1967-9

Advanced School in High Performance and GRID Computing

3 - 14 November 2008

Using Compilers and Profilers to Optimize your Code for Performance.

COZZINI Stefano

CNR-INFM Democritos
c/o SISSA
via Beirut 2-4
34014 Trieste
ITALY

Advanced School in

High Performance

and GRID Computing

OPTIMIZATION TECHNIQUE

Stefano Cozzini

CNR-INFM DEMOCRITOS, Trieste

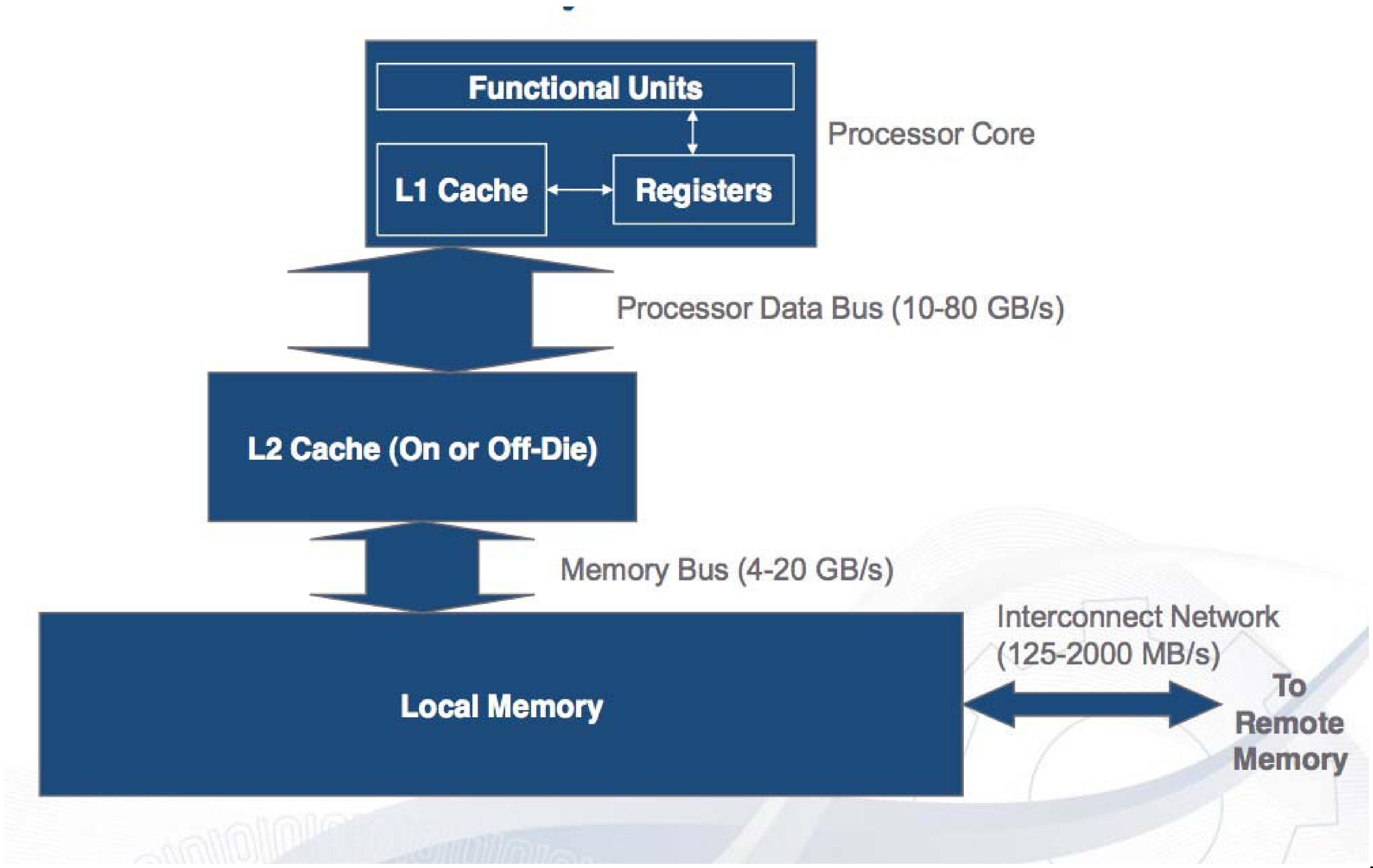
Agenda:

- Introduction
- Performance and evaluating process (Profiling and timing your code)
- Optimization techniques
- General Performance techniques
  - Use of Libraries: see next lecture..

Introduction

- Discuss how to measure performances of your cluster/system
- Discuss performance tuning techniques common to most modern architecture (mainly 32/64 bit commodity processor)
- Using optimization techniques users have control over
  - Code modification
  - Compiler options
- Optimization is a dirty work (and dangerous one for your code...)
- Compiler is your best friend..

Memory hierarchy



Locality of Reference

- ***Most programs have a high degree of locality in their accesses***
- Memory hierarchy tries to exploit locality

- ***Temporal locality:***

  Recently referenced items (instr or data) are likely to be *referenced again* in the near future:

  -iterative loops, subroutines, local variables

  -*working set* concept
- ***Spatial locality:***

  programs access data which is *near to each other*:

  -operations on tables/arrays

  -*cache line size* is determined by spatial locality
- ***Sequential locality:***

  processor executes instructions in *program order*:

  -branches/in-sequence ratio is typically 1 to 5

Performance Evaluation process

- Monitoring System:
  - Use monitoring tools to better understand your machine's limits and usage
    - is the system limit well suited to run my application ?
  - Observe both overall system performance and single-program execution characteristics. Monitoring your own code
    - Is the system doing well ? Is my program running in a pathological situation ?
- Monitoring your own code:
  - Timing the code:
    - timing a whole program (time command :/usr/bin/time)
    - timing a portion (all portions) of the program
  - Profiling the program

What to measure ?

- Scire est Mensurare - J.Keppler
- Examples:
  - Elapsed execution time of (part of) a program
    - Which algorithm runs faster in practice?
  - Number of instructions executed, absolute total or rate per second
    - Compare to theoretical peak performance e.g. instructions per cycle (IPC)
  - Rate of floating point operations per second (Mflops)
    - Compare to theoretical or sustained Mflops peak performance
  - System events, e.g. cache hits, system calls
    - Determine how much and when/where overhead occurs
  - Communication latency
  - Reorganize communications and computation (improve latency hiding)
  - Parallel speedup
    - How effective is parallelism?

Useful Monitoring Commands (Linux)

• **Uptime(1)**	returns information about system usage and user load
• **ps(1)**	lets you see a “snapshot” of the process table
• **top**	process table dynamic display
• free	memory usage
• **vmstat**	memory usage monitor

Monitoring your own code (time)

```
NAME
       time - time a simple command or give resource usage

SYNOPSIS
       time [options] command [arguments...]

DESCRIPTION
       The  time  command runs the specified program command with
       the given arguments.  When command finishes, time writes a
       message  to standard output giving timing statistics about
       this program ..


--------------->time ./a.out
 [program output]

real    0m1.361s
user    0m0.770s
sys     0m0.590s
```

user time: Cpu-time dedicated to your program
sys time: time used by your program to execute system calls
real time: total time aka walltime

User/System/Walltime

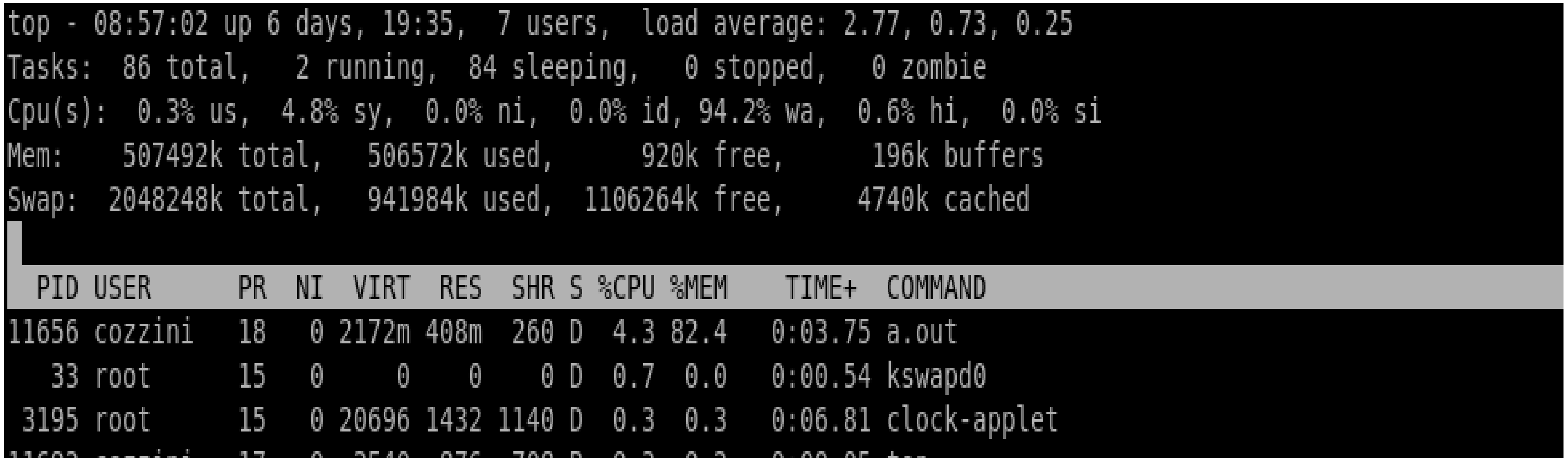
- *Real time* (or *wall clock time*) is the total elapsed time *from start to end of a timed task*
- *CPU user time* is the time spent executing in user space
  - Does not include time spent in system (OS calls) and time spent executing other processes
- *CPU system time* is the time spent executing system *calls (kernel code)*
  - System calls for I/O, devices, synchronization and locking, threading, memory allocation
  - Typically does not include process waiting time for non-ready devices such as disks
- CPU user time + CPU system time < real time
  - CPU percentage spent on process = 100% * (user+system) / real

a top disaster: swapping..

•virtual or swap memory:

This memory, is actually space on the hard drive. The operating system reserves a space on the hard drive for “swap space”.

• time to access virtual memory VERY large:
• this time is done by the system not by your program !
•sometimes the system assumes a killer to kill your program.. (oom killer)



```
top - 08:57:02 up 6 days, 19:35,  7 users,  load average: 2.77, 0.73, 0.25
Tasks:  86 total,   2 running,  84 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.3% us,  4.8% sy,  0.0% ni,  0.0% id, 94.2% wa,  0.6% hi,  0.0% si
Mem:    507492k total,   506572k used,      920k free,      196k buffers
Swap:  2048248k total,   941984k used,  1106264k free,     4740k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11656 cozzini   18   0 2172m 408m  260 D  4.3 82.4   0:03.75 a.out
   33 root      15   0     0    0    0 D  0.7  0.0   0:00.54 kswapd0
 3195 root      15   0 20696 1432 1140 D  0.3  0.3   0:06.81 clock-applet
```

top disaster example (1)

```
[cozzini@stroligo optimization]$ /usr/bin/time ./a.out
 provide an integer (suggested range 100-250)
 larger values can be very memory and time-consuming
300
 inizialisation time= 11.787208
10.86user 0.98system 0:14.22elapsed 83%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (5major+106090minor)pagefaults 0swaps

[cozzini@stroligo optimization]$ /usr/bin/time ./a.out
 provide an integer (suggested range 100-250)
 larger values can be very memory and time-consuming
320
Command terminated by signal 2
0.18user 1.81system 0:29.27elapsed 6%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (5846major+170788minor)pagefaults 0swaps
```

top disaster example (2)

```
[cozzini@stroligo optimization]$ /usr/bin/time ./a.out <300 &
[cozzini@stroligo optimization]$ free
             total       used       free     shared    buffers     cached
Mem:        507492     484916      22576          0       1156      10172
-/+ buffers/cache:     473588      33904
Swap:      2048248      78108    1970140

[cozzini@stroligo optimization]$ /usr/bin/time ./a.out <320 &
[cozzini@stroligo optimization]$ free
             total       used       free     shared    buffers     cached
Mem:        507492     506412       1080          0        252       3936
-/+ buffers/cache:     502224       5268
Swap:      2048248     546348    1501900
```

Timing portion of the code

- Record the time before portion A
- execute portion A
- record the time after portion A
- print/save the difference in time for subsequent analysis

```
clock_t c0, c1;
c0 = clock();
 section to code..
c1= clock();
 cputime = (c1 - c0)/(CLOCKS_PER_SEC );
```

- C function to compute time:
  - clock
- Fortran90 function to compute time:
  - cpu_time routine (f95)

```
call cpu_time(t0)

section to code..
call cpu_time(t1)
cputime = (t1 - t0)
```

Well written codes have their own timing report..

```
!!Specific TIMING for section:  MD INTEGRATION !!!!!!!!!!
              !Serial subroutines :
c
!section               times  avg-time         max(PE)     min(PE)
!vscale                    2    0.0600    0.0600(   0)    0.0600(   0)
!scanpairs_prot          100   72.5500   72.5500(   0)   72.5500(   0)
!vertest_prot            100    2.2600    2.2600(   0)    2.2600(   0)
!link_list                 7   70.2900   70.2900(   0)   70.2900(   0)
!spme_prot               100  727.8700  727.8700(   0)  727.8700(   0)
!fill_charge_gri         100  214.5000  214.5000(   0)  214.5000(   0)
!fft_back                100   79.2700   79.2700(   0)   79.2700(   0)
!scalar_sum              100   43.2400   43.2400(   0)   43.2400(   0)
!fft_forw                100   78.8400   78.8400(   0)   78.8400(   0)
!grad_sum                100  303.6600  303.6600(   0)  303.6600(   0)
!ewcorr_prot             100   24.4000   24.4000(   0)   24.4000(   0)
!ewald3_prot         5870100   15.4300   15.4300(   0)   15.4300(   0)
!pair_force_prot         100    0.0000    0.0000(   0)    0.0000(   0)
!srfew2_prot         5855979  817.5300  817.5300(   0)  817.5300(   0)
!dihfrc_prot             100    3.0800    3.0800(   0)    3.0800(   0)
```

Processes/programs

- A *CPU bound* process is compute intensive
  - Very few I/O operations
  - Execution speed of algorithm is determined by CPU
  - Use performance profiling
- A *Memory bound* process is memory intensive
  - Limited I/O operations and large memory footprint
  - Execution speed is determined by Memory performances
  - Use performance profiling with RAM checkers
- An *I/O bound* process is I/O intensive
  - Process includes I/O operations such as file access, message passing over network
  - Execution speed is limited by system’s I/O latencies
  - Performance analysis method depends on I/O operations:operations on file on disk, messages over pipes and sockets,

Analysis Techniques

- there are three generally available techniques for analyzing code performance:
  - Compiler reports and listings
  - Profiling
  - Hardware performance counters

Compiler Reports and Listings

- Compilers on most modern high performance computers are capable of doing a wide range of optimizations,
- By default, compilers generally do not describe in much detail what kinds of optimizations they were able to perform on a given piece of code.
- However, many compilers will optionally generate *optimization reports* and/or *listing files*.
  - Optimization reports are typically sent to **`stderr`** at compile time and contain messages describing what optimizations could or could not be applied at various points in the source code.
  - Listing files usually consist of a listing of the source code with messages about optimizations interspersed through the listing.

Reporting and Listing Compiler Options

GNU compilers	None
PGI compilers	
`-Minfo=option[,option,...]`	Prints information to `stderr` on `option`; option can be one or more of **`time`**, **`loop`**, **`inline`**, **`sym`**, **or** **`all`**
`-Mneginfo=option[,option]`	Prints information to `stderr` on why optimizations of type `option` were not performed; option can be **`concur`** **or** **`loop`**
`-Mlist`	Generates a listing file
Intel compilers	
`-opt_report`	Generates an optimization report on stderr
`-opt_report_file filename`	Generates an optimization report to **`filename`**

Profiling

- Profiling is an approach to performance analysis in which the amount of time spent in sections of code is measured (using either a sampling technique or on entry/exit of a code block) and presented as a histogram.
- This allows a developer to identify the routines which are taking the most execution time, as these are typically the best candidates for optimization.
- Profiling can done at varying levels of granularity:
  - Subroutine
  - Basic block
  - Source code line
- Profiling usually requires special compilation.
  - The specially compiled executable will generate a file containing execution profile data as it runs.
  - This data file can be analyzed after the code is run
  - a profiling analysis program should be employed

Profiling Compiler Options

```
FORTRAN INTEL:
-prof-dir <d>   specify directory for profiling output files (*.dyn and *.dpi)
-prof-file <f>  specify file name for profiling summary file
-prof-gen       instrument program for profiling
-prof-use       enable use of profiling information during optimization
-qp             compile and link for function profiling with UNIX gprof tool
-p              same as -qp

FORTRAN PGF90
-Mprof[=dwarf|func|hwcts|lines|mpich1|mpich2|time]
                    Generate additional code for profiling
    dwarf           Add limited DWARF info for third party profilers
    func            Function-level profiling
    hwcts           PAPI-based profiling using hardware counters, 64-bit only
    lines           Line-level profiling
    mpich1/2        Use profiled MPI communication library; implies -Mmpich1/2
    time            Sample-based instruction-level profiling

GNU:
-p  Generate extra code to write profile information suitable for the analysis progr
prof.
 -pg Generate extra code to write profile information suitable for the analysis
program gprof.
```

05/11/08

Demo: Profiling (g95)

```
program profiled
  integer,parameter :: N=1000,ntimes=100
  real,dimension(N,N) :: b,c,d
  real,dimension(N) :: a
  real :: begin,end
  real,dimension(2) :: rtime

  call random_number(b)
  call random_number(c)
  call random_number(d)
  begin=dtime(rtime)
  do it=1,ntimes
    do j=1,N
      a(j)=myvsum(b,j)+myvprod(c,j)*myvsum(d,j)
    end do
    if (mod(it,100).eq.0) write (*,*) a(1),b(1,1),c(1,1),d(1,1)
  end do
  end=dtime(rtime)
  write (*,*) "loop time = ",end," seconds"
  flops=(ntimes*N*(3*N+2))/end*1.0e-6
  write (*,*) "loop ran at ",flops," MFLOPS"
```

Demo: Profiling (g95, con't)

```
 contains

real function myvsum(x,j)
  real,dimension(N,N),intent(IN) :: x
  integer,intent(IN) :: j
  myvsum=x(1,j)
  do i=2,N
    myvsum=myvsum+x(i,j)
  end do
end function myvsum

real function myvprod(x,j)
  real,dimension(N,N),intent(IN) :: x
  integer,intent(IN) :: j
  myvprod=x(1,j)
  do i=2,N
    myvprod=myvprod*x(i,j)
  end do
end function myvprod

end program profiled
```

Demo: Profiling (g95 con’t)

```
[cozzini@stroligo optimization]$ g95 -pg profiled.f90
[cozzini@stroligo optimization]$ ./a.out
[cozzini@stroligo optimization]$ gprof a.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 95.63     53.18    53.18        1    53.18    53.18  MAIN_
  2.40     54.52     1.34                             _g95_random_4
  1.96     55.61     1.09                             xorshf96
  0.01     55.61     0.01                             check_seed

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.
```

Hardware Performance Counters

- Most modern microprocessors have one or more event counters which can be used to count low level processor events such as floating point operations, cache line misses, and total instructions executed.
- The output from these counters can be used to infer how well a program is utilizing the processor.
- In many cases, there are utilities for accessing these hardware counters, through either a library or a command line timing interface.

ADVANCED TOPIC FOR II WEEK ACTIVITY

How to optimize...

- Iterative optimization
  - 1. Check for correct answers (program must be correct!)
  - 2. Profile to find the hotspots, e.g. most time-consuming routines
  - 3. Optimize these routines using compiler options, compiler directives (pragmas), and source code modifications
  - Repeat 1-3
- Optimizing the hotspots of a program improves overall performance
- Programs with “flat profiles” (flat timing histogram)
  - Programs with lots of routines that each take a small amount of time are difficult to optimize

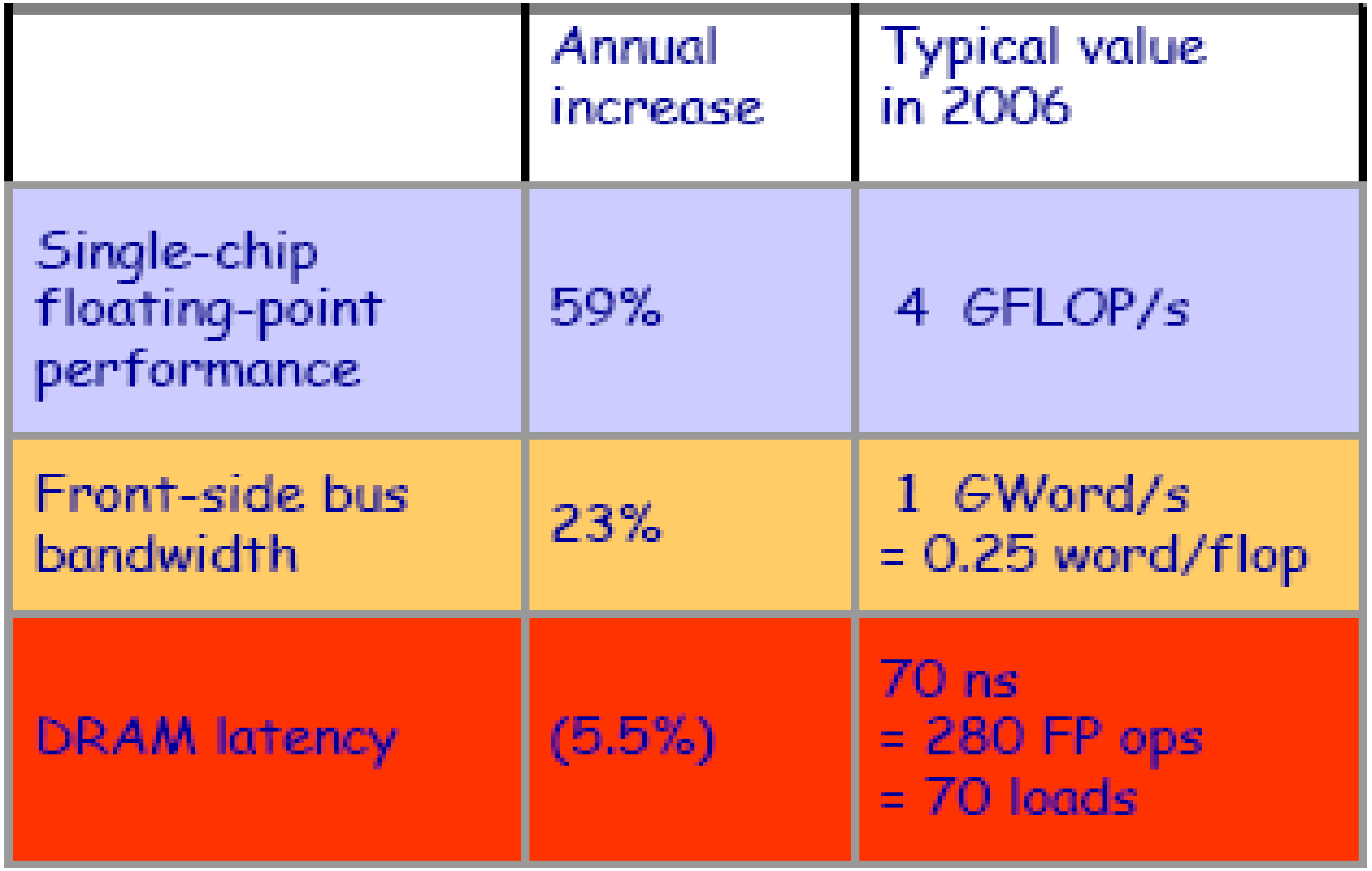
Optimization Methodology

- Optimize one loop/routine at a time
- Start with the most time consuming routines
- Then the second most....
- Then the third most....
- Parallelise your program..

Optimization Techniques

- There are basically three different categories:
  - Improve memory performance (The most important)
    - Better memory access pattern
    - Optimal usage of cache lines (improve spatial locality)
    - Re-usage of cached data (improve temporal locality)
  - Improve CPU performance
    - Create more opportunities to go superscalar (high level)
    - Better instruction scheduling (low level)
  - Use already highly optimized libraries/subroutines

Where to optimize ?



	Annual increase	Typical value in 2006
Single-chip floating-point performance	59%	4 GFLOP/s
Front-side bus bandwidth	23%	1 GWord/s = 0.25 word/flop
DRAM latency	(5.5%)	70 ns = 280 FP ops = 70 loads

Optimization Techniques for memory

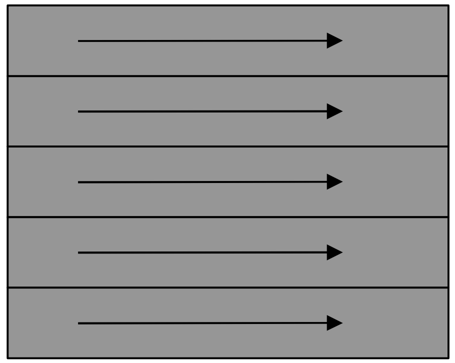
- Loop Interchanges
- Effective Reuse of Data Cache
- Loop Unrolling
- Loop Fusion/Fission
- Prefetching
- Floating Point Division

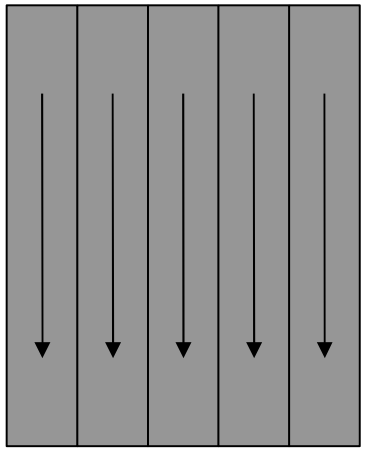
Storage in Memory

The storage order is language dependent:

Fortran stores “column-wise”

C stores “row-wise”





Accessing elements in storage order greatly enhances the performance for problem sizes that do not fit in the cache(s)

(spatial locality: stride 1 access)

Array Indexing

There are several ways to index arrays:

```
Do j=1,M
    Do i=1,N
      ..A(i, j)
    END DO
END DO               Direct
```

```
Do j=1,M
    Do i=1,N
      ..A(i+(j-1)*N)
    END DO
END DO               Explicit
```

```
Do j=1,M
    Do i=1,N
        k=k+1
      ..A(k)
    END DO
END DO               Loop carried
```

```
Do j=1,M
    Do i=1,N
      ..A(index(i,j))..
    END DO
END DO               Indirect
```

The addressing scheme can (and will) have an impact on the performance

Data Dependencies

- Independent instructions can be scheduled at the same time on the multiple execution units in superscalar CPU.
- Independent operations can be (software) pipelined on the CPU

Loop-carried dependencies

```
do i=1,n
     a (index (1,i)) = b(i)
     a (index (2,i)) = c(i)
 end do
```

$$\begin{tikzcd}
\texttt{index(1,i)} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \texttt{index(1,i+k)} \arrow[d, leftrightarrow] & \texttt{Loop-carried dependencies} \\
\texttt{index(2,i)} \arrow[r, leftrightarrow] & \texttt{index(2,i+k)} & \texttt{Non-loop-carried dependencies}
\end{tikzcd}$$

- Standard prog language (F77/F90/C/C$_{++}$) do not provide explicit information on data dependencies.
- Compilers assume worse case for the data dependencies:

  -problem for indirectly addressed arrays in Fortran

  -problem for all pointers C

Loop Interchange

Basic idea: In a nested loop, examine and possibly change the order of the loop

Advantages:

Better memory access patterns (leading to improved cache and memory usage)

Elimination of data dependencies (to increase the opportunities for CPU optimization and parallelization)

Disadvantage:

May make a short loop innermost (which is not good for optimal performances

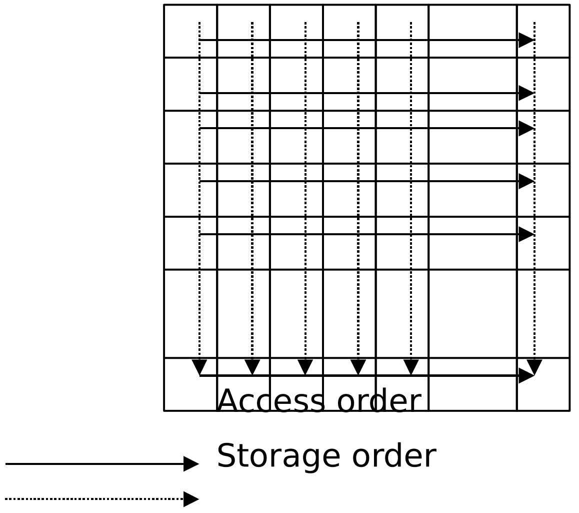
Loop Interchange - Example 1

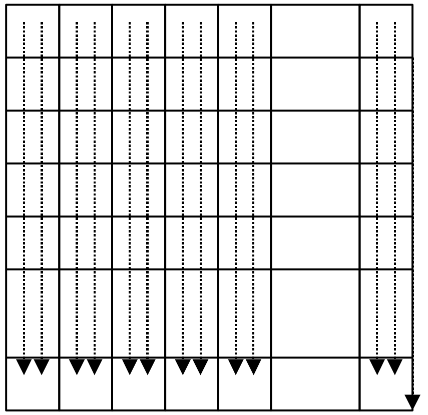
Original

```
DO i=1,N
   DO j=1,M
     C(i,j)=A(i,j)+B(i,j)
   END DO
END O
```

Interchanged loops

```
DO j=1,M
   DO i=1,N
     C(i,j)=A(i,j)+B(i,j)
   END DO
END DO
```





Loop Interchange in C

In C, the situation is exactly the opposite

interchange

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

index reversal

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

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

- The performance benefit is the same in this case
- In many practical situations, loop interchange is much easier to achieve than index reversal

Loop Interchange – Mnemonic rule

- With row-major, the column or "rightmost" index varies most quickly (C/C+)
- With column-major, the row of "leftmost" index varies most quickly.(Fortran/F90)

Loop Interchange - Example 2

```
DO i=1,300
   DO j=1,300
      DO k=1,300
         A (i,j,k) = A (i,j,k)+ B (i,j,k)* C (i,j,k)
      END DO
   END DO
END DO
```

Loop order	x335 (P4 2.4Ghz)	x330 (P3 1.4Ghz)
i j k	8.77	9.06
i k j	7.61	6.82
j i k	2	2.66
j k i	0.57	1.32
k i j	0.9	1.95
k j i	0.44	1.25

Timings are in seconds

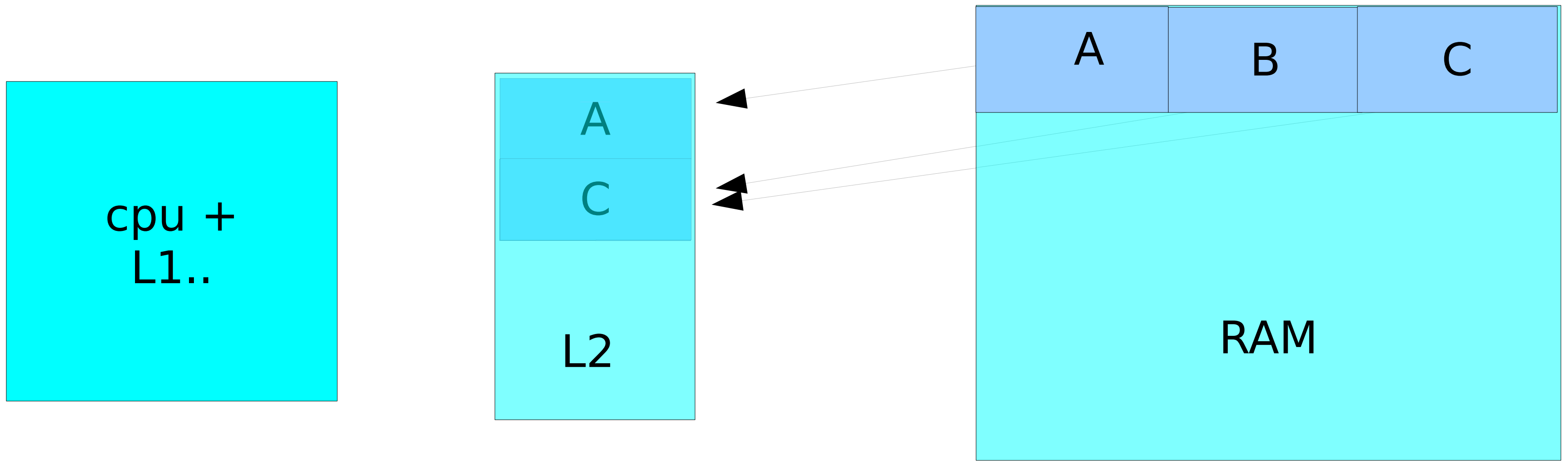
Loop Interchange Compiler Options

GNU compilers:	
	None
PGI compilers:	
`-Mvect`	Enable vectorization, including loop interchange
Intel compilers:	
`-O3`	Enable aggressive optimization, including loop transformations

TEST IF WHAT THEY CLAIM TO DO IS WHAT THEY ACTUALLY DO

Cache Thrashing

- Typical problem in code performance is *cache thrashing*.
- Cache trashing happens when data in cache are rewritten and fully reused.
- In the previous case, the cache thrashing was minimized by loop interchange.
- Another optimization technique aimed at minimizing cache thrashing is *COMMON block padding*.
- consider: a(i) +b(i)=c(i)



demo on cache trashing

- In the next two programs arrays `b` and `c` both fill the cache so when they are used in the addition loop considerable cache thrashing occurs as `c(1)` knocks out the line `b(1)` is in, then `b(2)` knocks out the line `c(1)` was in and so on.
- The second program illustrates the use of common-block padding. By putting a fake array - `space(4)` - between `b` and `c` in the COMMON block we shift the memory addresses of `b` and `c` and thus shift the cache lines they map to. Arrays `b` and `c` no longer collide in the cache.

Demo: Cache Thrashing

```
 program thrash
  integer,parameter :: N=4*1024*1024
  real,dimension(N) :: c,b,a
  real:: begin,end
  real,dimension(2):: rtime
  common/saver/a,b,c
!DIR$ CACHE_ALIGN /saver/
  call random_number(b)
  call random_number(c)
  begin=dtime(rtime)
  do i=1,N
    a(i)=b(i)+c(i)
  end do
  end=dtime(rtime)
  print *,'my loop time (s) is ',end
  flop=N/end*1.0e-6
  print *,'loop runs at ',flop,' MFLOP'
  print *,a(1),b(1),c(1)
 end program thrash
```

Considerable cache thrashing occurs as `c(1)` knocks out the line `b(1)` is in, then `b(2)` knocks out the line `c(1)` was in and so on

Solution: COMMON Block Padding

```
 program pad
  integer,parameter :: N=4*1024*1024
  real,dimension(N) :: c,b,a
  real:: begin,end
  real,dimension(2):: rtime
  common/saver/a,b,space(4),c
!DIR$ CACHE_ALIGN /saver/
  call random_number(b)
  call random_number(c)
  begin=dtime(rtime)
  do i=1,N
    a(i)=b(i)+c(i)
  end do
  end=dtime(rtime)
  print *,'my loop time (s) is ',end
  flop=N/end*1.0e-6
  print *,'loop runs at ',flop,' MFLOP'
  print *,a(1),b(1),c(1)
 end program pad

.2000000
 loop runs at     20.97152      MFLOP
    1.611511       0.9079230       0.7035879
```

By putting the fake array, **`space(4)`**, in the COMMON block, arrays **`b`** and **`c`** no longer collide in the cache.

COMMON Block Padding Compiler Options

GNU compilers:

`-malign-double`	Align double precision variables on 64-bit boundaries, including in COMMON blocks

PGI compilers:

`-Mdalign`	Align doubles [i.e. REAL*8] in COMMON blocks and structures on 8-byte boundaries

Intel compilers:

`-Zp8`	Specify alignment constraint for structures on 8-byte boundaries, including in COMMON blocks; default
`-pad`	Enable changing of variable and array memory layout
`-nopad`	Disable changing of variable and array memory layout; default

TEST IF WHAT THEY CLAIM TO DO IS WHAT THEY ACTUALLY DO

Prefetching

- *Prefetching* is the retrieval of data from memory to cache before it is needed in an upcoming calculation. This is an example of general optimization technique called latency hiding in which communications and calculations are overlapped and occur simultaneously.
- The actual mechanism for prefetching varies from one machine to another.

Prefetching Compiler Options

GNU:
-fprefetch-loop-arrays
   If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.

PGI:
-Mprefetch[=option:n] -Mnoprefetch
    Add (don't add) prefetch instructions for those processors that support them (Pentium 4,Opteron); -Mprefetch is default on Opteron; -Mnoprefetch is default on other processors.

INTEL:
-O3   Enable -O2 optimizations and in addition, enable more aggressive optimizations such as loop and memory access transformation, and prefetching.

TEST IF WHAT THEY CLAIM TO DO IS WHAT THEY ACTUALLY DO

Demo: Prefetching ()

```
 program sum
  integer,parameter :: N=5000,ntimes=20000
  real :: x(N),y(N)
  real:: begin,end
  real:: rtime(2)
  common/saver/x,y
!DIR CACHE_ALIGN
  call random_number(x)
  call random_number(y)
  begin=dtime(rtime)
  do it=1,ntimes
    do i=1,N
      y(i)=x(i)+i
    end do
    if (mod(it,1000).eq.0) print *,x(1),y(1)
  end do
  end=dtime(rtime)
  print *,' my loop time (s) is ',end
  flop=(N*ntimes)/end*1.0e-6
  print *,' loop runs at ',flop,' MFLOP'
 end program sum
```

Loop Unrolling

- Loop unrolling is an optimization technique which can be applied to loops which perform calculations on array elements.
- Consists of replicating the body of the loop so that calculations are performed on several array elements during each iteration.
- Reason for unrolling is to take advantage of pipelined functional units. Consecutive elements of the arrays can be in the functional unit simultaneously.
- Programmer usually does not have to unroll loops “by hand” -- compiler options and directives are usually available. Performing unrolling via directives and/or options is preferable
  - Code is more portable to other systems
  - Code is self-documenting and easier to read
- Loops with small trip counts or data dependencies should not be unrolled!

Loop Unrolling (con't)

Advantages:

- Can be done automatically by compiler.
- Makes use of pipelined functional units.
- Makes use of multiple pipelined functional units on superscalar processors.
- Compiler may be able to pre-load array operands into registers, hiding load latencies.

Disadvantages:

- Limited number of FP registers
  - Pentium III: 8
  - Pentium 4: 8
  - Athlon: 8
  - Opteron: 4
  - Itanium: 128(!)

Loop Unrolling Example

- Normal loop

```
do i=1,N
   a(i)=b(i)+x*c(i)
enddo
```

- Manually unrolled loop

```
do i=1,N,4
   a(i)=b(i)+x*c(i)
   a(i+1)=b(i+1)+x*c(i+1)
   a(i+2)=b(i+2)+x*c(i+2)
   a(i+3)=b(i+3)+x*c(i+3)
enddo
```

Loop Unrolling Compiler Options

GNU compilers:

`-funroll-loops`	Enable loop unrolling
`-funroll-all-loops`	Unroll all loops; not recommended

PGI compilers:

`-Munroll`	Enable loop unrolling
`-Munroll=c:N`	Unroll loops with trip counts of at least **`N`**
`-Munroll=n:M`	Unroll loops up to **`M`** times

Intel compilers:

`-unroll`	Enable loop unrolling
`-unrollM`	Unroll loops up to **`M`** times

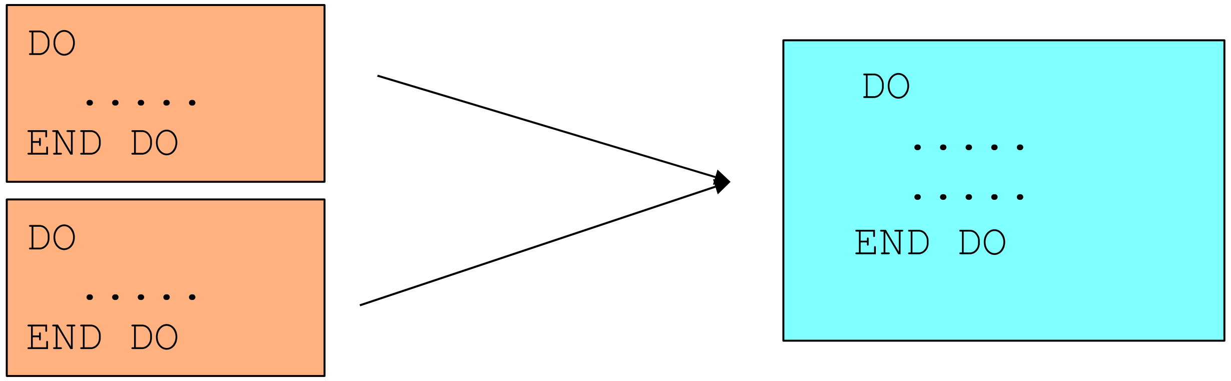
TEST IF WHAT THEY CLAIM TO DO IS WHAT THEY ACTUALLY DO

Loop unrolling directives...

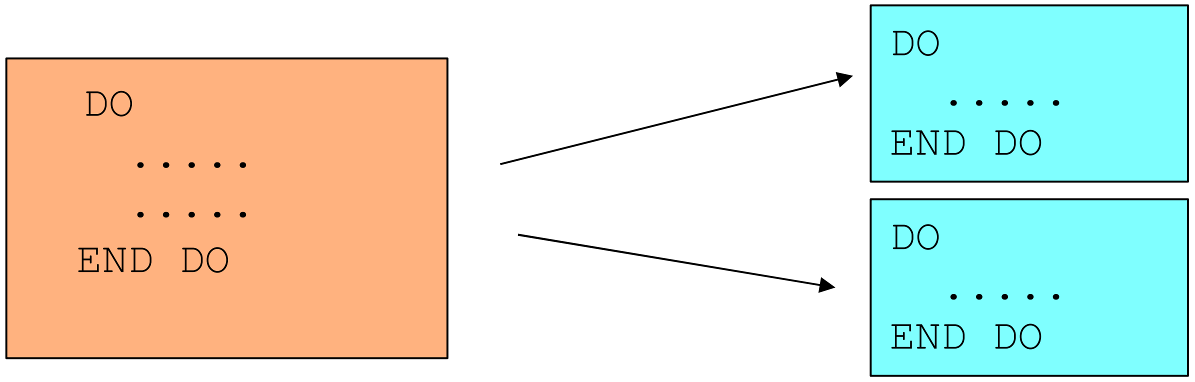
```
  program dirunroll
   integer,parameter :: N=1000000
   real,dimension(N):: a,b,c
   real:: begin,end
   real,dimension(2):: rtime
   common/saver/a,b,c
     call random_number(b)
     call random_number(c)
     x=2.5
     begin=dtime(rtime)
!DIR$ UNROLL 4
     do i=1,N
       a(i)=b(i)+x*c(i)
     end do
     end=dtime(rtime)
     print *,' my loop time (s) is ',(end)
     flop=(2.0*N)/(end)*1.0e-6
     print *,' loop runs at ',flop,' MFLOP'
     print *,a(1),b(1),c(1)
  end
```

Loop Fusion and Fission

Fusion: Merge multiple loops into one



Fission: Split one loop into multiple loops



Example of Loop Fusion

```
DO i=1,N
   B(i)=2*A(i)
END DO
```

```
DO k=1,N
   C(k)=B(k)+D(k)
END DO
```

```
DO ii=1,N
   B(ii)=2*A(ii)
   C(ii)=B(ii)+D(ii)
END DO
```

Potential for Fusion: dependent operations in separate loops

Advantage:

- Re-usage of array B()

Disadvantages:

- In total 4 arrays now contend for cache space
- More registers needed

Example of Loop Fission

```
DO ii=1,N
   B(i)=2*A(i)
   D(i)=D(i-1)+C(i)
END DO
```

```
DO ii=1,N
   B(ii)=2*A(ii)
END DO
```

```
DO ii=1,N
   D(ii)=D(ii-1)+C(ii)
END DO
```

Potential for Fission: independent operations in a single loop

Advantage:

- First loop can be scheduled more efficiently and be parallelised as well

Disadvantages:

- Less opportunity for out-of-order superscalar execution
- Additional loop created (a minor disadvantage)

Floating Point Division

- Floating point division is an expensive operation
  - Takes 22-60 CPs to complete (average about 32 CPs)
  - Usually not pipelined
  - According to the IEEE floating point standard, divisions **must** be carried out as such and not replaced with a multiplication by a reciprocal (even for division by a constant!)
- A possible optimization technique is to “relax” the IEEE requirements and replace a division with multiplication by a reciprocal. Most compilers do this automatically at higher levels of optimization.

Floating Point Division Compiler Options

GNU:
`-funsafe-math-optimizations`
Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards.

PGI:
`--Kieee -Knoieee (default)`
Perform floating-point operations in strict conformance with the IEEE 754 standard. Some optimizations are disabled with -Kieee, and a more accurate math library is used. The default -Knoieee uses faster but very slightly less accurate methods.

INTEL:
`--no-prec-div (i32 and i32em)`
Enables optimizations that give slightly less precise results than full IEEE division. With some optimizations, such as -xN and -xB, the compiler may change floating-point division computations into multiplication by the reciprocal of the denominator.

Floating Point Division Example

- SEE YESTERDAY'S LAB..

Floating Point Division With Arrays

- Consider the following loop nest in which the array `A(i,j)` is scaled by different factors stored in array `B(i)`:

```
do j=1,N
    do i=1,N
        A(i,j)=A(i,j)/B(i)
    enddo
enddo
```

- The compiler can do no automatic optimization to this, because `B(i)` is not a scalar. However, you can manually do the following:
  - Create a temporary array to hold the inverses of the **`B(i)`** array.
  - Replace the division in the inner loop with multiplication by the temporary array.
  - The resulting code can be unrolled and/or pipelined.

Optimization based on Microprocessor Architectures

- Pipelined Functional Units
- Superscalar Processors
  - Processors which have multiple functional units are said to be *superscalar*.
- Instruction Set Extensions
  - Newer processors have additional instructions beyond the usual floating point add and multiply instructions:
    - SSE2/SSE3/3DNow ! Etc...
    - Cat /proc/cpuinfo..
  –

Pipelined Functional Units

- For the processors in most modern parallel machines, the circuitry on the chip which performs a given type of operation on operands in registers is known as a *functional unit*.
- Most integer and floating point functional units are *pipelined*, meaning that they can have multiple independent executions of the same instruction placed in a queue. The idea is that after an initial startup latency, the functional unit should be able to generate one result every clock period (CP).
- Each stage of a pipelined operation can be working simultaneously on different sets of operands.

Superscalar Processors

- Processors which have multiple functional units are said to be *superscalar*.
- Examples:
  - Intel Pentium 3
    - 1 Floating point unit
    - 1 MMX/SSE unit
    - 2 Integer units
    - 2 Load/store units
  - Intel Pentium 4
    - 2 Floating point
    - 1 /MMX/SSE units
    - 2 Integer units
    - 2 Load/store units

Pipelining Compiler Options

```
GNU:
  none

PGI:
--Mvect[=option[,option,...]] -Mnovect (default)
              Pass options to the internal vectorizer.

INTEL:
    none
```

Instruction Set Extensions

Newer processors have additional instructions beyond the usual floating point add and multiply instructions:

- Intel MMX (Matrix Math eXtensions):
  - introduced in 1997 supported by all current processors
- Intel SSE (Streaming SIMD Extensions):
  - introduced on the Pentium III in 1999
  - useless for scientific computation: single precision
- Intel SSE2 (Streaming SIMD Extensions 2):
  - introduced on the Pentium 4 in Dec 2000

Instruction Set Extensions (2)

- AMD 3DNow! :
  - introduced in 1998 (extends MMX) "
- AMD 3DNow!+ (or 3DNow! Professional, or 3DNow! Athlon):
  - introduced with the Athlon (includes SSE)
- To check what you have on your machine:
  - cat /proc/cpuinfo

Instruction Set Extension Compiler Options

```
GNU:
  -mmmx/no-mmx           These switches enable or disable the use of
                    built-in functions that allow direct access to    -msse
               the MMX, SSE, SSE2, SSE3 and 3Dnow
  -mno-sse               extensions of the instruction set
  -msse2 /  -mno-sse2
  -msse3 /  -mno-sse3
  -m3dnow / -mno-3dnow

PGI:
  --fastsse
     Chooses generally optimal flags for a processor that supports
     SSE instructions (Pentium 3/4, AthlonXP/MP, Opteron) and SSE2
     (Pentium 4, Opteron).  Use pgf90 -fastsse -help to see the
     equivalent switches.

INTEL:
  -arch SSE   Optimizes  for Intel Pentium 4 processors with Streaming
         SIMD Extensions (SSE).
  -arch SSE2  Optimizes for Intel Pentium 4 processors with Streaming
          SIMD Extensions 2 (SSE2).
```

General techniques

- Blocking/ tiling
- !! Use of optimized libraries !!

Blocking for cache (tiling)

Blocking for cache is:

-An optimization that applies for datasets that do not entirely fit in the (2nd level) data cache

-A way to increase spatial locality of reference i.e. exploit full cache lines

-A way to increase temporal locality of reference i.e. Improves data re-usage

By way of example, let discuss the transpose of a matrix...

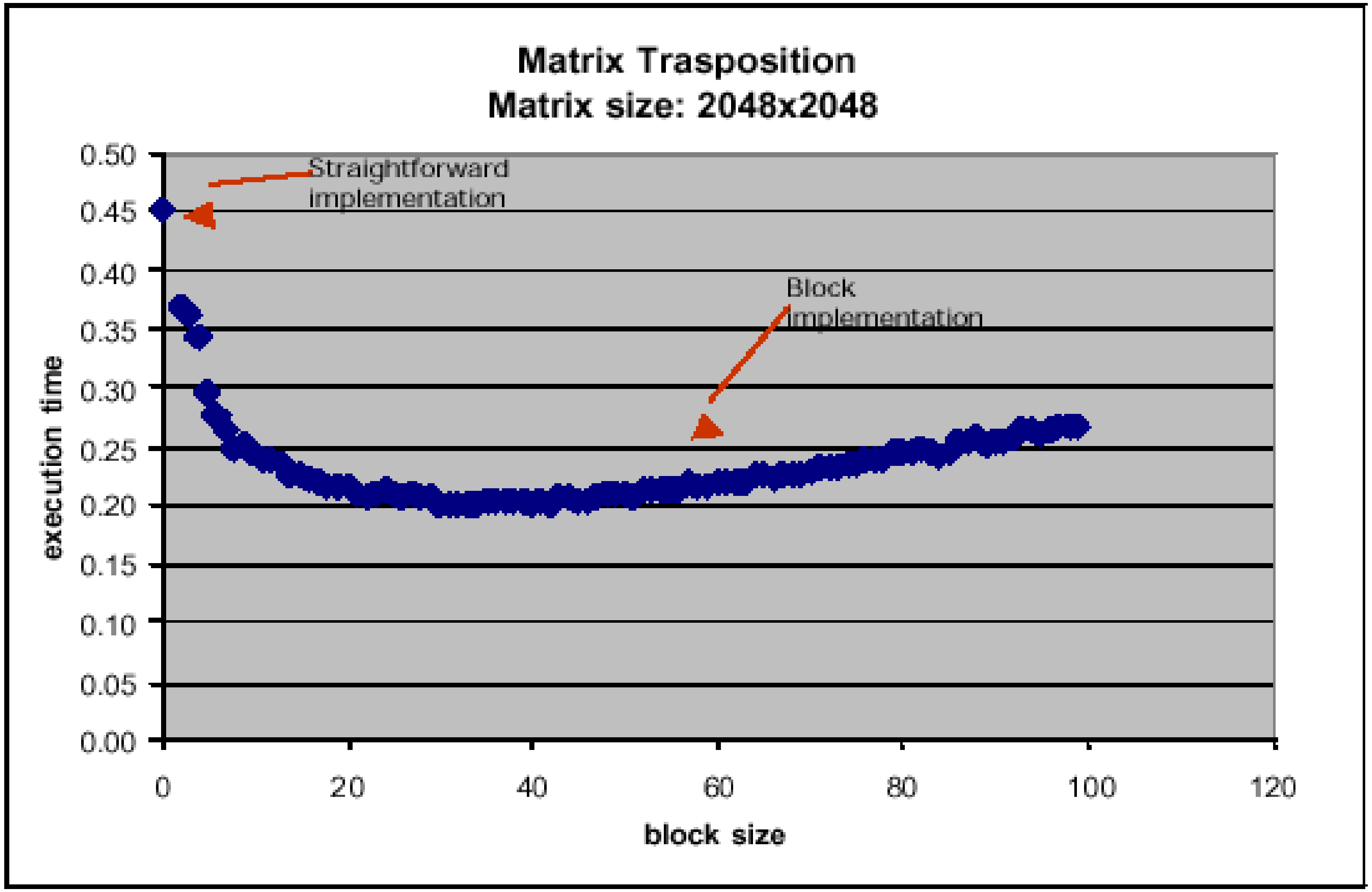
```
do i=1,n
  do j=1,n
     a(i,j)=b(j,i)
  end do
end do
```

Block algorithm for transposing a matrix:

- block data size= bsize
  - mb=n/bsize
  - nb=n/bsize
- Code is a little bit more complicated if
  - MOD(n,bize) is not zero
  - MOD(m,bize) is not zero

```
do ib = 1, nb
 ioff = (ib-1) * bsiz
 do jb = 1, mb
   joff = (jb-1) * bsiz
    do j = 1, bsiz
      do i = 1, bsiz
        buf(i,j) = x(i+ioff, j+joff)
     enddo
    enddo
    do j = 1, bsiz
      do i = 1, j-1
       bswp = buf(i,j)
       buf(i,j) = buf(j,i)
       buf(j,i) = bswp
      enddo
     enddo
    do i=1,bsiz
      do j=1,bsiz
       y(j+joff, i+ioff) = buf(j,i)
      enddo
    enddo
  enddo
enddo
```

Results... (Carlo Cavazzoni data)



Optimizing Matrix Multiply for Caches

- Several techniques for making this faster on modern processors
  - heavily studied
- Some optimizations done automatically by compiler, but can do much better
- In general, you should use optimized libraries (often supplied by vendor) for this and other very common linear algebra operations
  - BLAS = Basic Linear Algebra Subroutines
- Other algorithms you may want are not going to be supplied by vendor, so need to know these techniques

Summary

- Performance programming on uniprocessors requires
  - understanding of memory system
    - levels, costs, sizes
  - understanding of fine-grained parallelism in processor to produce good instruction mix
  - understanding your program
- Compilers are good at instruction level optimization and loop transformation
- User is responsible to present code in most natural way for compiler optimizations..
- The techniques work for any architecture, but choosing details depends on the architecture
- Blocking (tiling) is a basic approach that can be applied to many matrix algorithms