# Getting good performance from your application

Tuning techniques for serial programs on cache-based computer systems

# Application Tuning

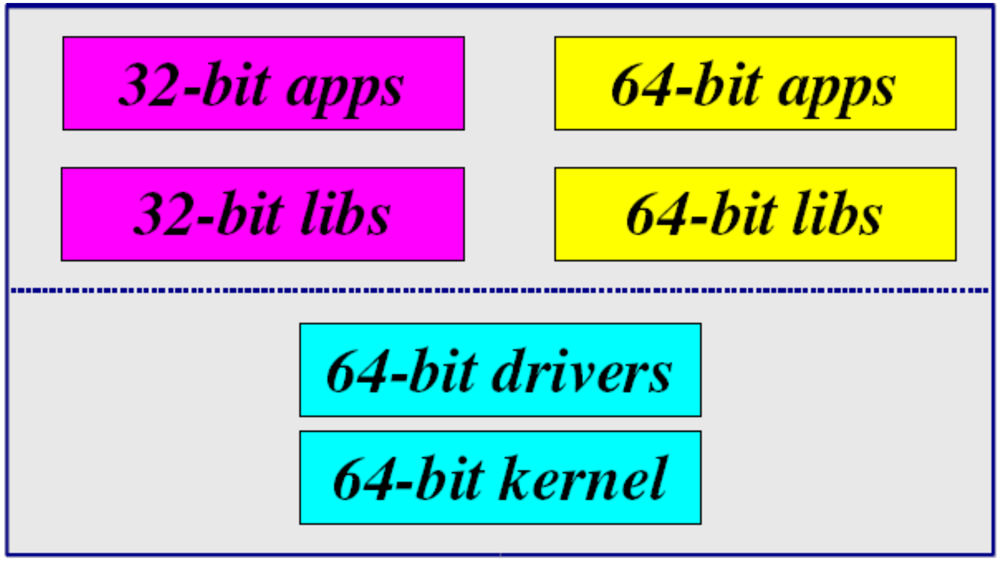
## Selected Topics

# Application Tuning

- Selected Topics:
  - 32- vs 64-bit
  - binary data portability
  - floating point numbers and IEEE 754
    - compiler options
  - a case study
- Summary

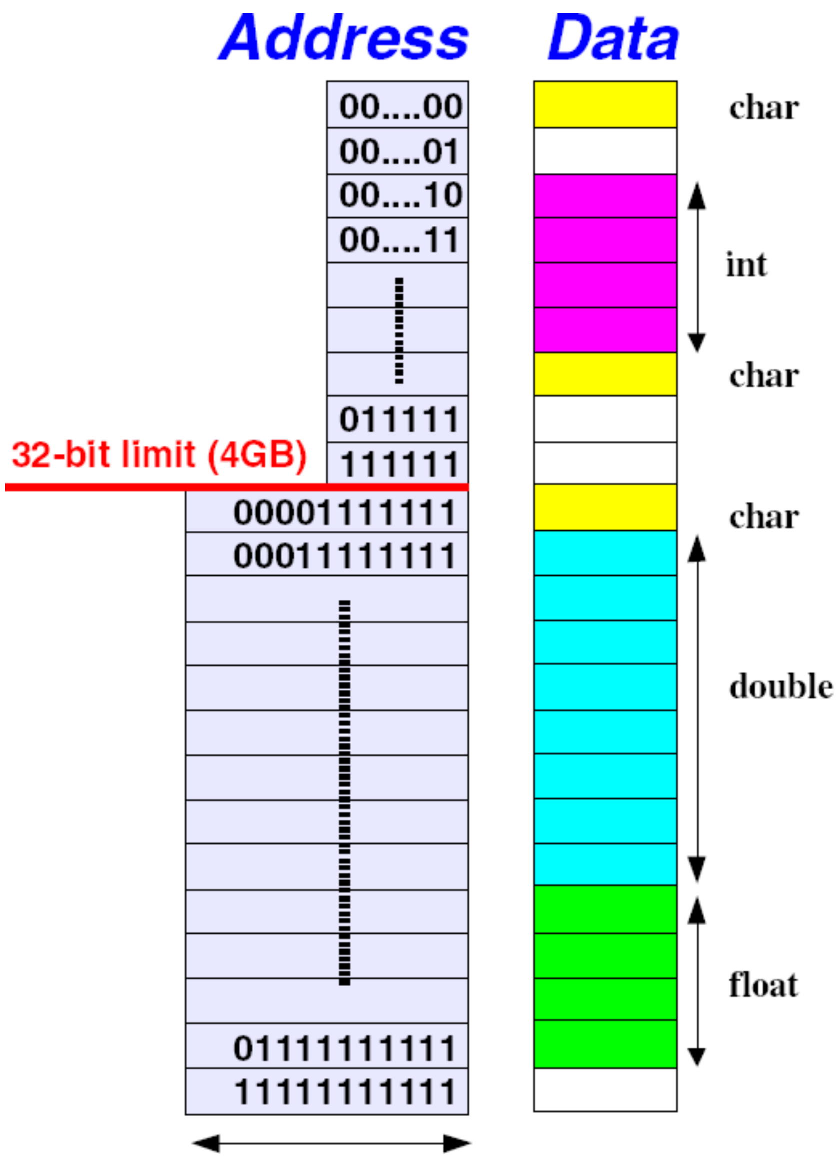
# 32-bit vs 64-bit issues

- 64-bit operating systems
- Implication: The address space of a single application can be larger than 4 GB



# 32-bit vs 64-bit issues






- Addresses ≠ Data
- An 'n'-byte data type fills always n bytes in memory (byte addressable)
- I.e. the next element is n bytes further in memory
- This increment is not related to the size of the addresses (32-bit or 64-bit)

# 32-bit vs 64-bit issues

| *C data type* | *ILP32 (bits)* | *LP64 (bits)* |
|---|---|---|
| *char* | *8* | *same* |
| *short* | *16* | *same* |
| *int* | *32* | *same* |
| *long* | *32* | *64* |
| *long long* | *64* | *same* |
| *pointer* | *32* | *64* |
| *enum* | *32* | *same* |
| *float* | *32* | *same* |
| *double* | *64* | *same* |
| *long double* | *128* | *same* |

UNIX and Linux support LP64; Windows 64-bit uses LLP64, where long stays 32 bits

# (p)ldd and LD_LIBRARY_PATH

How to check which shared-libraries are loaded?

- Static check: use the ldd command
  - `$ ldd executable`

- Dynamic check: use pldd on the PID
  - `$ pldd pid`
  - Solaris only
  - there are scripts available for Linux as well
  - we have installed pldd on the DTU HPC cluster

# (p)ldd and LD_LIBRARY_PATH

- How to change the search path for dynamic libraries?
  - Use LD_LIBRARY_PATH – but use it with care!
- Best practice:
  - Compile the path into your application:
    - GCC:  -Wl,-rpath <path_to_lib>
    - ld.so will then use this path
  - Avoid LD_LIBRARY_PATH in your shell environment – use a wrapper script for the application
  - Check out this blog note (www.hpc.dtu.dk/?page_id=1180), too!

# Binary data storage

- Storing your data in binary format
- Advantages:
  - compact
  - fast
  - no loss of precision
- Drawbacks:
  - not “human readable”
  - data analysis more complicated
  - and ...

# Binary data storage

- Example: integer 0x12345678 (hexadecimal)

```
value = 0x12345678;         // 305419896
printf("%d\n", value);
fwrite(&value, sizeof(value), 1, fptr);
```

- Write it ...
  - ... on i386:

```
305419896
Architecture: i386
Value written to endian_i386.dat.
```

  - ... on SPARC:

```
305419896
Architecture: sparc
Value written to endian_sparc.dat.
```

# Binary data storage

- Read it:

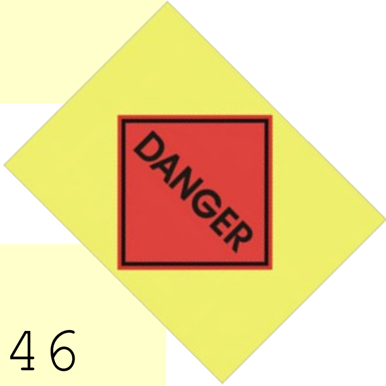
```
fread(&value, sizeof(value), 1, fptr);
printf("%d\n", value);
```

- on i386 data from i386:

```
Architecture: i386
Read from endian_i386.dat: 305419896
```

- on i386 data from SPARC:

```
Architecture: i386
Read from endian_sparc.dat: 2018915346
```

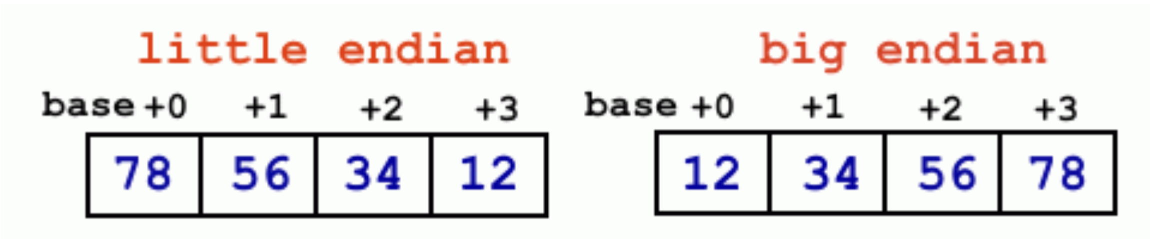


# Little Endian vs Big Endian

- The order in which the bits are interpreted has not been standardized!
- Two 'popular' formats in use
  - Big Endian – SPARC, PowerPC, ...
  - Little Endian – Intel x86, AMD64, ...
- This is an issue when using the same binary data file on both platforms ...

# Little Endian vs Big Endian

- Example: integer 0x12345678 (hexadecimal)




- Check with 'od' command:

```
$ od -x endian_sparc.dat
0000000 1234 5678
0000004

$ od -x endian_i386.dat
0000000 7856 3412
0000004
```

# Little Endian vs Big Endian

- This is something you should be aware of when working with binary data!
- Tools:
  - Sun Fortran: -xfilebyteorder option
  - Portland Fortran compiler
  - swab() subroutine (low level)
  - ...

# Floating point numbers & IEEE 754

## Lesser known side effects of IEEE 754:

- Will this code run or fail?

```
#include <stdio.h>
#include <math.h>

int
main(int argc, char *argv[]) {

    double x;

    for(int i = 0; i < 10; i++) {
        x = sqrt(5.0 - i);
        printf("%lf\n", x);
    }
}
```

# Floating point numbers & IEEE 754

## Lesser known side effects of IEEE 754:

- What do you prefer?

```
$ gcc -otrapex trapex.c -lm
$ ./trapex
2.236068
2.000000
1.732051
1.414214
1.000000
0.000000
-nan
-nan
-nan
-nan
$
```

IEEE 754 compliant!

```
$ suncc -ftrap=common -o trapex
$ ./trapex
2.236068
2.000000
1.732051
1.414214
1.000000
0.000000
Floating point exception
$
```

not(!) IEEE 754 compliant!

# Floating point numbers & IEEE 754

## Lesser known side effects of IEEE 754:

- The IEEE 754 standard doesn't “allow” traps on floating point exceptions, like invalid arguments, division by zero, over- and underflows
- Some compilers provide options to change that.
  - Studio: -ftrap=<exception_list>, e.g. common
  - Intel: -fp-trap=<exception_list>, e.g. common
- However: GCC has no such option, needs to be implemented by the programmer via library calls (see ‘man fenv’ – and next slide)

# Floating point numbers & IEEE 754

## Enabling floating point traps with GCC

```
#include <stdio.h>
#include <math.h>

#define __USE_GNU
#include "fenv.h"
void enable_ftraps(void) {
    feenableexcept (FE_DIVBYZERO);
    feenableexcept (FE_INVALID);
    feenableexcept (FE_OVERFLOW); }

int main(int argc, char *argv[]) {
    double x;
    enable_ftraps();
    for(int i = 0; i < 10; i++) {
        x = sqrt(5.0 - i);
        printf("%lf\n", x);
    }
}
```

# Floating point numbers & IEEE 754

## Now it works with GCC ...

```
$ gcc -o trapex2 trapex2.c -lm
$ ./trapex2
2.236068
2.000000
1.732051
1.414214
1.000000
0.000000
Floating point exception
$
```

not(!)
IEEE 754 compliant!

# Floating point numbers & IEEE 754

- Compilers do not re-arrange your arithmetic expressions – unless you ask for it!
- This is part of the optimization flags (and varies from compiler to compiler!)
- Example on the next slides shows this for a division inside a for-loop
  - this is done with an old Studio compiler, but illustrates nicely what happens
  - the optimization option is '-fsimple=[0|1|2]', from none (0) to aggressive arithmetic optimizations (2).

# Floating point numbers & IEEE 754

## Effects of -fsimple:

- compiled with -fast -xrestrict -fsimple=0:

```
void
divvec(int n, double div, double *a, double *b) {

    int i;

Source loop below has tag L1
L-1 scheduled with steady-state cycle count = 17
L-1 has 1 loads, 1 stores, 2 prefetches, 0 FPadds,
        0 FPmuls, and 1 FPdivs per iteration
    for(i = 0; i < n; i++)
        b[i] = a[i]/div;
}
```

UltraSPARC

# Floating point numbers & IEEE 754

## Effects of -fsimple:

- compiled with -fast -xrestrict -fsimple=2:

```
void
divvec(int n, double div, double *a, double *b) {

    int i;

Source loop below has tag L1
L-1 scheduled with steady-state cycle count = 1
L-1 unrolled 8 times
L-1 has 1 loads, 1 stores, 4 prefetches, 0 FPadds,
        1 FPmuls, and 0 FPdivs per iteration
    for(i = 0; i < n; i++)
        b[i] = a[i]/div;
}
```

UltraSPARC

# A closer look ...

- on x86_64 Linux, we have no compiler that does give us the information we want ... i.e. the number of fpmul and fpdiv
- What now???
- We need some other tools, to get this information.
  - objdump – a Linux tool “to look into” executables
  - objdump has a lot of options – we need just one
    - objdump -S file.o : show disassembly and source intermixed
  - ... but how to interprete the assembly code???

# A closer look ...

The really quick guide to x86 assembly

- instructions of interest: double precision floating point multiplication and division
- multiplication: *mul*d
  - mulsd, mulpd, vmulsd, vmulpd
- division: *div*d
  - divsd, divpd, vdivsd, vdivpd
- ‘s’ is for single data, ‘p’ for packed data (vector)
- the leading letter indicates vector type

# A closer look ...

compiled with ‘suncc -g -fast -fsimple=0’:

```
$ objdump -S divvec.o | egrep "mul.d|div.d"
  9d:   c5 e3 5e e0             vdivsd %xmm0,%xmm3,%xmm4
  b8:   c5 d3 5e f0             vdivsd %xmm0,%xmm5,%xmm6
  c8:   c5 43 5e c0             vdivsd %xmm0,%xmm7,%xmm8
  d8:   c5 33 5e d0             vdivsd %xmm0,%xmm9,%xmm10
  e8:   c5 23 5e e0             vdivsd %xmm0,%xmm11,%xmm12
  f8:   c5 13 5e f0             vdivsd %xmm0,%xmm13,%xmm14
 108:   c5 83 5e c8             vdivsd %xmm0,%xmm15,%xmm1
 118:   c5 eb 5e d8             vdivsd %xmm0,%xmm2,%xmm3
 149:   c5 f3 5e d0             vdivsd %xmm0,%xmm1,%xmm2
 1cf:   c5 35 5e d6             vdivpd %ymm6,%ymm9,%ymm10
 1ed:   c5 25 5e e6             vdivpd %ymm6,%ymm11,%ymm12
 21a:   c5 45 5e c6             vdivpd %ymm6,%ymm7,%ymm8
 259:   c5 db 5e e8             vdivsd %xmm0,%xmm4,%xmm5
```

- only divisions!
- single data and packed: multi-versioning of loop

# A closer look ...

compiled with ‘suncc -g -fast -fsimple=2’:

```
$ objdump -S divvec.o | egrep  "mul.d|div.d"
  82:   c5 d3 5e f0             vdivsd %xmm0,%xmm5,%xmm6
  a8:   c5 4b 59 04 06          vmulsd (%rsi,%rax,1),%xmm6,%xmm8
  ba:   c5 4b 59 4c 06 08       vmulsd 0x8(%rsi,%rax,1),%xmm6,%xmm9
  ca:   c5 4b 59 54 06 10       vmulsd 0x10(%rsi,%rax,1),%xmm6,%xmm10
  d6:   c5 4b 59 5c 06 18       vmulsd 0x18(%rsi,%rax,1),%xmm6,%xmm11
  e2:   c5 4b 59 64 06 20       vmulsd 0x20(%rsi,%rax,1),%xmm6,%xmm12
  ee:   c5 4b 59 6c 06 28       vmulsd 0x28(%rsi,%rax,1),%xmm6,%xmm13
  fa:   c5 4b 59 74 06 30       vmulsd 0x30(%rsi,%rax,1),%xmm6,%xmm14
 106:   c5 4b 59 7c 06 38       vmulsd 0x38(%rsi,%rax,1),%xmm6,%xmm15
 130:   c5 cb 59 3c 06          vmulsd (%rsi,%rax,1),%xmm6,%xmm7
 19b:   c4 41 45 5e c8          vdivpd %ymm8,%ymm7,%ymm9
 1cf:   c4 41 1d 59 e9          vmulpd %ymm9,%ymm12,%ymm13
 1ee:   c4 41 0d 59 f9          vmulpd %ymm9,%ymm14,%ymm15
 21a:   c4 41 2d 59 d9          vmulpd %ymm9,%ymm10,%ymm11
 25e:   c5 e3 5e c0             vdivsd %xmm0,%xmm3,%xmm0
 270:   c5 fb 59 24 06          vmulsd (%rsi,%rax,1),%xmm0,%xmm4
```

- mostly multiplications – it works!

# A closer look ...

compiled with ‘suncc -g -fast -fsimple=2’

- ... and using the ‘restrict’ keyword:

```
$ objdump -S divvec.o | egrep  "mul.d|div.d"
 7d:   c4 41 35 5e da        vdivpd %ymm10,%ymm9,%ymm11
 af:   c4 41 0d 59 fb        vmulpd %ymm11,%ymm14,%ymm15
 ce:   c4 c1 75 59 d3        vmulpd %ymm11,%ymm1,%ymm2
 fa:   c4 41 1d 59 eb        vmulpd %ymm11,%ymm12,%ymm13
140:   c5 db 5e e8           vdivsd %xmm0,%xmm4,%xmm5
150:   c4 a1 53 59 34 0e     vmulsd (%rsi,%r9,1),%xmm5,%xmm6
```

- mostly multiplications – it works!
- the multi-versioning is gone – only the vectorized code (and a clean-up loop) left!

# A closer look ... now with GCC

- -O3 does not enable math optimizations, like replacing a division with a constant by a multiplication with the inverse
- to enable this, we need -ffast-math option
- to get some optimization information (at compile time), we can use -fopt-info

# A closer look ... now with GCC

## compiled with ‘gcc  -g -O3  -fopt-info’

```
 $ gcc  -g -O3  -fopt-info -c divvec.c
divvec.c:11:5: optimized: loop vectorized using 16 byte
vectors
divvec.c:11:5: optimized:  loop versioned for vectorization
because of possible aliasing
```

```
 $ objdump -S divvec.o | egrep  "mul.d|div.d"
  35:   66 0f 5e c8             divpd  %xmm0,%xmm1
  55:   f2 0f 5e c2             divsd  %xmm2,%xmm0
  70:   f2 0f 5e c2             divsd  %xmm2,%xmm0
```

- GCC does multi-versioning, too!
- no multiplications – only divisions!

# A closer look ... now with GCC

## compiled with ‘gcc  -g -O3 -ffast-math -fopt-info’

```
 $ gcc  -g -O3 -ffast-math -fopt-info -c divvec.c
divvec.c:11:5: optimized: loop vectorized using 16 byte
vectors
divvec.c:11:5: optimized:  loop versioned for vectorization
because of possible aliasing
```

```
 $ objdump -S divvec.o | egrep  "mul.d|div.d"
  1e:   f2 0f 5e c1             divsd  %xmm1,%xmm0
  45:   66 0f 59 ca             mulpd  %xmm2,%xmm1
  60:   f2 0f 59 04 c6          mulsd  (%rsi,%rax,8),%xmm0
  80:   f2 0f 59 c8             mulsd  %xmm0,%xmm1
```

- GCC does multi-versioning, too!
- mostly multiplications – it works!

# A closer look ... now with GCC

compiled with 'gcc  -g -O3 -ffast-math -fopt-info'

- ... and using the 'restrict' keyword

```
 $ gcc  -g -O3 -ffast-math -fopt-info -c divvec.c
divvec.c:11:5: optimized: loop vectorized using 16 byte
vectors
```

```
 $ objdump -S divvec.o | egrep  "mul.d|div.d"
  10:   f2 0f 5e c1             divsd  %xmm1,%xmm0
  35:   66 0f 59 ca             mulpd  %xmm2,%xmm1
  52:   f2 0f 59 04 c6          mulsd  (%rsi,%rax,8),%xmm0
```

- multi-versioning is gone!
- less instructions than with the suncc compiler!?!

# A closer look ... now with GCC

## GCC does not unroll loops with -O3!

- we need to add -funroll-loops, too!

```
gcc -g -O3 -ffast-math -funroll-loops -fopt-info -c divvec.c
divvec.c:11:5: optimized: loop vectorized using 16 byte
vectors
divvec.c:11:16: optimized: loop unrolled 7 times
```

```
 $ objdump -S divvec.o | egrep  "mul.d|div.d" | grep -c mulpd
  15
```

- now we have more mulpd instructions (14 more from loop unrolling)
- Lesson learned: check what ‘-O3’ does!
  - Reminder:  ‘gcc -g -Q --help=optimizers ...’

# A closer look ... summary

- Even with no compiler commentary, we can still get useful information, using
  - extra compiler options like -fopt-info (gcc)
  - tools like objdump – plus some basic knowledge about assembly code
- Caveat: this kind of analysis is feasible on small code kernels, only!
- Best practice: extract a small kernel from larger application, do the tests/tuning (- and reinsert).

# Summary

- You have now heard about
  - tuning techniques
  - tools: compilers, analysis tools
  - libraries
  - other performance parameters
  - debuggers: try Totalview
- Now you have to apply that and get experience!
- But never forget:

# Correct code has the highest priority – not speed!