

Optimization of applications

- Execution time T of the program

$$T = T_e + T_m = \sum N_i t_i + \sum N_m t_m$$

T_e : time to execute instructions

T_m : time to move data (and instructions) between CPU and memory

N_i : number of instructions executed

t_i : (average) time to execute one instruction

N_m : number of memory operations

t_m : (average) time of one memory operation

- To get the execution time shorter the four factors N_i , t_i , N_m , and t_m should be cut down:

N_i : optimize the code (SW)

t_i : increase processor speed, pipelines, parallel execution of instructions, ... (HW)

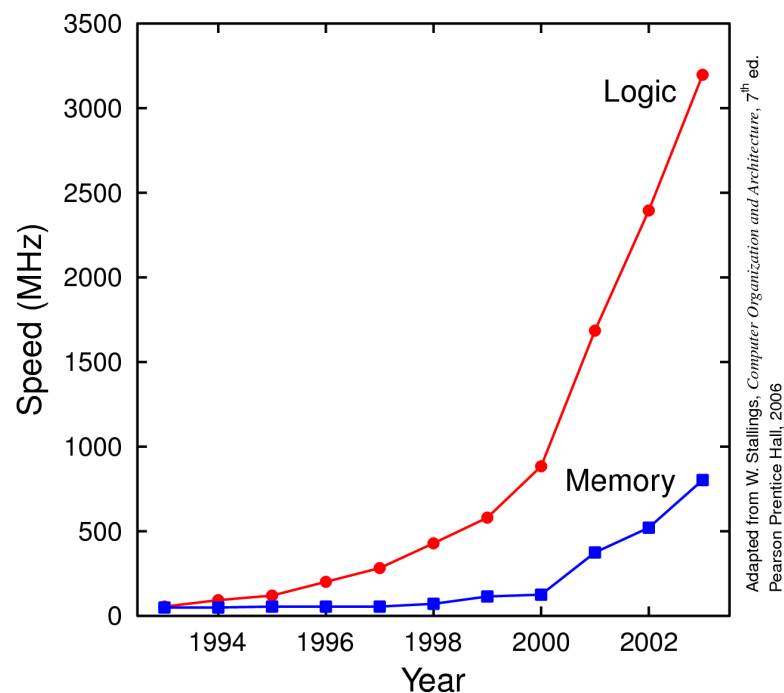
N_m : optimize the code (SW)

t_m : cache memories, prefetch, ... (HW)

Optimization of applications

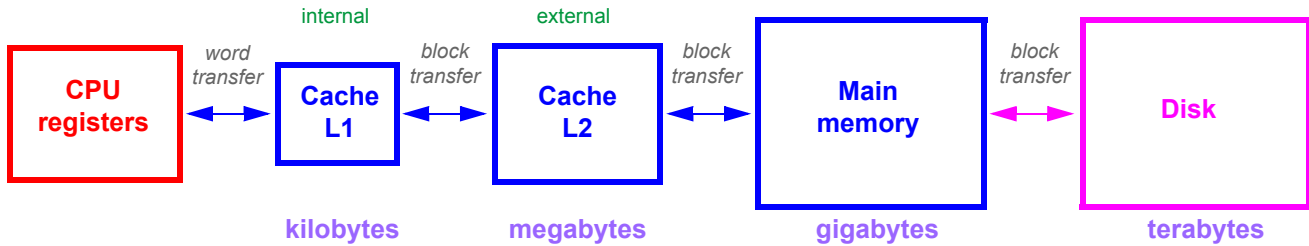
- **Cache memories**

- Processor speeds have increased much faster than the speed of main memory
→ memory access has become the bottleneck



Optimization of applications

- Fast cache memories are used to store data (and instructions) inside or near the processor¹.



- Relative speeds in the memory hierarchy²

Level	Access time	Typical size	Managed by
CPU registers	1-3 ns	1 kB	Compiler
L1 cache	2-8 ns	8-128 kB	Hardware
L2 cache	5-12 ns	0.1-1 MB	Hardware
Main memory	10-60 ns	1-8 GB	OS
Disk	3-10 ms	100-1000 GB	OS/user

1. In the virtual memory system the calculation of physical addresses is also cached by using the translation lookaside buffer (TLB).

2. Adapted from <http://arstechnica.com/articles/paedia/cpu/caching.ars/>

Optimization of applications

- *Why do these small caches help to increase the execution speed?*

- There is locality in the memory access patterns of programs:

- *Temporal locality:*

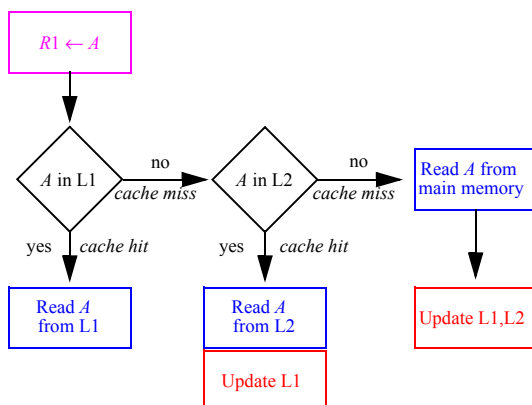
The most recently accessed memory locations are likely to be accessed again in the future.

- *Spatial locality:*

Memory near to those locations that have recently been accessed are likely to be accessed again in the future.

- Main memory access with caches.

- E.g. a load instruction: load a word from memory location *A* to CPU register *R1*.



Note that when a cache is updated a larger chunk of consecutive words is read in (a *cache line*). A typical cache line size is tens of bytes. Each line consists of the address of the memory block (tag), flags giving information on the usage of the line and the memory content.

Because cache memories are small, old entries must be deleted (*evicted*) by applying a suitable algorithm (eviction policy). In practice an approximation of a LRU (least recently used) algorithm is used.

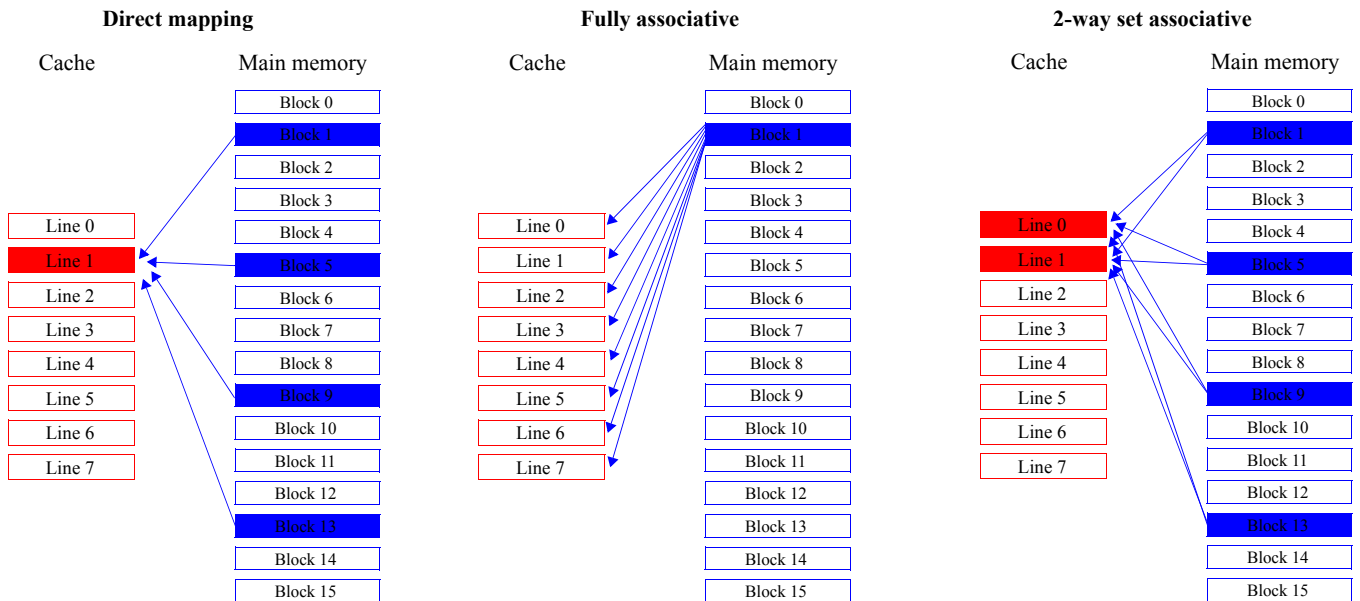
address	flags	data
address	flags	data
address	flags	data
address	flags	data
address	flags	data
address	flags	data
address	flags	data
address	flags	data
address	flags	data
address	flags	data

- The percentage of cache hits depends on the application.
- Commonly one can expect hit percentage of the order of 90%.

Optimization of applications

- Mapping of main memory blocks to cache lines can be done in three different ways (in all cases you have to store also the memory block address to the cache array):

- 1) Direct mapping: (cache line) = (address of the block) mod (number of lines)
- 2) Fully associative: A block can be stored in any cache line.
- 3) N-way set associative: Combination of 1 and 2:
a memory block can be mapped to any line within a bunch of cache lines.



Optimization of applications

• A simple example of cache effects

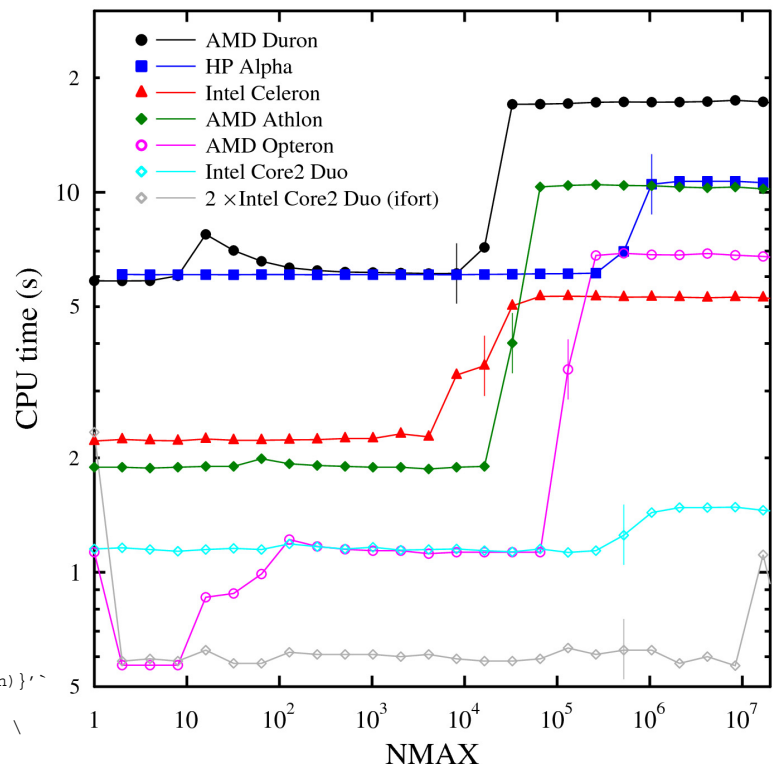
```

program cachetest
  use sizes
  implicit none
  integer :: n,m,mmax
  real(rk) :: t0,t1,z,x(NMAX)

  forall (n=1:NMAX) x(n)=n
  mmax=TOTAL/NMAX
  call cpu_time(t0)
  do m=1,mmax
    do n=1,NMAX
      z=z+x(n)
    end do
  end do
  call cpu_time(t1)
  write(6,'(i12,2g16.8,i10,g16.6)') &
    & NMAX, (t1-t0), (t1-t0)/ &
    & (mmax*NMAX), mmax, z
end program cachetest
  
```

System	L2 cache size ^a
AMD Duron	64 kB
HP Alpha	8 MB
Intel Celeron	128 kB
AMD Athlon	256 kB
AMD Opteron	1 MB
Intel Core2 Duo	4 MB

a. Marked with vertical lines in the plot



Run script (runcachetest)

```

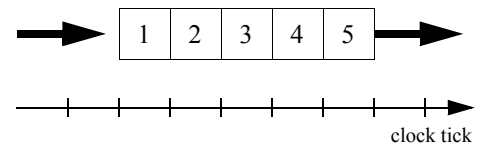
#!/bin/bash
TOTAL=1000000000
F90=ifort
FOPT="-O3"
$F90 $FOPT -c sizes.f90
for NMAX in `gawk 'BEGIN {for (n=1;n<=2^25;n*=2) printf "%d ",int(n)}'`
do
  cat cachetest.f90 | sed "s/NMAX/${NMAX}/g;s/TOTAL/${TOTAL}/g" \
    > cachetest_tmp_$$.$F90
  $F90 $FOPT cachetest_tmp_$$.$F90 #sizes.o
  nice ./a.out
done
rm -f cachetest_tmp_$$.$F90
  
```

Optimization of applications

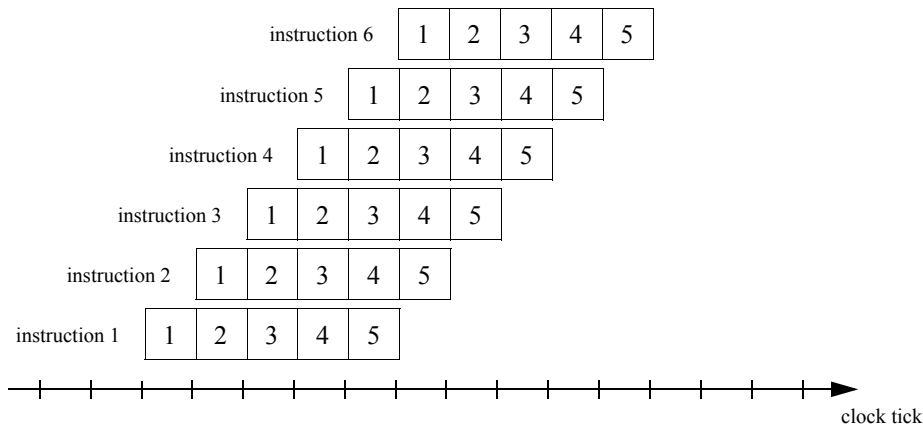
• Pipelines

- Many CPU clock cycles are needed to execute a single machine instruction.

- Execution consists of many *stages*; typically the following
 - 1) *Instruction fetch* read instruction from memory
 - 2) *Instruction decode* recognize and decode the instruction
 - 3) *Operand fetch* operands are read from memory or registers
 - 4) *Execute*
 - 5) *Writeback* write results back to memory or registers



- The idea of speeding up the execution of code is to have many instructions proceeding simultaneously in the pipeline.



Optimization of applications

- In this way we get one instruction per CPU clock cycle when the pipeline is full.
- There is a *latency* in filling the pipeline.
- Many issues, however, make the optimal use of pipelines difficult.
 - 1) Instructions use different amount of cycles. → Other parts of the pipeline must wait.
 - 2) Instructions upstream the pipeline may need the results of instructions downstream.
 - 3) Branches (jumps) in code make the rest of the pipeline useless. This is particularly true for conditional branches.
- The effect of issues 1 and 2 can be reduced by the *out of order execution* of instructions.
 - Instructions are reordered in such a way that the pipeline is optimally filled and at the same time data integrity is maintained (i.e. results are right!).
 - A simple example¹ (in a fictive assembler):

Pipeline stalled because the contents of r0 is not available **here**.

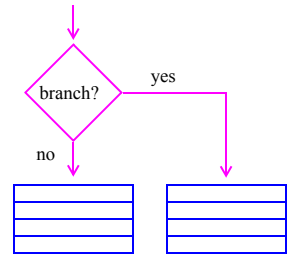
Original code		Executed code
load r0,a	; load register r0 from memory location a	load r0,a
add r2,r0,r3	; r2 ← r0+r3	load r1,b
load r1,b	; load register r1 from memory location b	add r2,r0,r3
sub r2,r1,r2	; r2 ← r1-r2	sub r2,r1,r2

- The task of reordering the instructions is done by both the hardware and the compiler.
- This is based on the *data flow analysis* of the code: Deduce which instructions are dependent on each other's results or data.

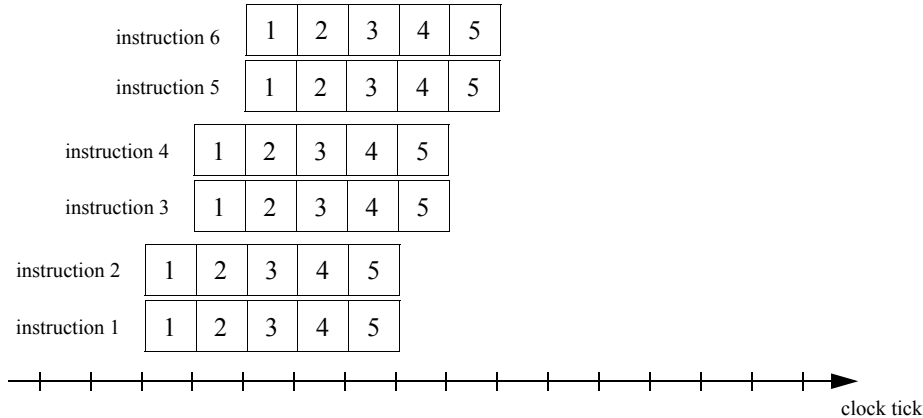
1. From K. Dowd, High Performance Computing, O'Reilly & Associates, 1993.

Optimization of applications

- There are many ways to attempt to decrease the performance penalty of branches.
 - By *branch prediction* the processor makes an educated guess which path is taken in the case of a conditional branch and begins to fetch instructions from that address.
 - Instructions can even be executed (*speculative execution*). However, if the branch doesn't go as the processor expected, it must undo or discard the results of these instructions.
 - The guess can be a static one (always either yes or no) or be based on the instruction opcode or on some statistics from previous executions of the current instruction.
 - If the guess was wrong the pipeline must be flushed and filled again from the right source.



- The processor speed can be further increased by adding more pipelines that can execute instructions in parallel (instruction-level parallelism):



- This is called *superscalar* architecture.
- Most modern processors (including the IA-32, Intel 64 and IA-64) are superscalar.

Optimization of applications

- In order to be able to execute instructions concurrently there must not be any dependences between them.
 - This must be handled by the processor control circuitry which (you can imagine) becomes quite complex.

• Examples:

- Intel Pentium architecture¹
 - Three-way superscalar (3 instructions/clock cycle)
 - 12-stage superpipelines supporting out-of-order execution
 - Level 1 cache: 8 kB instruction, 8 kB data
 - Level 2 cache: varies
 - Deep branch prediction, dynamic data flow analysis, speculative execution
- AMD64
 - Three execution engines for integer and floating point operations
 - 12-stage pipeline for integers 17-stage pipeline for floating point
 - Level 1 cache: 64 kB instruction, 64 kB data
 - Level 2 cache: 512, 1024 kB

1. From *Intel Architecture Software Developer's Manual*

Optimization of applications

• Tuning (or optimization) of applications

- A good application fulfills the requirements of locality of memory access and parallelism in the code.
 - Modern compilers can do a lot to improve the execution speed of an application.
 - However, programmer can also affect the speed of his code and the ability of the compiler to optimize it.

- First we see what basic optimizations compilers normally do.
Optimizations can be switched on and off by the compiler option.

- They are in many cases of the form `-On`, where `O` is capital oh not zero, and `n=0,...,3` or `5`.
 - These options normally switch on groups of optimization.
 - There may be individual options for each optimization technique.

- In the case of `ifort` the general options do the following:

`-O0` Disables all `-O<n>` optimizations. On IA-32 and Intel(R) EM64T systems, this option sets the `-fp` option.

`-O1` On IA-32 and Intel(R) EM64T systems, enables optimizations for speed. Also disables intrinsic recognition and the `-fp` option. This option is the same as the `-O2` option.

`-O2` or `-O`
This option is the default for optimizations. However, if `-g` is specified, the default is `-O0`.
On IA-32 and Intel(R) EM64T systems, this option is the same as the `-O1` option.

`-O3` Enables `-O2` optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop transformations. Enables optimizations for maximum speed, but does not guarantee higher performance unless loop and memory access transformations take place.
On IA-32 and Intel(R) EM64T systems, when the `-O3` option is used with the `-ax` and `-x` options, it causes the compiler to perform more aggressive data dependency analysis than for `-O2`, which may result in longer compilation times.
On Itanium-based systems, the `-O3` option enables optimizations for technical computing applications (loop-intensive code): loop optimizations and data prefetch.

Optimization of applications

- If you are curious enough you can output the assembler code generated by the `ifort` compiler as shown below.

doloop.f90

```
1: program doloop
2: integer :: i,j
3: j=0
4: do i=1,1234
5:   j=j+i
6: end do
7: print *,j
8: end program doloop
```

```
progs> ifort -fsource-asm -S doloop.f90
progs> less doloop.s
...
    .globl MAIN__
MAIN__:
..B1.1:                                # Preds ..B1.0
;;; program doloop
    pushl    %ebp                                #1.8
    movl     %esp, %ebp                        #1.8
    andl     $-16, %esp                        #1.8
    subl     $48, %esp                         #1.8
    call     __intel_proc_init                 #1.8
    push     $LITPACK_0                        #1.8
...
;;; integer :: i,j
;;; j=0
    xorl     %edx, %edx                        #3.2
;;; do i=1,1234
    movl     $1, %eax                          #4.2
..B1.3:                                # LOE eax edx ebx esi edi
                                # Preds ..B1.3 ..B1.2
;;;   j=j+i
    addl     %eax, %edx                        #5.5
;;; end do
    lea      1(%edx,%eax), %edx                #6.2
    lea      2(%edx,%eax), %ecx                #6.2
    lea      3(%ecx,%eax), %edx                #6.2
    lea      4(%edx,%eax), %edx                #6.2
    lea      5(%edx,%eax), %edx                #6.2
    addl     $6, %eax                          #6.2
    cmpl     $1228, %eax                       #6.2
    jle      ..B1.3                            # Prob 99%
                                # LOE eax edx ebx esi edi
..B1.4:                                # Preds ..B1.3
;;; print *,j
    movl     $0, (%esp)                        #7.2
...
```

Optimization of applications

- **gfortran** has similar optimization options¹ though the defaults and details are different:

-O

-O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

-O turns on the following optimization flags: -fdefer-pop -fdelayed-branch
-fguess-branch-probability -fcprop-registers -floopt-optimize -fif-conversion -fif-conversion2
-ftree-ccp -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-ter -ftree-lrs -ftree-sra
-ftree-copyrename -ftree-fre -ftree-ch -funit-at-a-time -fmerge-constants

-O also turns on -fomit-frame-pointer on machines where doing so does not interfere with debugging.

-O2 Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.

-O2 turns on all optimization flags specified by -O. It also turns on the following optimization flags: -fthread-jumps -fcrossjumping -foptimize-sibling-calls -fcse-follow-jumps
-fcse-skip-blocks -fgcse -fgcse-lm -fexpensive-optimizations -fstrength-reduce
-frerun-cse-after-loop -fre-run-loop-opt -fcaller-saves -fpeephole2 -fschedule-insns
-fschedule-insns2 -fsched-interblock -fsched-spec -fregmove -fstrict-aliasing
-fdelete-null-pointer-checks -freorder-blocks -fre-order-functions -falign-functions
-falign-jumps -falign-loops -falign-labels -ftree-vrp -ftree-pre

-O3 Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops and -fgcse-after-reload options.

-O0 Do not optimize. This is the default.

Note: Most other compilers have some optimization as default.

1. These can be found in the **gcc** manual because **gfortran** is just a front-end of the GNU compiler.

Optimization of applications

- **gfortran/gcc** has a huge list of detailed optimization options¹

-falign-functions=n -falign-jumps=n -falign-labels=n -falign-loops=n -fbounds-check -fmudflap
-fmudflapth -fmudflapir -fbranch-probabilities -fprofile-values -fvpt -fbranch-target-load-optimize
-fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves -fcprop-registers
-fcse-follow-jumps -fcse-skip-blocks -fcx-limited-range -fdata-sections -fdelayed-branch
-fdelete-null-pointer-checks -fearly-inlining -fexpensive-optimizations -ffast-math -ffloat-store
-fforce-addr -ffunction-sections -fgcse -fgcse-lm -fgcse-sm -fgcse-las -fgcse-after-reload
-fcrossjumping -fif-conversion -fif-conversion2 -finline-functions -finline-functions-called-once
-finline-limit=n -fkeep-inline-functions -fkeep-static-consts -fmerge-constants
-fmerge-all-constants -fmodulo-sched -fno-branch-count-reg -fno-default-inline -fno-defer-pop
-fmove-loop-invariants -fno-function-cse -fno-guess-branch-probability -fno-inline -fno-math-errno
-fno-peephole -fno-peephole2 -funsafe-math-optimizations -funsafe-loop-optimizations
-ffinite-math-only -fno-toplevel-reorder -fno-trapping-math -fno-zero-initialized-in-bss
-fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls -fprefetch-loop-arrays
-fprofile-generate -fprofile-use -fregmove -frename-registers -freorder-blocks
-freorder-blocks-and-partition -freorder-functions -frerun-cse-after-loop -frounding-math
-frtl-abstract-sequences -fschedule-insns -fschedule-insns2 -fno-sched-interblock -fno-sched-spec
-fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns=n -fsched-stalled-insns-dep=n
-fsched2-use-superblocks -fsched2-use-traces -fsee -freschedule-modulo-scheduled-loops
-fsection-anchors -fsignaling-nans -fsingle-precision-constant -fstack-protector
-fstack-protector-all -fstrict-aliasing -fstrict-overflow -ftracer -fthread-jumps -funroll-all-loops
-funroll-loops -fpeel-loops -fsplit-ivs-in-unroller -funswitch-loops
-fvariable-expansion-in-unroller -ftree-pre -ftree-ccp -ftree-dce -ftree-loop-optimize
-ftree-loop-linear -ftree-loop-lm -ftree-loop-ivcanon -fivopts -ftree-dominator-opts -ftree-dse
-ftree-copyrename -ftree-sink -ftree-ch -ftree-sra -ftree-ter -ftree-lrs -ftree-fre
-ftree-vectorize -ftree-vec-loop-version -ftree-salias -fipa-pta -fweb -ftree-copy-prop
-ftree-store-ccp -ftree-store-copy-prop -fwhole-program -param name=value -O -O0 -O1 -O2 -O3 -Os

- In most cases using the bundled options **-On** is enough.

1. See e.g. <http://gcc.gnu.org/onlinedocs/>

Optimization of applications

- Now we go through the most common and simple optimization techniques performed by compilers¹.

- Common expression elimination

- Different variables that get identical expressions assigned to them are replaced by a single variable.

```
t1 = ((j-1)25+(i-1))*4
t2 = ((j-1)25+(i-1))*4
a(t1)=b(t2)
```

```
t = ((j-1)25+(i-1))*4
a(t)=b(t)
```

- Strength reduction

- Replacing an expensive operation with a cheaper one:

```
real :: x,y
integer :: j,k

y = x**2
j = k*2
```

```
real :: x,y
integer :: j,k

y = x*x
j = k+k
```

- Constant folding

- As much as possible is computed at compile time

```
program main
integer,parameter::i=10
integer :: k
k=200
j=i+k
print *,j
end program main
```

```
program main
integer,parameter::j=210
print *,j
end program main
```

1. Partly adopted from K. Dowd, High Performance Computing, O'Reilly & Associates, 1993.

Optimization of applications

- Dead code removal

- Often the program contains code that can never be reached and can be safely removed.

- Also statements that produce results that are never used can be removed.

```
program main
integer :: k
k=2
print *,k
stop
k=4
print *,j
end program main
```

```
program main
integer :: k
k=2
print *,k
end program main
```

- Variable renaming

- If recycling of a variable is observed the different (independent) uses of the variable are modified so that the parallelism is more obvious.

```
x=y*z
q=r+x+x
x=a+b
```

```
x0=y*z
q=r+x0+x0
x=a+b
```

- Copy propagation

- Eliminate useless assignments that may cause dependencies.

```
x=y
z=1.0+x
```

```
x=y
z=1.0+y
```


Optimization of applications

- Loop invariant code motion

- Remove code that does not depend on the loop iteration outside the loop.



- Induction variable simplification

- Loops may contain induction variables. Their value is a linear function of the loop iteration count.



- This optimization is actually applied in calculating array element addresses.

- Memory address a_i of array element $A(i)$ is computed as

$$a_i = \text{base}(A) + (i - 1) \times \text{size}(A)$$

- However, in a loop all this need not be computed:

```
a_i = base(A) - size(A)
do i=1,n
  a_i = a_i + size(A)
  ...
end do
```

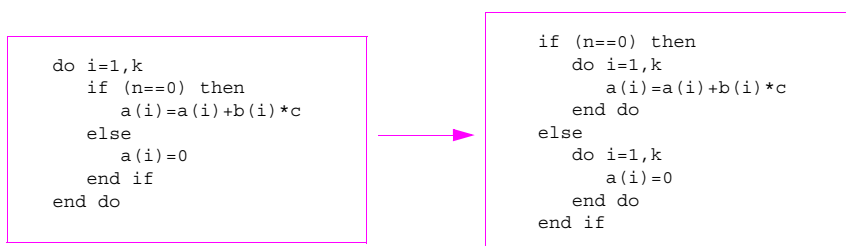
- Register variable detection

- Based on the data flow analysis the compiler decides which variables should be kept in the CPU registers.

Optimization of applications

- *Not all optimization is left to the compiler. In these cases it is the programmer that has to do the tuning.*

- Later we will deal with optimization of loops and memory access.
- Now we discuss a few basic methods to speed up the program executions
- One goal, of course, is to remove code that consumes CPU time but does not contribute to results.
- Keeping in mind the pipelines and the parallel execution capabilities of modern processors we also have to remove or rewrite those parts of the code that would prevent the utilization of parallel execution.
- **Subroutine or function calls** always include some overhead
 - Pushing parameters to stack, jumping to the routine, popping the stack and returning.
 - Calls may prevent the instruction-level parallelization.
 - Solution: inline functions by hand (or let the compiler do it).
- **Branches within loops**
 - Most CPU time in scientific applications is spent in loops. So, try to move all unnecessary code outside loops.
 - Branches add dependencies that prevent parallelization.
- **Loop invariant conditionals**
 - The conditional does not depend on the loop iteration → Change the order of the conditional and the loop.



Optimization of applications

- Loop index dependent conditionals

- The conditional is true for certain ranges of the loop index variables and false for others but there is a pattern that we can utilize.

```
do i=1,n
  do j=1,n
    if (j<1) then
      a(j,i)=a(j,i)+b(i,j)*c
    else
      a(j,i)=0
    end if
  end do
end do
```

```
do i=1,n
  do j=1,i-1
    a(j,i)=a(j,i)+b(j,i)*c
  end do
  do j=i,n
    a(j,i)=0
  end do
end do
```

- Could we use F90 array constructs in this case?

- Independent loop conditionals

- Here the conditionals of different iterations are independent of each other.
- In the example below not much can be done. However, by *loop unrolling* some parallelism could be achieved.

```
do i=1,n
  do j=1,n
    if (b(j,i)>1.0) a(j,i)=a(j,i)+b(j,i)*c
  enddo
enddo
```

```
do i=1,n
  do j=1,n,2
    if (b(j,i)>1.0) a(j,i)=a(j,i)+b(j,i)*c
    if (b(j+1,i)>1.0) a(j+1,i)=a(j+1,i)+b(j+1,i)*c
  enddo
enddo
```

Optimization of applications

- Dependent loop conditionals

- Loops with *if*-statements can have dependencies between iterations.
- Not much can be done to these; for example:

```
do i=1,n
  if (x<a(i)) x=x+b(i)
enddo
```

By unrolling this we see that the next iteration can not be started before the previous one has finished.

```
if (x<a(1)) x=x+b(1)
if (x<a(2)) x=x+b(2)
if (x<a(3)) x=x+b(3)
if (x<a(4)) x=x+b(4)
if (x<a(5)) x=x+b(5)
if (x<a(6)) x=x+b(6)
...
```

- Reduction operations

- Vector and matrix reduction operations are a special case of loop constructs.
- If possible, use the F90/F95 array constructs or intrinsic functions (*where*, *forall*, *matmul*, *dot_product*, *sum*, *maxloc*, *maxval*, *any*, *all*, *count*,...)
- It is possible to introduce some parallelism into these operations. For example computing the maximum value of elements of an array:

```
x=-huge(x)
do n=1,nmax
  if (a(n)>=x) x=a(n)
enddo
```

```
x0=-huge(x) ; x1=-huge(x)
do n=1,nmax,2
  if (a(n)>=x0) x0=a(n)
  if (a(n+1)>=x1) x1=a(n+1)
enddo
if (x0>=x1) then
  x=x0
else
  x=x1
endif
```

Optimization of applications

- Example

```
program imaximum
  use sizes
  implicit none
  real(rk) :: t1,t2
  integer :: a(NMAX),i,x
  forall (i=1:NMAX)
    a(i)=mod(i,M)+mod(i+1,K)
  end forall
  x=-huge(x)
  call cpu_time(t1)
  do i=1,NMAX
    if (a(i)>=x) x=a(i)
  end do
  call cpu_time(t2)
  print *, 'imaximum ',t2-t1,x
end program imaximum
```

```
program imaximum2
  use sizes
  implicit none
  real(rk) :: t1,t2
  integer :: a(NMAX),i,x,x0,x1
  forall (i=1:NMAX)
    a(i)=mod(i,M)+mod(i+1,K)
  end forall
  x0=-huge(x)
  x1=-huge(x)
  call cpu_time(t1)
  do i=1,NMAX,2
    if (a(i) >=x0) x0=a(i)
    if (a(i+1) >=x1) x1=a(i+1)
  end do
  x=max(x0,x1)
  call cpu_time(t2)
  print *, 'imaximum2 ',t2-t1,x
end program imaximum2
```

```
program imaximum4
  use sizes
  implicit none
  real(rk) :: t1,t2
  integer :: a(NMAX),i,x,x0,x1,x2,x3
  forall (i=1:NMAX)
    a(i)=mod(i,M)+mod(i+1,K)
  end forall
  x0=-huge(x)
  x1=-huge(x)
  x2=-huge(x)
  x3=-huge(x)
  call cpu_time(t1)
  do i=1,NMAX,4
    if (a(i) >=x0) x0=a(i)
    if (a(i+1) >=x1) x1=a(i+1)
    if (a(i+2) >=x2) x2=a(i+2)
    if (a(i+3) >=x3) x3=a(i+3)
  end do
  x=max(x0,x1,x2,x3)
  call cpu_time(t2)
  print *, 'imaximum4 ',t2-t1,x
end program imaximum4
```

```
progs> make -f Makefile.imaximum
rm -f imaximum imaximum2 imaximum4
ifort -O0 -fpp -DNMAX=50000000 -DM=20 -DK=2001 -o imaximum imaximum.f90
ifort -O0 -fpp -DNMAX=50000000 -DM=20 -DK=2001 -o imaximum2 imaximum2.f90
ifort -O0 -fpp -DNMAX=50000000 -DM=20 -DK=2001 -o imaximum4 imaximum4.f90
./imaximum ; ./imaximum2 ; ./imaximum4
imaximum      0.163975000000000      2019
imaximum2     0.142978000000000      2019
imaximum4     0.138979000000000      2019
progs> make -f Makefile.imaximum clean
rm -f imaximum imaximum2 imaximum4
progs> make -f Makefile.imaximum
rm -f imaximum imaximum2 imaximum4
ifort -unroll14 -fpp -DNMAX=50000000 -DM=20 -DK=2001 -o imaximum imaximum.f90
ifort -unroll14 -fpp -DNMAX=50000000 -DM=20 -DK=2001 -o imaximum2 imaximum2.f90
ifort -unroll14 -fpp -DNMAX=50000000 -DM=20 -DK=2001 -o imaximum4 imaximum4.f90
./imaximum ; ./imaximum2 ; ./imaximum4
imaximum      0.116983000000000      2019
imaximum2     0.115982000000000      2019
imaximum4     0.119982000000000      2019
```

Note the use of **ifort** options
- **fpp** (preprocessor) and **-D** (define a symbol).

Optimization of applications

- The previous was an example of **loop unrolling**.

- Loop stride is larger than one and part of the loop is written out explicitly. E.g. unrolling a loop to a *depth* of 4:

```
do n=1,NMAX
  a(n)=a(n)+b(n)*c
end do
```

```
m=mod(NMAX,4)
do n=1,m
  a(n)=a(n)+b(n)*c
end do

do n=m+1,NMAX,4
  a(n)=a(n)+b(n)*c
  a(n+1)=a(n+1)+b(n+1)*c
  a(n+2)=a(n+2)+b(n+2)*c
  a(n+3)=a(n+3)+b(n+3)*c
end do
```

- By unrolling loops one can save a few cycles by decreasing the effect of loop overhead.
- More important is the fact that unrolling generates a block of code within the loop which give the compiler a chance to better optimize it: pipelining and instruction level parallelization.
- Unrolling works best for long (i.e. many iterations) loops with simple content.
 - No subroutine calls: Call overhead, prevents optimization. (However: inlining, see below.)
 - No branches: Disrupt instruction pipelining.
 - Not for *fat loops* (loops with a lot of stuff between **do i=...** and **end do**): They already have a big block of code for the compiler to optimize.
- In many cases the compiler does the unrolling.

Optimization of applications

- In the case of many loops within each other (*loop nests*) one can also try to unroll not the innermost but one or more outer loops.

```
program unroll_outer
  use sizes
  implicit none
  integer,parameter :: NMAX=3000,MMAX=3000,K=10
  real(rk) :: a(NMAX,MMAX),b(NMAX,MMAX),c=11.0
  integer :: n,m,nn
  real(rk) :: t1,t2

  !-----

  forall (n=1:NMAX,m=1:MMAX)
    a(n,m)=mod(n+1,K)+mod(m+1,K)
    b(n,m)=mod(n,K+10)+mod(m,K-5)
  end forall

  call cpu_time(t1)
  nn=mod(NMAX,4)
  do n=1,nn
    do m=1,MMAX
      a(n,m)=a(n,m)+b(n,m)*c
    end do
  end do
  do n=nn+1,NMAX,4
    do m=1,MMAX
      a(n,m)=a(n,m)+b(n,m)*c
      a(n+1,m)=a(n+1,m)+b(n+1,m)*c
      a(n+2,m)=a(n+2,m)+b(n+2,m)*c
      a(n+3,m)=a(n+3,m)+b(n+3,m)*c
    end do
  end do
  call cpu_time(t2)
  print *,sum(a),t2-t1
```

```
!-----

forall (n=1:NMAX,m=1:MMAX)
  a(n,m)=mod(n+1,K)+mod(m+1,K)
  b(n,m)=mod(n,K+10)+mod(m,K-5)
end forall

call cpu_time(t1)
do n=1,NMAX
  do m=1,MMAX
    a(n,m)=a(n,m)+b(n,m)*c
  end do
end do
call cpu_time(t2)
print *,sum(a),t2-t1

!-----

end program unroll_outer
```

```
progs> gfortran unroll_outer.f90
progs> a.out
1219500000.000000      0.1929700000000000
1219500000.000000      0.2989550000000000
```

- Loop nest optimization will also be dealt with when talking about optimization of memory access.

Optimization of applications

- Note that sometimes optimization of arithmetics may produce different results compared with the unoptimized code.
 - Because of the finite precision of floating point numbers arithmetics operations are not always associative:
 $(x+y)+z$ may be different from $x+(y+z)$
 - Many compilers can do optimizations that can produce different results. Usually there are warnings in the compiler documentation for the use of these options.
- As an example of a loop unrolling in a reduction operator where the effect of a finite precision might be seen:

```
program unroll_dotprod
  use sizes
  implicit none
  integer,parameter :: NMAX=10000000,K=10,rr=rk
  real(rr) :: a(NMAX),b(NMAX),c=11.0
  integer :: n,m
  real(rr) :: t1,t2,s,s0,s1,s2,s3

  forall (n=1:NMAX)
    a(n)=mod(n+1,K)/(10000.0*real(n,rr))
    b(n)=mod(n,K+10)
  end forall

  call cpu_time(t1)
  s=dot_product(a,b)
  call cpu_time(t2)
  print *,s,t2-t1
```

```
call cpu_time(t1)
s0=0.0; s1=0.0; s2=0.0; s3=0.0
do n=1,NMAX,4
  s0=s0+a(n)*b(n)
  s1=s1+a(n+1)*b(n+1)
  s2=s2+a(n+2)*b(n+2)
  s3=s3+a(n+3)*b(n+3)
end do
s=s0+s1+s2+s3
call cpu_time(t2)
print *,s,t2-t1

s=0.0
call cpu_time(t1)
do n=1,NMAX
  s=s+a(n)*b(n)
end do
call cpu_time(t2)
print *,s,t2-t1
end program unroll_dotprod
```

Intel Fortran

GNU Fortran

```
progs> ifort unroll_dotprod.f90; a.out
6.918235090048858E-002  0.1009840000000000
6.918235090048259E-002  0.1069840000000000
6.918235090048858E-002  0.1039840000000000
progs> ifort -O0 unroll_dotprod.f90; a.out
6.918235090048858E-002  0.1379790000000000
6.918235090048259E-002  0.1049840000000000
6.918235090048858E-002  0.1389790000000000
```

```
progs> gfortran -O0 unroll_dotprod.f90 ; a.out
6.3805178E-02  9.3984991E-02
6.8390638E-02  3.9994001E-02
6.3805178E-02  9.2985988E-02
progs> gfortran -O2 unroll_dotprod.f90 ; a.out
6.9182351E-02  2.4996012E-02
6.9182351E-02  2.2996008E-02
6.9182351E-02  2.4995983E-02
```

Optimization of applications

- Procedure inlining

- Write the procedure statements instead of the subroutine or function call.
- Can be used in loop unrolling when the procedure is short. Example:

```
do n=1,nmax
  a(i)=func(b(i),c)
end do
```

```
real function func(b,c)
  real :: b,c
  func=sin(b)*c
  return
end function func
```

```
do n=1,nmax
  a(i)=sin(b(i))*c
end do
```

- Most compilers can do inlining if asked to.

- Common expression elimination

- There are cases where the compiler can not do this because of procedure calls:

```
x=a*myfunc(b)+c
y=myfunc(b)**2+b
z=myfunc(b)+e
```

```
tmp=myfunc(b)
x=a*tmp+c
y=tmp**2+b
z=tmp+e
```

- Why can't compiler do this optimization?

- If it does not know whether **myfunc** has any side-effects or not it wants to be on the safe side and does the optimization assuming that there are side-effects.
- Use the interprocedural optimization feature some compilers have or write the procedure to the same file as the calling program if you want the compiler to inline your own function.
- Note also the Fortran 95 specifier **pure**. With it you can tell the compiler that your procedure has no side-effects.

Optimization of applications

- Loop invariant code motion

- Sometimes the compiler can't do this either.

```
program codemotion
  use sizes
  implicit none
  integer :: i,j
  integer,parameter :: NMAX=10000000
  real(rk) :: a(NMAX),b(NMAX),x,y,t1,t2
  real(rk),external :: func

  read(5,*) x,y

  call cpu_time(t1)
  do i=1,NMAX
    a(i)=func(i)
    b(i)=a(i)/sqrt(x**2+y**2)
  end do
  call cpu_time(t2)

  print *,t2-t1,sum(a),sum(b)
end program codemotion
```

```
program codemotion1
  use sizes
  implicit none
  integer :: i,j
  integer,parameter :: NMAX=10000000
  real(rk) :: a(NMAX),b(NMAX),x,y,z,t1,t2
  real(rk),external :: func

  read(5,*) x,y
  z=1.0/sqrt(x**2+y**2)

  call cpu_time(t1)
  do i=1,NMAX
    a(i)=func(i)
    b(i)=a(i)*z
  end do
  call cpu_time(t2)

  print *,t2-t1,sum(a),sum(b)
end program codemotion1
```

```
progs> ifort -o codemotion codemotion.f90 inline_func.o
progs> ifort -o codemotion1 codemotion1.f90 inline_func.o
progs> ./codemotion
1,2
 1.823723000000000    2076445.65499713    928614.727234675
progs> ./codemotion1
1,2
 1.661748000000000    2076445.65499713    928614.727234675
```

By investigating the assembler listing one can assure that the compiler doesn't move the sqrt expression outside the loop. (See next page.)

It is the call of function **func** that prevents the optimization.

Optimization of applications

- Assembler listing of the loop of the program `codemotion1`¹:

```
;;;
;;; do i=1,NMAX

        movl    $1, 80(%esp)
        .align  4,0x90
        # LOE ebx esi edi
..B1.5:      # Preds ..B1.6 ..B1.4

;;;      a(i)=func(i) !real(i,rk)/real(NMAX,rk)

        lea     80(%esp), %eax
        pushl   %eax
        call    func_
        # LOE ebx esi edi f1
..B1.17:    # Preds ..B1.5
        popl    %ecx
        # LOE ebx esi edi f3
..B1.6:    # Preds ..B1.17
        movl    80(%esp), %eax
```

```
;;;      b(i)=a(i)/sqrt(x**2+y**2)

        fldl    40(%esp)
        fldl    56(%esp)
        fxch    %st(1)
        fmul    %st(0), %st
        fxch    %st(1)
        fmul    %st(0), %st
        fxch    %st(2)
        fstl    -8+codemotion_$A(,%eax,8)
        fxch    %st(2)
        faddp   %st, %st(1)
        fsqrt
        fdivrp  %st, %st(1)
        fstpl   -8+codemotion_$B(,%eax,8)

;;; end do

        addl    $1, %eax
        movl    %eax, 80(%esp)
        cmpl    $1000000, %eax
        jle     ..B1.5      # Prob 99%
                          # LOE ebx esi edi
..B1.7:      # Preds ..B1.6
```

1. Don't be frightened. There's no need to understand all of this.

Optimization of applications

- As we have seen the pattern of the application's *memory access* has a large effect on its performance.
- Cache and TLB¹ misses should be minimized.
- Arrays should be accessed in the order they are stored in the memory.
 - For multidimensional arrays in F90 this means that the innermost `do` loop should be the one that runs over the left-most index of the array.
 - In C it should be the rightmost index.

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

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

- In this way the memory is accessed with unit *stride*.

- By a simple loop interchange the `n` stride can be changed to unit stride:

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

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

1. Translation Lookaside Buffer = cache of virtual to physical address translation

Optimization of applications

- Optimization is not always this simple. For example:

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



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

- Some unrolling of both loops might help:

```
program memaccess_block
  use sizes
  implicit none
  integer,parameter :: N=3000,P=21
  real(rk) :: a(N,N),b(N,N)
  integer :: i,j
  real(rk) :: t1,t2

  call fillarrays()
  call cpu_time(t1)
  do i=1,N
    do j=1,N
      a(j,i)=a(j,i)+b(i,j)
    end do
  end do
  call cpu_time(t2)
  print *,t2-t1,maxval(a),maxloc(a),sum(a)
```

```
call fillarrays()
call cpu_time(t1)
do i=1,N,2
  do j=1,N,2
    a(j,i)=a(j,i)+b(i,j)
    a(j+1,i)=a(j+1,i)+b(i,j+1)
    a(j,i+1)=a(j,i+1)+b(i+1,j)
    a(j+1,i+1)=a(j+1,i+1)+b(i+1,j+1)
  end do
end do
call cpu_time(t2)
print *,t2-t1,maxval(a),maxloc(a),sum(a)

contains
  subroutine fillarrays()
    forall (i=1:N,j=1:N)
      a(i,j)=mod(i+1,P)+mod(j+1,P)
      b(i,j)=mod(i,P+10)+mod(j,P-5)
    end forall
  end subroutine fillarrays
end program memaccess_block
```

```
progs> ifort memaccess_block.f90
progs> a.out
0.53191900      87.00    103    526    0.182231E+09
0.28395600      87.00    103    526    0.182231E+09
```

Optimization of applications

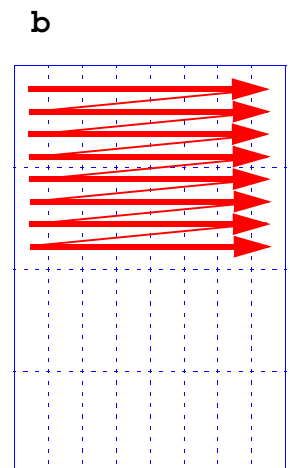
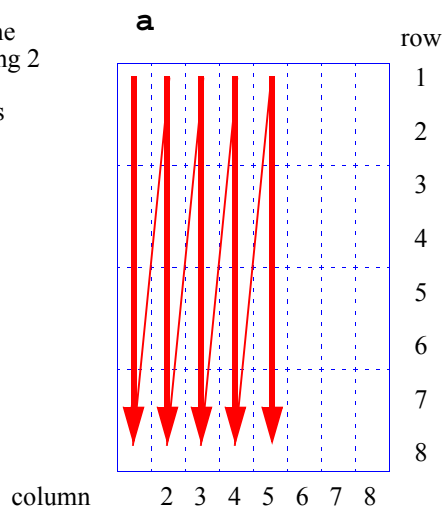
- Access of arrays a and b can be depicted in a way that clearly shows the usage of cache lines:

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

cache line
= containing 2
memory
locations

storage order

1	9						
2	10						
3	11						
4	12						
5	13						
6	...						
7							
8							

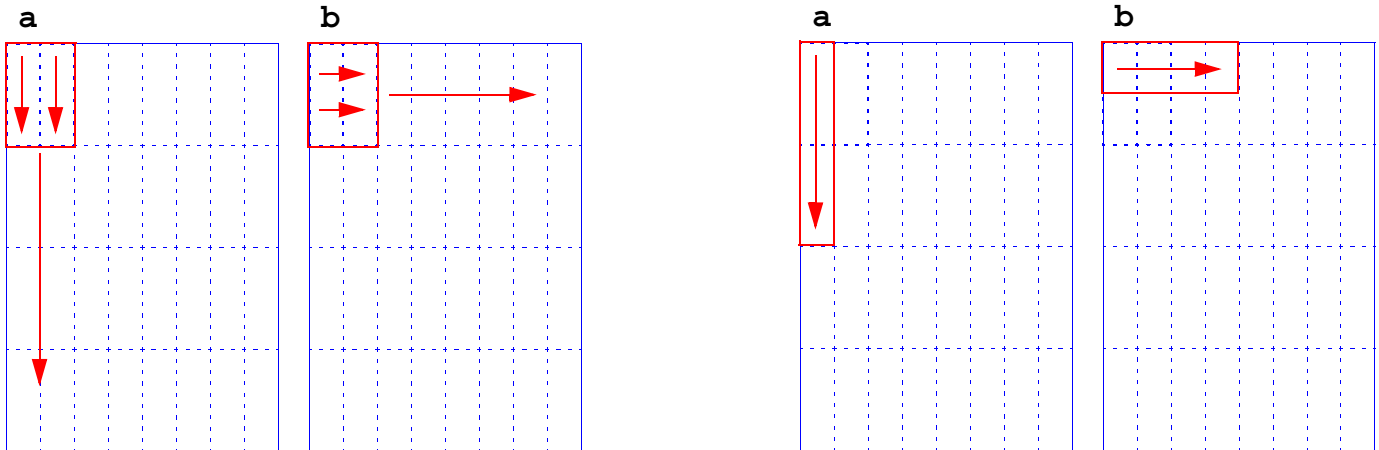


Optimization of applications

- What we actually did when we unrolled both loops to depth 2 in the previous example can also be viewed as consuming the arrays in small 2x2 rectangles:

```
do i=1,N,2
  do j=1,N,2
    a(j,i)=a(j,i)+b(i,j)
    a(j+1,i)=a(j+1,i)+b(i,j+1)
    a(j,i+1)=a(j,i+1)+b(i+1,j)
    a(j+1,i+1)=a(j+1,i+1)+b(i+1,j+1)
  end do
end do
```

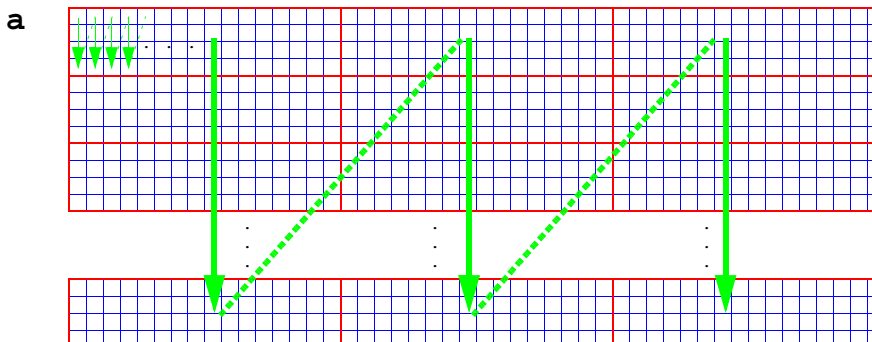
Compare to this



- This is called *blocking*.
- We can immediately see that the number cache misses is reduced compared to the case where both arrays are traversed as one single block.
- On the other hand there are more cache misses than in the case where both arrays are scanned with unit stride, i.e. in their storage order.

Optimization of applications

- A more complicated pattern of the array access gives even larger performance boost¹:



Example from page 26:

CPU times in seconds for 2048x2048 arrays

Direct double loop	0.49392500
2x2 blocking	0.27895800
Blocking show above	0.12898100

1. Example from K. Dowd: High Performance Computing, O'Reilly & Associates, 1993, Chapter 11, code in the file [memaccess_block.f90](#)

Optimization of applications

- Sometimes compiler can not do the optimization due to *ambiguity in memory references*.
- For example when doing computations with sparse matrices the indexing is indirect:

```
do i=1,n
  a(k(i))=b(k(i))+c(k(i))
enddo
```

- Now the compiler can not tell whether different iterations are independent.
- Other form of ambiguity is *aliasing*.
 - Aliasing may happen when dealing with pointers: It is possible that two pointers point to same target; they become aliases.
 - This is a more serious problem in C because there arrays are always pointers¹.
 - Two variables can become aliases also by a subroutine call:

```
call sub(a,a)
.
.
.

subroutine sub(x,y)
.
.
.
```

1. Well, depending how you define e.g. 2D arrays the compiler might know more about the possible aliasing between them.

Optimization of applications

- Data alignment

- Processor load data from memory in 4-32 byte chunks.
- The address of a chunk is a multiple of it size.
- This means that if memory is allocated for a double precision variable in such a way that its bytes belong to two 8-byte chunks the processor has to access memory twice in order to load the variable.
 - In this case data is *misaligned*.
- In many cases the compiler does the alignment; if not by default there are always options which force the natural data alignment.
- For best performance, data should be aligned as follows¹:

Size of data in bits	Address should be a multiple of
8	any
16	4
32	4
64	8
80	16
128	16

1. Intel® Fortran Compiler for Linux® Systems, User's Guide, Volume II: Optimizing Applications

Optimization of applications

- Example

```

program misaligned
  use sizes
  implicit none
  integer,parameter :: ik=selected_int_kind(1)
  integer,parameter :: NMAX=100000,ITER=10000
  character(len=80) :: argu
  real(rk) :: t1,t2
  integer :: i,n,c
  type rec
    real(rk) :: x
    integer(ik) :: i
    integer(ik) :: j
  end type rec
  type(rec) :: r(NMAX)

  print *,bit_size(r(1)%i)

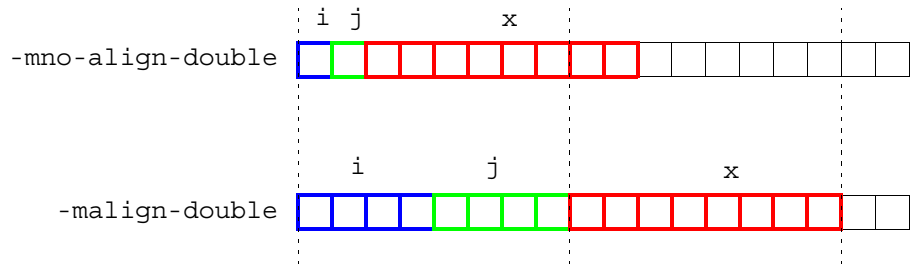
  call getarg(1,argu); read(argu,*) c
  r%x=0.0
  do n=1,NMAX
    r(n)%x=0.0
    r(n)%i=c
    r(n)%j=2*c
  end do
  call cpu_time(t1)
  do i=1,ITER
    do n=1,NMAX
      r(n)%x=r(n)%x+r(n)%i+r(n)%j
    end do
  end do
  call cpu_time(t2)
  print *,t2-t1,r(c)%x
end program misaligned

```

```

progs> gfortran -mno-align-double misaligned.f90 ; a.out 2
17.1143990000000 60000.0000000000
progs> gfortran -malign-double misaligned.f90 ; a.out 2
14.4118100000000 60000.0000000000

```



Optimization of applications

- CPU time as a function of array size:

