2068-19

Advanced School in High Performance and GRID Computing - Concepts and Applications

30 November - 11 December, 2009

Using Compilers and Profilers to Optimize your Code for Performance (part 1)

S.T. Brown

Carnegie Mellon University
Pittsburgh
USA

Optimization and Profiling

Shawn T. Brown
Senior Scientific Specialist
Pittsburgh Supercomputing Center
stbrown@psc.edu

Philosophy...

- Real processors have
  - registers, cache, parallelism, ... they are bloody complicated
- Why is this your problem?
  - In theory, compilers understand all of this and can optimize your code; in practice they don't.
  - Generally optimizing algorithms across all computational architectures is an impossible task, hand optimization will always be needed.
- We need to learn how...
  - to measure performance of codes on modern architectures
  - to tune performance of the codes by hand (32/64 bit commodity processors)

Philosophy...

- When you are charged with optimizing an application...
  - Don't optimize the whole code
    - Profile the code, find the bottlenecks
    - They may not always be where you thought they were
  - Break the problem down
    - Try to run the shortest possible test you can to get meaningful results
    - Isolate serial kernels
  - Keep a working version of the code!
    - Getting the wrong answer faster is not the goal.
  - Optimize on the architecture on which you intend to run
    - Optimizations for one architecture will not necessarily translate
  - The compiler is your friend!
    - If you find yourself coding in machine language, you are doing to much.

Performance

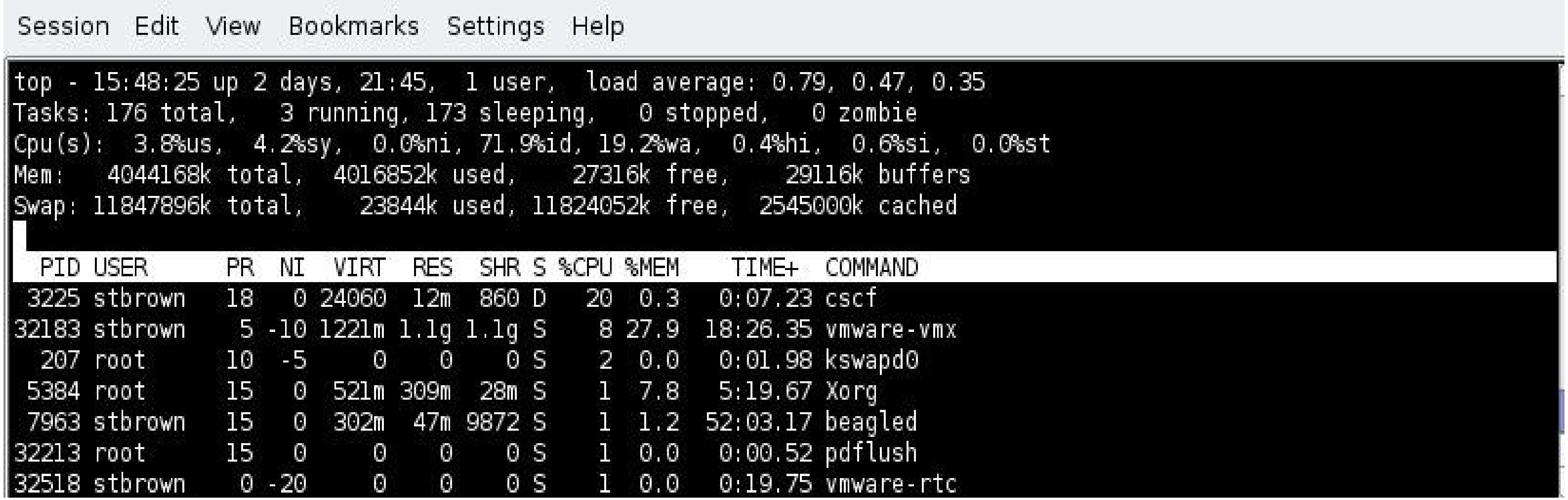
- The peak performance of a chip
  - The number of theoretical floating point operations per second
    - e.g. 2.4 Ghz Operon can theoretically do 2 fops per cycle, for a peak performance of 4.8 Gflops
- Real performance
  - Algorithm dependent, the actually number of floating point operations per second
    - Generally, most programs get about 10% or lower of peak performance
    - 40% of peak, and you can go on holiday
- Parallel performance
  - The scaling of an algorithm relative to its speed on 1 processor
    - more tomorrow!

Performance Evaluation process

- Monitoring System
  - Observe both overall system performance and single-program execution characteristics.
    - Look to see if the system is doing well and what percentage of the resources your program is using.
    - Pro: easy   Con: not very detailed
- Profiling and Timing the code
  - Timing a whole programs (time command :/usr/bin/time)
  - Timing portions of the program (code modification)
  - Profiling

Useful Monitoring Commands (Linux)

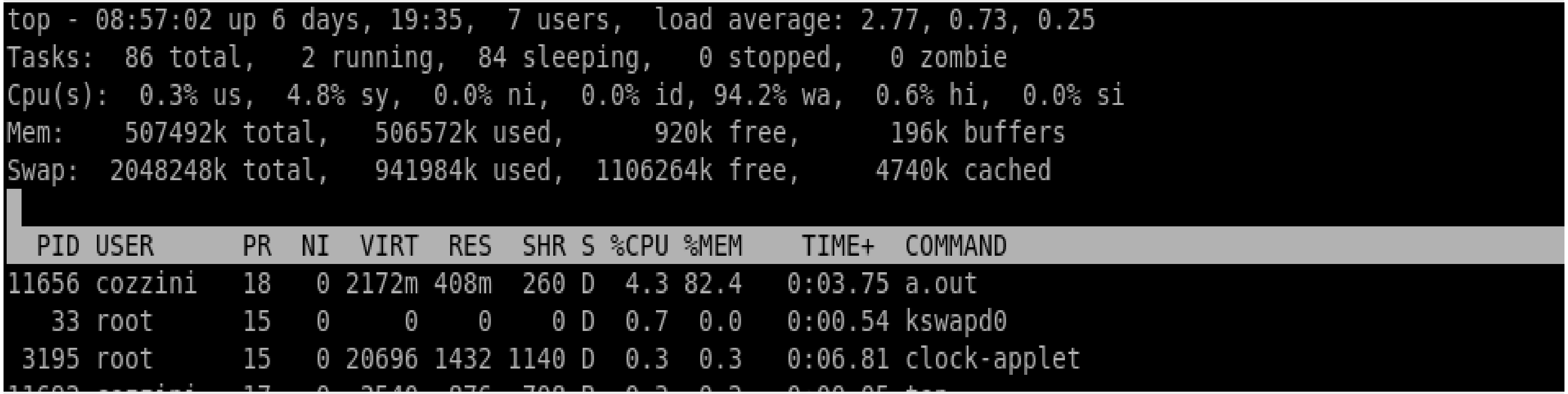
Uptime	returns information about system usage and user load
ps(1)	lets you see a “ snapshot” of the process table
top	process table dynamic display
free	memory usage
vmstat	memory usage monitor



```
top - 15:48:25 up 2 days, 21:45,  1 user,  load average: 0.79, 0.47, 0.35
Tasks: 176 total,   3 running, 173 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.8%us,  4.2%sy,  0.0%ni, 71.9%id, 19.2%wa,  0.4%hi,  0.6%si,  0.0%st
Mem:   4044168k total,  4016852k used,    27316k free,    29116k buffers
Swap: 11847896k total,    23844k used, 11824052k free,  2545000k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3225 stbrown   18   0 24060  12m  860 D   20  0.3   0:07.23 cscf
32183 stbrown    5 -10 1221m 1.1g 1.1g S    8 27.9  18:26.35 vmware-vmx
  207 root      10  -5     0    0    0 S    2  0.0   0:01.98 kswapd0
 5384 root      15   0  521m 309m  28m S    1  7.8   5:19.67 Xorg
 7963 stbrown   15   0  302m  47m 9872 S    1  1.2  52:03.17 beagled
32213 root      15   0     0    0    0 S    1  0.0   0:00.52 pdflush
32518 stbrown    0 -20     0    0    0 S    1  0.0   0:19.75 vmware-rtc
```

Swapping... A top disaster

- virtual or swap memory:
  - This memory, is actually space on the hard drive. The operatingsystem reserves a space on the hard drive for “ swap space” .
- time to access virtual memory VERY large:
- this time is done by the system not by your program !



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

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

Monitoring your own code (time)

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

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

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

```
--------------->time ./a.out
[program output]
real 0m1.361s
user 0m0.770s
sys 0m0.590s
```

user time: Cputime dedicated to your program

sys time: time used by your program to execute system calls

real time: total time aka walltime

Timing A Portion of the Code

- Most programming languages provide a means to access the systems own timing functions

- C function: clock

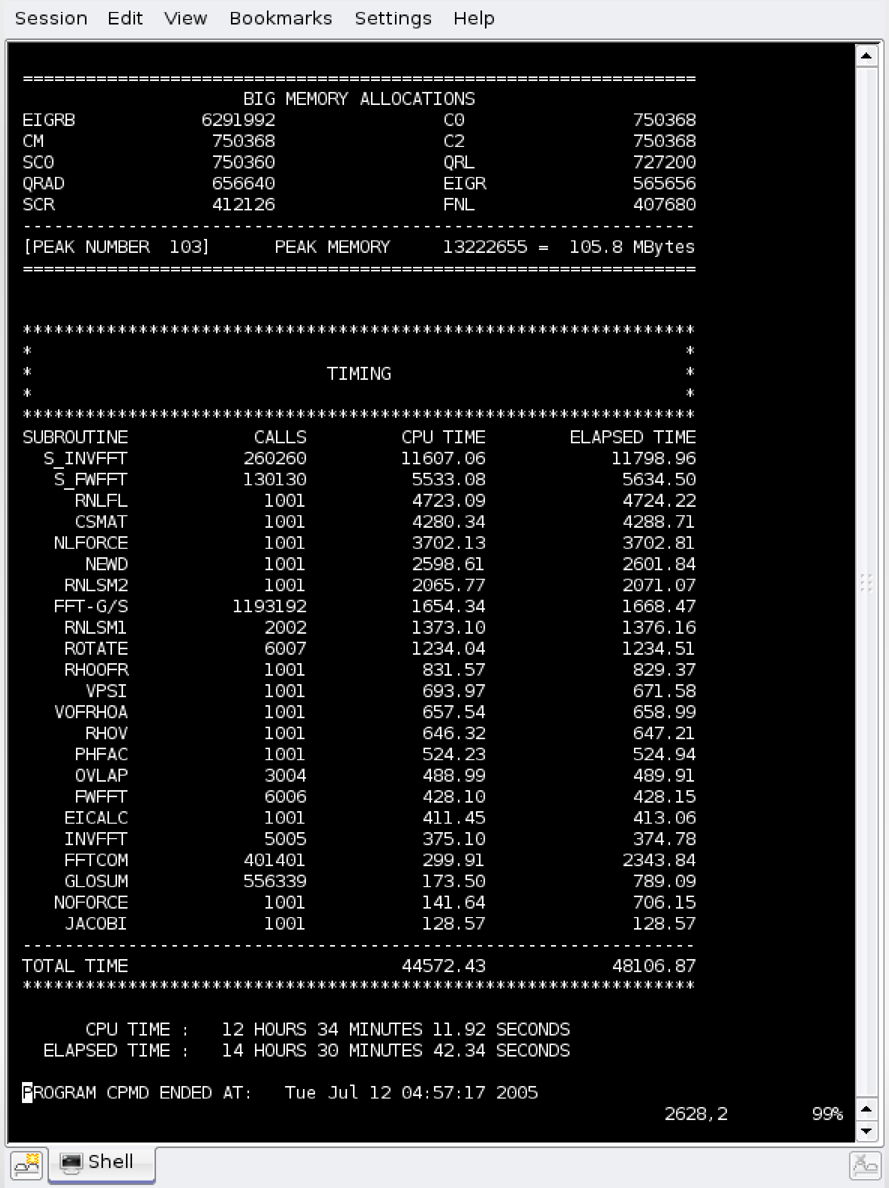
```
clock_t c0, c1;
c0 = clock();
      section to code..
c1= clock();
cputime = (c1 - c0)/(CLOCKS_PER_SEC );
```

- Fortran Subroutine: cpu_time

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

It is good practice….

- Good application writers will take full advantage of these to give users insight into code performance.



```
================================================================
                     BIG MEMORY ALLOCATIONS
EIGRB            6291992                C0                750368
CM                750368                C2                750368
SC0               750360                QRL               727200
QRAD              656640                EIGR              565656
SCR               412126                FNL               407680
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
[PEAK NUMBER  103]      PEAK MEMORY     13222655 =  105.8 MBytes
================================================================


****************************************************************
*                                                              *
*                            TIMING                            *
*                                                              *
****************************************************************
SUBROUTINE            CALLS         CPU TIME        ELAPSED TIME
  S_INVFFT           260260         11607.06            11798.96
   S_FWFFT           130130          5533.08             5634.50
     RNLFL             1001          4723.09             4724.22
     CSMAT             1001          4280.34             4288.71
   NLFORCE             1001          3702.13             3702.81
      NEWD             1001          2598.61             2601.84
    RNLSM2             1001          2065.77             2071.07
   FFT-G/S          1193192          1654.34             1668.47
    RNLSM1             2002          1373.10             1376.16
    ROTATE             6007          1234.04             1234.51
    RHOOFR             1001           831.57              829.37
      VPSI             1001           693.97              671.58
   VOFRHOA             1001           657.54              658.99
      RHOV             1001           646.32              647.21
     PHFAC             1001           524.23              524.94
     OVLAP             3004           488.99              489.91
     FWFFT             6006           428.10              428.15
    EICALC             1001           411.45              413.06
    INVFFT             5005           375.10              374.78
    FFTCOM           401401           299.91             2343.84
    GLOSUM           556339           173.50              789.09
   NOFORCE             1001           141.64              706.15
    JACOBI             1001           128.57              128.57
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
TOTAL TIME                          44572.43            48106.87
****************************************************************

      CPU TIME :   12 HOURS 34 MINUTES 11.92 SECONDS
  ELAPSED TIME :   14 HOURS 30 MINUTES 42.34 SECONDS

PROGRAM CPMD ENDED AT:   Tue Jul 12 04:57:17 2005
                                                             2628,2        99%
```

Profiling

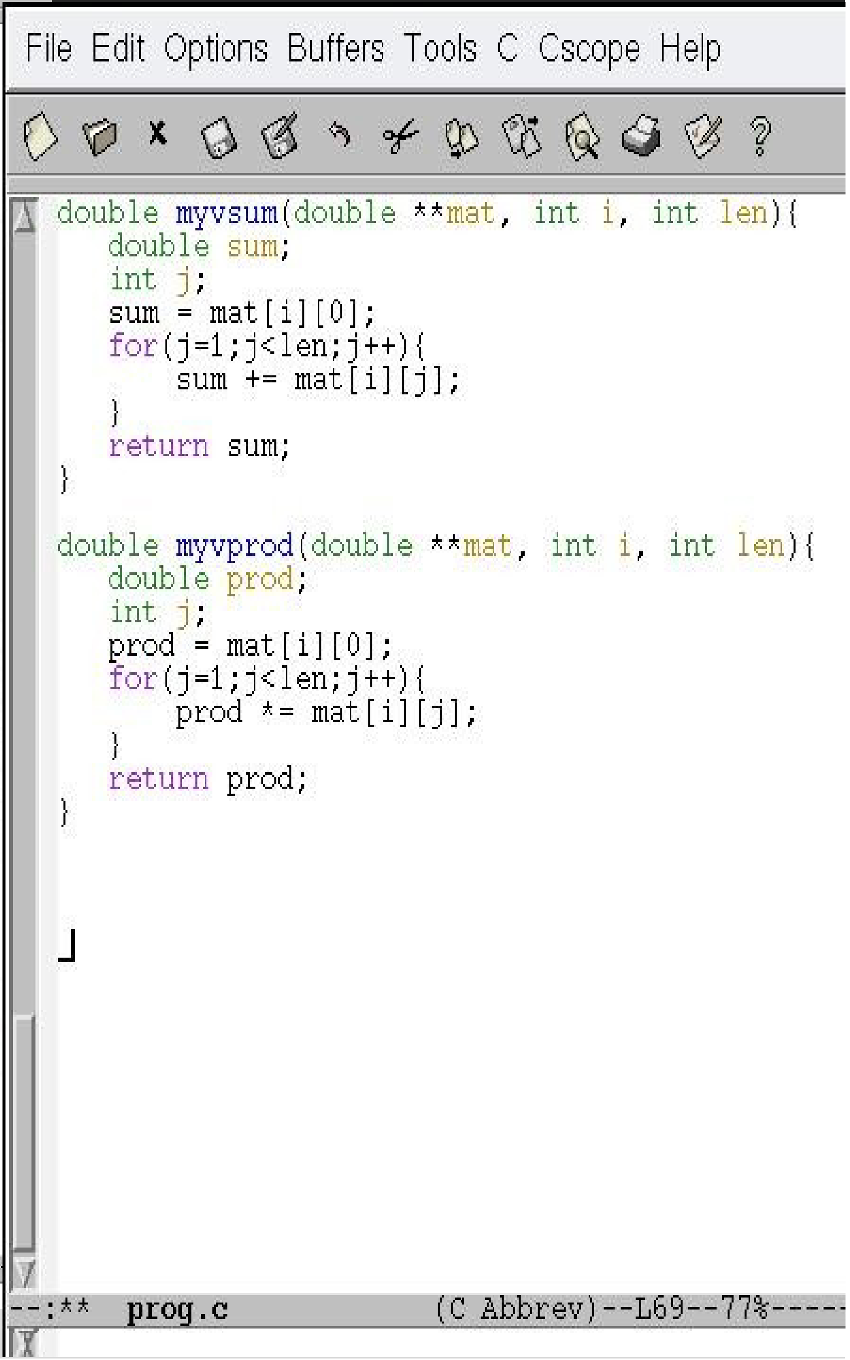
- Profiling is an approach to performance analysis in which the amount of time spent in sections of code is measured (using either a sampling technique or on entry/exit of a code block) and presented as a histogram.
- Allows a developer to target key time consuming portions of codes.
- Profiling can be done at varied levels of granularity
  - Subroutine, code block, loop and source code line

GCC profiling and gprof

- Simple gcc compiler flags can be used to get profiling information.
  - Great place to start
- GNU:
  - -p Generate extra code to write profile information suitable for analysis program prof
  - -pg Generate extra code to write profile information suitable for analysis by program gprof.
- Procedure
  - gcc -pg prog.c -o prog
  - ./prog
  - gprof prog.c gmon.out

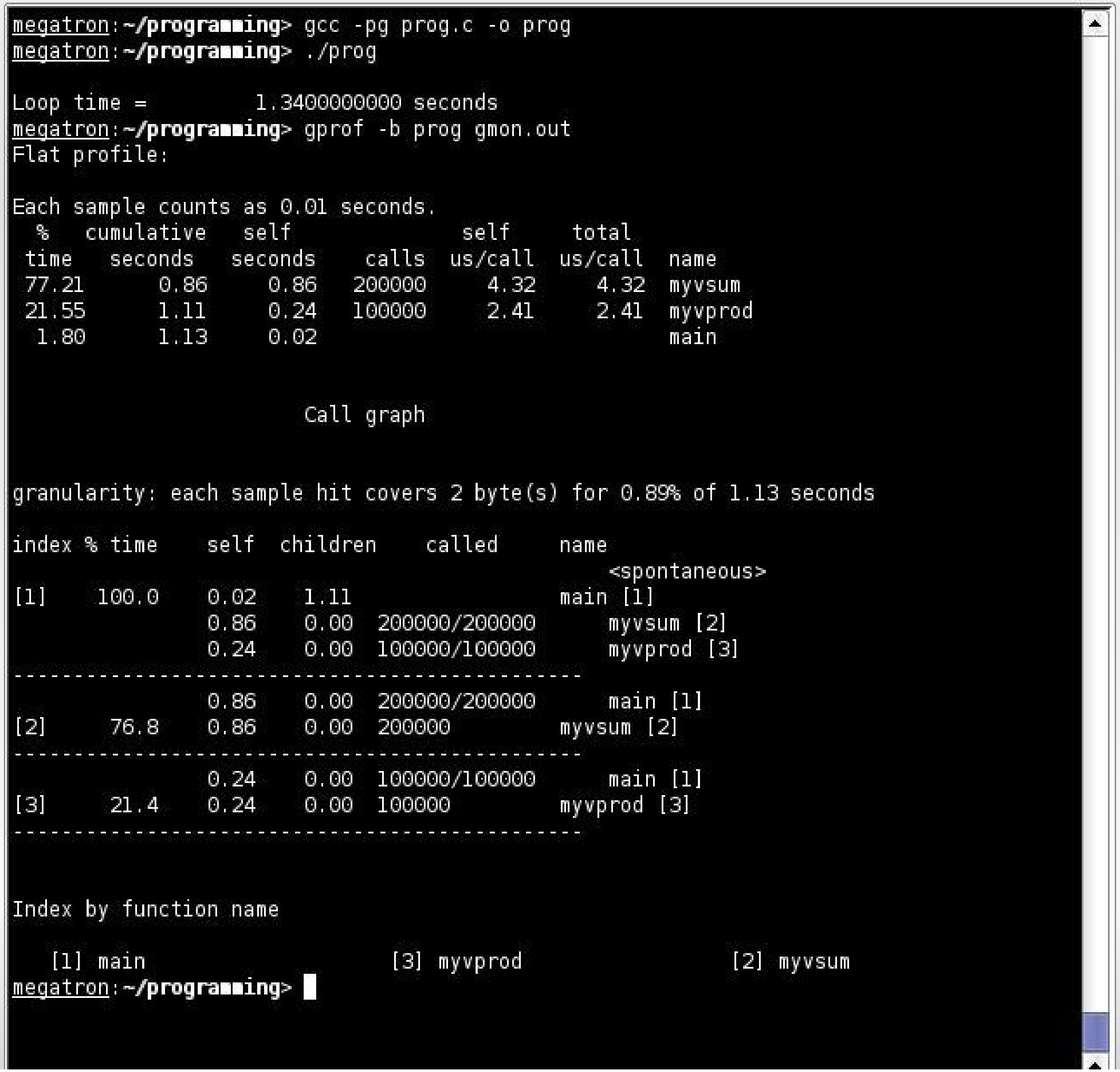
Example

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
double myvsum(double **mat, int i, int len);
double myvprod(double **mat, int i, int len);

int main (void){
   double **b,**c,**d;
   double *a;
   double begin, end;
   double flops;
   int i,j;
   int N = 1000;
   int ntimes = 100;
   b = (double **)malloc(N*sizeof(double*));
   for (i=0;i<N;i++)
     b[i] = (double *)malloc(N*sizeof(double));
   c = (double **)malloc(N*sizeof(double*));
   for (i=0;i<N;i++)
     c[i] = (double *)malloc(N*sizeof(double));
   d = (double **)malloc(N*sizeof(double*));
   for (i=0;i<N;i++)
     d[i] = (double *)malloc(N*sizeof(double));
   a = (double*)malloc(N*sizeof(double));

   for (i=0;i<N;i++){
     for(j=0;j<N;j++){
        b[i][j] = (double)(i+j);
        c[i][j] = (double)(i-j);
        d[i][j] = (double)(i);

      }
   }
   begin = clock();
   for(i=0;i<ntimes;i++){
     for(j=0;j<N;j++){
       a[j] = myvsum(b,j,N) + myvprod(c,j,N) + myvsum(d,j,N);
     }
   }
   end = clock();
   printf("\nLoop time = %20.10lf seconds\n",(end-begin)/(CLOCKS_PER_SEC));
   return 0;
}
```



```
double myvsum(double **mat, int i, int len){
   double sum;
   int j;
   sum = mat[i][0];
   for(j=1;j<len;j++){
       sum += mat[i][j];
   }
   return sum;
}

double myvprod(double **mat, int i, int len){
   double prod;
   int j;
   prod = mat[i][0];
   for(j=1;j<len;j++){
       prod *= mat[i][j];
   }
   return prod;
}
```

Example



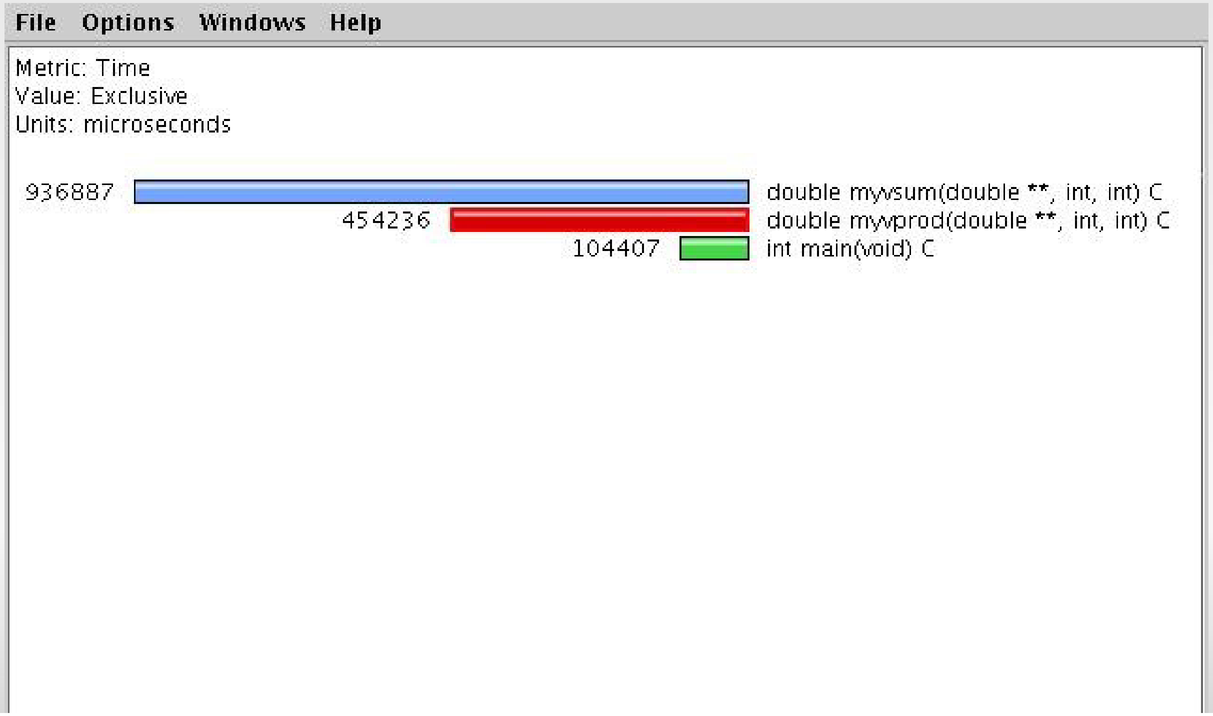
```
megatron:~/programming> gcc -pg prog.c -o prog
megatron:~/programming> ./prog

Loop time =         1.3400000000 seconds
megatron:~/programming> gprof -b prog gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 77.21      0.86     0.86   200000     4.32     4.32  myvsum
 21.55      1.11     0.24   100000     2.41     2.41  myvprod
  1.80      1.13     0.02                             main


                        Call graph


granularity: each sample hit covers 2 byte(s) for 0.89% of 1.13 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.02    1.11                 main [1]
                0.86    0.00  200000/200000      myvsum [2]
                0.24    0.00  100000/100000      myvprod [3]
-----------------------------------------------
                0.86    0.00  200000/200000      main [1]
[2]     76.8    0.86    0.00  200000         myvsum [2]
-----------------------------------------------
                0.24    0.00  100000/100000      main [1]
[3]     21.4    0.24    0.00  100000         myvprod [3]
-----------------------------------------------


Index by function name

   [1] main                   [3] myvprod                [2] myvsum
megatron:~/programming> 
```

Hardware Performance Counters

- Most modern processors have one or more registers dedicated to count low level hardware information
  - e.g. floating point operations, L1 cache misses, etc.
- This information is really useful to understand at a very fine grain of detail what a program is doing on the architecture.
- PAPI (Performance API)
  - The API provides function handles for setting and accessing these counters.
  - http://icl.cs.utk.edu/papi/

Tuning and Analysis Utilities

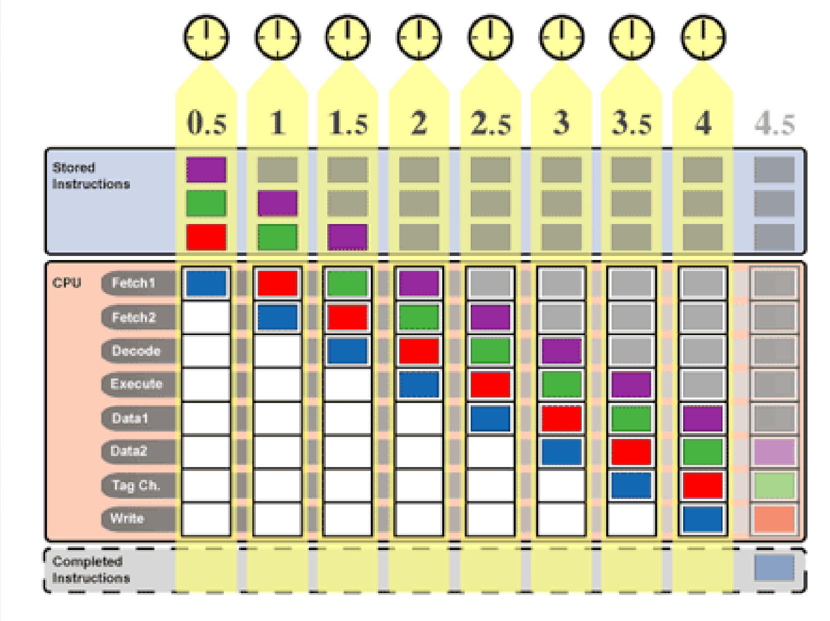
- TAU is a portable profiling and tracing toolkit for performance analysis of parallel programs.
- www.cs.uoregon.edu/research/tau/home.php



Metric: Time
Value: Exclusive
Units: microseconds

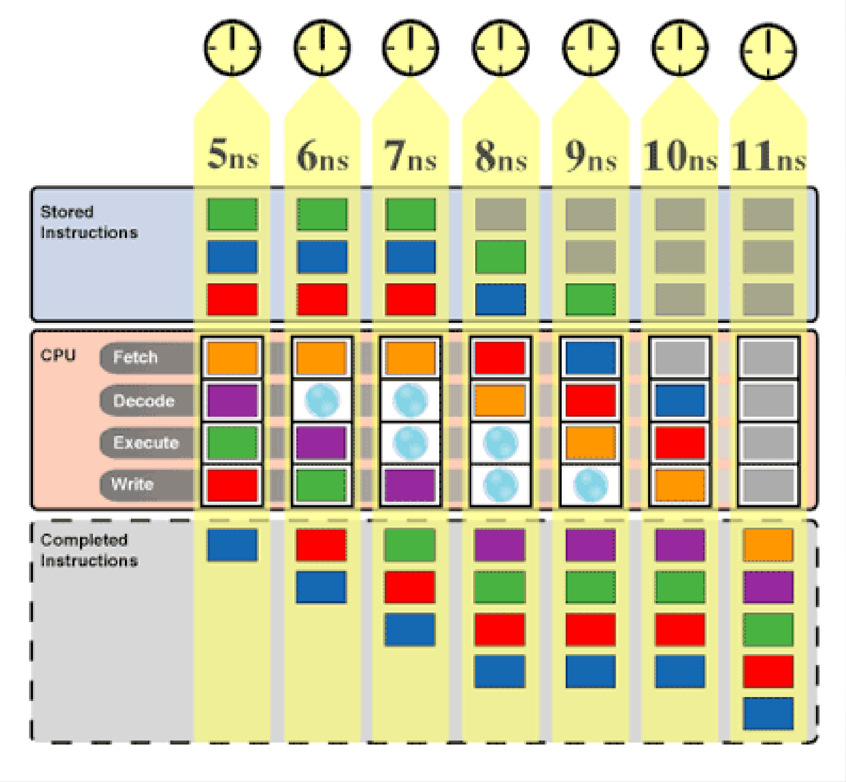
936887	double mywsum(double **, int, int) C
454236	double mywprod(double **, int, int) C
104407	int main(void) C

Pipelining



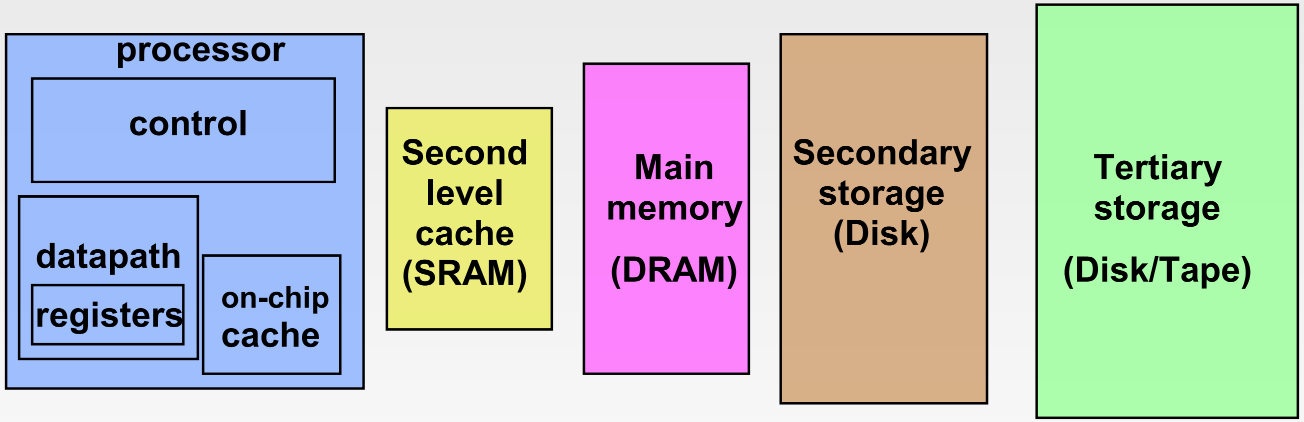
- Pipelining allows for a smooth progression of instructions and data to flow through the processor
- Any optimization that facilitate pipelining will speed the serial performance of your code.
- As chips support more SSE like character, filling the pipeline is more difficult.

- Stalling the pipeline slows codes down
  - Out of cache reads and writes
  - Conditional statements



Memory locality

- **Effective use of the memory heirarchy can facilitate good pipelining**
- **Temporal locality:**
  - Recently referenced items (instr or data) are likely to be referenced again in the near future
  - -iterative loops, subroutines, local variables
  - -working set concept
- **Spatial locality:**
  - programs access data which is near to each other:
  - operations on tables/arrays
  - cache line size is determined by spatial locality
- **Sequential locality:**
  - processor executes instructions in program order:
  - branches/in-sequence ratio is typically 1 to 5


processor

control

datapath

registers

on-chip cache

Second level cache (SRAM)

Main memory (DRAM)

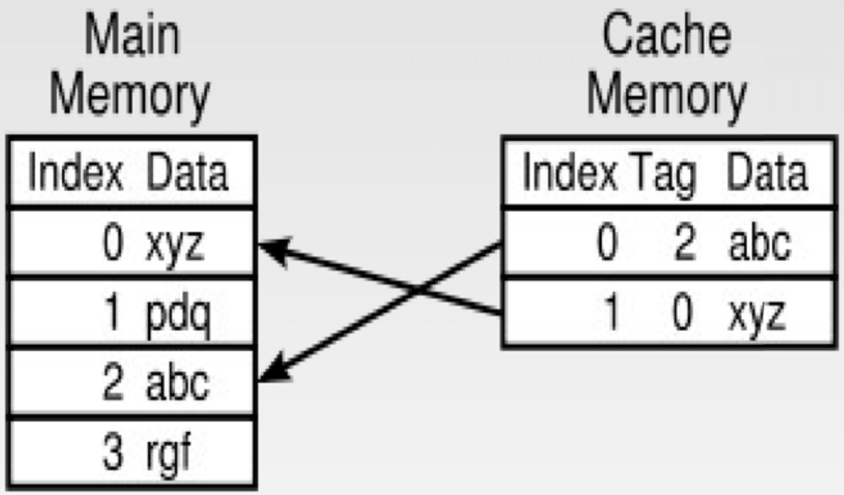
Secondary storage (Disk)

Tertiary storage (Disk/Tape)

Speed	1ns	10ns	100ns	10ms	10sec
Size	B	KB	MB	GB	TB

Caching

- CPU cache is generally set up as a series of lines that can pull in a specified amount of data a given time.
- Accessing Cache infinitely faster than main memory
  - Get as much data in at a time
  - Use that data to its fullest!



Main Memory

Index	Data
0	xyz
1	pdq
2	abc
3	rgf

Cache Memory

Index	Tag	Data
0	2	abc
1	0	xyz

Optimization Methodology

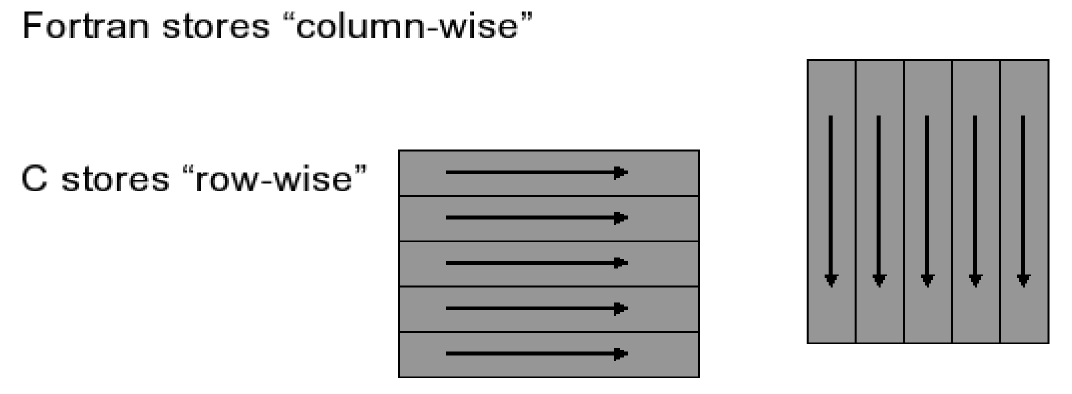
- So I profiled my code... found bottle necks...
- Optimize one loop/routine at a time
- Start with the most time consuming routines (that is why we profile)
- Then the second and the third most...
- Parallelize your program..
  - Then work on parallel performance (communication, load balancing, etc..)

Optimization Techniques

- There are basically two different categories:
  - Improve memory performance (taking advantage of locality)
    - Better memory access patterns
    - Optimal usage of cache lines
    - Re-use of cached data
  - Improve CPU performance
    - Reduce flop count
    - Better instruction scheduling
    - Use optimal instruction set

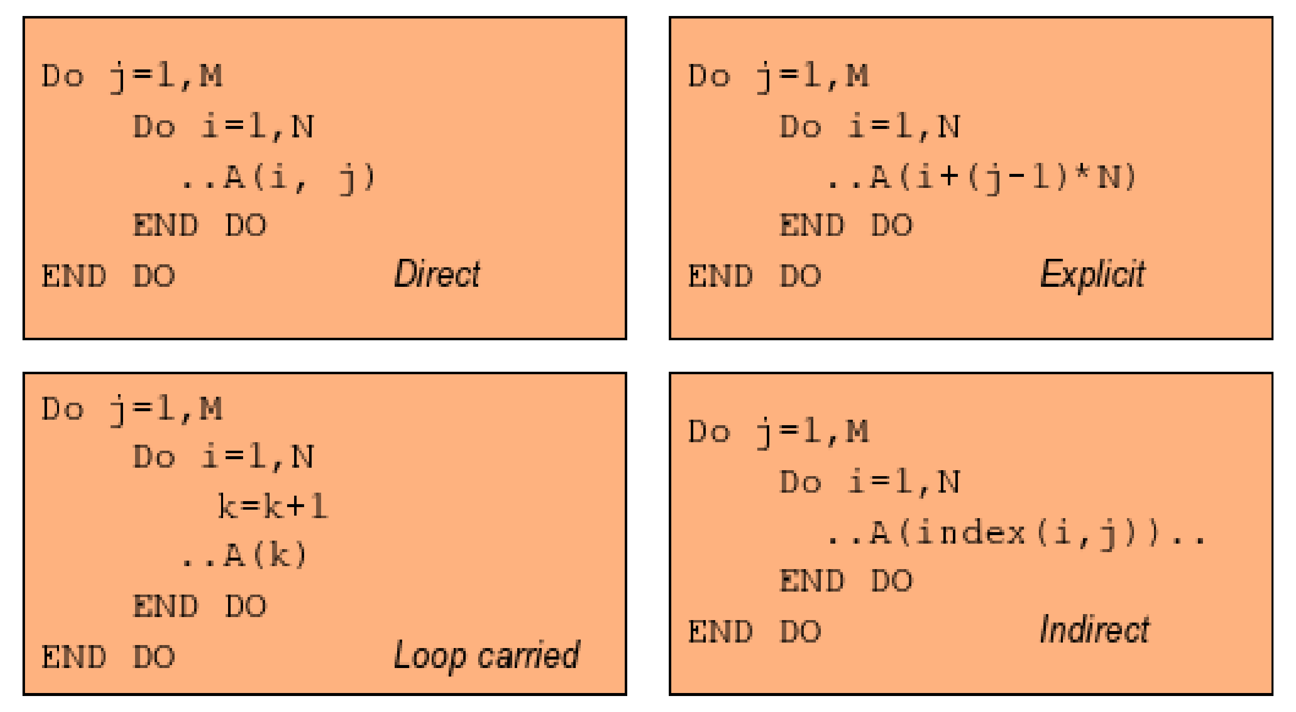
Optimization Techniques for Memory

- Stride
    - contiquous blocks of memory
- Accessing memory in stride greatly enhances the performance



Array indexing

- Ther are several ways to index arrays:



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

Direct

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

Explicit

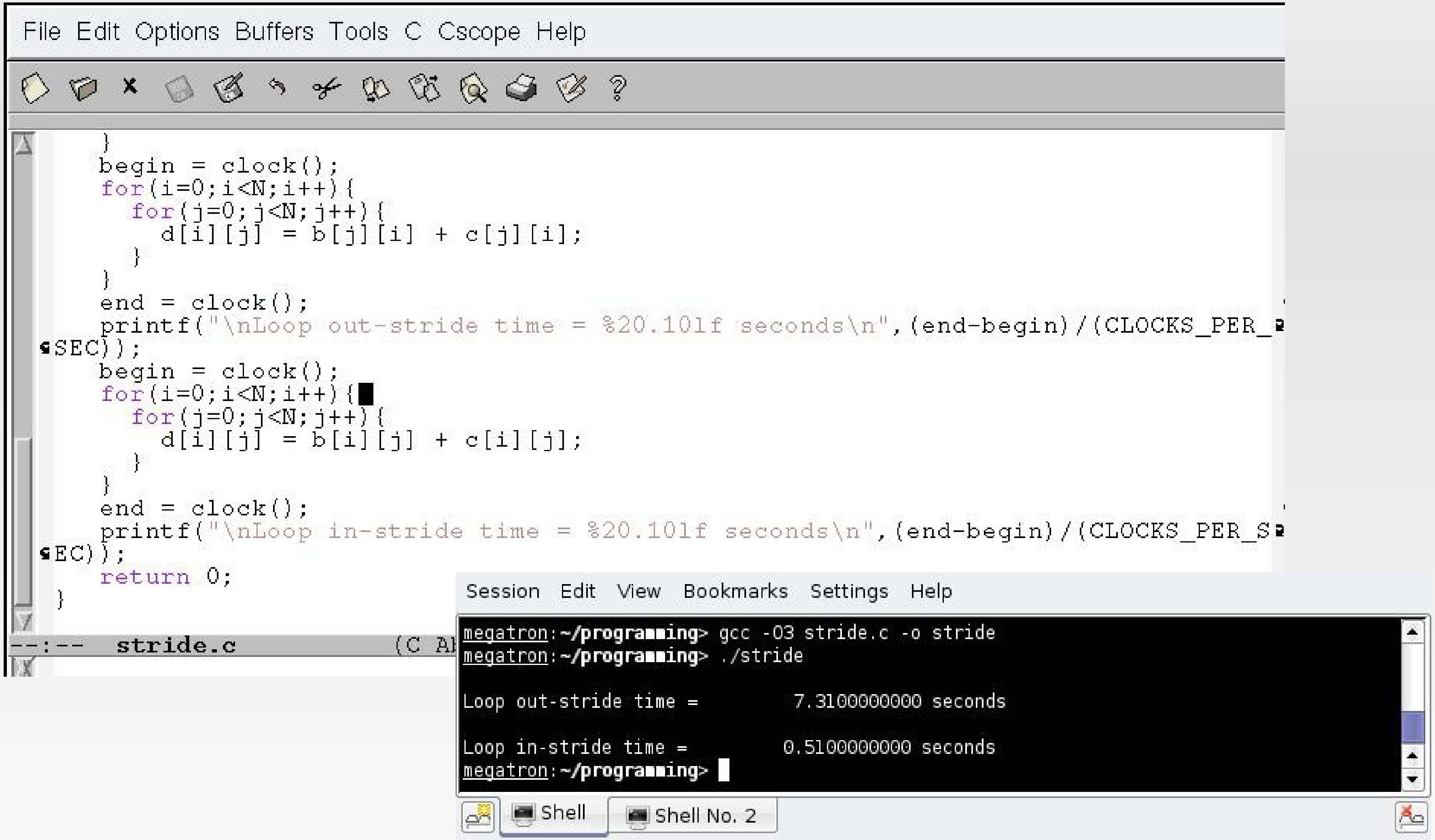
```
Do j=1,M
    Do i=1,N
        k=k+1
      ..A(k)
    END DO
END DO
```

Loop carried

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

Indirect

Example (stride)

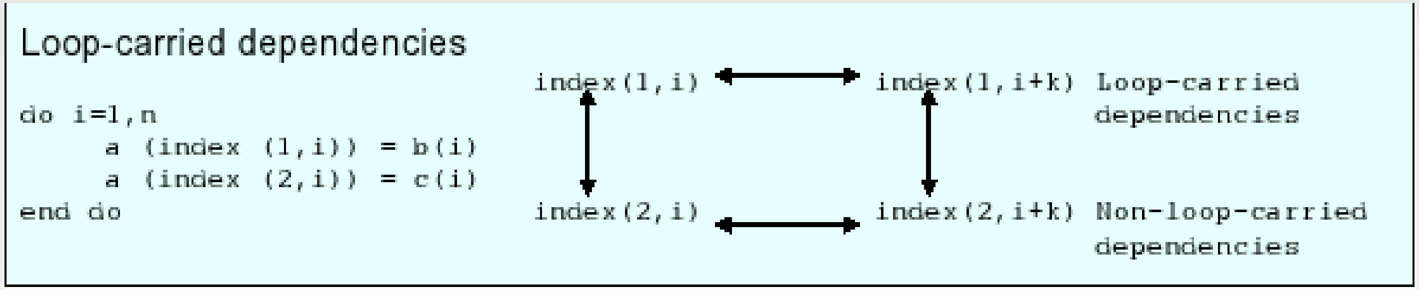


```c
    }
    begin = clock();
    for(i=0;i<N;i++){
      for(j=0;j<N;j++){
        d[i][j] = b[j][i] + c[j][i];
      }
    }
    end = clock();
    printf("\nLoop out-stride time = %20.10lf seconds\n",(end-begin)/(CLOCKS_PER_
SEC));
    begin = clock();
    for(i=0;i<N;i++){
      for(j=0;j<N;j++){
        d[i][j] = b[i][j] + c[i][j];
      }
    }
    end = clock();
    printf("\nLoop in-stride time = %20.10lf seconds\n",(end-begin)/(CLOCKS_PER_S
EC));
    return 0;
  }
```

```
megatron:~/programming> gcc -O3 stride.c -o stride
megatron:~/programming> ./stride

Loop out-stride time =         7.3100000000 seconds

Loop in-stride time =          0.5100000000 seconds
megatron:~/programming> 
```

Data Dependencies

- In order to perform hand optimization, you really need to get a handle on the data dependencies of your loops.
    - Operations that do not share data dependencies can be performed in tandum.



Loop-carried dependencies

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

        - Automatically determining data dependencies is tough for the compiler.
        - great opportunity for hand optimization

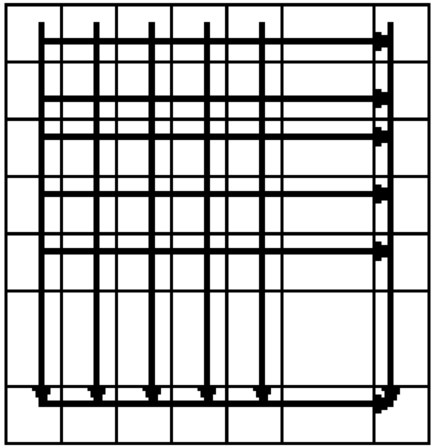
Loop Interchange

- Basic idea: change the order of data independent nested loops.
- *Advantages:*
  - Better memory access patterns (leading to improved cache and memory usage)
  - Elimination of data dependencies (to increase opportunity for CPU optimization and parallelization
- *Disadvantage:*
  - Make make a short loop innermost

Loop Interchange – Example 1

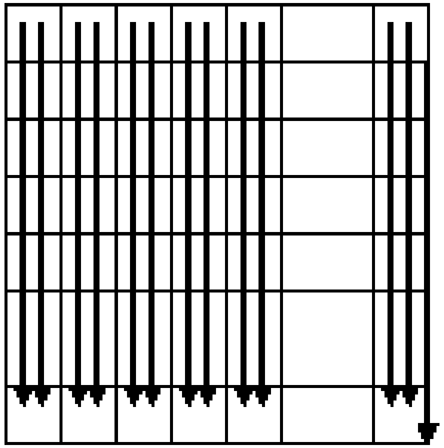
Original

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



Interchanged loops

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



Access order
Storage order

Loop Interchange in C/C++

In C, the situation is exactly the opposite

interchange

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

index reversal

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

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

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

Loop Interchange – Example 2

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

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

Compiler Loop Interchange

- GNU compilers: No support
- PGI compilers:
  - -Mvect Enable vectorization, including loop interchange
- Intel compilers:
  - -O3 Enable aggressive optimization, including loop transformations

CAUTION: Make sure that your program still works after this!

Loop Unrolling

- Computation cheap... branching expensive
  - Loops, conditionals, etc. Cause branching instructions to be performed.
  - Looking at a loop...

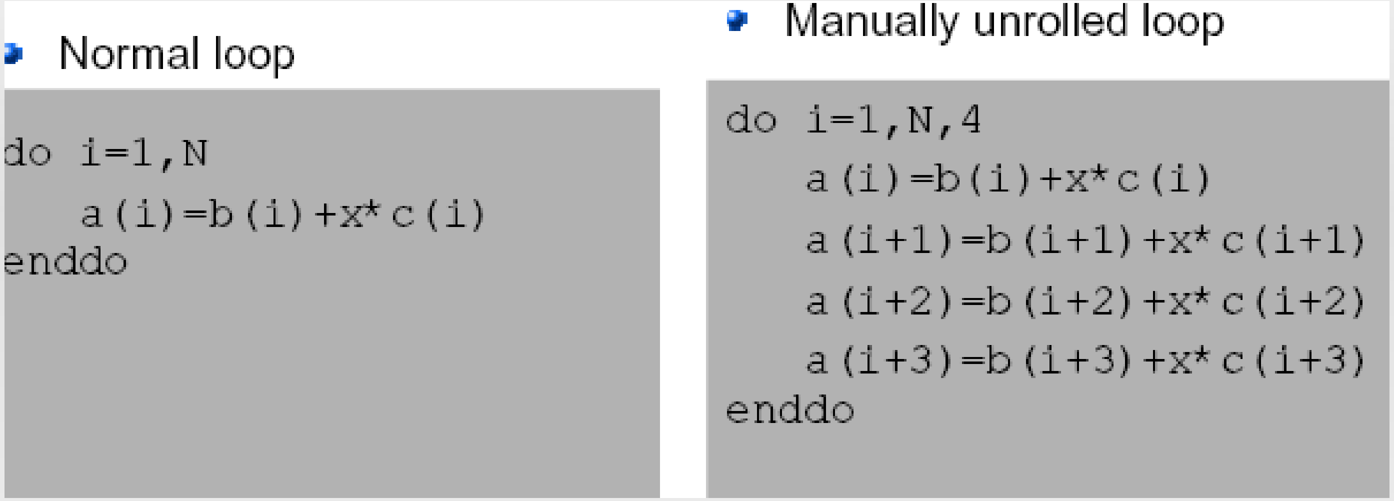
```
for( i = 0; i < N; i++){
    do work....
}
```

Every time this statement is hit, a branching instruction is called.

So optimizing a loop would involve increasing the work per loop iteration.

More work, less branches

Loop unrolling



- Normal loop

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

- Manually unrolled loop

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

- Good news – compilers can do this in the most helpful cases (not itanium, more later)
- Bad news – compilers sometimes do this where it is not helpful and or valid.
- This is not helpful when the work inside the loop is not mostly number crunching.

Loop Unrolling - Compiler

GNU compilers:

-funrollloops	Enable loop unrolling
-funrollallloops	Unroll all loops; not recommended

PGI compilers:

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

Intel compilers:

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

CAUTION: Make sure that your program still works after this!

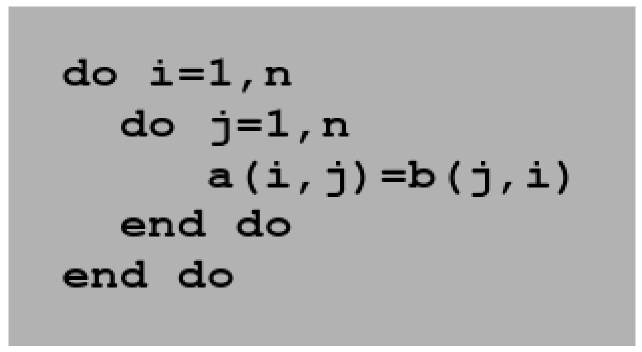
Loop Unrolling Directives

```
program dirunroll
integer,parameter :: N=1000000
real,dimension(N):: a,b,c
real:: begin,end
real,dimension(2):: rtime
common/saver/a,b,c
call random_number(b)
call random_number(c)
x=2.5
begin=dtime(rtime)
!DIR$ UNROLL 4
do i=1,N
a(i)=b(i)+x*c(i)
end do
end=dtime(rtime)
print *,' my loop time (s) is ',(end)
flop=(2.0*N)/(end)*1.0e6
print *,' loop runs at ',flop,'
MFLOP'
print *,a(1),b(1),c(1)
end s ) is 5.9999999E02
```

- Directives provide a very portable way for the compiler to perform automatic loop unrolling.
  - Compiler can choose to ignore it.

Blocking for cache (tiling)

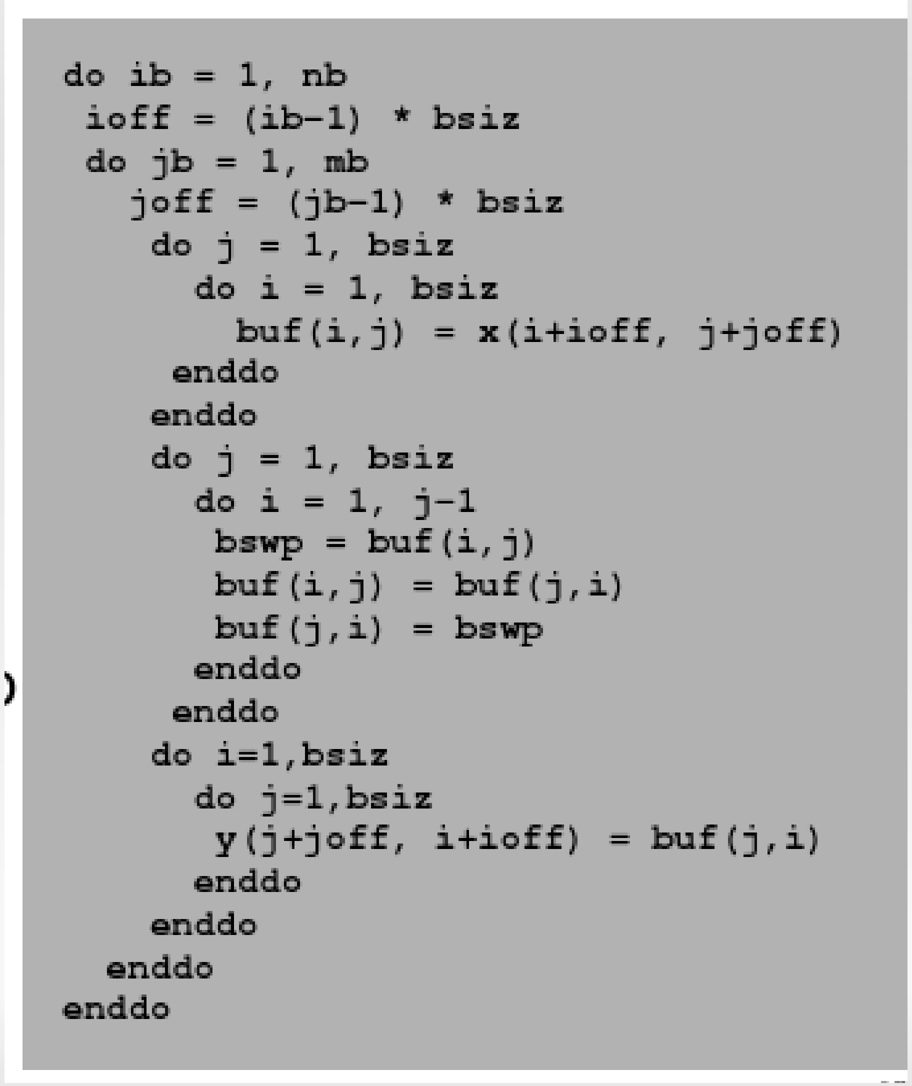
- Blocking for cache is
  - An optimization that applies for datasets that do not fit entirely into cache
  - A way to increase spatial locality of reference i.e. exploit full cache lines
  - A way to increase temporal locality of reference i.e. improves data reuse
- Example, the transposing of a matrix



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

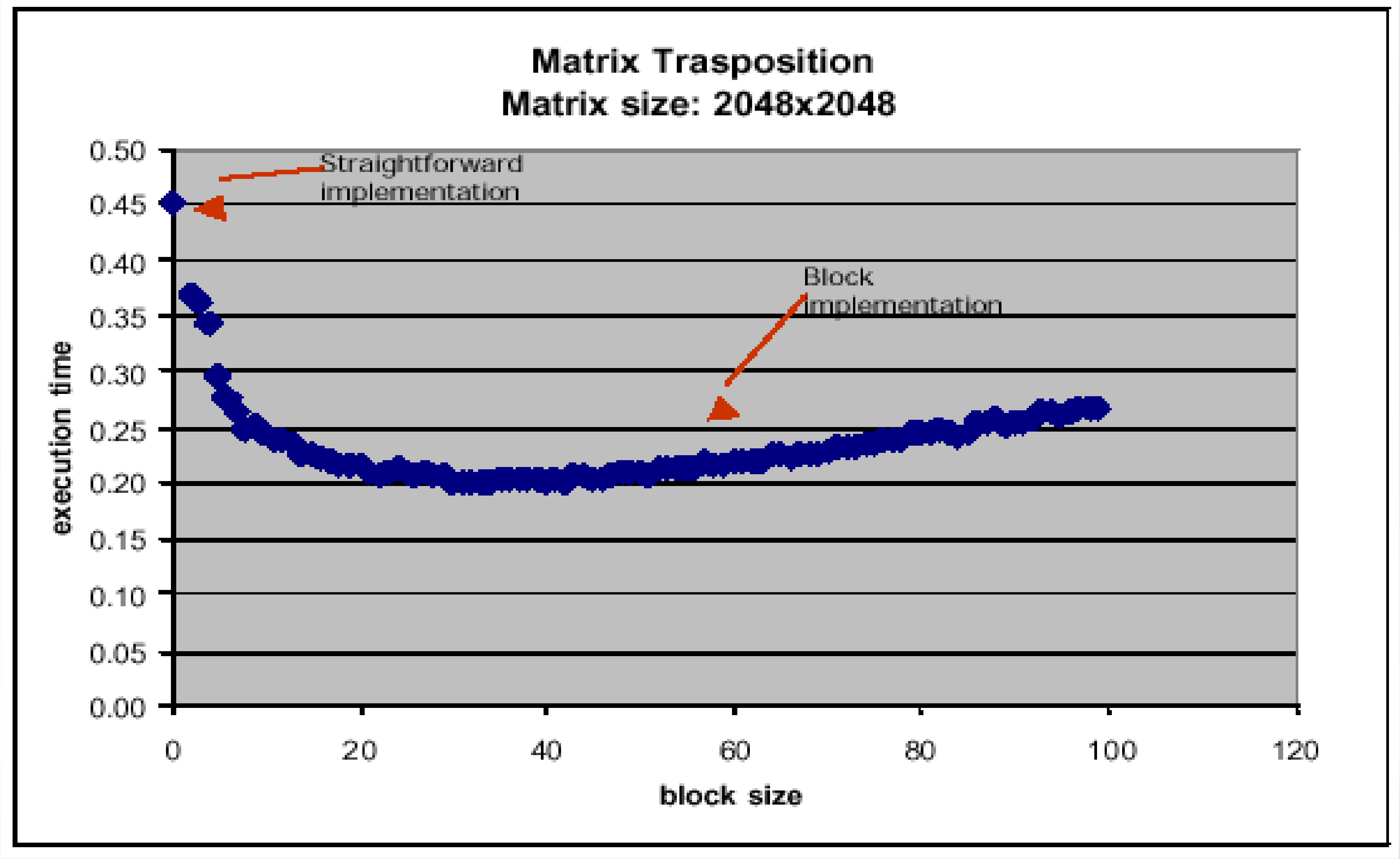
Block algorithm for transposing a matrix

- block data size = bsize
    - mb = n/bsize
    - nb = n/bsize
- These sizes can be manipulated to coincide with actual cache sizes on individual architectures.



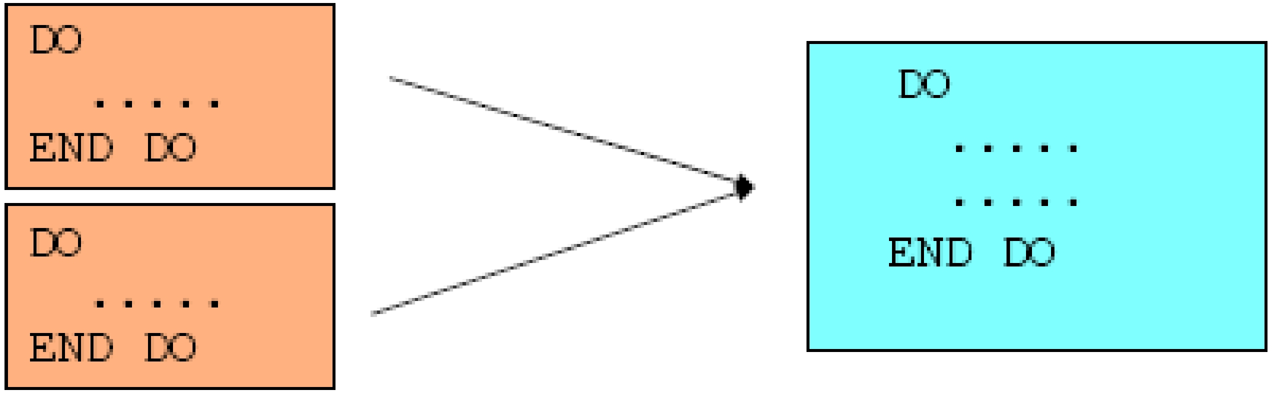
```
do ib = 1, nb
 ioff = (ib-1) * bsiz
 do jb = 1, mb
   joff = (jb-1) * bsiz
    do j = 1, bsiz
      do i = 1, bsiz
        buf(i,j) = x(i+ioff, j+joff)
     enddo
    enddo
    do j = 1, bsiz
      do i = 1, j-1
       bswp = buf(i,j)
       buf(i,j) = buf(j,i)
       buf(j,i) = bswp
      enddo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

Results...


Matrix Trasposition
Matrix size: 2048x2048
Straightforward implementation
Block implementation
execution time
block size
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
120

Loop Fusion and Fission

Fusion: Merge multiple loops into one

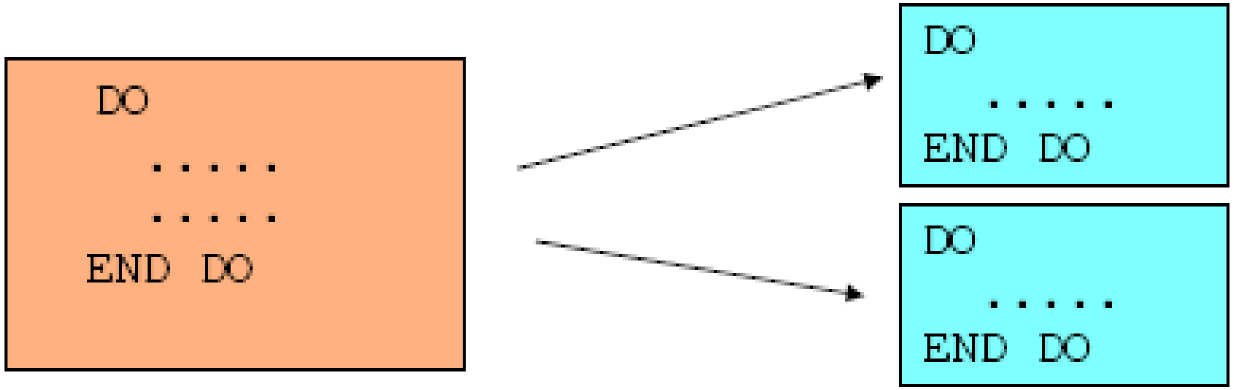


```
DO
   ......
END DO
```

```
DO
   ......
END DO
```

```
DO
   ......
   ......
END DO
```

Fission: Split one loop into multiple loops



```
DO
   ......
   ......
END DO
```

```
DO
   ......
END DO
```

```
DO
   ......
END DO
```

Loop Fusion Example

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

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

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

Potential for Fusion: dependent operations in separate loops

Advantage:

- Re-usage of array B()

Disadvantages:

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

Loop Fission Example

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

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

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

Potential for Fission: independent operations in a single loop

Advantage:

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

Disadvantages:

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

Prefetching

- Modern CPU's can perform anticipated memory lookups ahead of their use for computation.
  - Hides memory latency and overlaps computation
  - Minimizes memory lookup times
- This is a very architecture specific item
- Very helpful for regular, in-stride memory patterns

GNU:

`-fprefetch-loop-arrays`

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

PGI:

`-Mprefetch[=option:n] -Mnoprefetch`

Add (don’t add) prefetch instructions for those processors that support them (Pentium 4,Opteron); -Mprefetch is default on Opteron; -Mnoprefetch is default on other processors.

Intel:

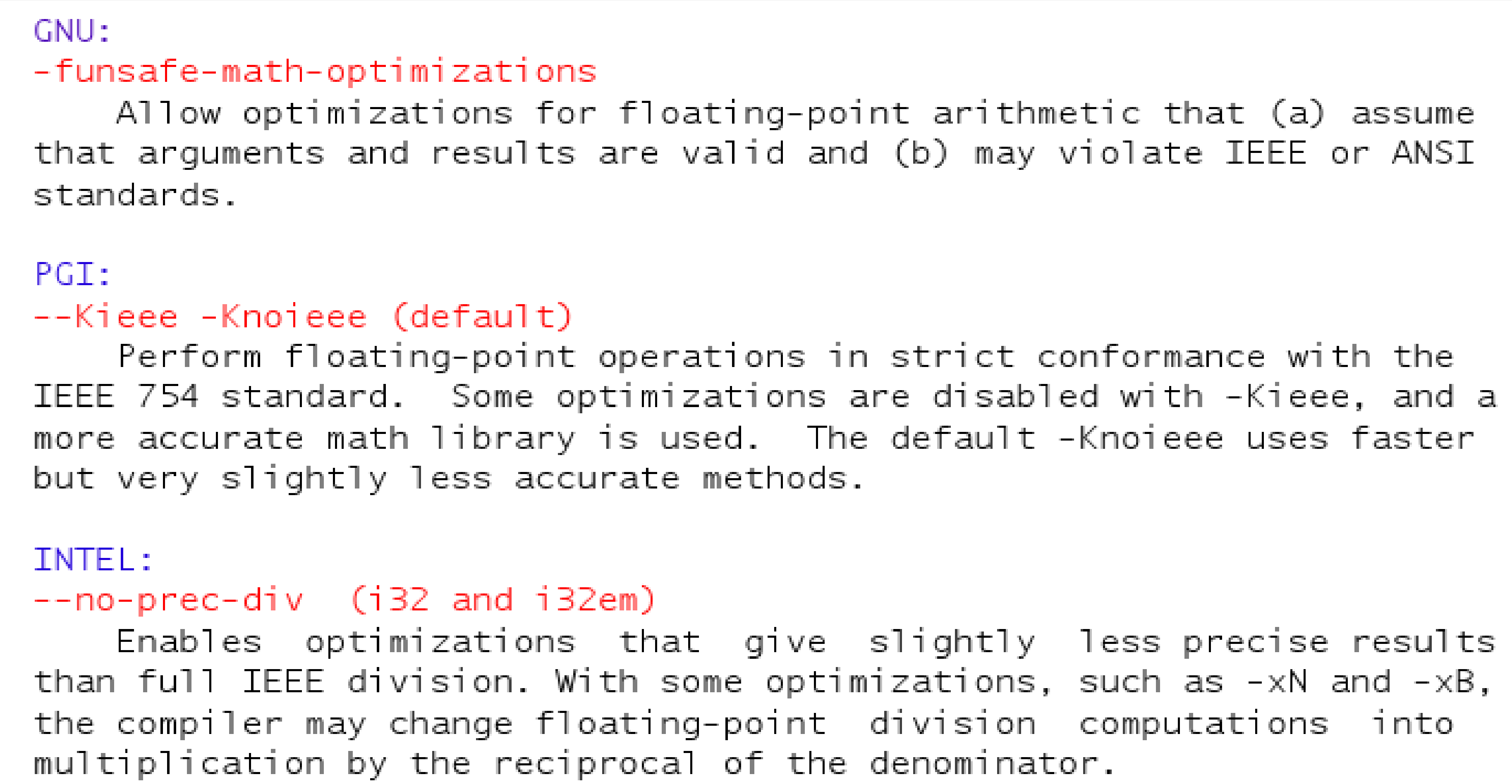
`-O3`

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

Optimizing Floating Point performance

- Operation replacement
  - Replacing individual time consuming operations with faster ones
  - Floating point division
    - Notoriously slow, implemented with a series of instructions
    - So does that mean we cannot do any division if we want performance?
  - IEEE standard dictates that the division must be carried out
    - We can relax this and replace the division with multiplication by a reciprocal
    - Compiler level optimization, rarely helps doing this by hand.
    - Much more efficient in machine language than straight division, because it can be done with approximates

IEEE relaxation



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

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

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

Keep in mind! This does reduce the precision of the math!

Elimination of Redundant Work

- Consider the following piece of code

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

It is clear that the division by B(j) is redundant and can be pulled out of the loop

```
do j = 1,N
   sum = 0.0D0
   do i = 1,N
      sum = sum + C(i,j)
   enddo
   A(j) = A(j) + sum/B(j)
enddo
```

Elimination of Redundant Work

```
do k = 1,N
   do j = 1,N
      do i = 1,N
         A(k) = B(k) + C(j) + D(i)
      enddo
   enddo
enddo
```

Array lookups cost time

By introducing constants and precomputing values, we eliminate a bunch of unnecessary fops

This is the type of thing compilers can do quite easily.

```
do k = 1,N
   Bk = B(k)
   do j = 1,N
      BkCj = Bk + C(j)
      do i = 1,N
         A(k) = BkCj + D(i)
      enddo
   enddo
enddo
```

Function (Procedure) Inlining

- Calling functions and subroutines requires overhead by the CPU to perform
  - The instructions need to be looked up in memory, the arguments translated, etc..
- Inlining is the process by which the compiler can replace a function call in the object with the source code
  - It would be like creating your application in one big function-less format.
- Advantage
  - Increase optimization opportunities
  - Particularly advantegeous (necessary) when a function is called a lot, and does very little work (e.g. max and min functions).

Function (Procedure) Inlining Compiler Options

GNU compilers:

`-fno-inline`	Disable inlining
`-finline-functions`	Enable inlining of functions

PGI 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`

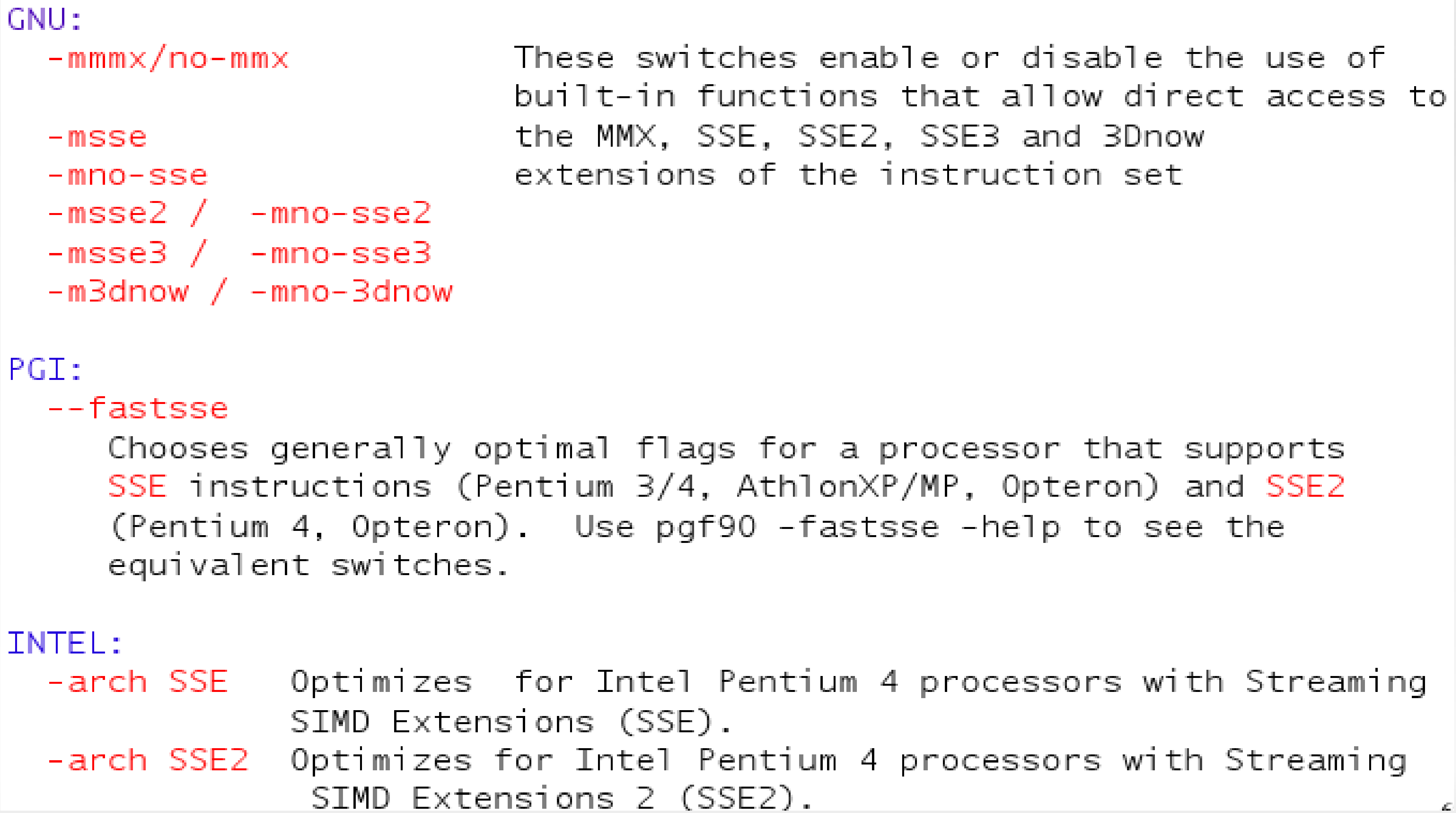
Intel compilers:

`-ip`	Enable single-file interprocedural optimization, including enhanced inlining
`-ipo`	Enable interprocedural optimization across files

Superscalar Processors

- Processors which have multiple functional units are called superscalar (instruction level parallelism)
- Examples:
  - Athlons, Opterons, Pentium 4's
  - All can do multiple floating point and integer procedures in one clock cycle
- Special instructions
  - SSE (Streaming SIMD Extensions)
    - Allow users to take advantage of this power by packing mutliple operations into one register.
    - SSE2 for double-precision
    - Right now, 2 way is very common (Opteron, P4), but 4-way to 16-way on the horizon.
    - Much much more difficult to get peak performance.

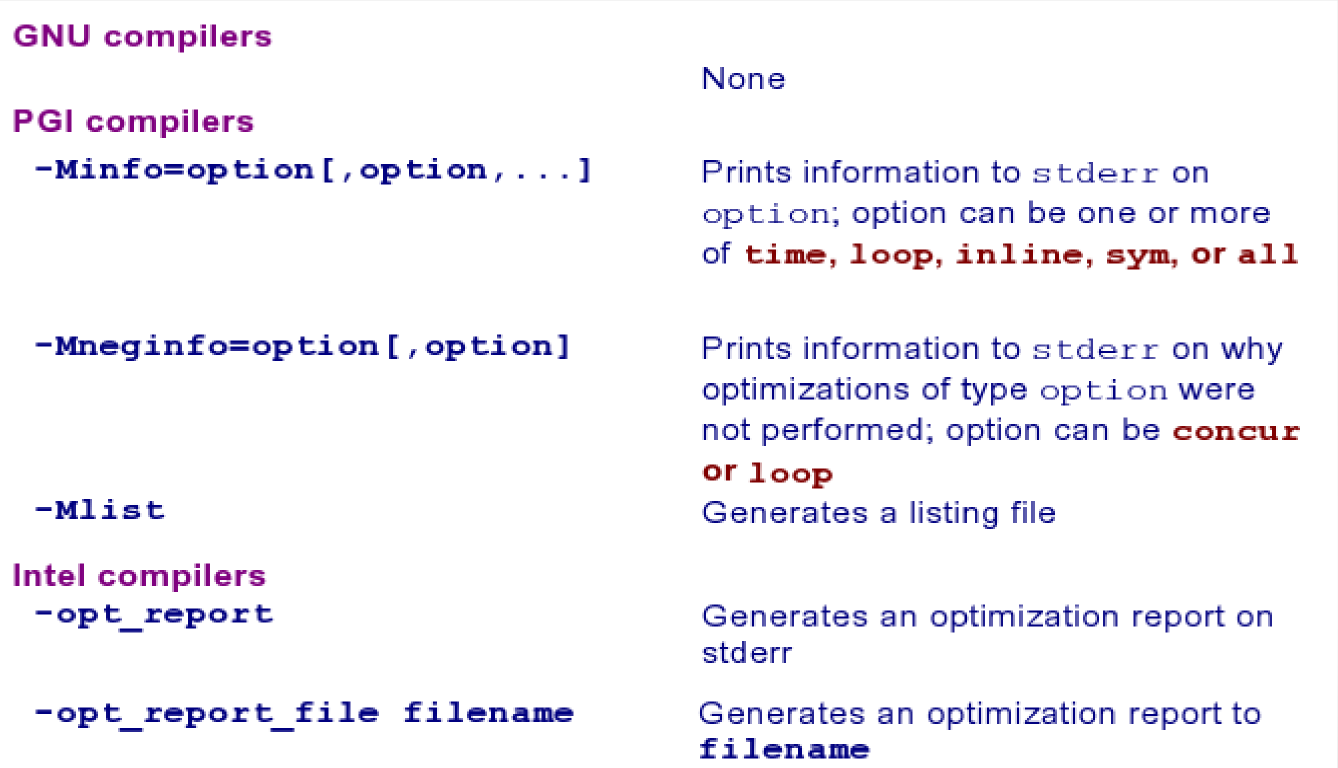
Instruction Set Extension Compiler Options



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

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

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

How do you know what the compiler is doing?

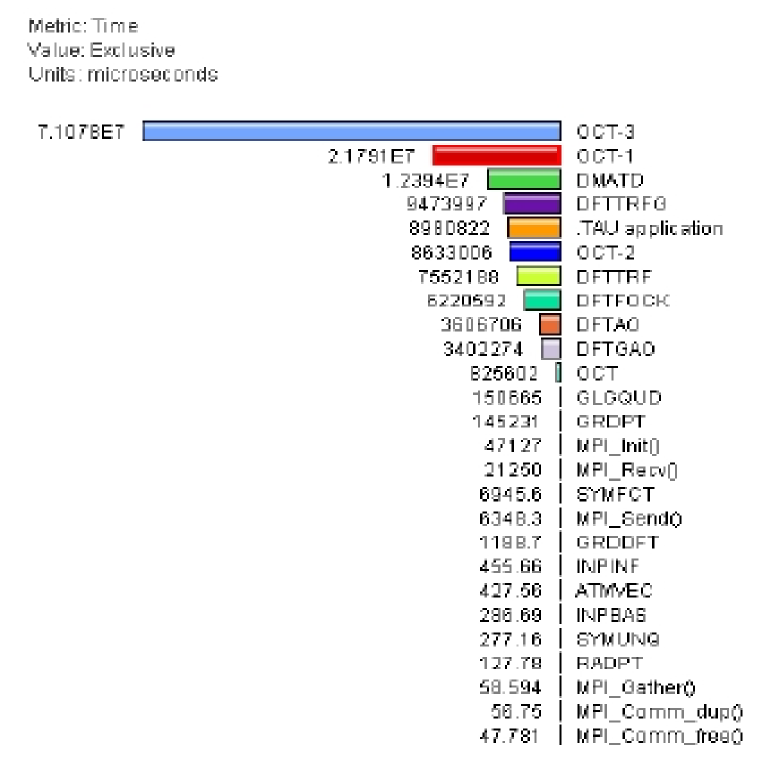
- Compiler Reports and Listings
  - By default, compilers don't say much unless you screwed up.
  - One can generate optimization reports and listing files to yeild output that shows what optimizations are performed



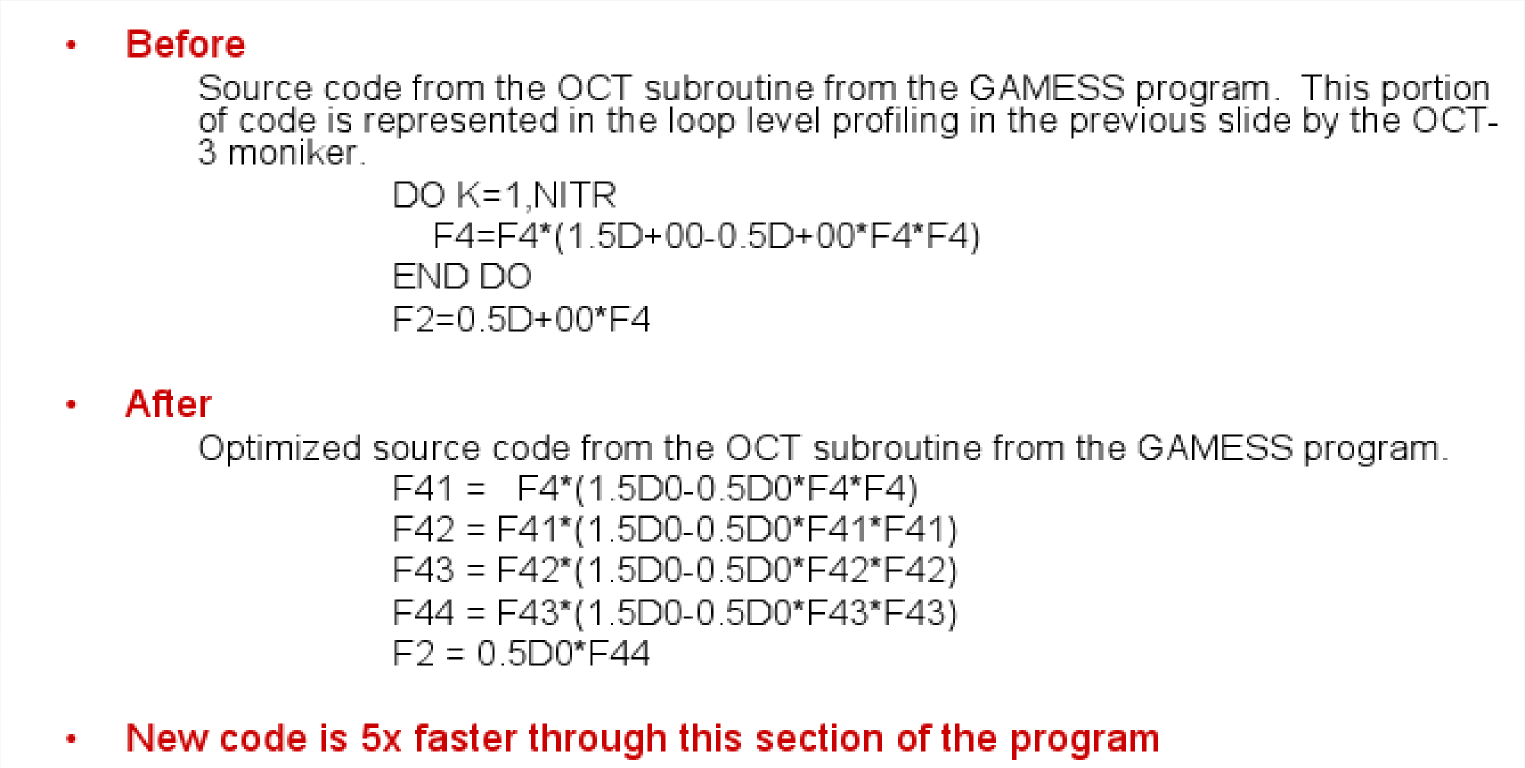
GNU compilers	
	None
PGI compilers	
`-Minfo=option[,option,...]`	Prints information to `stderr` on `option`; option can be one or more of **`time`**, **`loop`**, **`inline`**, **`sym`**, **or** **`all`**
`-Mneginfo=option[,option]`	Prints information to `stderr` on why optimizations of type `option`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

Case Study: GAMESS

- Mission from the DoD – Optimize GAMESS DFT code on an SGI Altix
- First step: profile the code



Case Study: GAMESS



- **Before**

  Source code from the OCT subroutine from the GAMESS program. This portion of code is represented in the loop level profiling in the previous slide by the OCT-3 moniker.

```
DO K=1,NITR
   F4=F4*(1.5D+00-0.5D+00*F4*F4)
END DO
F2=0.5D+00*F4
```

- **After**

  Optimized source code from the OCT subroutine from the GAMESS program.

```
F41 =   F4*(1.5D0-0.5D0*F4*F4)
F42 = F41*(1.5D0-0.5D0*F41*F41)
F43 = F42*(1.5D0-0.5D0*F42*F42)
F44 = F43*(1.5D0-0.5D0*F43*F43)
F2 = 0.5D0*F44
```

- **New code is 5x faster through this section of the program**

- Further inspection of the Itanium archtecture showed 2 things:
  - The compilers were really bad at loop optimization
  - The overhead for conditionals is enormous

Future...

- Multi-core CPU's
  - The key issue is memory bandwidth, and good caching performance will be key.
    - This problem is worsened as more cores are added.
  - Caching and memory performance vary greatly
    - Some share L2 cache between all cores, some have their own
    - Varying number of pipelines to memory
- Increasing SIMD operations
  - SSE2 and beyond
  - 4-way here, 8 and 16-way down the pike
    - Makes it increasingly more difficult to get peak performance of a chip
    - Stalling the pipeline gives a relatively bigger hit.

Take Home Messages...

- Performance programming on single processors requires
  - Understanding memory
    - levels, costs, sizes
  - Understand SSE and how to get it to work
    - In the future this will one of the most important aspects of processor performance.
  - Understand your program
    - No subsitute for speding quality time with your code.
- Do not spend a lot of time doing what I compiler will do automatically.
  - Start with compiler optimizations!
- Code optimization is hard work!
  - We haven't even talked about parallel applications yet!