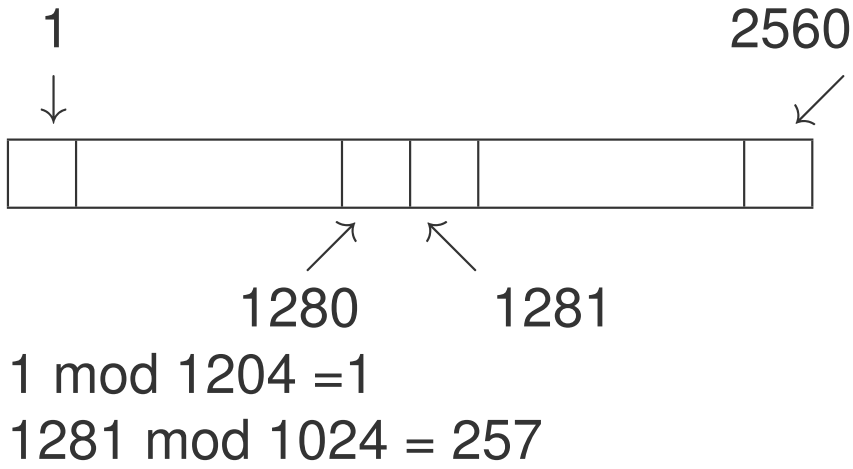
1 mod 1024=1 1025 mod 1024= 1

In the cache:

trashing

Cache padding

```
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

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

Misaligned accesses

- 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

Misaligned Accesses

```
      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
            somma1 = somma1 + (a(i)-b(i))
         enddo
      enddo
```

Different performances for:

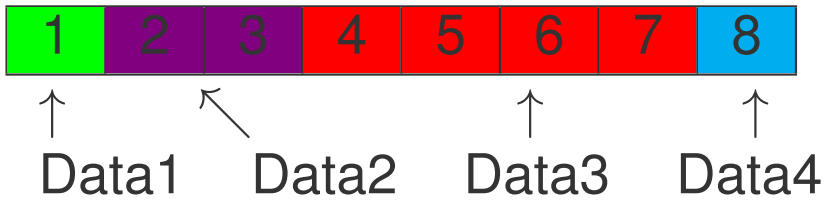
```
      common /data1/ a,c,b
      common /data1/ b,c,a
      common /data1/ a,b,c
```

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

Memory alignment

In order to optmize 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
                    }
```

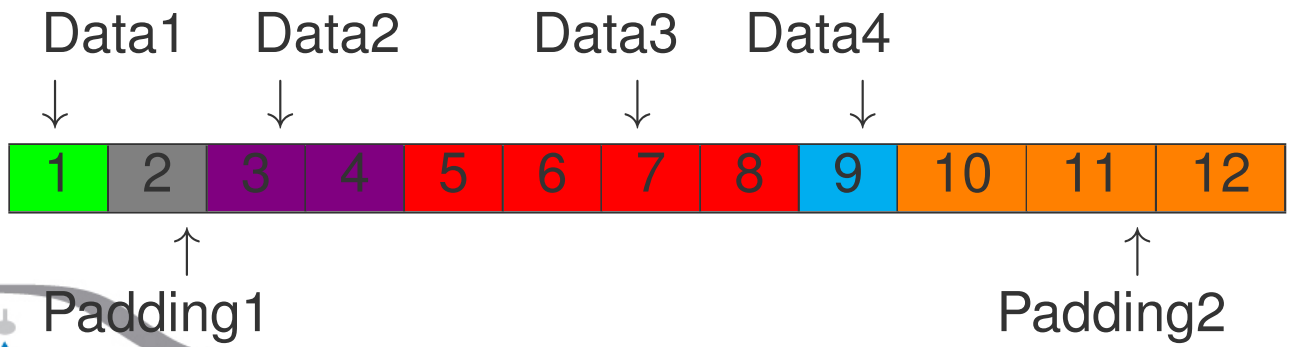


To have Data3 value two reading from memory need.

Memory alignment

With alignment:

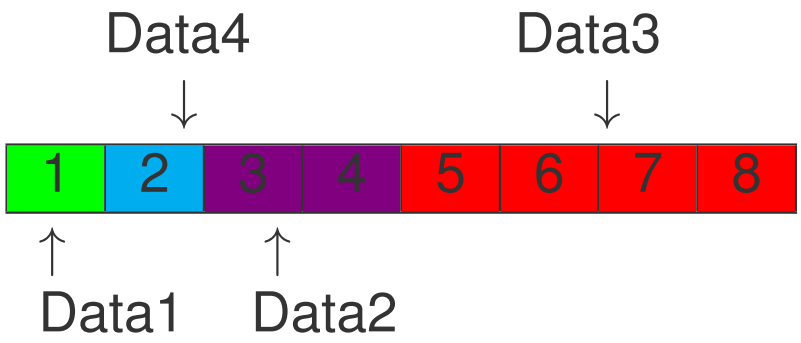
```
struct MixedData{
char Data1;
char Padding1[1];
short Data2;
int Data3
char Data4
char Padding2[3];
                 }
```



Memory alignment

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
                }
```



How to detect the problem?

- 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
  - . . .

Cache is a memory

- 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

Valgrind

- 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

Cachegrind

```
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,DLlmr
  - D cache writes, Dw,D1mw,DLlmw
- Optionally provides branches and mispredicted branches

Cachegrind:example I

```
==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%  )
```

Cachegrind:example II

```
==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:cg_annotate

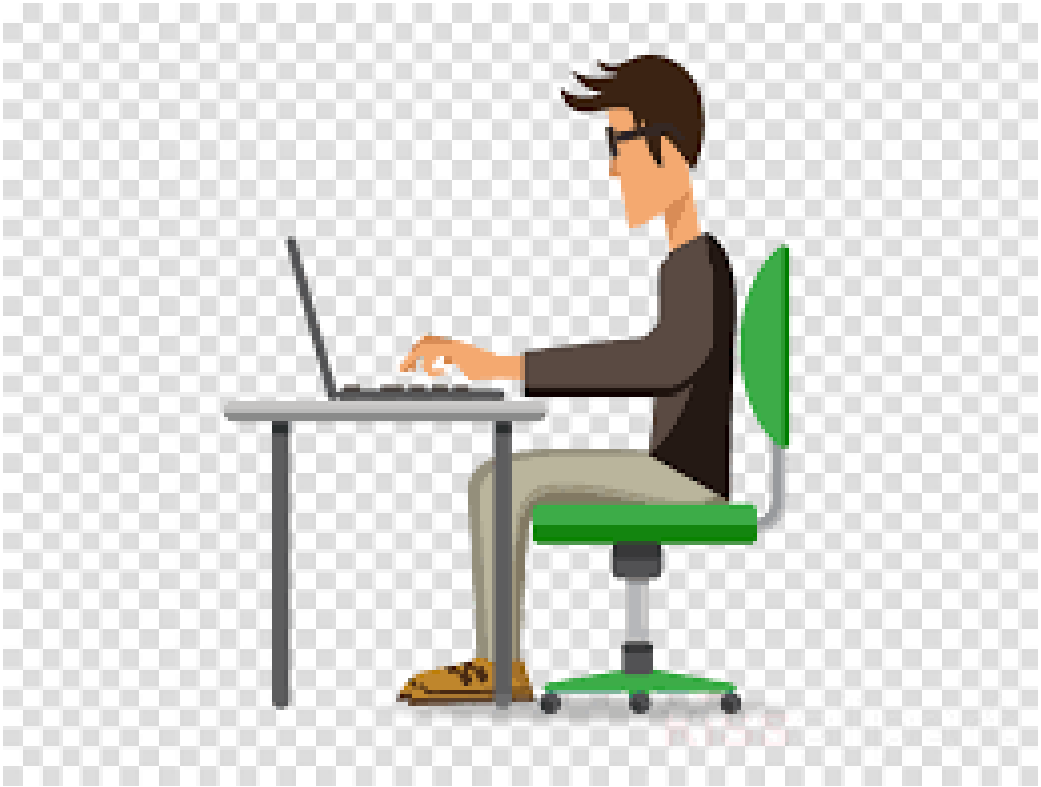
- Cachegrind automatically produces the file cachegrind.out.<pid>
- In addition to the previous information, more detailed statistics for each function is made available

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

Cachegrind:options

- —I1=<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>

Hands-on 2



Outline

CPU: internal parallelism?

- CPU are entirely parallel
  - pipelining
  - superscalar execution
  - units SIMD MMX, SSE, SSE2, SSE3, SSE4, AVX
- To achieve performances comparable to the peak performance:
  - give a large amount of instructions
  - give the operands of the instructions

Pipelining:Definitions

- Pipelining is an implementation technique where multiple operations on a number of instructions are overlapped in execution.
- An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction.
- Each step is called a pipe stage or a pipe segment.
- The stages or steps are connected one to the next to form a pipe – instructions enter at one end and progress through the stage and exit at the other end.
- Throughput of an instruction pipeline is determined by how often an instruction exists the pipeline.
- The time to move an instruction one step down the line is is equal to the machine cycle and is determined by the stage with the longest processing delay

Latency and Throughput

- Latency : the number of clocks to complete an instruction when all of its inputs are ready
- Throughput : the number of clocks to wait before starting an identical instruction
  - Identical instructions are those that use the same execution unit

Basic Performance Issues In Pipelining

- Pipelining increases the CPU instruction throughput: The number of instructions completed per unit time. Under ideal condition instruction throughput is one instruction per machine cycle, or CPI = 1
- Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).
- It usually slightly increases the execution time of each instruction over unpipelined implementations due to the increased control overhead of the pipeline and pipeline stage registers delays.

The pipeline

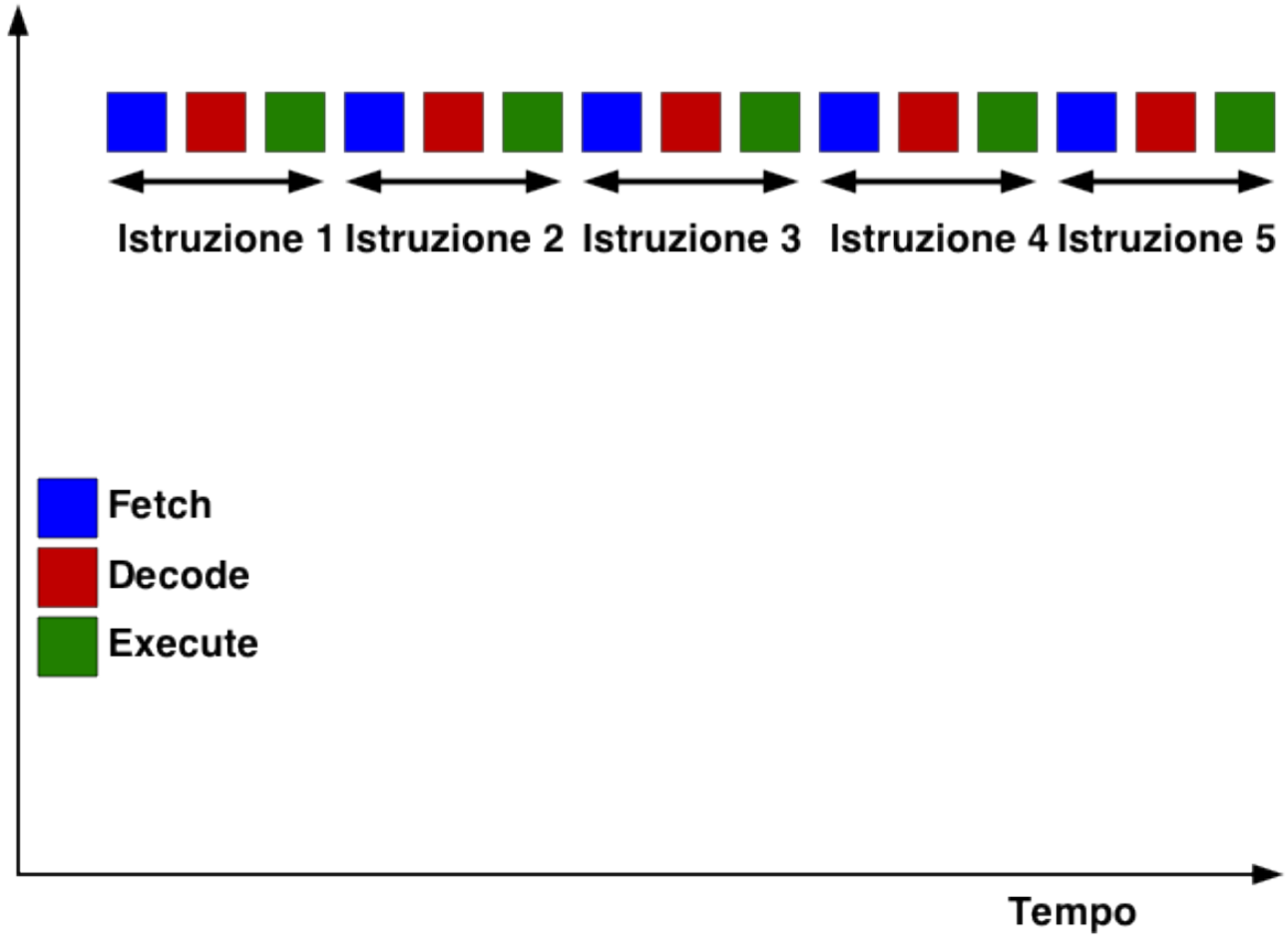
- Pipeline, channel or tube for carrying oil
- An operation is split in independent stages and different stages are executed simultaneously
  - fetch (get, catch) gets the instruction from memory and the pointer of Program Counter is increased to point to the next instruction
  - decode instruction gets interpreted
  - execute send messages which represent commands for execution
- Parallelism with different operation stages
- Processors significantly exploit pipelining to increase the computing rate

CPU cycle

- The time to move an instruction one step through the pipeline is called a machine cycle
- CPI (clock Cycles Per Instruction)
  - the number of clock cycles needed to execute an instruction
  - varies for different instructions
  - its inverse is IPC (Instructions Per Cycle)

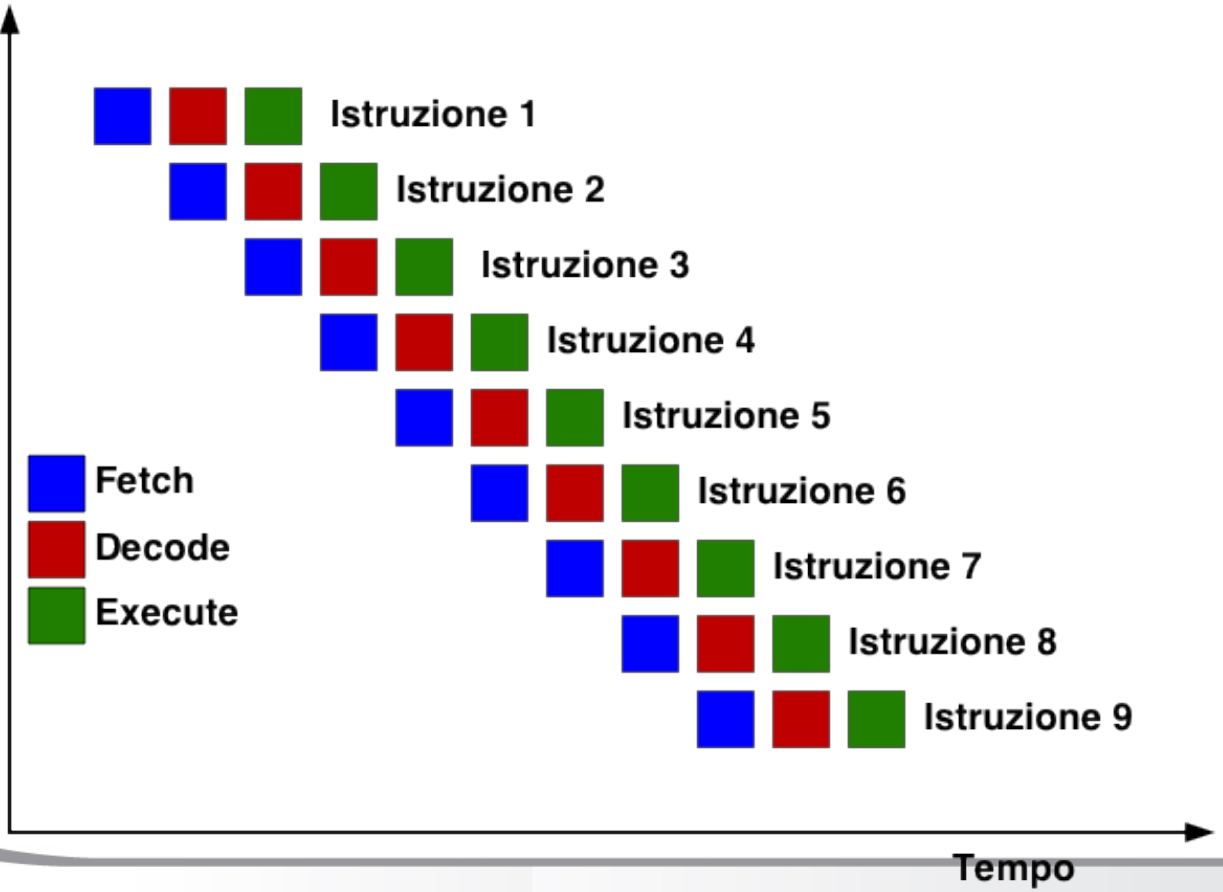
NON pipelined computing units

- Each instruction is completed after three cycles



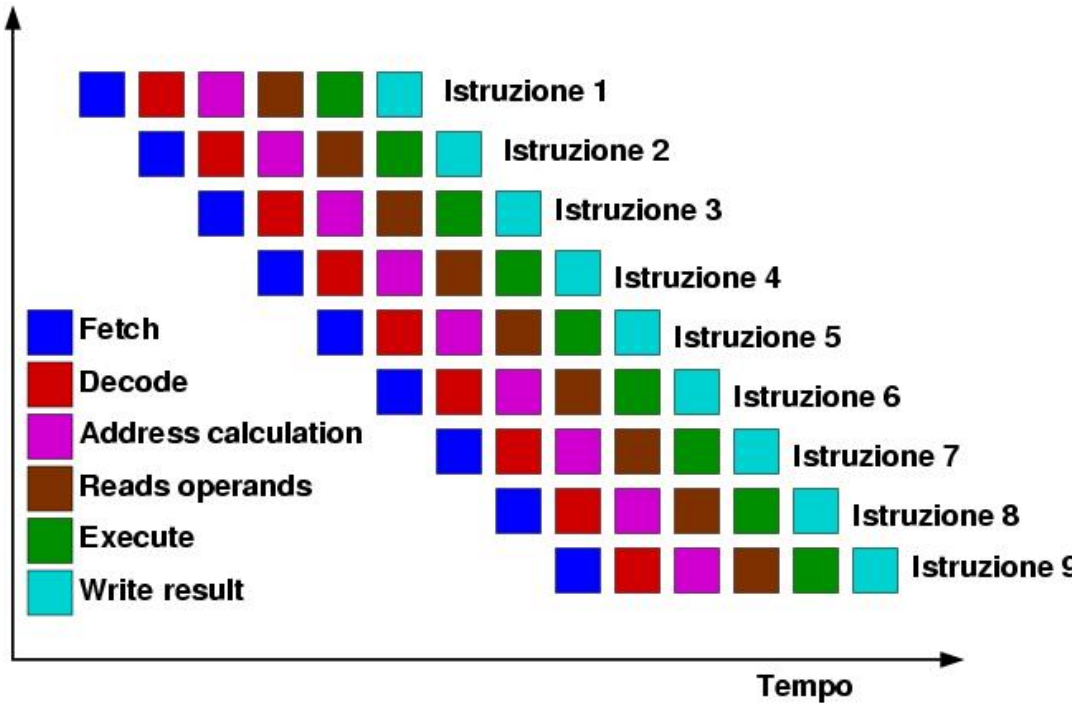
Pipelined units

- After 3 clock cycles, the pipeline is full
- A result per cycle when the pipeline is completely filled
- To fill it 3 independent instructions are needed (including the operands)



Superpipelined computing units

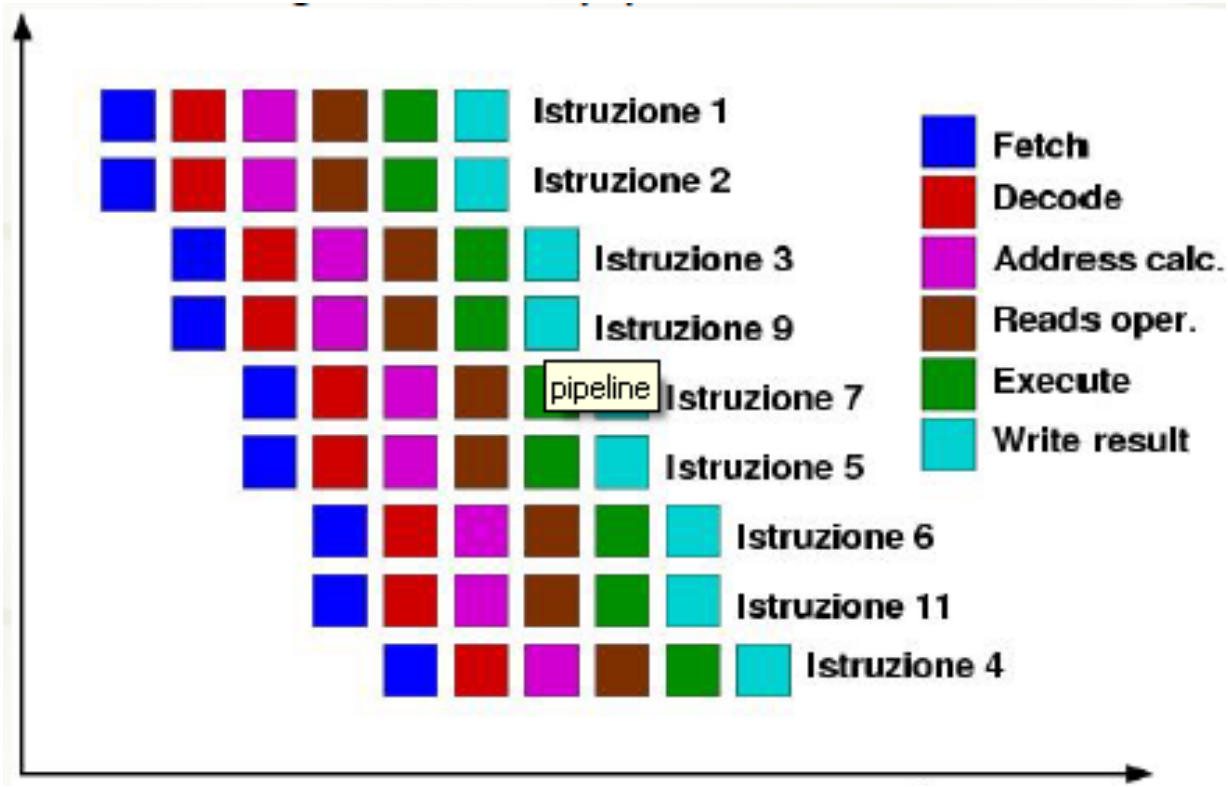
- After 6 clock cycles, the pipeline is full
- A result per cycle when the pipeline is completely filled
- To fill it 6 independent instructions are needed (including the operands)
- It is possible to halve the clock rate, i.e. doubling the frequency



Out of order execution

- Dynamically reorder the instructions
  - move up instructions having operands which are available
  - postpone instructions having operands still not available
  - reorder reads/write from/into memory
  - always considering the free functional units
- Exploit significantly:
  - register renaming (physical vs architectural registers)
  - branch prediction
  - combination of multiple read and write from/to memory
- Crucial to get high performance on present CPUs
- The code should not hide the reordering possibilities

Out of order execution



Superscalar execution

- CPUs have different independent units
  - functional differentiation
  - functional replication
- Independent operations are executed at the same time
  - integer operations
  - floating point operations
  - skipping memory
  - memory accesses
- Instruction Parallelism
- Hiding latencies
- Processors exploit superscalarity to increase the computing power for a fixed clock rate

How to exploit internal parallelism?

- Processors run at maximum speed (high instruction per cycle rate (IPC)) when
  1. There is a good mix of instructions (with low latencies) to keep the functional units busy
  2. Operands are available quickly from registers or D-cache
  3. The FP to memory operation ratio is high (FP : MEM > 1)
  4. Number of data dependences is low
  5. Branches are easy to predict
- The processor can only improve #1 to a certain level with out-of-order scheduling and partly #2 with hardware prefetching
- Compiler optimizations effectively target #1-3
- The programmer can help improve #1-5

Strategies

- loop unrolling $\rightarrow$ unroll the loop
- loop merging $\rightarrow$ merge loops into a single loop
- loop splitting $\rightarrow$ decompose complex loops
- function inlining $\rightarrow$ avoid breaking instruction flow

Loop unrolling

- Repeat the body of a loop k times and go through the loop with a step length k
- k is called the unrolling factor

```
do j = 1, nj                    -> do j = 1, nj
do i = 1, ni                    -> do i = 1, ni, 2
 a(i,j)=a(i,j)+c*b(i,j)         ->  a(i  ,j)=a(i  ,j)+c*b(i  ,j)
                                ->  a(i+1,j)=a(i+1,j)+c*b(i+1,j)
```

  - The unrolled version of the loop has increased code size, but in turn, will execute fewer overhead instructions.
  - The same number of operations, but the loop index is incremented half of the times
  - The performance of this loop depends upon both the trace cache and L1 cache state.
  - In general the unrolled version runs faster because fewer overhead instrunctions are executed.
  - It is not valid when data dependences exist.

Reduction & Unroll

```
    do j = i, nj ! normal case 1)
       do i = i, ni
          somma = somma + a(i,j)
       end do
    end do
.....
    do j = i, nj !reduction to 4 elements.. 2)
       do i = i, ni, 4
          somma_1 = somma_1 + a(i+0,j)
          somma_2 = somma_2 + a(i+1,j)
          somma_3 = somma_3 + a(i+2,j)
          somma_4 = somma_4 + a(i+3,j)
       end do
    end do
    somma = somma_1 + somma_2 + somma_3 + somma_4
f77 -native -O2 (-O4)
time 1) ---> 4.49785 (2.94240)
time 2) ---> 3.54803 (2.75964)
```

What inhibits loop unrolling?

- Conditional jumps (`if` ...)
- Calls to intrinsic functions and library (sin, exp, ...)
- I/0 operations in the loop

Avoid Branches! I

- Tight loops: loops containing few operations. Ideal candidates for pipelining or loop unrolling.
- Compiler cannot do this in presence of branch statements.
- Branch prediction can guess wrong branch (branch miss).

Avoid Branches! II

```
int i, j, sign;
double A[N][N], B[N], C[N];
for (j = 0; j < N; j++) {
 for (i = 0; i < N; i++) {
  if (i >= j) {
    sign = 1;
  } else if (i < j) {
    sign = -1;
  } else {
    sign = 0;
  }
  C[j] += sign*A[i][j]*B[i];
 }
}
```

```
int i, j, sign;
double A[N][N], B[N], C[N];
for (j = 0; j < N; j++) {
 for (i = j+1; i < N; i++){
   C[j] += A[i][j]*B[i];
 }
}


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

Compiler options

- Can I know how compiler works?

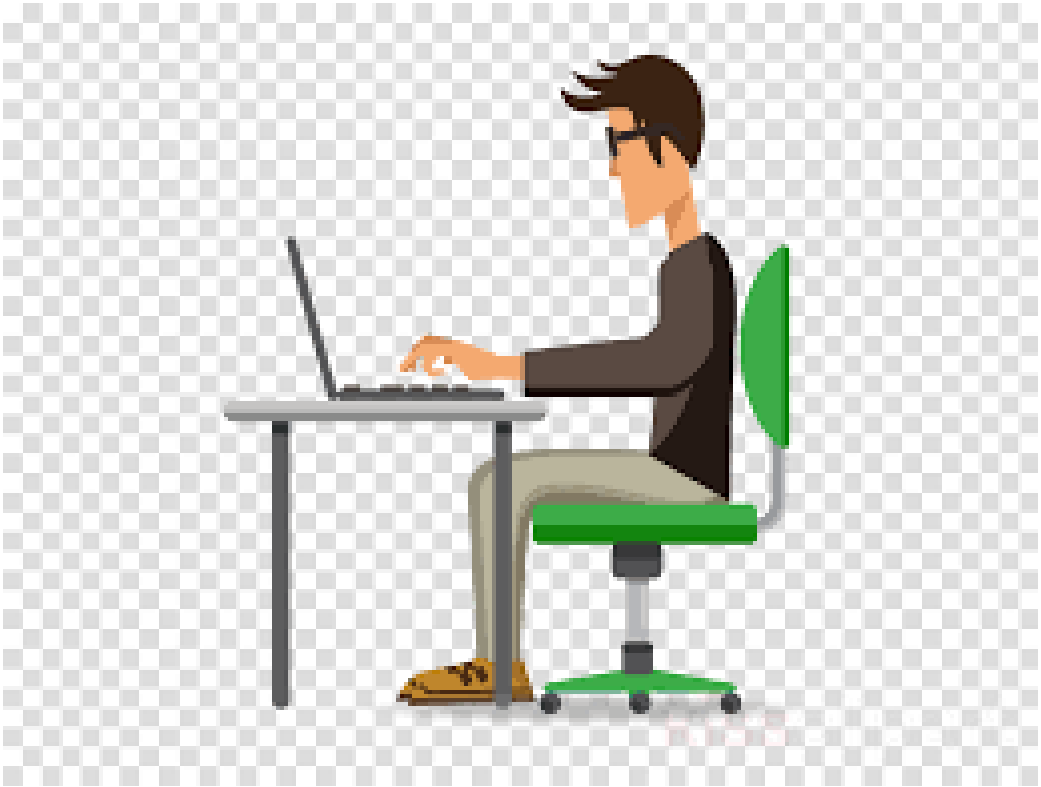
Compiler options

- Can I know how compiler works?
- See reference documentation for the compiler.

Compiler options

- Can I know how compiler works?
- See reference documentation for the compiler.
- Use, for example, the intel compiler with flag **-qopt-report**.

Hands-on 3



Outline

Programming languages

- Many programming languages were defined...
- http://foldoc.org/contents/language.html

```
20-GATE; 2.PAK; 473L Query; 51forth; A#; A-0; a1; a56;
Abbreviated Test Language for Avionics Systems; ABC;
ABC ALGOL; ABCL/1; ABCL/c+; ABCL/R; ABCL/R2; ABLE;
ABSET; abstract machine; Abstract Machine Notation;
abstract syntax; Abstract Syntax Notation 1;
Abstract-Type and Scheme-Definition Language; ABSYS;
Accent; Acceptance, Test Or Launch Language; Access;
ACOM; ACOS; ACT++; Act1; Act2; Act3; Actalk; ACT ONE;
Actor; Actra; Actus; Ada; Ada++; Ada 83; Ada 95; Ada 9X;
Ada/Ed; Ada-O; Adaplan; Adaplex; ADAPT; Adaptive Simulated
Annealing; Ada Semantic Interface Specification;
Ada Software Repository; ADD 1 TO COBOL GIVING COBOL;
ADELE; ADES; ADL; AdLog; ADM; Advanced Function Presentation;
Advantage Gen; Adventure Definition Language; ADVSYS; Aeolus;
AFAC; AFP; AGORA; A Hardware Programming Language; AIDA;
AIr MAterial COmmand compiler; ALADIN; ALAM; A-language;
A Language Encouraging Program Hierarchy; A Language for Attributed ...
```

Programming languages

- Interpreted:
  - statement by statement translation during code execution
  - no way to perform optimization between different statements
  - easy to find semantic errors
  - e.g. scritping languages, Java (bytecode),. . .
- Compiled:
  - code is translated by the compiler before the execution
  - possibility to perform optimization between different statements
  - e.g. Fortran, C, C++

CPU/1

- It is composed by (first approximation):
  - Registers: hold instruction operands
  - Functional units: performs instructions
- Functional units
  - logical operations (bitwise)
  - integer arithmetic
  - floating-point arithmetic
  - computing address
  - load & store operation
  - branch prediction and branch execution

CPU/2

- RISC: Reduced Instruction Set CPU
  - simple "basic" instructions
  - one statement $\rightarrow$ many istructions
  - simple decode and execution
- CISC: Complex Instruction Set CPU
  - many "complex" instructions
  - one statement $\rightarrow$ few istructions
  - complex decode and execution

Architecture vs. Implementation

- Architecture:
  - instruction set (ISA)
  - registers (integer, floating point, . . .)
- Implementation:
  - physical registers
  - clock & latency
  - # of functional units
  - Cache's size & features
  - Out Of Order execution, Simultaneous Multi-Threading, ...
- Same architecture, different implementations:
  - Power: Power3, Power4, . . ., Power8
  - x86: Pentium III, Pentium 4, Xeon, Pentium M, Pentium D, Core, Core2, Athlon, Opteron, . . .
  - different performances
  - different way to improve performance

The Compiler

- "Translate" source code in an executable
- Rejects code with syntax errors
- Warns (sometimes) about "semantic" problems
- Try (if allowed) to optimize the code
  - code independent optimization
  - code dependent optimization
  - CPU dependent optimization
  - Cache & Memory oriented optimization
  - Hint to the CPU (branch prediction)
- It is:
  - powerfull: can save programmer's time
  - complex: can perform "complex" optimization
  - limited: it is an expert system but can be fooled by the way you write the code . . .

Building an executable

A three-step process:

1. Pre-processing:
   - every source code is analyzed by the pre-processor
     - MACROs substitution (**`#define`**)
     - code insertion for **`#include`** statements
     - code insertion or code removal (**`#ifdef`** . . .)
     - removing comments . . .
2. Compiling:
   - each code is translated in object files
     - object files is a collection of "symbols" that refere to variables/function defined in the program
3. Linking:
   - All the object files are put together to build the finale executable
   - Any symbol in the program must be resolved
     - the symbols can be defined inside your object files
     - you can use other object file (e.g. external libraries)

Example: gfortran compilation

- With the command:

```
user@caspur$> gfortran dsp.f90 dsp_test.f90 -o dsp.x
```

all the three steps (preprocessing, compiling, linking) are performed at the same time

- Pre-processing

```
user@caspur$> gfortran -E -cpp dsp.f90
user@caspur$> gfortran -E -cpp dsp_test.f90
```

  - `-E -cpp` options force `gfortran` to stop after pre-processing
  - no need to use `-cpp` if file extension is `*.F90`

- Compiling

```
user@caspur$> gfortran -c dsp.f90
user@caspur$> gfortran -c dsp_test.f90
```

  - `-c` option force `gfortran` only to pre-processing and compile
  - from every source file an object file `*.o` is created

Example: gfortran linking

- Linking: we must use object files

```
user@caspur$> gfortran dsp.o dsp_test.o -o dsp.x
```

- To solve symbols from external libraries
  - suggest the libraries to use with option `-l`
  - suggest the directory where looking for libraries with option `-L`
- e.g.: link `libdsp.a` library located in `/opt/lib`

```
user@caspur$> gfortran file1.o file2.o -L/opt/lib -ldsp -o dsp.x
```

- How create and link a static library

```
user@caspur$> gfortran -c dsp.f90
user@caspur$> ar curv libdsp.a dsp.o
user@caspur$> ranlib libdsp.a
user@caspur$> gfortran test_dsp.f90 -L. -ldsp
```

- `ar` creates the archive `libdsp.a` containing `dsp.o`
- `ranlib` builds the library

Compiler: Transformation Issues

For a compiler to apply an optimization to a program, it must do three things:

1. **Decide** upon a part of the program to optimize and a particular transformation to apply to it;
2. **Verify** that the transformation either does not change the meaning of the program or changes it in a restricted way that is acceptable to the user; and
3. **Transform** the program.

Compiler: what it can do

- It performs many code modifications
  - Register allocation
  - Register spilling
  - Copy propagation
  - Code motion
  - Dead and redundant code removal
  - Common subexpression elimination
  - Strength reduction
  - Inlining
  - Index reordering
  - Loop pipelining , unrolling, merging
  - Cache blocking
  - . . .
- Everything is done to maximize performances!!!

Compiler: what it cannot do

- Global optimization of "big" source code, unless switch on interprocedural analisys (IPO) but it is very time consuming . . .
- Understand and resolve complex indirect addressing
- Strenght reduction (with non-integer values)
- Common subexpression elimination through function calls
- Unrolling, Merging, Blocking with:
  - functions/subroutine calls
  - I/O statement
- Implicit function inlining
- Knowing at run-time variabile's values

Optimizations: levels

- All compilers have “predefined” optimization levels **`-O<n>`**
  - with **`n`** from 0 a 3 (IBM compiler up to 5)
- Usually :
  - **`-O0`**: no optimization is performed, simple translation (tu use with **`-g`** for debugging)
  - **`-O`**: default value (each compiler has it’s own default)
  - **`-O1`**: basic optimizations
  - **`-O2`**: memory-intensive optimizations
  - **`-O3`**: more aggressive optimizations, it can alter the instruction order (see floating point section)
- Some compilers have **`-fast/-Ofast`** option (**`-O3`** plus more options)

Intel compiler: -O3 option

```
icc (or ifort) -O3
```

- Automatic vectorization (use of packed SIMD instructions)
- Loop interchange (for more efficient memory access)
- Loop unrolling (more instruction level parallelism)
- Prefetching (for patterns not recognized by h/w prefetcher)
- Cache blocking (for more reuse of data in cache)
- Loop peeling (allow for misalignment)
- Loop collapsing (for loop count)
- Memcpy recognition (call Intel's fast memcpy, memset)
- Loop splitting (facilitate vectorization)
- Loop fusion (more efficient vectorization)
- Scalar replacement (reduce array accesses by scalar temps)
- Loop rerolling (enable vectorization)
- Loop reversal (handle dependencies)

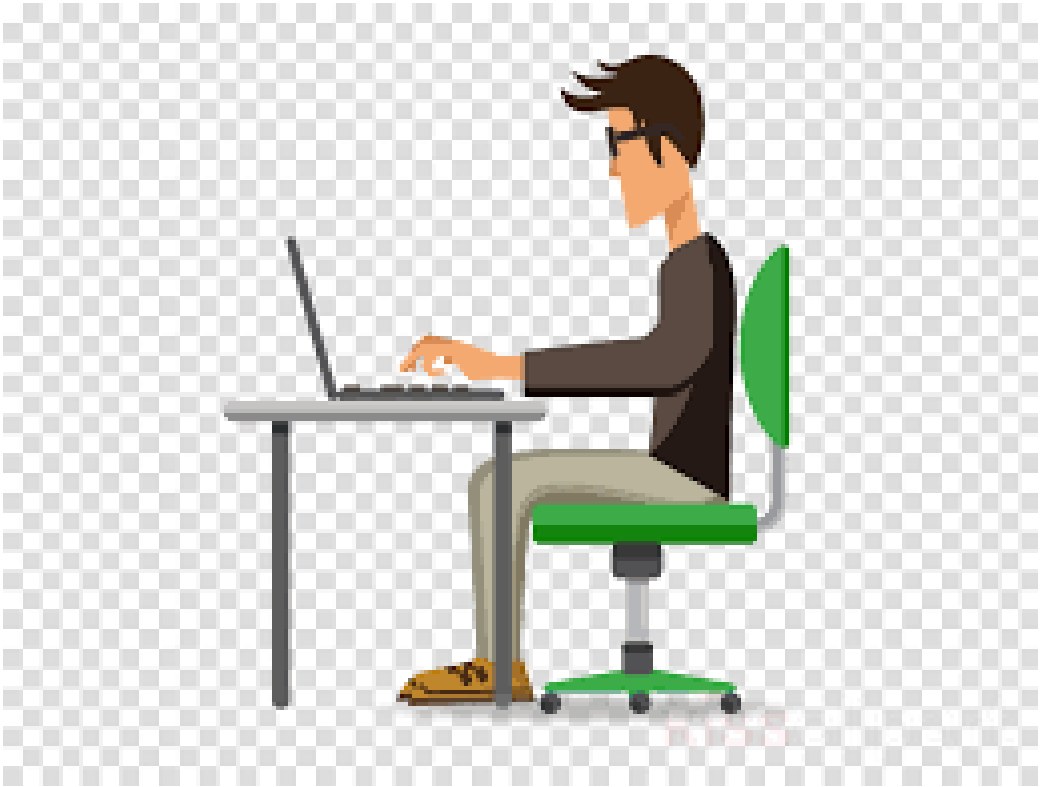
From source code to executable

- Executable (i.e. istructions performed by CPU) is very very different from what you think writing a code
- Example: matrix-matrix production

```
do j = 1, n
   do k = 1, n
      do i = 1, n
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
   end do
end do
```

- Computational kernel
  - load from memory three numbers
  - perform one product and one sum
  - store back the result

Hands-on 4



Hands-on: compiler optimization flags

- Matrix-Matrix product, 1024×1024, double precision
- Cache friendly loop
- The Code is in **`matrixmul`** directory (both C & Fortran)
- to load compiler:
  - GNU –> **`gfortran, gcc : module load gnu`**
  - Intel –> **`ifort, icc : module load intel`**
  - You can load one compiler at time, **`module purge`** to remove previous compiler

	GNU	Intel	GNU	Intel
flags	seconds	seconds	GFlops	GFlops
-O0				
-O1				
-O2				
-O3				
-O3 -funroll-loops		—		—
-Ofast	—		—-	

Matmul: performance

- Size 1024×1024, duoble precision
- Fortran core, cache friendly loop
  - FERMI: IBM Blue Gene/Q system, single-socket PowerA2 with 1.6 GHz, 16 core
  - PLX: 2 esa-core XEON 5650 Westmere CPUs 2.40 GHz

FERMI - xlf

Option	seconds	Mflops
-O0	65.78	32.6
-O2	7.13	301
-O3	0.78	2735
-O4	55.52	38.7
-O5	0.65	3311

PLX - ifort

Option	seconds	MFlops
-O0	8.94	240
-O1	1.41	1514
-O2	0.72	2955
-O3	0.33	6392
-fast	0.32	6623

- Why ?

Compiler: report

- What happens at different optimization level?
  - Why performance degradation using **`-O4`**?
- Hint: use report flags to investigate
- Using IBM **`-qreport`** flag for **`-O4`** level shows that:
  - The compiler understant matrix-matrix pattern (it is smart) ad perform a substitution with external BLAS function (**`__xl_dgemm`**)
  - But it is slow because it doesn't belong to IBM optimized BLAS library (ESSL)
  - At **`-O5`** level it decides not to use external library
- As general rule of thumb performance increase as the optimization level increase . . .
  - . . . but it's bettet to check!!!

Who does the dirty work?

- option **`-fast`** (ifort on PLX) produce a $\simeq$ 30x speed-up respect to option **`-O0`**
  - many different (and complex) optimizations are done . . .
- **Hand-made optimizations?**
- The compiler is able to do
  - Dead code removal: removing branch

```
b = a + 5.0;
if ((a>0.0) && (b<0.0)) {
    ......
}
```

  - Redudant code removal

```
integer, parameter :: c=1.0
f=c*f
```

- **But coding style can fool the compiler**

Loop counters

- Always use the correct data type
- If you use as loop index a real type means to perform a implicit casting real $\rightarrow$ integer every time
- I should be an error according to standard, but compilers are (sometimes) sloppy...

```
real :: i,j,k
....
do j=1,n
do k=1,n
do i=1,n
c(i,j)=c(i,j)+a(i,k)*b(k,j)
enddo
enddo
enddo
```

Time in seconds

compiler/level	integer	real
(PLX) gfortran -O0	9.96	8.37
(PLX) gfortran -O3	0.75	2.63
(PLX) ifort -O0	6.72	8.28
(PLX) ifort -fast	0.33	1.74
(PLX) pgif90 -O0	4.73	4.85
(PLX) pgif90 -fast	0.68	2.30
(FERMI) bgxlf -O0	64.78	104.10
(FERMI) bgxlf -O5	0.64	12.38

Compilers limitations

- A compiler can do a lot of work . . . but it is a program
- It is easy to fool it!
  - loop body too complex
  - loop values not defined a compile time
  - to much nested `if` structure
  - complicate indirect addressing/pointers

index reordering

- For simple loops there's no problem
  - . . . using appropriate optimization level

```
do i=1,n
   do k=1,n
      do j=1,n
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
   end do
end do
```

- Time in seconds

	j-k-i	i-k-j
(PLX) ifort -O0	6.72	21.8
(PLX) ifort -fast	0.34	0.33

index reordering/2

- For more complicated loop nesting could be a problem . . .
  - also at higher optimization levels
  - solution: always write cache friendly loops, if possible

```
do jj = 1, n, step
   do kk = 1, n, step
      do ii = 1, n, step
         do j = jj, jj+step-1
            do k = kk, kk+step-1
               do i = ii, ii+step-1
                  c(i,j) = c(i,j) + a(i,k)*b(k,j)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

- Time in seconds

Otimization level	j-k-i	i-k-j
(PLX) ifort -O0	10	11.5
(PLX) ifort -fast	1.	2.4

Loop splitting

Also called loop distribution or loop fission: breaks a single loop into multiple loops with the same iteration space.

original loop

```
do i=1, n
    a[i] = a[i]+c
    x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

after splitting loop

```
do all i=1, n
  a[i] = a[i]+c
end do all
do i=1, n
  x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

- create perfect loop nests;
- create sub-loops with fewer dependences;
- improve instruction cache due to shorter loop bodies;
- increase register re-use by decreasing register pressure;

Loop fusion

The inverse transformation of splitting also called jamming.
Needs compatible loops

Original loop

```
for(i = 1 to 100)
   a[i] =
for(j = 1 to 100)
    b[j] =
```

Fused loop

```
 for(i = 1 to 100)
   a[i] =
   b[i] =
```

- reducing loop overhead
- improving register use
- increasing instruction parallelism;
- fork/join based parallelisation
- improving the load balance of parallel loops

Loop unrolling

Replicate he body of a loop some number of times called the unrolling factor (u), and then iterates by step u instead of step 1

Original loop

```
for(i = 1 to 100)
  a[i] = i
```

Loop unrolled twice

```
for(i = 1 to 100 step 2)
  a[i] = i
  a[i+1] = i+1
```

- reducing loop overhead;
- increasing instruction parallelism;
- improving register, data cache locality.

Loop collapsing

The number of dimensions of the array is actually reduced

Original loop

```
for (i=0;i<100;++i)
  for (j=0;j<50;++j)
```

Collapsed loop

```
for (ij=0;ij<5000;++ij)
```

- eliminates the overhead of multiple nested loops and multi-dimensional araray indexing.
- increase the number of parallelizable loop iterations

Loop peeling

- to remove dependences created by the first or last few loop iterations
  - thereby enabling parallelization
- to match the iteration control of adjacent loops to enable fusion.

Original loop

```
do i=2,n
   b(i) = b(i) + b(2)
end do
do (parallel) i = 3,n
   a(i) = a(i) + c
end do
```

Peeled loop

```
 if( 2<=n) then
   b(2) = b(2) + b(2)
end do
do (parallel) i = 3,n
   b(i) = b(i) + b(2)
   a(i) = a(i) + c
end do
```

after peeling one iteration from first loop and fusing the resulting loops

Loop rerolling

Original code

```
for (int i = 0; i < 3200; i += 5) {
  a[i] += alpha * b[i];
  a[i + 1] += alpha * b[i + 1];
  a[i + 2] += alpha * b[i + 2];
  a[i + 3] += alpha * b[i + 3];
  a[i + 4] += alpha * b[i + 4];
}
```

Rerolled code

```
for (int i = 0; i < 3200; ++i) {
  a[i] += alpha * b[i];
}
```

- Code-size reduction (especially relevant, obviously, when compiling for code size).
- Providing greater choice to the loop vectorizer (and generic unroller) to choose the unrolling factor (and a better ability to vectorize).

Loop reversal (1)

Reverse loop direction

Original code

```
for(i = 1 to N)
for(j = 1 to M)
 a[i,j] = a[i,j-1]+b[i]
```

New code

```
for(i = N to 1 step -1)
for(j = 1 to M)
 a[i,j] = a[i,j-1]+b[i]
```

Loop reversal (2)

- Rarely used in isolation
- Can combine interchange and reversal

Original cose: interchange is not possible

```
do i = 1, n
do j = 1, n
  a[i,j] = a[i-1, j+1] + 1
end do
end do
```

New code:inner loop reversed. Loops can be interchanged

```
do i = 1, n
do j = n, 1, -1
 a[i,j] = a[i-1, j+1] + 1
end do
end do
```

Copy propagation

Propagates the original name of the value and eliminates redudant coipies

Original code

```
t = i*4
s = t
print *, a[s]
r = t
a[r] = a[r] + c
```

After copy propagation

```
t = i*4
print *, a[t]
a[t] = a[t] + c
```

- reduces register pressure
- eliminates redundant register-to-register move instructions

Different operations, different latencies

For a CPU different operations present very different latencies

- sum: few clock cycles
- product: few clock cycles
- sum+product: few clock cycles
- division: many clock cycle ($O(10)$)
- sin,sos: many many clock cycle ($O(100)$)
- exp,pow: many many clock cycle ($O(100)$)
- I/O operations: many many many clock cycles ($O(1000 - 10000)$)

Strength reduction

- Optimization of arithmetic expression by replacing computationally expensive operations with cheaper ones.
- When applied to loops it can magnify performance impact

Unoptimized

```
z(i) = x(i)/y(i)
w(i) = u(i)/v(i)
```

Optimized

```
tmp = 1.0/(y(i)*v(i))
z(i) = x(i)*v(i)*tmp
w(i) = u(i)*y(i)*tmp
```

Cache & subroutine

```
do i=1,nwax+1
    do k=1,2*nwaz+1
       call diffus (u_1,invRe,qv,rv,sv,K2,i,k,Lu_1)
       call diffus (u_2,invRe,qv,rv,sv,K2,i,k,Lu_2)
....
    end do
 end do


 subroutine diffus (u_n,invRe,qv,rv,sv,K2,i,k,Lu_n)
   do j=2,Ny-1
     Lu_n(i,j,k)=invRe*(2.d0*qv(j-1)*u_n(i,j-1,k)-(2.d0*rv(j-1)
                +K2(i,k))*u_n(i,j,k)+2.d0*sv(j-1)*u_n(i,j+1,k))
   end do
 end subroutine
```

- non unitary access (stride MUST be $\simeq 1$)

Cache & subroutine/2

```
call diffus (u_1,invRe,qv,rv,sv,K2,Lu_1)
call diffus (u_2,invRe,qv,rv,sv,K2,Lu_2)
....

subroutine diffus (u_n,invRe,qv,rv,sv,K2,i,k,Lu_n)
 do k=1,2*nwaz+1
  do j=2,Ny-1
   do i=1,nwax+1
    Lu_n(i,j,k)=invRe*(2.d0*qv(j-1)*u_n(i,j-1,k)-(2.d0*rv(j-1)
               +K2(i,k))*u_n(i,j,k)+2.d0*sv(j-1)*u_n(i,j+1,k))
   end do
  end do
 end do
end subroutine
```

- "same" results as the the previous one
- stride = 1
- Sometimes compiler can perform the transformations, but `inlining` option must be activated

Inlining

- means to substitue the functon call with all the instruction
  - no more jump in the program
  - help to perform interpocedural analysis
- the keyword **`inline`** for C and C++ is a “hint” for compiler
- Intel (n: 0=disable, 1=inline functions declared, 2=inline any function, at the compiler’s discretion)

```
-inline-level=n
```

- GNU (n: size, default is 600):

```
-finline-functions
-finline-limit=n
```

- It varies from compiler to compiler, read the manpage . . .

Common Subexpression Elimination

- Using Common Subexpression for intermediate results:
  A= B+C+D
  E= B+F+C

- ask for: 4 load, 2 store, 4 sums
  A=(B+C) + D
  E=(B+C) + F
- ask for 4 load, 2 store, 3 sums, 1 intermediate result.
- WARNING: with floating point arithmetics results can be different
- “Scalar replacement” if you access to a vector location many times
  - compilers can do that (at some optimization level)

Functions & Side Effects

- Functions returns a values but
  - sometimes global variables are modified
  - I/O operations can produce side effects
- side effects can “stop” compiler to perform inlining
- Example (no side effect):

```
function f(x)
   f=x+dx
end
```

so `f(x)+f(x)+f(x)` it is equivalent to `3*f(x)`

- Example (side effect):

```
function f(x)
   x=x+dx
   f=x
end
```

so `f(x)+f(x)+f(x)` it is different from `3*f(x)`

CSE & function

- reordering function calls can produce different results
- It is hard for a compiler understand is there are side effects
- Example: 5 calls to functons, 5 products:

```
x=r*sin(a)*cos(b);
y=r*sin(a)*sin(b);
z=r*cos(a);
```

- Example: 4 calls to functons, 4 products, 1 temporary variable:

```
temp=r*sin(a)
x=temp*cos(b);
y=temp*sin(b);
z=r*cos(a);
```

- Correct if there's no side effect!

CSE: limitations

- Core loop too wide:
  - Compiler is able to handle a fixed number of lines: it could not realize that there's room for improvement
- Functions:
  - there is a side effect?
- CSE mean to alter order of operations
  - enabled at "high" optimization level (`-qnostrict` per IBM)
  - use parentheis to "inhibit" CSE
- "register spilling": when too much intermediate values are used

What can do a compiler?

```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         acc =1./(1.-coe*aciv(i)*(1.-int(forclo(nve,i,j,k))))
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         ami(jj,i)=-coe*amiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         fi(jj,i)=qcap(i,j,k)*acc
     enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         acc =1./(1.-coe*ackv(k)*(1.-int(forclo(nve,i,j,k))))
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
         amk(jj,k)=-coe*amkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
```

. . . this . . .

```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*aciv(i)*temp)
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*temp
         ami(jj,i)=-coe*amiv(i)*acc*temp
         fi(jj,i)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*ackv(k)*temp)
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*temp
         amk(jj,k)=-coe*amkv(k)*acc*temp
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
```

. . . but not that!!! (20% faster)

```
do k=1,n3m
   do j=n2i,n2do
      do i=1,n1m
         temp_fact(i,j,k) = 1.-int(forclo(nve,i,j,k))
      enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         temp = temp_fact(i,j,k)
         acc =1./(1.-coe*ackv(k)*temp)
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*temp
         amk(jj,k)=-coe*amkv(k)*acc*temp
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
...
...
!  the same for the other loop
```

Array Syntax

- in place 3D-array translation (512^3)
- Explixcit loop (Fortran77): 0.19 seconds
  - CAVEAT: the loop order is “inverse” in order not to overwirte data

```
do k = nd, 1, -1
  do j = nd, 1, -1
    do i = nd, 1, -1
      a03(i,j,k) = a03(i-1,j-1,k)
    enddo
  enddo
enddo
```

- Array Syntax (Fortran90): 0.75 seconds
  - According to the Standard $\rightarrow$ store in an intermediate array to avoid to overwrite data

```
a03(1:nd, 1:nd, 1:nd) = a03(0:nd-1, 0:nd-1, 1:nd)
```

- Array Syntax with hint: 0.19 seconds

```
a03(nd:1:-1,nd:1:-1,nd:1:-1) = a03(nd-1:0:-1, nd-1:0:-1, nd:1:-1)
```

Optimization Report/1

- A report of optimization performed can help to find “problems”
- Intel

```
-qopt-report[n]      n=0(none),1(min),2(med),3(max)
-qopt-report-file<file>
-qopt-report-phase<phase>
-qopt-report-routine<routine>
```

- one or more `*.optrpt` file are generated

```
...
Loop at line:64 memcopy generated
...
```

- Is this `memcopy` necessary?

Optimization Report/2

- There's no equivalent flag for GNU compilers
  - Best solution:

```
-fdump-tree-all
```

  - dump all compiler operations
  - very hard to understand

- PGI compilers

```
-Minfo
-Minfo=accel,inline,ipa,loop,opt,par,vect
```

Info at standard output

Give hints to compiler

- Loop size known at compile-time o run-time
  - Some optimizations (like unrolling) can be inhibited

```
real a(1:1024,1:1024)
real b(1:1024,1:1024)
real c(1:1024,1:1024)
...
read(*,*) i1,i2
read(*,*) j1,j2
read(*,*) k1,k2
...
do j = j1, j2
do k = k1, k2
do i = i1, i2
c(i,j)=c(i,j)+a(i,k)*b(k,j)
enddo
enddo
enddo
```

- Time in seconds (Loop Bounds Compile-Time o Run-Time)

flag	LB-CT	LB-RT
(PLX) ifort -O0	6.72	9
(PLX) ifort -fast	0.34	0.75

- WARNING: compiler dependent...

Static vs. Dynamic allocation

- Static allocation gives more information to compilers
  - but the code is less flexible
  - recompile every time is really boring

```
integer :: n
parameter(n=1024)
real a(1:n,1:n)
real b(1:n,1:n)
real c(1:n,1:n)
```

```
real, allocatable, dimension(:,:) :: a
real, allocatable, dimension(:,:) :: b
real, allocatable, dimension(:,:) :: c
print*,'Enter matrix size'
read(*,*) n
allocate(a(n,n),b(n,n),c(n,n))
```

Static vs. Dynamic allocation/2

- for today compilers there's no big difference
  - Matrix-Matrix Multiplication (time in seconds)

	static	dynamic
(PLX) ifort -O0	6.72	18.26
(PLX) ifort -fast	0.34	0.35

- With static allocation data are put in the "stack"
  - at run-time take care of stacksize (e.g. segmentation fault)
  - bash: to check

```
ulimit -a
```

  - bash: to modify

```
ulimit -s unlimited
```

Dynamic allocation using C/1

- Using C matrix → arrays of array
  - with static allocation data are contiguos (columnwise)

```
double A[nrows][ncols];
```

- with dynamic allocation ....
  - “the wrong way”

```
/* Allocate a double matrix with many malloc */
double** allocate_matrix(int nrows, int ncols) {
   double **A;
   /* Allocate space for row pointers */
   A = (double**) malloc(nrows*sizeof(double*) );
   /* Allocate space for each row */
   for (int ii=1; ii<nrows; ++ii) {
      A[ii] = (double*) malloc(ncols*sizeof(double));
   }
   return A;
}
```

Dynamic allocation using C/2

- allocate a linear array

```
/* Allocate a double matrix with one malloc */
double* allocate_matrix_as_array(int nrows, int ncols) {
   double *arr_A;
   /* Allocate enough raw space */
   arr_A = (double*) malloc(nrows*ncols*sizeof(double));
   return arr_A;
}
```

- using as a matrix (with index linearization)

```
arr_A[i*ncols+j]
```

- MACROs can help
- also use pointers

```
/* Allocate a double matrix with one malloc */
double** allocate_matrix(int nrows, int ncols, double* arr_A) {
   double **A;
   /* Prepare pointers for each matrix row */
   A = new double*[nrows];
   /* Initialize the pointers */
   for (int ii=0; ii<nrows; ++ii) {
      A[ii] = &(arr_A[ii*ncols]);
   }
   return A;
}
```

Aliasing & Restrict

- Aliasing: when two pointers point at the same area
- Aliasing can inhibit optimization
  - you cannot alter order of operations
- C99 standard introduce **`restrict`** keyword to point out that aliasing is not allowed

```
void saxpy(int n, float a, float *x, float* restrict y)
```

- C++: aliasing not allowed between pointer to different type (strict aliasing)

Input/Output

- Handled by the OS:
  - many system calls
  - pipeline goes dry
  - cache coerency can be destroyed
  - it is very slow (HW limitation)

- Golden Rule #1: NEVER mix computing with I/O operations
- Golden Rule #2: NEVER read/write a single data, pack them in a block

Different I/O

```
do k=1,n  ;  do j=1,n  ;  do i=1,n
write(69,*) a(i,j,k)                                  ! formatted I/O
enddo     ;  enddo     ;  enddo

do k=1,n  ;  do j=1,n  ;  do i=1,n
write(69) a(i,j,k)                                    ! binary I/O
enddo     ;  enddo     ;  enddo

do k=1,n  ;  do j=1,n
write(69) (a(i,j,k),i=1,n)                            ! by colomn
enddo     ;  enddo

do k=1,n
write(69) ((a(i,j,k),i=1),n,j=1,n)                    ! by matrix
enddo

write(69) (((a(i,j,k),i=1,n),j=1,n),k=1,n)            ! dump (1)

write(69) a                                           ! dump (2)
```

Different I/O: some figures

	seconds	Kbyte
formatted	81.6	419430
binary	81.1	419430
by colunm	60.1	268435
by matrix	0.66	134742
dump (1)	0.94	134219
dump (2)	0.66	134217

- WARNING: the filesystem used could affect performance (e.g. RAID)...

I/O

- read/write operations are slow
- read/write format data are very very slow
- ALWAYS read/write binary data

- Golden Rule #1: NEVER mix computing with I/O operations
- Golden Rule #2: NEVER read/write a single data, pack them in a block
- For HPC is possibile use:
  - I/O libraries: MPI-I/O, HDF5, NetCDF,...

Hands-on 5