{{lastupdated_at}} by {{lastupdated_by}}

Computation Benchmarks

Mac OS X benchmarks

Here a summary of some benchmarks that have been obtained on a 2.66 GHz Intel Core i7 Mac OS X 10.6.8 system for executing 100000000 (one hundred million) times a given computation (execution times in seconds). The benchmarks have been obtained in double precision and single precision:

Computation	double	float	Comment
pow(x,2)	1.90	1.19	
x*x	0.49	0.48	**Prefer multiplication over pow(x,2)**
pow(x,2.01)	7.96	4.19	pow is a very time consuming operation
x/a	0.98	1.22	
x*b (where b=1/a)	0.46	0.66	**Prefer multiplication by the inverse over division**
x+1.5	0.40	0.40	
x-1.5	0.49	0.49	**Prefer addition over subtraction**
sin x_mark.png	4.66	2.39	
cos x_mark.png	4.64	2.46	
tan x_mark.png	5.40	2.84	tan is pretty time consuming
acos x_mark.png	2.18	0.94	
sqrt x_mark.png	1.29	1.37	
log10 x_mark.png	2.60	2.48	
log x_mark.png	2.72	2.33	
exp x_mark.png	7.17	7.20	exp is a very time consuming operation (comparable to pow)

Note that pow, exp and the trigonometric functions are significantly (a factor of about 2) faster using single precision compared to double precision.

Benchmark comparison for different systems

And here a comparison for various computing systems (double precision). In this comparison, the Mac is about the fastest, galileo (which is a 32 Bit system) is pretty fast for multiplications, kepler is the laziest (AMD related? Multi-core related?), fermi and the CI13 virtual box are about the same (there is no notable difference between gcc and clang on the virtual box).

Computation	Mac OS X	galileo	kepler	dirac	fermi	CI13 (gcc 4.8.0)	CI13 (clang 3.1)
pow(x,2)	1.90	**5.73**	4.83	3.5	2.65	1.94	1.99
x*x	0.49	0.31	**1.04**	**1.06**	0.5	0.58	0.57
pow(x,2.01)	7.96	10.96	**17.53**	**17.73**	11.11	8.71	8.44
x/a	0.98	1.24	**1.87**	**1.92**	1.03	1.15	1.16
x*b (where b=1/a)	0.46	0.27	**0.99**	**0.99**	0.51	0.54	0.54

x+1.5	0.40	0.27	**0.96**	**1.02**	0.43	0.47	0.47
x-1.5	0.49	0.27	**1.08**	**1.1**	0.57	0.47	0.47
sin x_mark.png	4.66	4.76	**10.46**	**10.44**	6.72	5.62	5.52
cos x_mark.png	4.64	4.68	**10.16**	**10.28**	6.35	5.65	5.62
tan x_mark.png	5.40	6.27	**15.23**	**15.4**	8.61	8.11	7.98
acos x_mark.png	2.18	**9.57**	7.49	7.75	4.48	3.86	2.93
sqrt x_mark.png	1.29	**3.29**	2.33	2.4	0.97	2.02	1.84
log10 x_mark.png	2.60	5.33	**12.91**	**12.58**	7.71	6.54	6.47
log x_mark.png	2.72	5.15	**10.64**	**10.66**	6.32	5.26	5.09
exp x_mark.png	7.17	**10**	4.78	4.8	1.85	2.03	2.02

Underlined numbers show the fastest, **bold** numbers the slowest computations.

And the same for single precision:

Computation	**Mac OS X**	**galileo**	**kepler**	**dirac**	**fermi**	**CI13 (gcc 4.8.0)**	**CI13 (clang 3.1)**
pow(x,2)	1.19	1.77	**3.27**	3	1.35	1.54	0.9
x*x	0.48	0.3	**0.99**	**1**	0.47	0.54	0.54
pow(x,2.01)	4.19	10.64	**29.81**	**30.21**	14.42	13	12.29
x/a	1.22	1.24	**2.77**	**2.79**	1.2	1.37	1.4
x*b (where b=1/a)	0.66	0.27	**1.72**	**1.74**	0.67	0.76	0.79
x+1.5	0.40	0.27	**1.03**	**1.04**	0.4	0.46	0.47
x-1.5	0.49	0.27	**1.13**	**1.14**	0.54	0.47	0.47
sin x_mark.png	2.39	4.92	**116.41**	**119.06**	54	41.2	40.22
cos x_mark.png	2.46	4.85	**116.47**	**119.27**	53.93	40.91	40.3
tan x_mark.png	2.84	6.47	**120.69**	**122**	55.14	42.36	41.83
acos x_mark.png	0.94	**9.02**	8.6	8.71	3.86	2.81	2.38
sqrt x_mark.png	1.37	2.27	**3.77**	**3.75**	1.5	1.84	1.55
log10 x_mark.png	2.48	4.15	**12.74**	**12.59**	6.28	5.74	4.97
log x_mark.png	2.33	3.83	**10.07**	**10.42**	5.16	4.88	4.11
exp x_mark.png	7.20	9.96	**17.51**	**18.32**	10.77	10.21	10.18

Note the **enormous speed penalty of trigonometric functions on most of the systems**. Floating point arithmetics are only faster on Mac OS X.

Here the specifications of the machines used for benchmarking:

- Mac OS X: 2.66 GHz Intel Core i7 Mac OS X 10.6.8, gcc 4.2.1
- galileo: 32 Bit, Intel Xeon, 2.8 GHz, gcc 3.2.2
- kepler: 64 Bit, AMD Opteron 6174, 12C, 2.20 GHz, gcc 4.1.2
- dirac: 64 Bit, AMD Opteron 6174, 12C, 2.20 GHz, gcc 4.1.2
- fermi: 64 Bit, Intel(R) Xeon(R) CPU E5450 @ 3.00GHz, gcc 4.1.2
- CI13: 64 Bit (virtual box)

Behind the scenes

Here now some information to understand what happens.

Kepler

I did some experiments to see how the compiled code differs for different variants of the sin function call. In particular, I tested

- std::sin(double)
- std::sin(float)
- sin(double)
- sin(float)

It turned out that the call to std::sin(float) calls the function sinf, while all other codes call sin. The execution time difference is therefore related to different implementations of sin and sinf on Kepler.

Note that sin and sinf are implement in /lib64/libm.so.6 on Kepler. This library is part of the GNU C library libc (see http://www.gnu.org/software/libc/).

Using std::sin(double)

When std::sin(double) is used, the sin function will be called by the processor. Note that the same behavior is obtained when calling sin(double) (without the std prefix).

```
$ nano stdsin.cpp
 #include <cmath>
 int main(void)
 {
    double arg    = 1.0;
    double result = std::sin(arg);
    return 0;
 }
$ g++ -S stdsin.cpp
$ more stdsin.s
main:
.LFB97:
   pushq    %rbp
.LCFI0:
   movq    %rsp, %rbp
.LCFI1:
   subq    $32, %rsp
.LCFI2:
   movabsq    $4607182418800017408, %rax
   movq    %rax, -16(%rbp)
   movq    -16(%rbp), %rax
   movq    %rax, -24(%rbp)
   movsd    -24(%rbp), %xmm0
   call    sin
   movsd    %xmm0, -24(%rbp)
   movq    -24(%rbp), %rax
   movq    %rax, -8(%rbp)
   movl    $0, %eax
   leave
   ret
```

Using std::sin(float)

When std::sin(float) is used, the sinf function will be called by the processor.

```
$ nano floatstdsin.cpp
  #include <cmath>
  int main(void)
  {
      float arg    = 1.0;
      float result = std::sin(arg);
      return 0;
  }
$ g++ -S floatstdsin.cpp
$ more floatstdsin.s
main:
.LFB97:
    pushq    %rbp
.LCFI3:
    movq    %rsp, %rbp
.LCFI4:
    subq    $32, %rsp
.LCFI5:
    movl    $0x3f800000, %eax
    movl    %eax, -8(%rbp)
    movl    -8(%rbp), %eax
    movl    %eax, -20(%rbp)
    movss    -20(%rbp), %xmm0
    call    _ZSt3sinf
    movss    %xmm0, -20(%rbp)
    movl    -20(%rbp), %eax
    movl    %eax, -4(%rbp)
    movl    $0, %eax
    leave
    ret
_ZSt3sinf:
.LFB57:
    pushq    %rbp
.LCFI0:
    movq    %rsp, %rbp
.LCFI1:
    subq    $16, %rsp
.LCFI2:
    movss    %xmm0, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, -12(%rbp)
    movss    -12(%rbp), %xmm0
    call    sinf
    movss    %xmm0, -12(%rbp)
    movl    -12(%rbp), %eax
    movl    %eax, -12(%rbp)
    movss    -12(%rbp), %xmm0
    leave
    ret
```

Using sin(float)

When sin(float) is used, the compiler will perform an implicit conversion to double and then call the sin function.

```
$ nano floatsin.cpp
  #include <cmath>
  int main(void)
  {
```

```
    float arg    = 1.0;
    float result = sin(arg);
    return 0;
 }
$ g++ -S floatsin.cpp
$ more floatsin.s
main:
.LFB97:
   pushq    %rbp
.LCFI0:
   movq    %rsp, %rbp
.LCFI1:
   subq    $16, %rsp
.LCFI2:
   movl    $0x3f800000, %eax
   movl    %eax, -8(%rbp)
   cvtss2sd    -8(%rbp), %xmm0
   call    sin
   cvtsd2ss    %xmm0, %xmm0
   movss    %xmm0, -4(%rbp)
   movl    $0, %eax
   leave
   ret
```

Mac OS X

And now the same experiment on Mac OS X. It turns out that the code generated by the compiler has the same structure, and the functions that are called are again _sin and _sinf (all function names have a _ prepended on Mac OS X). This means that the implementation of the _sinf function on Mac OS X is considerably faster than the implementation on kepler.

Note that _sin and _sinf are implement in /usr/lib/libSystem.B.dylib on my Mac OS X.

Using std::sin(double)

When std::sin(double) is used, the _sin function will be called by the processor. Note that the same behavior is obtained when calling sin(double) (without the std prefix).

```
$ nano stdsin.cpp
 #include <cmath>
 int main(void)
 {
    double arg    = 1.0;
    double result = std::sin(arg);
    return 0;
 }
$ g++ -S stdsin.cpp
$ more stdsin.s
_main:
LFB127:
      pushq   %rbp
LCFI0:
      movq    %rsp, %rbp
LCFI1:
      subq    $16, %rsp
LCFI2:
      movabsq $4607182418800017408, %rax
      movq    %rax, -8(%rbp)
      movsd   -8(%rbp), %xmm0
      call    _sin
      movsd   %xmm0, -16(%rbp)
      movl    $0, %eax
      leave
      ret
```

Using std::sin(float)

When std::sin(float) is used, the _sinf function will be called by the processor.

```
$ nano floatstdsin.cpp
  #include <cmath>
  int main(void)
  {
      float arg    = 1.0;
      float result = std::sin(arg);
      return 0;
  }
$ g++ -S floatstdsin.cpp
$ more floatstdsin.s
_main:
LFB127:
        pushq   %rbp
LCFI3:
        movq    %rsp, %rbp
LCFI4:
        subq    $16, %rsp
LCFI5:
        movl    $0x3f800000, %eax
        movl    %eax, -4(%rbp)
        movss   -4(%rbp), %xmm0
        call    __ZSt3sinf
        movss   %xmm0, -8(%rbp)
        movl    $0, %eax
        leave
        ret
LFB87:
        pushq   %rbp
LCFI0:
        movq    %rsp, %rbp
LCFI1:
        subq    $16, %rsp
LCFI2:
        movss   %xmm0, -4(%rbp)
        movss   -4(%rbp), %xmm0
        call    _sinf
        leave
        ret
```

Using sin(float)

When sin(float) is used, the compiler will perform an implicit conversion to double and then call the _sin function.

```
$ nano floatsin.cpp
  #include <cmath>
  int main(void)
  {
      float arg    = 1.0;
      float result = sin(arg);
      return 0;
  }
$ g++ -S floatsin.cpp
$ more floatsin.s
_main:
LFB127:
        pushq   %rbp
LCFI0:
        movq    %rsp, %rbp
```

```
LCFI1:
        subq    $16, %rsp
LCFI2:
        movl    $0x3f800000, %eax
        movl    %eax, -4(%rbp)
        cvtss2sd        -4(%rbp), %xmm0
        call    _sin
        cvtsd2ss        %xmm0, %xmm0
        movss   %xmm0, -8(%rbp)
        movl    $0, %eax
        leave
        ret
```

Files

arithmetics.cpp	7.7 KB	03/26/2013	Knödlseder Jürgen
arithmetics.hpp	213 Bytes	03/26/2013	Knödlseder Jürgen