Log in to edit a copy. Download. Other published documents...

Untitled

2 hours ago by wbhart

Benchmark 2

Core 2	GMP	MPIR	K8	GMP	MPIR
Squaring			**Squaring**		
128 x 128	54446732	55370350	128 x 128	42226804	53762100
512 x 512	9320676	8172208	512 x 512	10295600	12481800
8192 x 8192	104065	101386	8192 x 8192	165214	168034
131072 x 131072	1620	1722	131072 x 131072	2562	2767
2097152 x 2097152	70.1	76.7	2097152 x 2097152	81.0	83.4
Multiplication			**Multiplication**		
128 x 128	54400830	55315582	128 x 128	45649804	53767752
512 x 512	7342969	8160125	512 x 512	10913936	12428363
8192 x 8192	71306	75225	8192 x 8192	114962	118476
131072 x 131072	1165	1289	131072 x 131072	1754	2075
2097152 x 2097152	47.8	52.9	2097152 x 2097152	52.3	63.3
Unbalanced			**Unbalanced**		
15000 x 10000	34790	36592	15000 x 10000	57365	59908
20000 x 10000	26612	28447	20000 x 10000	44094	47322
30000 x 10000	15707	16786	30000 x 10000	24894	27565
16777216 x 512	224	234	16777216 x 512	345	332
16777216 x 262144	9.01	9.89	16777216 x 262144	9.34	11.3
Division			**Division**		
8192 / 32	807890	675542	8192 / 32	1507178	1319736
8192 / 64	801590	686421	8192 / 64	1530848	1319605
8192 / 128	527984	377947	8192 / 128	931519	478680
8192 / 4096	118750	110330	8192 / 4096	189753	188476
8192 / 8064	1651613	1653280	8192 / 8064	2347446	2333862
131072 / 65536	1382	1371	131072 / 65536	2170	2229
8388608 / 4194304	4.05	4.39	8388608 / 4194304	5.27	6.01

16777216 / 262144	2.64	2.80	16777216 / 262144	4.12	4.46
GCD			**GCD**		
128 x 128	2172359	2259180	128 x 128	1436187	1364651
512 x 512	236660	212043	512 x 512	227624	196581
8192 x 8192	5846	5497	8192 x 8192	7833	6243
131072 x 131072	89.2	89.6	131072 x 131072	140	136
1048576 x 1048576	4.20	4.29	1048576 x 1048576	6.04	6.58
XGCD			**XGCD**		
128 x 128	1028823	693319	128 x 128	910501	338531
512 x 512	176163	109519	512 x 512	173108	62580
8192 x 8192	3720	2419	8192 x 8192	5400	3007
131072 x 131072	52.3	51.6	131072 x 131072	84.8	82.4
1048576 x 1048576	2.81	2.57	1048576 x 1048576	3.88	4.08
RSA			**RSA**		
512	16319	15019	512	20450	22476
1024	3048	3134	1024	4152	5078
2048	482	479	2048	783	882
Pi			**Pi**		
10000	482	489	10000	644	652
100000	20.7	22.7	100000	28.7	32.1
1000000	1.17	1.32	1000000	1.42	1.66
Overall	**1065**	**1043**	**Overall**	**1435**	**1446**

Contributors

- Jason Moxham - K8, Core2, Penryn, Nehalem, Pentium 4 assembly optimisation
- Brian Gladman - MSVC port
- Jason Martin - Core 2 assembly
- Pierrick Gaudry - AMD 64 assembly
- Anonymous Japanese contributor - assembly support
- Robert Gerbicz - Root testing
- William Hart - Sun, Apple, Cygwin, MSYS support, Toom 3/4/7 optimisation, Yasm switch, Fast Extended GCD
- Paul Zimmermann, Marco Bodrato - Toom 4/7
- Niels Moller - Fast GCD
- Paul Zimmermann, Pierrick Gaudry, Alexander Kruppa, Torbjorn Granlund - Fermat/Mersenne FFT
- Michael Abshoff - fix build issues, valgrinding, Sage integration
- Mariah Lennox - work on mpirbench, build farm maintenance

- Many others - contributions to build testing

Fast Code in MPIR

At the mpz level

```
struct
{
   mp_size_t _mp_size;
   mp_size_t _mp_alloc;
   mp_limb_t * _mp_data;
} __mpz_struct
```

```
typedef mpz_t __mpz_struct[1];
```

- Checking for zero

  Don't use mpz_cmp, use:

  ```
  if (mpz_sgn(a) == 0)
  ```

- Combined multiplication and addition:

  `mpz_addmul(x, a, b)`

  - set $x = x + ab$

  `mpz_submul(x, a, b)`

  - set $x = x - ab$

  `mpz_addmul_ui(x, a, b)`

  - set $x = x + ab$

  `mpz_submul_ui(x, a, b)`

  - set $x = x - ab$

If you did the addition separately in addmul_ui it would take 40% longer!

- Multiplication and division by powers of 2:

  `mpz_mul_2exp(x, a, exp)`

  - set $x = 2^{exp} a$

  `mpz_tdiv_q_2exp(x, a, exp)`

  - set $x = a/2^{exp}$

- Exact division is faster than division with remainder:

  `mpz_divexact(x, a, b)`

  - set $x = a/b$ assuming b divides a

  `mpz_divexact_ui(x, a, b)`

- Don't use mpz_import or mpz_export

  EVER!!

MPIR Tools

GMP compatibility

Use

```
./configure --enable-gmpcompat
```

and

```
make install-gmpcompat
```

if you wish to link MPIR against a library which is expecting GMP

Build from source - run make check

Binaries will be slower - build from source. But always do "make check".

Many functions now support SSE, SSE2, SSE3, LAHF, etc, where available - binaries aren't built with all optimisations.

Testing with make try

In /tests/devel/ you can "make try"

DEMO

Timing with make speed

In /tune/ you can "make speed"

DEMO

Performance tuning with make tune

In /tune/ you can "make tune -f 1000000"

DEMO

Developer documentation

Some developer documentation is available in /doc/devel

Fat binaries

```
./configure --enable-fat
```

Fat binaries will pick best assembly core at runtime - but know that there is a performance deficit for small operands

Enable asserts

```
./configure --enable-assert
```

Can help with debugging code, whether at mpz/mpq/mpf or the mpn level.

Enter the mpn's!

What is an mpn?

An mpn is a pair

```
{mp_limb_t * x, mp_size_t xn}
```

where x is an array of limbs, i.e. mp_limb_t's where xn is the number of limbs, i.e. an mp_size_t. There is NO MEMORY MANAGEMENT done for you.

Let's have a short example:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpir.h"
```

```
int main(void)
{
   mp_limb_t * a, * b;
   a = malloc(1001*sizeof(mp_limb_t));
   b = malloc(1000*sizeof(mp_limb_t));

   mpn_random2(a, 1000);
   mpn_random2(b, 1000);
   a[1000] = 0;

   for (long i = 0; i < 1000000; i++)
      a[1000] += mpn_addmul_1(a, b, 1000, 34567890);

   printf("a[1000] = %ld\n", a[1000]);

   free(a);
   free(b);

   return 0;
}
```

New assembly functions in MPIR

Lot's of new mpn functions are available on x86_64 in MPIR.

- mpn_divexact_by3(rp, sp, sn) - {rp, sn} computes {sp, sn} divided by 3 (carry is non-zero if exact division doesn't occur)
- mpn_divexact_byBm1of(rp, sp, sn, f, (B-1)/f) computes {rp, sn} = {sp, sn} / f where f is a divisor of B-1, e.g. 5, 17, 15, 51, ...
- mpn_addadd_n(rp, sp, tp, up, sn) computes cy, {rp, sn} = {sp, sn} + {tp, sn} + {up, sn}
- mpn_addsub_n(rp, sp, tp, up, sn) computes cy, {rp, sn} = {sp, sn} + {tp, sn} - {up, sn}
- mpn_subadd_n(rp, sp, tp, up, sn) computes bw, {rp, sn} = {sp, sn} - {(tp, sn} + {up, sn})
- mpn_addlsh1_n(rp, sp, tp, sn) computes cy, {rp, sn} = {sp, sn} + 2{tp, sn}
- mpn_sublsh1_n(rp, sp, tp, sn) computes br, {rp, sn} = {sp, sn} - 2{tp, sn}
- mpn_mul_2(rp, sp, sn, cp) computes cy, {rp, sn + 1} = cp[0]{sp, sn} + cp[1]B{sp, tn}
- mpn_addmul_2(rp, sp, sn, cp) computes cy, {rp, sn + 1} = {rp, sn + 1} + cp[0]{sp, sn} + cp[1]B{sp, tn}
- mpn_sumdiff_n(rp, sp, tp, up, tn) computes cy, {rp, sn} = {tp, tn} + {up, tn} and bw, {sn, tn} = {tp, tn} - {up, tn} (function returns 2*cy+bw)
- mpn_mul_basecase(rp, sp, sn, tp, tn) computes {rp, sn + tn} = {sp, sn} * {tp, tn}
- mpn_sqr_basecase(rp, sp, sn) computes {rp, 2*sn} = {sp, sn} * {sp, sn}

Also functions for and, andn, ior, iorn, nand, nior, xor, xnor and redc_basecase.

ASSERTS

- ASSERT(condition) will raise an assert if the condition is not met
- ASSERT_ALWAYS(condition) will always check the condition, even when asserts are not enabled
- ASSERT_CARRY(mpn_blah(...)) will assert that the function should return a nonzero carry
- ASSERT_NOCARRY(mpn_blah(...)) asserts that the function should return a zero carry - useful for mpn_divexact_1, mpn_divexact_by3, etc
- ASSERT_CODE(expr) for rolling your own assert code, i.e. expr can be anything, not just a condition
- ASSERT_MPN_ZERO_P(ptr, size) asserts that the given mpn is zero (size equal to 0 is allowed)
- ASSERT_MPN_NONZERO_P(ptr,size) assert that the given mpn is nonzero

MACROS

- MPN_CMP(result, xp, yp, size) sets result to -ve, 0 or +ve depending on whether {xp, size} is less than equal to or greater than {yp, size}, leading zero limbs are allowed
- ABS(xn), MIN(xn, yn), MAX(xn, yn) - just what they say
- POW2_P(n) - whether n is an exact power of 2 (or zero)
- MPN_PTR_SWAP(x, xn, y, yn) - swaps {x, xn} and {y, yn} by swapping the pointers x, y and the lengths xn, yn, not the data
- MPN_SRCPTR_SWAP(xp, xn, yp, yn) - for swapping mpn's which are source operands, i.e. those basically declared const
- MP_SIZE_T_SWAP (xn, yn) - swap two mp_size_t's
- MPN_COPY(d, s, n) - copy {s, n} to {d, n}
- MPN_COPY_INCR(d, s, n) - copy {s, n} to {d, n} incrementing memory locations as the copy proceeds
- MPN_COPY_DECR(d, s, n) - copy {s, n} to {d, n} decrementing memory locations as the copy proceeds
- MPN_SAME_OR_SEPARATE_P (d, s, n) returns nonzero if the mpns {d, n} and {s, n} are either the same or completely non-overlapping
- MPN_SAME_OR_INCR_P (d, s, n) returns nonzero if the mpns are the same or if it would be safe to copy one to the other whilst incrementing memory locations
- MPN_SAME_OR_DECR_P (d, s, n) returns nonzero if the mpns are the same or if it would be safe to copy one to the other whilst decrementing memory locations
- MPN_OVERLAP_P (d, dn, s, sn) returns nonzero if {d, dn} overlaps {s, sn}
- MPN_REVERSE(d, s, n) set {d, n} to the reverse of {s, n}
- MPN_NORMALIZE(d, dn) normalises the mpn {d, dn} - note you have to start with dn as an upper bound on the number of limbs with possible zero leading limbs
- MPN_NORMALIZE_NOT_ZERO(d, dn) - same as MPN_NORMALIZE except that it assumes the final dn will not be zero
- MPN_STRIP_LOW_ZEROS_NOT_ZERO(s, sn, low) - start with low equal to s[0], this function will increment s and decrement sn until s[0] is nonzero and it will set low to the new s[0], assumes that {s, sn} is not zero
- MPN_LOGOPS_N_INLINE(d, s1, s2, n, operation) - applies the given operation between the limbs of {s1, n} and {s2, n} and sets d to the result, e.g.

  ```
  MPN_LOGOPS_N_INLINE(d, s1, s2, n, d[__n] = s1[__n] & s2[__n])
  ```

- MPN_ZERO(s, sn) - set {s, sn} to zero
- mpn_store(d, n, val) - set all limbs of {d, n} to val
- mpn_com_n(d, s, n) - set {d, n} to the twos complement of {s, n}
- ADDC_LIMB(cy, w, x, y) - set cy, w = x + y where x and y are limbs
- SUBC_LIMB(bw, w, x, y) - set bw, w = x - y where x and y are limbs
- LIMB_HIGHBIT_TO_MASK(n) - returns a limb of all 1's if n has its top bit set, otherwise returns 0
- MPN_INCR_U(s, sn, incr) - set {s, sn} = {s, sn} + incr where incr is a single limb (assuming no carry)
- MPN_DECR_U(s, sn, incr) - set {s, sn} = {s, sn} - incr where incr is a single limb (assuming no borrow)

HINTS

- if LIKELY(condition) - will give a hint to the CPU that the branch is likely to be taken
- if UNLIKELY(condition) - will give a hint to the CPU that the branch is unlikely to be taken

Temporary allocation

MPIR has a temporary memory allocation system, like Pari. Here is an example of it in action:

```
mp_limb_t * ws;

TMP_DECL;

/* do whatever */

TMP_MARK;
ws = TMP_ALLOC_LIMBS (count);

/* Use ws however you like */

TMP_FREE;
```

The temporary allocation allocates memory on the stack if it is a small quantity and on the heap if it is big. But if you know you always want a small amount use TMP_SDECL, TMP_SMARK, TMP_SALLOC_LIMBS, TMP_SFREE. If you know you need a big amount all the time, or you want to avoid the stack overflowing, use TMP_BDECL, TMP_BMARK, TMP_BALLOC_LIMBS, TMP_BFREE.

Two's complement

One can use mpn's for negative numbers by making use of two's complement format and working to a fixed precision where overflow can't occur.

Here is a specific example. We pass in three mpn's to a function, all of the same length, assuming the first two are positive and the third is signed. We also suppose the top limb of each is zero upon entry.

```
void myfunction(mp_limb_t rp, mp_size_t * rn, mp_limb_t sp, mp_limb_t up, mp_limb_t vp, mp_size_t sn)
{
   mp_size_t size = ABS(sn);

   mpn_add_n(sp, sp, up, size);

   if (sn < 0)
      mpn_add_n(vp, sp, vp, size);
   else
      mpn_sub_n(vp, sp, vp, size);
   /* vp is now in twos complement format */

   mpn_lshift1(vp, vp, size);

   mpn_submul_1(rp, vp, size, 64);
}
```

BEWARE: right shift doesn't necessarily work because the sign bit will be shifted right. However one can use MPN_HIGH_BIT_TO_MASK to fix the top bits. Multiplication, division and divexact won't work on two's complement, so one needs to make the mpn's unsigned first, e.g. do mpn_com_n(sp, sp, size) and mpn_add_1(sp, sp, size, 1) to negate them if negative.

Memory management savings

Saving memory can make a huge difference in algorithms where caching becomes important. Here are some tips:

- Break large computations up into smaller chunks to improve locality - only helps if you use the same data over and over

- Allocate as little temporary memory as possible
- Try using some of the output space for temporary storage during the computation - this can also save a copy of data at the end of the computation if part of the result happens to end up in the right place
- In some cases it is possible to store everything except for carry limbs, which overlap some other temporary space. It is often more efficient to make a copy of the small bit that would be overlapped by the carry limbs, and add it back in later, than to allocate a large temporary space and copy the whole result over when done.

Using longlong.h

Ever wanted to get carries in C? Use longlong.h in the top level source directory of MPIR.

WARNING: just doing

```
#include "longlong.h"
```

is not enough, and will return

> **WRONG ANSWERS**

on some platforms.

One either has to first include gmp-impl.h or one has to do something like the following (works on all C99 systems we know of):

```
#include

#define UWtype mp_limb_t
#define UHWtype mp_limb_t
#define UDWtype mp_limb_t
#define W_TYPE_SIZE {insert number of bits of UWtype here}
#define SItype int32_t
#define USItype uint32_t
#define DItype int64_t
#define UDItype uint64_t

#define LONGLONG_STANDALONE

#define ASSERT(condition)

#include "longlong.h"
```

On a machine where a limb is two unsigned longs, you might set UDWtype to mp_limb_t and UWtype and UHWtype to unsigned long. You can define UHWtype to be half the size of UWtype if you want. You can define ASSERT to be whatever you want, but it must be defined. On a 32 bit machine UWtype should typically be USItype; on a 64 bit machine, UWtype should typically be UDItype

Once we have longlong.h included we have access to the following functions:

- umul_ppmm(high_prod, low_prod, multipler, multiplicand) - multiplication of two UWtypes, returning high and low limbs
- __umulsidi3(a,b) - multiply two UWtypes, returning a single UDWtype
- udiv_qrnnd(quotient, remainder, high_numerator, low_numerator, denominator) - division returning quotient and remainder. On some systems the high bit of denominator must be 1. If so, longlong.h sets UDIV_NEEDS_NORMALIZATION to 1.
- sdiv_qrnnd(quotient, remainder, high_numerator, low_numerator, denominator) - as for udiv_qrnnd, but with signed integers - quotient is rounded towards zero.

- count_leading_zeros(count, x) - sets count to the number of leading zeroes of x. It sets count to COUNT_LEADING_ZEROS_0 if x is 0. You must define that macro if you wish to use it.
- count_trailing_zeros(count, x) - as for count_leading_zeros, but counts the trailing zeroes.
- add_ssaaaa(high_sum, low_sum, high_addend_1, low_addend_1, high_addend_2, low_addend_2) - add two 2 limb quantities
- sub_ddmmss(high_difference, low_difference, high_minuend, low_minuend, high_subtrahend, low_subtrahend) - subtract two 2 limb quantities