타키온에서의 성능 최적화

목 차 : 순차프로그램 성능 최적화

- ❑ 순차 프로그램 성능측정 및 분석
  - 순차코드의 성능 측정
  - 성능 측정 방법
- ❑ 시스템 특성
  - 프로세서
  - 계층구조의 메모리와 케쉬
- ❑ 순차코드의 성능 개선 방법 : Tachyon 컴파일러 적용
  - 인라이닝 (Inlining)
  - 루프 최적화 (Loop optimizations)
  - 메모리 최적화 (Memory optimizations)
  - 부동소수 연산 특성 분석 (Floating point arithmetic behavior)
  - 라이브러리의 사용 (Optimized mathematical libraries)

Single Processor Performance Tuning Techniques

순차코드의 성능 측정

- Wallclock time
  - 가장 기본적인 성능 측정방법
  - 프로그램 실행에 소모되는 전체시간 "time to solution"
  - CPU time 측정 vs Wallclock time

- 하드웨어 성능 측정
  - 프로그램 성능측정과 함께 필요
  - 부동소수점계산 성능 : floating point performance (GGLOP/s)
  - 메모리 bandwidth (GB/s)
  - Cache hit/miss rates

순차코드의 성능 측정 방법

- 성능 측정의 목적
  - 코드 성능 분석 및 이해
  - 병목지점 분석 : 순차 코드의 어떤 부분이 가장 최적화가 필요한가?
  - 컴파일러가 얼만큼의 성능개선에 도움을 주는가?

- 코드 성능 분석을 위한 기본적인 방법
  - 시간 측정 : Timing
  - 컴파일러에 의한 분석 : Compiler reports and listings
  - 프로파일링을 통한 세부 코드 분석
  - 하드웨어 성능 측정 (Hardware performance counters)
    - 메모리 및 cache hit/miss rates

Timing

- Time 명령어
  - Single node에서 지원하는 코드 시간측정 도구
  - /usr/bin/time

```
/usr/bin/time showq
....
0.00user 0.01system 0:00.02elapsed 88%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+2553minor)pagefaults 0swaps
```

- 절대 경로의 사용 권장
  - Shell에 따라 부족한 정보를 보여주는 time command 가 있을 수 있음.

- Report
  - user time : 프로그램 실행중 User process가 CPU를 사용한 시간
  - system time (kernel time) : Kernel process가 CPU를 사용한 시간
  - elapsed time (wallclock) : 프로그램 실제 실행 시간
  - % CPU --(user+system)/elapsed
  - memory, pagefault, and swap statistics
  - I/O statistics

Timing (con't.)

❑ 소스코드에 삽입되는 Timing 루틴들 (instrumented)

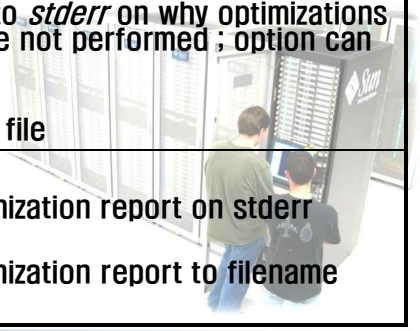
- C/C++
  - Wallclock: time(2), difftime(3), getrusage(2)
  - CPU: times(2)
- Fortran 77/90
  - Wallclock: SYSTEM_CLOCK(3)
  - CPU: DTIME(3), ETIME(3),
- MPI (C/C++/Fortran)
  - Wallclock: MPI_Wtime(3)]

Compiler Reports and Listings

❑ 최근의 고성능 컴퓨터의 컴파일러의 특징

- 다양한 최적화 능력 보유
- 최적화에 따른 코드의 변경 및 최적화 불가에 대한 결과를 보여줌
- 기본적으로 많은 정보를 보여주지 않음

GNU compilers	None
Portland Group 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



Exercise : Compiler Reporting

```
program report
implicit none
integer, parameter      :: N=1000,
itermax=1000
integer                 :: it,i,j
real   , dimension(N,N) :: a,b,c,d
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
real                    :: flops

call random_number(b)
call random_number(c)
call random_number(d)

st = dtime(rtime)

do it=1,itermax
  do j=1,N
    do i=1,N
      a(i,j) = b(i,j)+c(i,j)*d(i,j)
    end do
  end do
  if(it.eq.itermax)  &
    write(*,*) a(1,1),b(1,1),c(1,1),d(1,1)
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*2.0*N*N)/et
write(*,100) et
write(*,200) flops

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")
end program report
```

Exercise : Compiler Reporting (Portland Group)

```
$pgf90 -O2 -Munroll -Minfo=loop -Mneginfo=loop report.f90 -o test.x
report:
     18, Loop unrolled 4 times
$./test.x
 1.017629       0.9079230       0.8079612       0.1357810
loop time =       6.37000 seconds
loops runs at        313.97174 MFLOPS


$pgf90 -O2 -Munroll -Mvect=assoc -Minfo=loop -Mneginfo=loop report.f90 -o test.x
report:
     16, Loop not vectorized: contains call
     18, Loop unrolled 4 times
$./test.x
 1.017629       0.9079230       0.8079612       0.1357810
loop time =       6.37000 seconds
loops runs at        313.97174 MFLOPS
```

Exercise : Compiler Reporting (Portland Group, con't.)

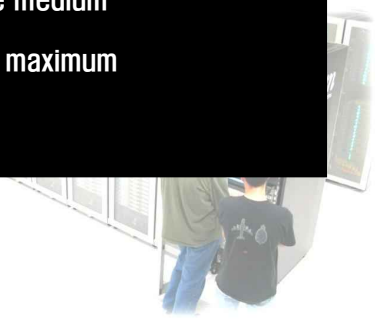
```
$pgf90 -O2 -Munroll -Mvect=assoc,prefetch -Minfo=loop -Mneginfo=loop report.f90 -o
test.x
report:
    16, Loop not vectorized: contains call
    18, Loop unrolled 4 times
        Generate 3 prefetch instructions for this loop
$./test.x
 1.017629       0.9079230       0.8079612       0.1357810
loop time =      5.77000 seconds
loops runs at       346.62045 MFLOPS

$pgf90 -O2 -Munroll -Mvect=assoc,sse -Minfo=loop -Mneginfo=loop report.f90 -o test.x
report:
    16, Loop not vectorized: contains call
    18, Generated vector sse code for inner loop
        Generated 3 prefetch instructions for this loop$./test.x
$./test.x
 1.017629       0.9079230       0.8079612       0.1357810
loop time =      5.48000 seconds
loops runs at       364.96350 MFLOPS
```

Exercise : Compiler Reporting (Intel)

```
$ifort -O3 -vec-report0 report.f90 -o test.x
.........
$man ifort
 -vec-report[n] (i32 and i64em)
            Specifies  the  amount  of vectorizer diagnostic information to
            report.  You can specify one of the following values for [n]:
            0 - Produces no diagnostic information.
            1 - Indicates vectorized loops. This is the default.
            2 - Indicates vectorized and non-vectorized loops.
            3 - Indicates vectorized and non-vectorized loops and prohibit-
            ing data dependence information.
            4 - Indicates non-vectorized loops.
            5  - Indicates non-vectorized loops and prohibiting data depen-
            dence information.
```

Exercise : Compiler Reporting (Intel , con't.)

```
$ifort -O3 -opt-report 0 report.f90 -o test.x
.........
$man ifort
  .........
  -opt-report  [n]
            Tells the  compiler  to  generate  an  optimization  report  to
            stderr. n is the level of detail in the report. You can specify
            one of the following values for [n]:
            0 - Tells the compiler to generate no optimization report.
            1 - Tells the compiler to generate a report  with  the  minimum
            level of detail.
            2  -  Tells  the  compiler to generate a report with the medium
            level of detail. This is the default if n is not specified.
            3 - Tells the compiler to generate a report  with  the  maximum
            level of detail.
  .........
```



Profiling

- 코드의 자세한 성능분석
  - 전체 계산시간의 세부 분석에 필요
  - 프로시져의 호출관계 분석
  - 프로시져 또는 섹션의 시간 소모 분석
  - Sampling technique or on entry/exit of a code block
  - 방대한 코드의 최적화를 위해 반드시 필요
- 세분화된 코드의 분석 (Levels of granularity)
  - Subroutine
  - Basic block
  - Source code line
- 프로파일을 위한 컴파일
  - 컴파일 과정에서 옵션 필요
  - 코드의 실행과 동시에 history와 profile을 생성
  - 프로파일링 툴들을 통해 분석 가능

프로파일링 컴파일러 옵션

GNU compilers	
-Pg	Enable function-level profiling using gprof
Portland Group compilers	
-Mprof=func	Enable function-level profiling using pgprof
-Mprof=lines	Enable source code line-level profiling using gprof; overhead may be extremely high!
Intel compilers	
-Pg	Enable function-level profiling using gprof
-prof-gen	Enable basic-block profiling for profile-guided optimization
-prof-use	Enable profile-guided optimization

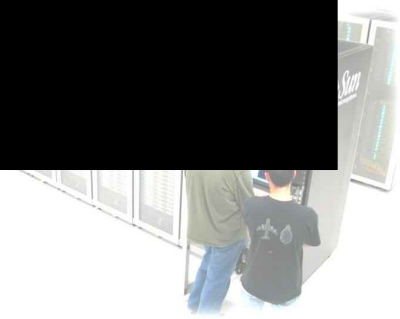
Exercise : Profiling

```
program profiling
implicit none
integer, parameter      :: N=1000, itermax=100
integer                 :: it,i,j
real   , dimension(N,N) :: b,c,d
real   , dimension(N)   :: a
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
real                    :: flops

call random_number(b)
call random_number(c)
call random_number(d)

st = dtime(rtime)

do it=1,itermax
  do j=1,N
    a(j) = vecsum(b,j) + vecprod(c,j)*vecsum(d,j)
  end do
  if(mod(it,100).eq.0) write(*,*)
a(1),b(1,1),c(1,1),d(1,1)
end do

et = dtime(rtime)
flops = (1.0E-6*itermax*N*(3*N+2))/et
```

```
write(*,100) et
write(*,200) flops

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

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

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

end program profiling
```

Exercise : Profiling (Portland Group)

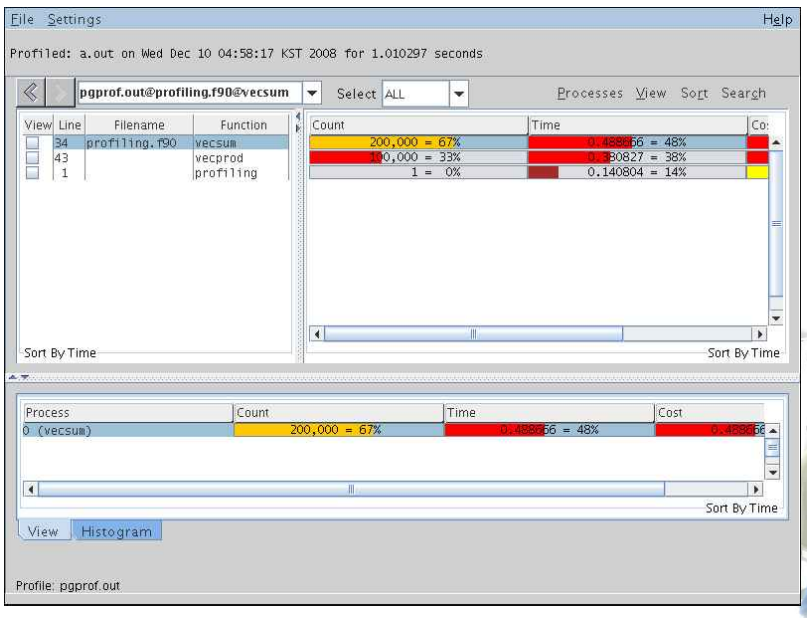
```
$pgf90 -Mprof=func -O2 -Munroll -Mvect=assoc,prefetch -Minfo=loop,inline -
Mneginfo=loop profiling.f90 -o test.x

profiling:
     18, Loop not vectorized: contains call
vecsum:
     38, Unrolled inner loop 8 times
         Generated 1 prefetch instructions for this loop
         Loop unrolled 7 times (completely unrolled)
vecprod:
     47, Unrolled inner loop 8 times
         Generated 1 prefetch instructions for this loop
         Loop unrolled 7 times (completely unrolled)

$./test.x
 498.9057       0.9079230       0.8079612       0.1357810
loop time =      0.95000 seconds
loops runs at       315.99997 MFLOPS
```



Exercise : Profiling (Portland Group, con't.)

```
$pgprof pgprof.out
```

PGPROF Profiler v7.1



Exercise : Profiling (Intel)

```
$ ifort -O3 -unroll profiling.f90 -o test.x -pg
profiling.f90(19): (col. 12) remark: LOOP WAS VECTORIZED.
profiling.f90(19): (col. 26) remark: LOOP WAS VECTORIZED.
profiling.f90(19): (col. 39) remark: LOOP WAS VECTORIZED.

$ ./test.x
 516.1282       3.9208680E-07  0.4334742       0.6844026
loop time =       0.48000 seconds
loops runs at        625.41663 MFLOPS

$ gprof ./test.x gmon.out > profile.txt
```

Exercise : Profiling (Intel, con't.)

```
$ more profile.txt
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 89.59      0.43     0.43        1   430.01   430.01  MAIN__
  9.38      0.48     0.05                             for_random_number_single
  1.04      0.48     0.01                             for_random_seed_bit_size

     ..........
```

```
                            Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 2.08% of 0.48 seconds

index % time    self  children    called     name
                0.43    0.00       1/1          main [2]
[1]     89.6    0.43    0.00       1          MAIN__ [1]
-----------------------------------------------
                                   <spontaneous>
[2]     89.6    0.00    0.43                 main [2]
                0.43    0.00       1/1          MAIN__ [1]
```

시스템 영향 (프로세서 성능)

- 최근의 프로세서 Architecture
  - 파이프라인 유닛
  - 슈퍼스칼라 실행
  - 비순차적 실행(Out-of order execution) : 명령어를 CPU에 넣어지는 순서가 아닌 재조합
  - 예측실행(speculative execution)
  - Floating point instruction set extensions : 부동소수점 연산의 효율성 증대
- 계층적 메모리/캐시 구조
  - Hierarchical memory
  - Cache layout
  - Prefetching

최근의 프로세서 구조

Modern microprocessor cores are complex :

- 다양한 성능 향상 유닛의 지원
  - 다수의 파이프라이닝을 가지는 다수의 연산 유닛
  - SIMD/vector 유닛
  - 비순차 실행 / 예측 실행
- 복잡해진 명령 집합 (instruction set)
  - RISC와 CISC의 경계가 모호해 짐
  - 명령해독기(Instruction decoder)들이 명령어들을 다수의 "micro-ops" 로 번역
  - 컴파일러의 역할 증대
    - Extra-hard to decide how to schedule instructions

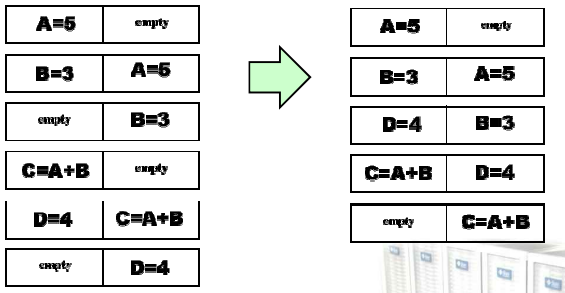
Superscalar Execution

- Superscalar
  - 한 프로세서 사이클 동안에 하나 이상의 명령어를 실행시킬 수 있는 프로세서 아키텍쳐
- Supearscalar 프로세서
  - AMD Opteron
    - 3 Floating point/MMX/SSE units
    - 3 Integer units
    - 3 Load/store units
  - IBM POWER6
    - 2 Floating point units
    - 2 Integer units
    - 2 Load/store units
  - Intel Itanium 2
    - 2 Floating point units
    - 4 Integer units
    - 2 Load/store units
  - Intel Xeon
    - 2 Floating point units
    - 2 Integer units
- 2 Load/store units

비순차 실행 및 예측실행

- 비순차 실행 (Out-of-order execution)
  - CPU가 명령어들을 CPU에 넣어지는 순서가 아닌 효율적인 방법으로 실행
  - 파이프라인내의 빈 공간이 생기는 것을 방지
  - 싸이클의 절약을 통해 CPU가 최고 성능을 내도록 함 (ex: 2cycles /instruction)

```
A = 5
B = 3
C = A + B
D = 4
```



- 예측 실행 (Speculative execution)
  - 코드 경로(분기점)에 대한 예측을 통해 수행
  - 예측 실패(mis-predicted branch)에 따른 손실 고려 필요
  - 대부분의 프로그램들이 지역성(locality)를 따르게 되므로, 90% 이상의 예측율을 보임

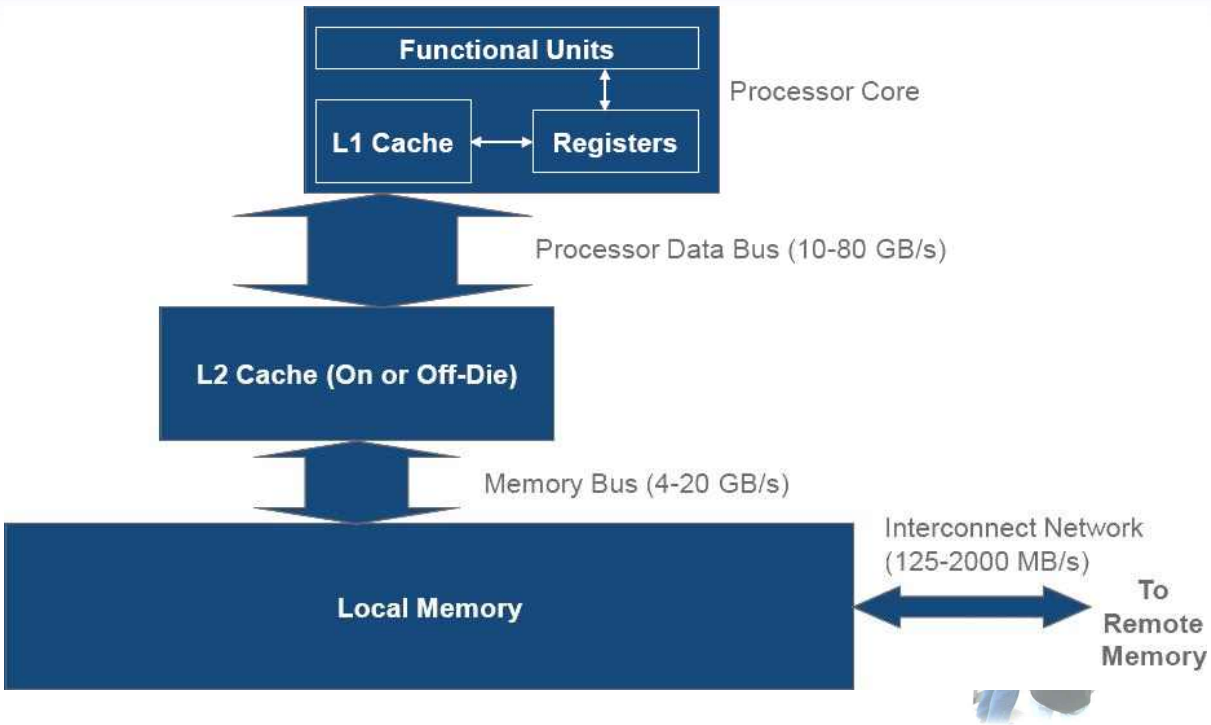
부동소수점 명령어 집합의 확장 (Floating Point Instruction Set Extensions)

- Newer processors have additional floating point instructions beyond the usual floating point add and multiply instructions:
- Square root instruction --usually not pipelined!
  - AMD Opteron
  - IBM POWER
  - Intel Itanium 2
  - Intel Xeon
- SIMD (a.k.a. vector) floating point instructions
  - AMD Opteron
  - IBM Cell -designed around the concept!
  - Intel Xeon
- Combined floating point multiply/add (MADD) instruction
  - AMD Opteron ("Barcelona" and after, using SIMD)
  - IBM POWER
  - Intel Itanium 2
  - Intel Xeon ("Woodcrest" and after, using SIMD)

계층적 메모리/ 캐시 구조

- 계층적 메모리 구조
  - 메모리 레벨
  - 레벨에 따른 지연(latency) 구성
- Cache Layout
- 프리패칭 (Prefetching)

메모리 계층 구조



메모리 계층 구조 (con't.)

- 레지스터 : 연산자 또는 연산유닛의 결과를 저장하는 on-chip circuitry
- L1 (Primary) Data Cache: Small (on-chip) cache used to hold data about to operated on by processor.
- L2 (Secondary) Cache: Larger (on-or off-chip) cache used to hold data and instructions retrieved from local memory. Some systems also have L3 and even L4 caches.
- Local Memory: Memory on the same node as the processor.
- Interconnect Network: Wiring which connects nodes to each other; topologies can vary:
  - Cray XT3/4/5, IBM BlueGene: 3D Torus
  - SGI Altix: Fat bristled hypercube
  - IBM SP, most clusters: Switch or tree of switches
  - TACHYON : Infiniband
- Remote Memory: Memory on another node but accessible to all processors in the network.

계층적 메모리 구조와 지연시간

- 계층적 메모리 구조의 핵심은 상위레벨의 메모리보다 하위레벨의 메모리가 더 적은 지연시간을 가지도록 하는 것

- Actual latencies for an Itanium 2 (1.4GHz) :
  - L1 data cache: 3 CPs
  - L2 cache: 9 CPs
  - L3 cache: 24 CPs
  - Local memory: 60 CPs

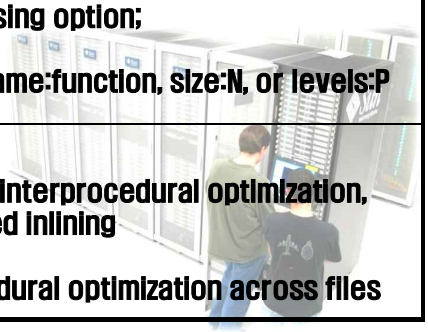
- Actual latencies for an Opteron 8218 (2.6GHz):
  - L1 data cache: 3 CPs
  - L2 cache: 12 CPs
  - Local memory: 166 CPs

General Performance Tuning Techniques

- 인라이닝 (Inlining)
- 루프 최적화 (Loop Optimization)
  - Pipelining and vectorization
  - Unrolling
  - Splitting
- 메모리 최적화 (Memory Optimization)
  - Cache alignment
  - Stride-One memory access
  - Avoiding cache thrashing
  - Prefetching
- 부동소수 연산
  - Instruction set extensions
  - Division
  - Exception handling
- 최적화된 수학라이브러리의 사용

인라이닝 컴파일러 옵션

GNU compilers	
-fno-inline	Disable inlining
-finline-functions	Enable inlining of functions
Portland Group compilers	
-Mextract=option[,option,…]	Extract functions selected by option for use in inlining; option may be name : function or size : N , where N is a number of statements
-Minline=option[,option,…]	Perform inlining using option; option may be lib:filename.ext, name:function, size:N, or levels:P
Intel compilers	
-ip	Enable single-file interprocedural optimization, including enhanced inlining
-ipo	Enable interprocedural optimization across files



Exercise : Inlining

```
program inlining
implicit none
integer, parameter      :: N=40000000
integer                 :: i
real   , parameter      :: a=12.34567
real                    :: x,y,z
real                    :: st,et,dtime
real   , dimension(2)   :: rtime

st = dtime(rtime)

do i=1,N
  call myfunc(a,x,y,z)
end do

et = dtime(rtime)
write(*,100) et
write(*,*) a,x,y,z

100 format("loop time = ",f12.5," seconds")
```

```
CONTAINS
subroutine myfunc(a,x,y,z)
real   , intent(IN)     :: a
real   , intent(INOUT)  :: x,y
real   , intent(OUT)    :: z
call random_number(x)
call random_number(y)
z = sqrt(y*y + a*x*x)
return
end subroutine myfunc

end program inlining
```

Exercise : Inlining (Portland Group)

```
$pgf90 -O2 inlining.f90 -o test.x
$./test.x
loop time =      2.22000 seconds
    12.34567       0.1012497       0.8724552       0.9421994

$pgf90 -O2 -Minline inlining.f90 -o test.x
$./test.x
loop time =      2.06000 seconds
    12.34567       0.1012497       0.8724552       0.9421994
```

Inlined loop runs 7.21% faster than non-inlined loop

Exercise : Inlining (Intel)

```
$ifort -O2 inlining.f90 -o test.x
$./test.x
loop time =      1.32000 seconds
   12.34567      0.5890008      8.0076814E-02   2.071086

$ifort -O2 -ip inlining.f90 -o test.x
$./test.x
loop time =      1.31000 seconds
   12.34567      0.5890008      8.0076814E-02   2.071086
```

Inlined loop runs 0.76% faster than non-inlined loop

루프 최적화 (Loop Optimizations)

- Vectorization and pipelining
- Loop unrolling
- Loop splitting

Vectorization and Pipelining 컴파일러 옵션

GNU compilers -ftree-vectorize	Perform loop vectorization on trees
Portland Group compilers -Mvect -Mvect=cachesize:N	Enable vectorization, including pipelining. Enable vectorization and assume L2 cache size is N bytes for the purposes of blocking and tiling optimizations.
Intel compilers -O3	Aggressive optimization, including vectorization and other loop transformations

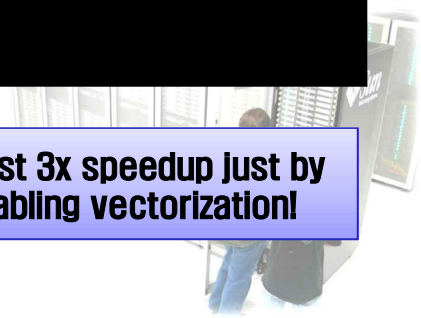
Exercise : Vectorization

```
program vectorization
implicit none
integer, parameter      :: N=1000000,
itermax=1000
integer                 :: it,i
real   , parameter      :: coeff=0.125
real   , dimension(N)   :: a,b,c
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
common/saver/a,b,c

call random_number(b)
call random_number(c)

st = dtime(rtime)

do it=1,itermax
  do i=1,N
    a(i) = b(i) + coeff*c(i)
  end do
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*2.0*N)/et
write(*,100) et
write(*,200) flops
write(*,*) a(1),b(1),c(1)

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program vectorization
```

Exercise : Vectorization (Portland Group, con't.)

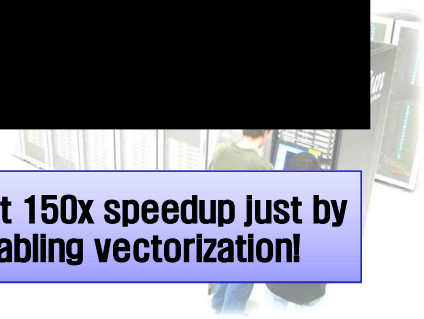
```
$pgf90 -O2 -Minfo=loop vectorization.f90 -o test.x
$./test.x
loop time =      5.05000 seconds
loops runs at       396.03961 MFLOPS
    1.008918       0.9079230       0.8079612

$pgf90 -O2 -Mvect -Minfo=loop vectorization.f90 -o test.x
vectorization:
    17, Created a tiled loop
    18, Created a tiled loop
$./test.x
loop time =      1.77000 seconds
loops runs at      1129.94360 MFLOPS
    1.008918       0.9079230       0.8079612
```



Almost 3x speedup just by enabling vectorization!

Exercise : Vectorization (Intel)

```
$ ifort -O2 vectorization.f90 -o test.x
vectorization.f90(18): (col. 3) remark: LOOP WAS VECTORIZED.

$ ./test.x
loop time =      3.02000 seconds
loops runs at       662.25171 MFLOPS
  5.4184671E-02  3.9208680E-07  0.4334742

$ ifort -O3 vectorization.f90 -o test.x
vectorization.f90(17): (col. 1) remark: PERMUTED LOOP WAS VECTORIZED.

$ ./test.x
loop time =      0.02000 seconds
loops runs at    100000.00781 MFLOPS
  5.4184671E-02  3.9208680E-07  0.4334742
```



Almost 150x speedup just by enabling vectorization!

Loop Unrolling

- 컴파일러에 의해 대부분 자동으로 unrolling이 수행됨
- 파이프라인의 활용도를 극대화 시킴
- 슈퍼스칼라 프로세서의 다중 파이프라인 연산 유닛의 활용도를 극대화
- Hiding load latencies : 컴파일러가 array 연산중 레지스터에 다음 연산 arrry의 pre-load를 가능하도록 함
- 제약사항 : Limited number of FP registers on some architectures
  - AMD Opteron: 16
  - IBM POWER6: 32
  - Intel Itanium: 128
  - Intel Xeon: 16

Loop Unrolling Example

- **Normal loop**

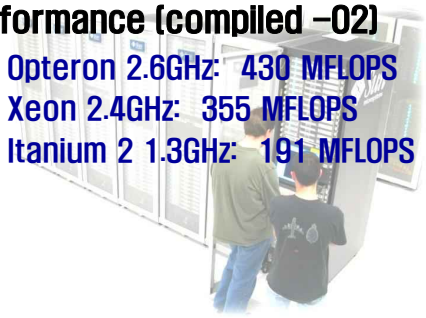
```
do i=1,N
    a(i)=b(i)+coeff*c(i)
enddo
```

- **Manually unrolling loop**

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

- **Performance (compiled -O2)**
  - Opteron 2.6GHz: 425 MFLOPS
  - Xeon 2.4GHZ: 356 MFLOPS
  - Itanium 2 1.3GHz: 190 MFLOPS

- **Performance (compiled -O2)**
  - Opteron 2.6GHz: 430 MFLOPS
  - Xeon 2.4GHz: 355 MFLOPS
  - Itanium 2 1.3GHz: 191 MFLOPS



Loop Unrolling 컴파일러 옵션

GNU compilers	
-funroll-loops	Enable loop unrolling
-funroll-all-loops	Unroll all loops; not recommended
Portland Group 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

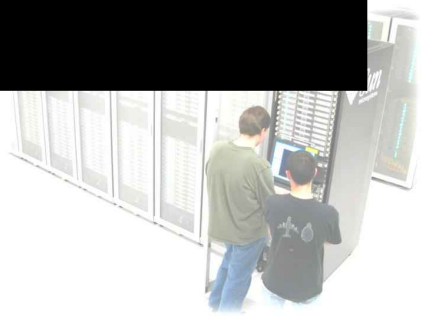
Exercise : Loop Unrolling Directives

```
program unrolling
implicit none
integer, parameter      :: N=1000000,
itermax=1000
integer                 :: it,i
real   , parameter      :: coeff=0.125
real   , dimension(N)   :: a,b,c
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
common/saver/a,b,c

call random_number(b)
call random_number(c)

st = dtime(rtime)

do it=1,itermax
  do i=1,N
    a(i) = b(i) + coeff*c(i)
  end do
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*2.0*N)/et
write(*,100) et
write(*,200) flops
write(*,*) a(1),b(1),c(1)

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program unrolling
```

Exercise : Loop Unrolling Directives (Portland Group)

```
$pgf90 -O2 -Minfo=loop unrolling.f90 -o test.x
$./test.x
loop time =      5.06000 seconds
loops runs at       395.25696 MFLOPS
    1.008918       0.9079230       0.8079612

$pgf90 -O2 -Munroll -Minfo=loop unrolling.f90 -o test.x
unrolling:
    18, Loop unrolled 8 times
$./test.x
loop time =      4.92000 seconds
loops runs at       406.50409 MFLOPS
    1.008918       0.9079230       0.8079612
```



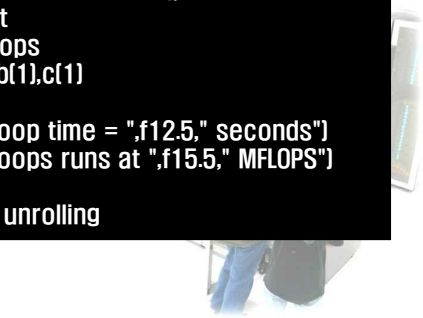
Exercise : Loop Unrolling Directives (Intel)

```
$ifort -O2 unrolling.f90 -o test.x
unrolling.f90(18): (col. 3) remark: LOOP WAS VECTORIZED.
$./test.x
loop time =      2.51000 seconds
loops runs at       796.81281 MFLOPS
  5.4184671E-02  3.9208680E-07  0.4334742

$ifort -O2 -unroll unrolling.f90 -o test.x
unrolling.f90(18): (col. 3) remark: LOOP WAS VECTORIZED.
$./test.x
loop time =      2.50000 seconds
loops runs at       800.00006 MFLOPS
  5.4184671E-02  3.9208680E-07  0.4334742
```

Exercise : Loop Unrolling Directives (Manually)

```
program unrolling
implicit none
integer, parameter      :: N=1000000,
itermax=1000
integer                 :: it,i1,i2,i3
real   , parameter      :: coeff=0.125
real   , dimension(N)   :: a,b,c
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
common/saver/a,b,c

call random_number(b)
call random_number(c)

st = dtime(rtime)
```

```
do it=1,itermax
  do i=1,N,4
    i1 = i + 1
    i2 = i + 2
    i3 = i + 3
    a(i ) = b(i ) + coeff*c(i )
    a(i1) = b(i1) + coeff*c(i1)
    a(i2) = b(i2) + coeff*c(i2)
    a(i3) = b(i3) + coeff*c(i3)
  end do
end do

et = dtime(rtime)
flops = (1.0E-6*itermax*2.0*N)/et
write(*,100) et
write(*,200) flops
write(*,*) a(1),b(1),c(1)

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program unrolling
```



Loop Splitting

```
p = 10;
for (i=0; i<10; ++i)
{
   y[i] = x[i] + x[p];
   p = i;
}
```

In the above code, only in the 1st iteration is p=10.
For all other iterations p=i-1
We get the following after *loop peeling*

```
y[0] = x[0] + x[10];
for (i=1; i<10; ++i)
{
   y[i] = x[i] + x[i-1];
}
```

Loop Splitting 컴파일러 옵션

GNU compilers	None
Portland Group compilers -Mvect	Enable vectorization, including loop tiling
Intel compilers -O3	Aggressive optimization, including loop transformations

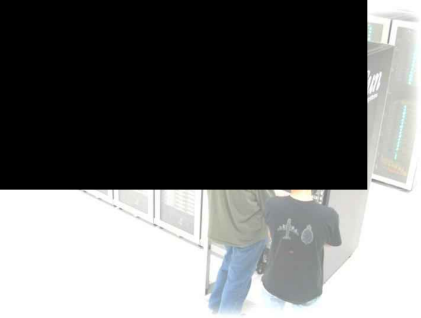
Exercise : Loop Splitting

```
program splitting
implicit none
integer, parameter      :: N=100000,
itermax=1000
integer                 :: it,i
real   , dimension(N)   :: x,a,b,c,d,e
real   , dimension(N)   :: y,f,g,h,k,l
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime

x=0.0 ; a=1.0 ; b=2.0 ; c=3.0 ; d=4.0 ; e=5.0
y=0.0 ; f=6.0 ; g=7.0 ; h=8.0 ; k=9.0 ; l=10.0

st = dtime(rtime)

do it=1,itermax
  do i=1,N
    x(i) = x(i) + a(i) + b(i) + c(i) + d(i) + e(i)
    y(i) = y(i) + f(i) + g(i) + h(i) + k(i) + l(i)
  end do
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*2.0*N)/et
write(*,100) et
write(*,200) flops
write(*,*) x(1),y(1)

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program splitting
```

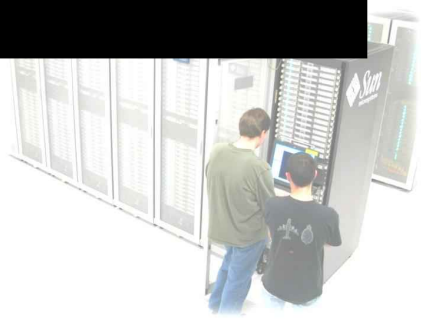
Exercise : Loop Splitting (Portland Group)

```
$pgf90 -O2 -Minfo=loop splitting.f90 -o test.x
splitting:
    12, Memory set idiom, loop replaced by memset call
$ ./test.x
loop time =      1.75000 seconds
loops runs at       571.28572 MFLOPS
    15000.00        40000.00

$ pgf90 -O2 -Mvect -Minfo=loop splitting.f90 -o test.x
splitting:
    12, Memory set idiom, loop replaced by memset call
    16, Created a tiled loop
    17, Created a tiled loop
$ ./test.x
loop time =      0.69000 seconds
loops runs at       1449.85510 MFLOPS
    15000.00        40000.00
```



Exercise : Loop Splitting (Intel)

```
$ifort -O2 splitting.f90 -o test.x
$./test.x
loop time =      1.46000 seconds
loops runs at       684.93152 MFLOPS
   15000.00       40000.00

$ ifort -O3 splitting.f90 -o test.x
splitting.f90(11): (col. 1) remark: LOOP WAS VECTORIZED.
splitting.f90(11): (col. 1) remark: LOOP WAS VECTORIZED.
$ ./test.x
loop time =      0.14000 seconds
loops runs at      7142.85742 MFLOPS
   15000.00       40000.00
```



메모리 최적화

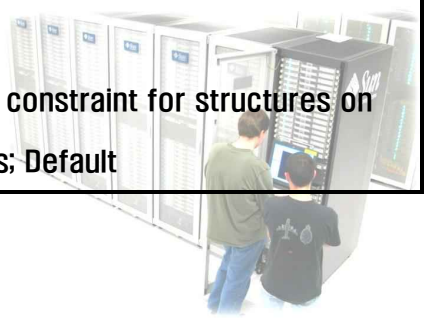
- 메모리 사이즈 증가율(Moore의 법칙) vs. 메모리 속도
  - DRAM density increases roughly 60% per year (i.e., quadrupling every *three* years)
  - DRAM cycle time decreases by 1/3 every *ten* years
- 성능 최적화의 대부분이 메모리 접근 성능에 기준함
  - 필요한 데이터를 최대한 프로세서에 근접하게 위치시킴으로써 데이터 지역성을 증대시시는 것이 목적
  - Hide memory access latency 기술의 필요

Cache Line Alignment of Arrays

- 대부분의 시스템이 지원하는 최적화 기법중의 하나로, 캐시라인 경계의 시작 위치에 배열 또는 common block들을 배치하도록 하는 방법
  - 사용자들이 필요한 변수들이 데이터 캐시에 위치한 레이아웃을 이해하기 편리해진다
  - 캐시라인의 활용도가 높아지게 됨. 캐시라인을 실제 메모리로 채워서 사용 가능
- 컴파일러가 자동으로 변수의 캐시라인 정렬을 지원하고 있음

Cache Alignment 컴파일러 옵션

GNU compilers -malign-double	Align double precision variables on 64-bit Boundaries
Portland Group compilers -Mcache align	Align arrays not in COMMON blocks on cache line Boundaries
Intel compilers -Zp8	Specify alignment constraint for structures on 8-byte boundaries; Default



Stride-One Memory Access

- 캐시사용에 있어서 가장 중요하고, 효과적인 방법
- 반드시 루프에서 배열의 연속적인 접근은 하나의 stride씩 이루어지도록
- 다차원 배열의 Nested loops에서 특히 중요

- Most programming languages (C, C++, etc.) are *row-major*; that is, in a 2D array, they store elements consecutively by row:
  - First array index should be outermost loop
  - Last array index should be innermost loop

- However, all versions of Fortran are *column-major*, so there the reverse is true:
  - Last array index should be outermost loop
  - First array index should be innermost loop

Stride-One Memory Access Examples

Real 배열 b(1024,1024) 를 접근하는 두 예제 프로그램

- 첫 번째 예제 프로그램 :
  - Array b는 column 단위로 정확히 접근되고 있다
  - 캐시라인에 연속적인 메모리 접근이 가능
  - 각각의 캐시라인은 한번의 fectching으로 4번 사용 가능 (3번의 cache hits)

- 두 번째 예제 프로그램 :
  - Array b는 row 단위로 정확하지 못하게 접근되고 있다
  - b(1,1)이 fetch되고 나서 다음 element b(1,2)는 b(1,1)과 같은 캐시라인에 존재하지 않음
  - Cache miss 의 발생 : 새로운 local momory의 접근이 필요하게되며, 기존의 캐시라인은 다른 데이터로 변경되어짐
  - 극심한 성능의 저하를 가져옴

Exercise : Good Cache Reuse

```
program stride1
implicit none
integer, parameter      :: N=1024,
M=1024,itermax=100
integer                 :: it,i,j
real   , dimension(N,M) :: a,b
real   , dimension(N)   :: c
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime

call random_number(b)
call random_number(c)

st = dtime(rtime)

do it=1,itermax
  do j=1,M
    do i=1,N
      a(i,j) = b(i,j) + c(i)
    end do
  end do
  write(*,*) a(1,1),b(1,1)
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*N*M)/et
write(*,100) et
write(*,200) flops

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program stride1
```

a and b are accessed correctly (by column)

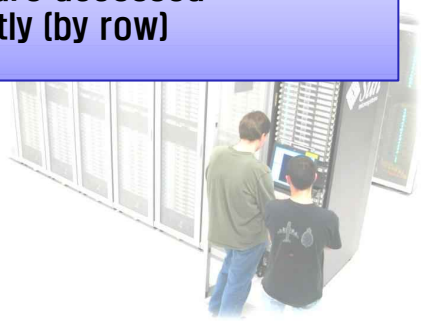
Exercise : Poor Cache Reuse

```
program stride2
implicit none
integer, parameter      :: N=1024,
M=1024,itermax=100
integer                 :: it,i,j
real   , dimension(N,M) :: a,b
real   , dimension(N)   :: c
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
```

```
call random_number(b)
call random_number(c)

st = dtime(rtime)

do it=1,itermax
  do i=1,N
    do j=1,M
      a(i,j) = b(i,j) + c(i)
    end do
  end do
  write(*,*) a(1,1),b(1,1)
end do
```

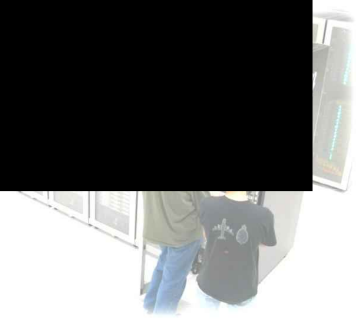
```
et = dtime(rtime)
flops = (1.0E-6*itermax*N*M)/et
write(*,100) et
write(*,200) flops

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program stride2
```

a and b are accessed incorrectly (by row)



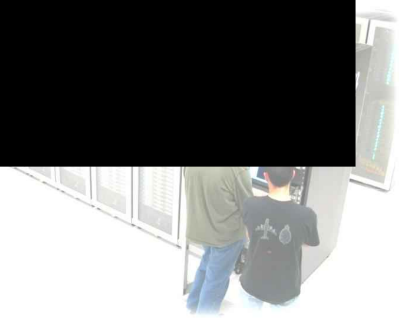
Exercise : Cache Reuse (Portland Group)

```
$pgf90 -O2 -Minfo=loop -Mneginfo=loop stride1.f90 -o test.x
$ ./test.x
......
loop time =      0.77000 seconds
loops runs at       136.17870 MFLOPS$

$pgf90 -O2 -Munroll -Minfo=loop stride1.f90 -o test.x
stride1:
    18, Loop unrolled 8 times
$ ./test.x
.......
loop time =      0.73000 seconds
loops runs at       143.64055 MFLOPS


$pgf90 -O2 -Munroll -Minfo=loop stride2.f90 -o test.x
$ ./test.x
.......
loop time =     32.72000 seconds
loops runs at        3.20469 MFLOPS
```



Exercise : Cache Reuse (Intel)

```
$ ifort -O2 stride1.f90 -o test.x
$ ./test.x
......
loop time =      0.28000 seconds
loops runs at       374.49142 MFLOPS

$ ifort -O2 -unroll stride1.f90 -o test.x
stride1:
    18, Loop unrolled 8 times
$ ./test.x
.......
loop time =      0.29000 seconds
loops runs at       361.57794 MFLOPS

$ ifort -O2 -unroll stride2.f90 -o test.x
$ ./test.x
.......
loop time =      8.93000 seconds
loops runs at       11.74217 MFLOPS
```



Loop Interchange for Stride-One Memory Access

- Loop interchange (also known as loop nest optimization)
  - 컴파일러가 자동적으로 루프를 재조정하여 stride-one memory access 가 가능하도록 하는 작업
  - 결과에 지장을 주지 않는 선에서 inner / outer loop들을 재구성

- 컴파일 옵션으로 정확한 memory access (stride-one memory access) 만큼의 효과를 기대할 수 있음
  - 컴파일러에 따라 변경된(또는 변경되지 못한) 루프를 상세히 리포팅

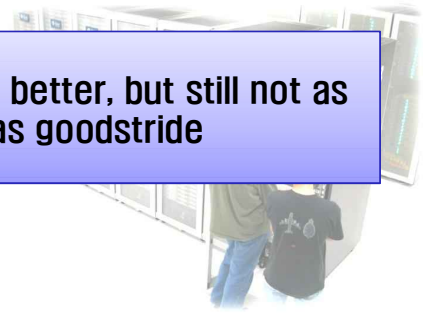
Loop Interchange 컴파일러 옵션

GNU compilers	None
Portland Group compilers -Mvect	Enable vectorization, including loop interchange
Intel compilers -O3	Enable aggressive optimization, including loop transformations

Exercise : Loop Interchange (Portland Group)

```
$pgf90 -02 -Mvect -Munroll -Minfo=loop -Mneginfo=loop stride2.f90 -o test.x
stride2:
    16, Loop not vectorized: contains call
    17, Interchange produces reordered loop nest: 18, 17
        Created a tiled loop
    18, Created a tiled loop
$ ./test.x
.....
loop time =      0.86000 seconds
loops runs at       121.92744 MFLOPS
```

Much better, but still not as fast as goodstride



Exercise : Loop Interchange (Intel)

```
$ifort -03 -unroll stride2.f90 -o test.x
stride2.f90(17): (col. 3) remark: PERMUTED LOOP WAS VECTORIZED.
$ ./test.x
.....
loop time =      0.28000 seconds
loops runs at       374.49142 MFLOPS
```

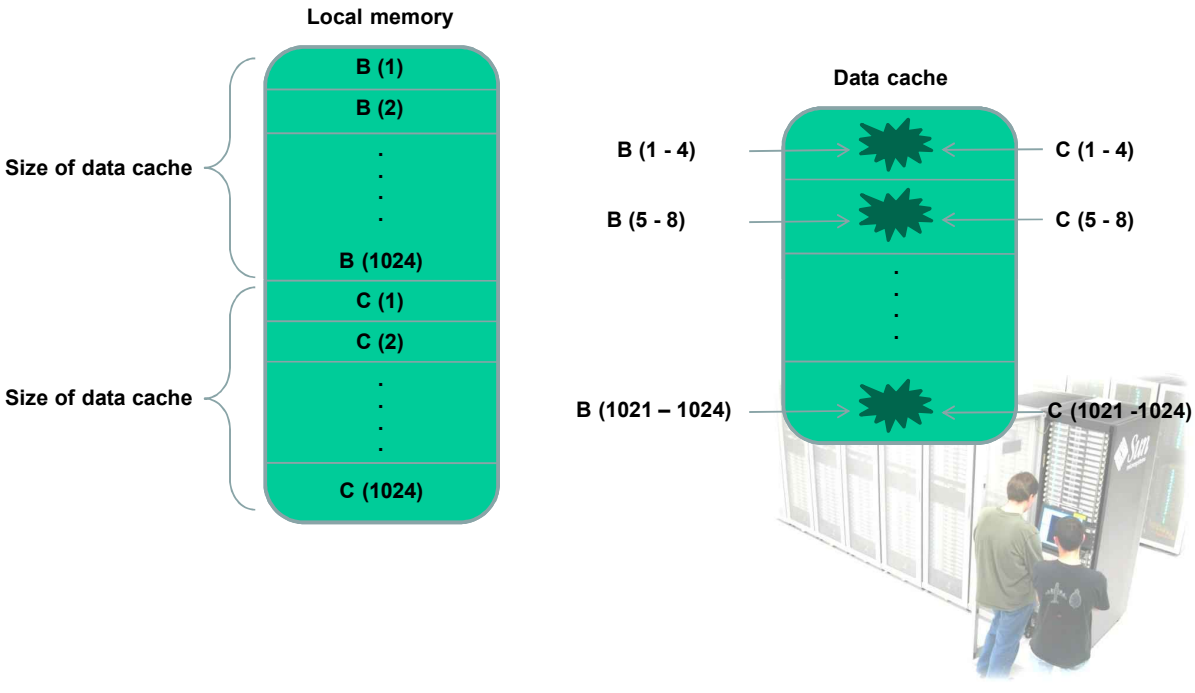
Almost as fast as goodstride

Cache Thrashing의 제거

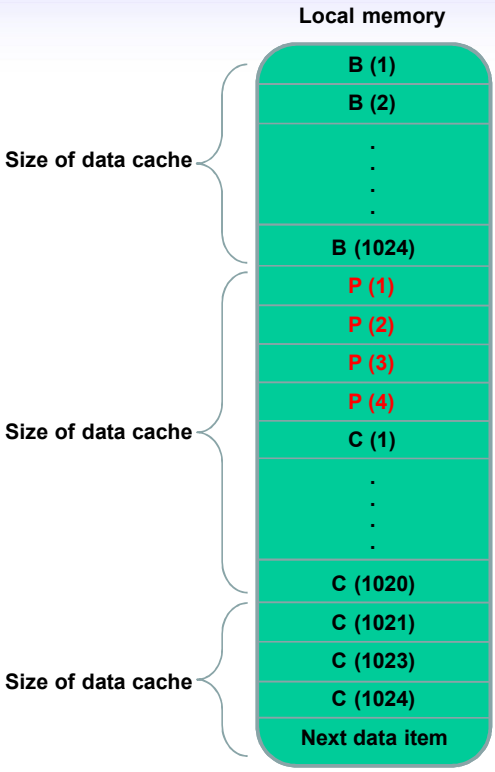
- **Cache thrashing**
  - 같은 캐시라인에 사상되는 서로 다른 메모리 위치에 대한 접근이 반복적으로 교차하는 경우 발생
  - 기존의 캐시라인에 있는 데이터를 사용하지 못하고, 새로운 local memory의 접근을 통해 캐시라인을 업데이트 해야 할 필요성이 있을 때 발생
  - Bad-stride 프로그램 예제에서 발생하는 cache missing의 경우에 이에 해당되며 loop interchange를 통해 cache thrashing을 제거하여 코드의 성능을 증대

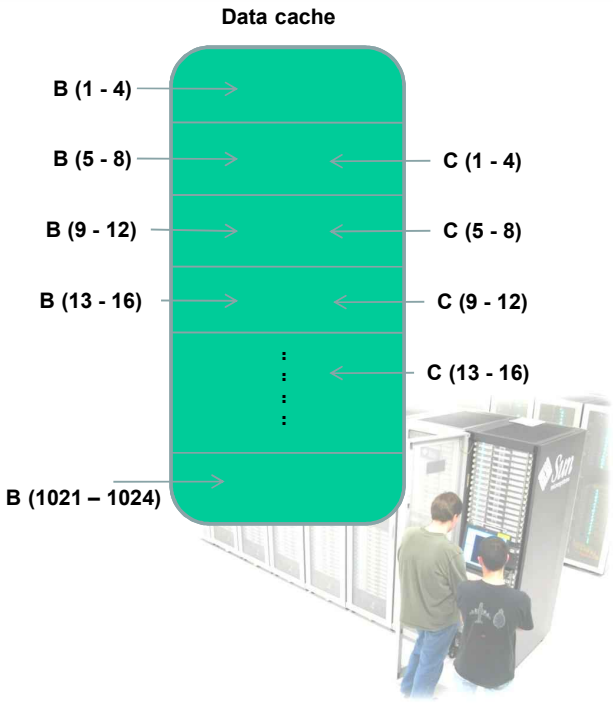
- **Common block padding**
  - 포트란 코드에서 사용하는 cache trashing 최소화를 위한 최적화 기법
  - 예제프로그램
    - Array c와 b가 같은 캐시라인에 mapping되게 됨으로 발생하는 cache trashing의 발생
    - Fake array-space(4)- 를 COMMON block 에 삽입하여 array b와 c의 캐시라인 mapping에서 발생하던 충돌(collide)을 막을 수 있다

Cache Thrashing



Avoiding Cache Thrashing





Exercise : Cache Thrashing

```
program trashing
implicit none
integer, parameter      :: N=4*1024*1024
integer, parameter      :: itermax=100
integer                 :: it,i
real   , dimension(N)   :: a,b,c
real   , dimension(4)   :: space
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
common/saver/a,b,c
!DIR$ cache_align /saver/
call random_number(b)
call random_number(c)

st = dtime(rtime)

do it=1,itermax
  do i=1,N
    a(i) = b(i) + c(i)
  end do
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*2.0*N)/et
write(*,100) et
write(*,200) flops
write(*,*) a(1),b(1),c(1)

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program trashing
```

Considerable cache trashing 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

Exercise : Cache Thrashing (Portland Group)

```
$pgf90 -O2 -Munroll -Minfo=loop trashing.f90 -o test.x
trashing:
    19, Loop unrolled 8 times

$ ./test.x
loop time =      4.82000 seconds
loops runs at       174.03751 MFLOPS
    1.611511       0.9079230       0.7035879
```

Exercise : Cache Thrashing (Intel)

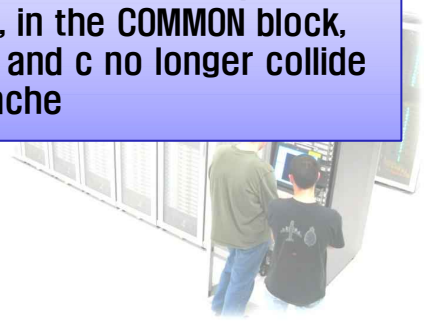
```
$ ifort -O2 -unroll trashing.f90 -o test.x
trashing.f90(19): (col. 3) remark: LOOP WAS VECTORIZED.

$ ./test.x
loop time =      1.08000 seconds
loops runs at       776.72290 MFLOPS
  0.7032483      3.9208680E-07  0.7032479
```

Exercise : COMMON Block Padding

```
program trashing
implicit none
integer, parameter      :: N=4*1024*1024
integer, parameter      :: itermax=100
integer                 :: it,i
real   , dimension(N)   :: a,b,c
real   , dimension(4)   :: space
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
common/saver/a,b,space(4),c
!DIR$ cache_align /saver/
call random_number(b)
call random_number(c)

st = dtime(rtime)

do it=1,itermax
  do i=1,N
    a(i) = b(i) + c(i)
  end do
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*2.0*N)/et
write(*,100) et
write(*,200) flops
write(*,*) a(1),b(1),c(1)

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program trashing
```

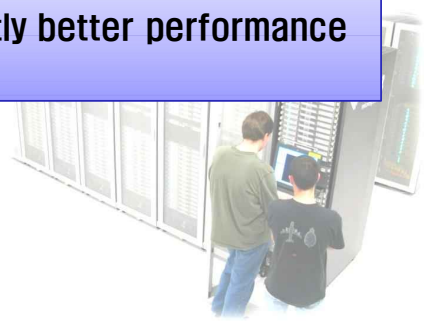
By putting the fake array, space(4), in the COMMON block, arrays b and c no longer collide in the cache



Exercise : COMMON Block Padding (Portland Group)

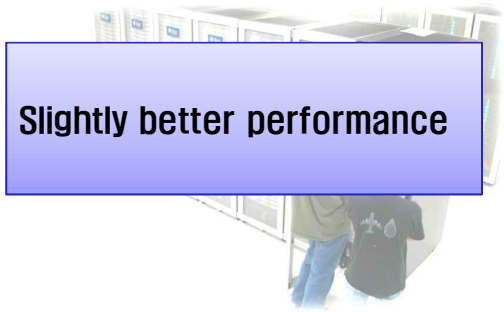
```
$pgf90 -O2 -Munroll -Minfo=loop trashing.f90 -o test.x
trashing:
    19, Loop unrolled 8 times

$ ./test.x
loop time =      4.58000 seconds
loops runs at       183.15738 MFLOPS
    1.611511       0.9079230       0.7035879
```

Slightly better performance



Exercise : COMMON Block Padding (Intel)

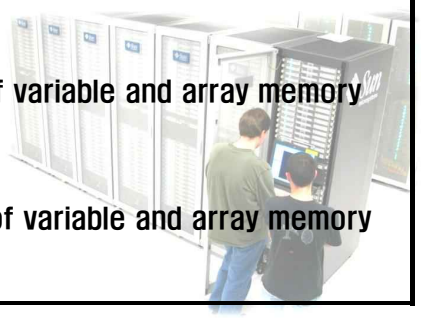
```
$ ifort -O2 -unroll trashing.f90 -o test.x
trashing.f90(19): (col. 3) remark: LOOP WAS VECTORIZED.

$ ./test.x
loop time =      1.06000 seconds
loops runs at       791.37811 MFLOPS
  0.7032483      3.9208680E-07  0.7032479
```



Slightly better performance

COMMON Block Padding 컴파일러 옵션

GNU compilers	
-malign-double	Align double precision variables on 64-bit boundaries, including in COMMON blocks
Portland Group 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



Dimension Extension

- 같은 캐시라인에 사상되는 배열 변수의 충돌을 막기 위해 “face row” 를 넣는 방법
- Common-block padding 과 흡사하며, 변수크기의 재설정을 통한 접근 방법
- 계산 결과에 대한 확인이 반드시 필요

Exercise : Column Cache Thrashing

```
program trashing2
implicit none
integer, parameter      ::
N=1024,M=64,itermax=10000
integer                 :: it,i,j
real   , dimension(N,M) :: a,b
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
common/saver/a,b
!DIR$ cache_align /saver/

call random_number(b)

st = dtime(rtime)

do it=1,itermax
  do j=1,M-1
  do i=1,N
    a(i,j) = b(i,j) - b(i,j+1)
  end do
  end do
  !write(*,*) a(1,1),b(1,1)
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*N*(M-1))/et
write(*,100) et
write(*,200) flops

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program trashing2
```

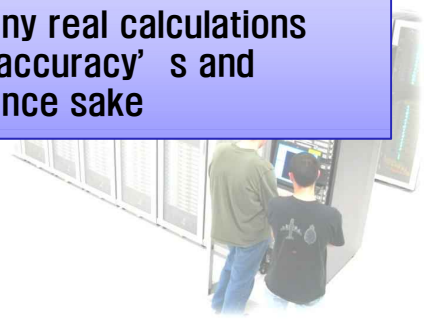
Columns on array b exactly fill the primary data cache and thus when different columns are subtracted excessive data thrashing occurs

Exercise : Dimension Extension

```
program trashing2
implicit none
integer, parameter      ::
N=1024,M=64,itermax=10000
integer                 :: it,i,j
real   , dimension(N+4,M) :: a,b
real                    :: flops
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
common/saver/a,b
!DIR$ cache_align /saver/

call random_number(b)

st = dtime(rtime)

do it=1,itermax
  do j=1,M-1
  do i=1,N
    a(i,j) = b(i,j) - b(i,j+1)
  end do
  end do
  !write(*,*) a(1,1),b(1,1)
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*N*(M-1))/et
write(*,100) et
write(*,200) flops

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")

end program trashing2
```

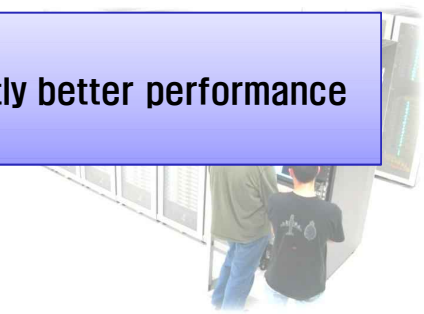
With this technique care should be taken not to use the fake rows in any real calculations both for accuracy' s and performance sake



Exercise : Column Cache Thrashing (Portland Group)

```
$pgf90 -O2 -Minfo=loop trashing2.f90 -o test.x

$ ./test.x
loop time =      1.23000 seconds
loops runs at      524.48779 MFLOPS

Modified Results

./test.x
loop time =      1.16000 seconds
loops runs at      556.13794 MFLOPS
```

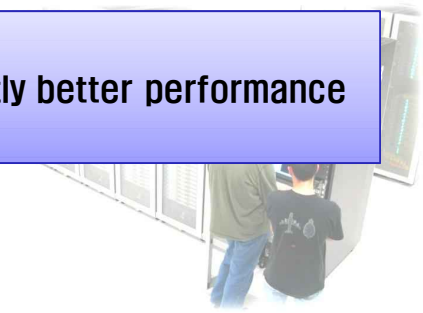
Slightly better performance



Exercise : Column Cache Thrashing (Intel)

```
$ifort -O2 trashing2.f90 -o test.x

$ ./test.x
loop time =      0.65000 seconds
loops runs at       992.49231 MFLOPS

Modified Results

./test.x
loop time =      0.55000 seconds
loops runs at      1172.94543 MFLOPS
```

Slightly better performance



Compiler-Initiated Prefetching

- Pentium 4와 같은 architecture들은 시스템이 prefetching을 지원
  - 프로세서가 메인 메모리로부터 다음 단계의 캐시(L2 또는 상위)로의 fetch traffic을 항시 점검하고 있음
  - 필요에 따라 prefetching을 시스템이 결정
- 컴파일시의 prefetching
  - 성능의 저하를 가져올 수도 있음
  - 컴파일 옵션에 따른 결과에 대한 점검이 필요

Prefetching Compiler Options

GNU compilers `-fprefetch-array-loops`	Enable prefetching, if supported.
Portland Group compilers `-Mvect=prefetch`	Enable prefetching
Intel compilers `-O3`	Enable aggressive optimization, including prefetching

Exercise : Prefetching

```
program prefetching
implicit none
integer, parameter      :: N=5000,
itermax=200000
integer                 :: it,i
real   , dimension(N)   :: a,b
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
real                    :: flops
common /saver/ a,b
call random_number(a)
call random_number(b)

st = dtime(rtime)

do it=1,itermax
  do i=1,N
    b(i) = a(i)+i
  end do
  if(mod(it,10000).eq.0) write(*,*) a(1),b(1)
end do
```

```
et = dtime(rtime)
flops = (1.0E-6*itermax*N)/et
write(*,100) et
write(*,200) flops

100 format("loop time = ",f12.5," seconds")
200 format("loops runs at ",f15.5," MFLOPS")
end program prefetching
```

Exercise : Prefetching (Portland Group)

```
$pgf90 -O2 -Munroll -Minfo=loop prefetching.f90 -o test.x
prefetching:
     16, Loop unrolled 4 times
$ ./test.x
    ...
loop time =      1.88000 seconds
loops runs at       531.91492 MFLOPS

$pgf90 -O2 -Munroll -Mvect=prefetch -Minfo=loop prefetching.f90 -o test.x
prefetching:
     16, Unrolled inner loop 8 times
         Generated 1 prefetch instructions for this loop
$ ./test.x
    ......
loop time =      1.97000 seconds
loops runs at       507.61423 MFLOPS
```

poorer performance

Exercise : Prefetching (Intel)

```
$ ifort -O2 -unroll prefetching.f90 -o test.x
prefetching.f90(16): (col. 3) remark: LOOP WAS VECTORIZED.

$ ./test.x
    ...
loop time =      0.39000 seconds
loops runs at      2564.10254 MFLOPS

$ ifort -O2 -prefetch -unroll prefetching.f90 -o test.x
prefetching.f90(16): (col. 3) remark: LOOP WAS VECTORIZED.

$ ./test.x
    ......
loop time =      0.39000 seconds
loops runs at      2564.10254 MFLOPS
```

Same as no W/O prefetch

부동소수 연산 특성 (Floating Point Behavior)

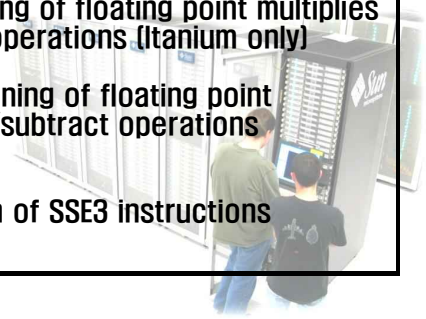
- 명령어집합의 확장 (Instruction set extensions)
- 나누기 (Division)
- 예외 사항(exception)의 처리

부동소수 연산 집합의 확장 (Floating Point Instruction Set Extensions)

- By default, most compilers will be fairly conservative about the floating point instructions they generate.
- In particular, they will typically not generate code that makes use of extended FP instruction sets (such as SSE2 or SSE3) without being told to do so.

SSE(Streaming SIMD Extensions) 사용 컴파일러 옵션

GNU compilers	
-mfpmath=sse	Use SSE instructions to perform floating point arithmetic where possible
Portland Group compilers	
-Mvect=sse	Use SSE instructions to implement vect or loops where possible
-fastsse	Use SSE instructions to implement vect or loops where possible
Intel compilers	
-IPF_fma	Enable the combining of floating point multiplies and add/subtract operations (Itanium only)
-IPF_fma-	Disables the combining of floating point multiplies and add/subtract operations (Itanium only)
-xT	Enables generation of SSE3 instructions (x86_64 only)



Floating Point Division Compiler Options

GNU compilers

None

Portland Group compilers

-Kieee — Require strict IEEE-754 compliance

-Knoieee — Use inline division and disable traps on underflow; default

Intel compilers

-mp — Maintain floating point precision by not transforming division into multiplication by a reciprocal

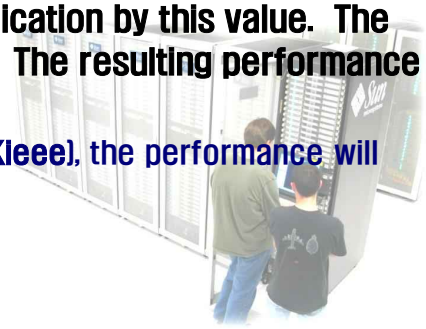
-mp1 — Improve floating point precision while still allowing some rearrangement of arithmetic operations

Floating Point Division Example

- Consider the following code

```
x=2.5
do i=1,N
    a(i)=a(i)/x
enddo
```

- On a 2.6GHz Opteron using the Portland Group pgf90compiler, the compiler will automatically compute (1.0/2.5) once outside the loop and replace the division in the loop with a multiplication by this value. The new multiplication loop will then be unrolled. The resulting performance will be 1151 MFLOPS.
  - If the user forces IEEE conformance (**pgf90 –Kieee**), the performance will be reduced to 310 MFLOPS.

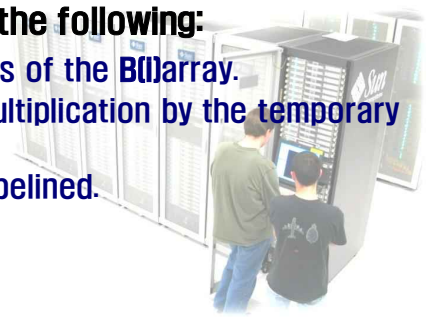


Floating Point Division With Arrays

- 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.

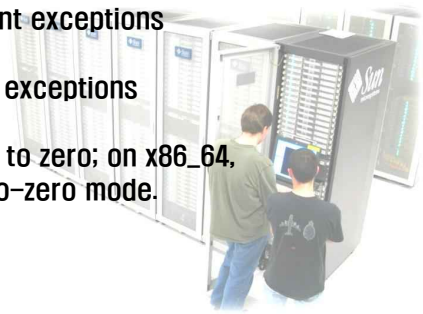


Floating Point Exception Handling

- The IEEE-754 floating point standard defines several exception conditions:
  - Overflow
  - Underflow
  - Divide by zero
  - Invalid operand
- By default, programs generated by most compilers will catch FP exceptions automatically and keep running; however, there are typically flags available to force a program to crash when an FP exception is encountered.
- FP exceptions are expensive on Itanium, as they are reported to the kernel and logged via syslog.
  - This is especially a problem for codes that assume that FP underflows are ignored and rounded to zero, which is usually the case on most other platforms.

Floating Point Division Compiler Options

GNU compilers	None
Portland Group compilers	
-Mflushz	Set SSE to flush-to-zero mode.
-Ktrap=fp	Abort on invalid operand, divide by zero, and overflow
Intel compilers	
-fpe0	Abort on invalid operand, divide by zero, and overflow
-fpe1	Abort on all floating point exceptions
-fpe3	Ignore all floating point exceptions
-ftz	Flush underflow results to zero; on x86_64, also set SSE to flush-to-zero mode.



Optimized Math Libraries

- By default the algorithms used to calculate intrinsic math functions ("calculator functions") are scalar in nature. On the other hand, loops which use some intrinsic function calls on array elements used can be "vectorized" so that the functions calls can be optimized. The vectorization of the loop occurs in several steps.
  - First the intrinsic calls are separated into their own loop where the results are assigned to a temporary array and
  - an optimized algorithm is used to make use of the pipelined functional units.
  - The resulting temporary array is then used in the original loop instead of the actual function calls.

- The functions which can be vectorized in this manner include log, alog, exp, pow, **, sin, cos, sqrt, and1.0/sqrt.
- Manufacturers of high performance computers also typically supply highly optimized versions of commonly used numerical libraries such as BLAS and LAPACK.

Exercise : Optimized LAPACK & BLAS

```
program uselapack
implicit none
integer, parameter      :: N=1000,
itermax=10
integer                 :: it,info
real   , dimension(N,N) :: a
real   , dimension(N)   :: b,c
integer, dimension(N)   :: pivots
real                    :: st,et,dtime
real   , dimension(2)   :: rtime
real                    :: flops

call random_number(a)
call random_number(b)
c = b

st = dtime(rtime)

do it=1,itermax
  call sgesv(N,1,a,N,pivots,b,N,info)
  if(mod(it,itermax/10).eq.0) write(*,*) b(1),b(N)
  b = c
end do
```

```
et = dtime(rtime)
write(*,100) et

100 format("iteration time = ",f12.5,"
seconds")
end program uselapack
```

Exercise : Optimized LAPACK & BLAS (Portland Group)

```
$pgf90 -O2 uselapack.f90 -o test.x -llapack -lblas
uselapack.f90:
$ ./test.x
    ......
iteration time =      2.60000 seconds


$ pgf90 -O2 uselapack.f90 -L/applic/lib.pgi/ACML/ACML.4.0.1/pgi64/lib -Bstatic -lacml -o
test.x
uselapack.f90:
$ ./test.x
    ........
iteration time =      1.01000 seconds
```

Exercise : Optimized LAPACK & BLAS (Intel)

```
$ ifort -O2 uselapack.f90 -o -L/applic/lib.intel/LAPACK -Bstatic test.x -llapack -lblas

$ ./test.x
    ........
iteration time =      2.23000 seconds

$ ifort -O2 uselapack.f90 -L/applic/lib.intel/ACML/ifort64/lib -Bstatic -lacml -o test.x
uselapack.f90:

$ ./test.x
    ........
iteration time =      1.04000 seconds
```