GMP gem Performance

Performance analysis of the GMP gem

28 November 2012

written by Sam Rawlins

Contents

Introduction to the performance benchmarks

The benchmarking system used to test the performance of the gmp gem is inspired by, and uses parts of, gmpbench 0.2. http://gmplib.org/gmpbench.html. gmpbench consists of two parts:

- `multiply`, `divide`, `gcd`, `gcdext`, `rsa`, and `pi` are 6 small programs that use GMP to measure a specific piece of functionality. `multiply`, `divide`, `gcd`, and `gcdext` are the "base" benchmarks that test small pieces of functionality. `rsa` and `pi` are the "application" benchmarks that measure the performance of a larger concept implemented with GMP.
- `runbench` is a shell script that coordinates an execution of each of the benchmarking programs, applying a weight to the results of each, and yielding a total score for GMP on the current system.

The benchmarking system in the gmp gem uses Ruby versions of each of the 6 programs (actually, `pi` is still being ported), attempting to be as identical to their C code siblings. This system also just uses `runbench` unmodified from the original gmpbench suite.

Due to a few issues with Ruby 1.8.7, and the gmp gem itself, there are actually 3x different versions of the benchmark suite that use the gmp gem:

- `benchmark/gmp/bin_op` uses binary operators, such as `*`, on `GMP::Z` integers. Since `a * b` creates a new `mpz_t` that it returns, the benchmark programs are constantly creating new objects, which is not what the GMP benchmark programs do. The real problem that this creates is Ruby 1.8.7 running out of memory.
- `benchmark/gmp/gc` also uses binary operators, but invokes Ruby's garbage collector every 512 iterations of each test. This allows all of the benchmarks to complete in Ruby 1.8.7, but is still not the best comparison with GMP's benchmark programs.
- `benchmark/gmp/functional` uses the "functional", `GMP::Z` singleton methods to perform what would otherwise be binary operations. For example, `x * y` is replaced with `GMP::Z.mul(z,x,y)` in order to use `z` as the "return argument" through each iteration of a benchmark. In this version, `z` is only created once, before the benchmark begins measuring time.

Run the Benchmarks

In order to run a set of benchmarks (a directory containing `multiply`, `runbench`, etc.), just use the command:

```
./runbench -n
```

Next to each test case, program, and category, a score will be printed, which is iterations per second. For program, category, and overall scores, this represents a weighted geometric mean, and so should just be thought of more like a "score" than an actual real-world metric.

Ruby benchmarks

In addition to the above variations of the benchmark suite located in `benchmark/gmp`, there is one more variation of the benchmark suite that measure's Ruby's Bignum algorithms. This suite is located at `benchmark/ruby`.

New `Bignum` methods

Several methods provided in `GMP::Z` are not provided in `Bignum`, in Ruby's standard library. In order to attempt a vague comparison between `Bignum` and `GMP::Z`, a simple and "fast enough" version of the following methods is provided in `benchmark/ruby/ruby-enhancements`:

- `Bignum.gcdext`
- `Bignum.invert`
- `Bignum.powmod`
- `Bignum#[]=`
- `Bignum#gcd`

`Bignum.gcdext`, `Bignum.invert`, and `Bignum.powmod` are all borrwed from John Nishinaga, available at https://gist.github.com/2388745.

Modifications to `benchmark/ruby` benchmarks

Ruby's `Bignum` class is not advanced enough to handle several of the benchmark test cases, namely:

- `multiply 16777216 512` (Ruby's `Bignum` cannot raise 2 to a 16777216-bit number.)
- `multiply 16777216 262144` (Ruby's `Bignum` cannot raise 2 to a 16777216-bit number.)
- `divide 8388608 4194304` (Ruby's `Bignum` cannot raise 2 to a 8388608-bit number.)
- `divide 16777216 262144` (Ruby's `Bignum` cannot raise 2 to a 16777216-bit number.)

Ruby can raise 2 to approximately 4,194,000.

In the `benchmark/ruby` suite, these have been removed, so that summary scores can still be produced. In order to compare these summary scores against `GMP::Z` benchmarks, there is also a `benchmark/gmp/reduced` suite that uses the same test cases. `benchmark/gmp/reduced` is the only test suite that should be compared against `benchmark/ruby` (or, with some work, one can manually calculate the weighted geometric means, using the same method found in `runbench`.

Results

Raw benchmark results can be found in `benchmark/benchmark-results.ods`, and OpenOffice spreadsheet. Here I show some interpreted results.

Ruby v Ruby

I benchmarked three different versions of Ruby's Bignum implementation: Ruby 1.8.7, Ruby 1.9.3, and Ruby 2.0.0-preview2 (the latest version of Ruby 2.0 at the time of the tests). These tests only measured Ruby's Bignum, and do not use GMP at all. Ruby 1.9.3 and Ruby 2.0.0-preview2 performed very similarly, within 5% of each other in most cases. The interesting result in this test is Ruby 1.8.7 v Ruby 1.9.3. With the exception of `divide`, 1.9.3 outperformed 1.8.7, and often dramatically:

Program	Ruby 1.8.7	Ruby 1.9.3	1.9.3 over 1.8.7*
multiply	1.98e+03	4.89e+03	2.47
divide	2.45e+04	2.32e+04	0.95
gcd	2.23e+01	3.08e+01	1.38
gcdext	6.41e+00	1.05e+01	1.64
[base]	8.34e+04	1.27e+03	1.52
rsa	1.17e+02	1.45e+02	1.24
[app]	1.17e+02	1.45e+02	1.24
[bench]	3.12e+02	4.29e+02	1.37

* Calculated as $\frac{\text{1.9.3 score}}{\text{1.8.7 score}}$ so that 2.47 means "2.47 times as fast" or equivalently "1.47 times faster."

We can look at individual tests to see where 1.9.3 specifically improves over 1.8.7:

- Firstly, in the `multiply` test, 1.9.3 and 1.8.7 are actually neck-and-neck for most of the tests, until we get to multiplying "very large" numbers together. Multiplying a 131072-bit number by a 131072-bit number is ~5 times as fast in 1.9.3 vs 1.8.7. Multiplying two 2,097,152-bit numbers together is 22x as fast!
- Second, the reverse phenomenon happens with `gcd` and `gcdext`, where 1.9.3 outperforms 1.8.7 at 3.4x and 5.1x, respectively, when using 128-bit inputs. With 512-bit inputs and above, however, the speedup fades to nothing. This suggests that the algorithms used in `Bignum` do not change, but the overhead costs are lower in Ruby 1.9.3. One can understand that when GCDing smaller numbers, the overhead of looping, making method calls, etc. is a larger percentage of the work being done, but when GCDing larger numbers, the overhead dissolves into almost nothing.

gmp gem: Binary Operators v Functional Operators

It is beneficial to look at the two different forms of methods sometimes offered: binary operators (such as `GMP::Z#+` which is used like `c = a + b`) and "functional" operators (such as `GMP::Z.add` which is used like `GMP::Z.add(c, a, b)`). At this time, only the `GMP::Z#*` binary operator is available as a functional operator (`GMP::Z.multiply`), which can change gears to a squaring algorithm if it detects that the operands are equal. (Squaring is thus faster than multiplication.) We can look at those results below:

Test Case	Bin Op	Functional	Functional over Bin Op
multiply(128)	9.30e+05	4.39e+06	4.72
multiply(512)	9.19e+05	3.10e+06	3.37
multiply(8192)	7.93e+04	9.24e+04	1.17
multiply(131072)	1.66e+03	1.75e+03	1.06
multiply(2097152)	6.24e+01	6.20e+01	0.99
multiply(128, 128)	9.57e+05	4.41e+06	4.61
multiply(512, 512)	8.40e+05	2.78e+06	3.31
multiply(8192, 8192)	5.44e+03	5.92e+04	1.09
multiply(131072, 131072)	1.20e+03	1.23e+03	1.02
multiply(2097152, 2097152)	4.08e+01	4.00e+01	0.98
multiply(15000, 10000)	2.95e+04	3.19e+04	1.08
multiply(20000, 10000)	2.32e+04	2.51e+04	1.08
multiply(30000, 10000)	1.54e+04	1.60e+04	1.04

We can see the effects of allocating new `GMP::Z` objects every iteration of the benchmark loop. When we are squaring or multiplying "small," 128-bit or 512-bit numbers, allocating objects and garbage collection can slow down the computation by three- or four-fold, if the computation is multiplying numbers (using `GMP::Z#*`) in a tight loop.

Once we get to squaring (or multiplying) 8192-bit numbers, however, the time spent inside GMP becomes great enough, that garbage collection and object allocation fades into the background. Above this size, binary operators can be only 17% slower. When squaring 131072-bit numbers, or multiplying 10000-bit numbers, binary operators are 8%, or less, slower.

GNU Multiple Precision Arithmetic Library, without Ruby

Here I present some raw benchmark results of GMP 5.0.5, using the original gmpbench 0.2 software. These tests do not involve the Ruby interpreter in any way.

Program	GMP 5.0.5	GMP 5.0.5, reduced
multiply(128, 128)	4.55e+07	4.55e+07
multiply(2097152, 2097152)	4.09e+01	4.09e+01
multiply(16777216, 262144)	9.97e+00	n/a
multiply	2.15e+04	5.58e+04
divide(8192, 32)	7.23e+05	7.23e+05
divide(16777216, 262144)	4.98e+00	n/a
divide	1.93e+04	2.77e+05
gcd	3.68e+03	3.68e+03
gcdext	2.22e+03	2.22e+03
[base]	1.06e+04	3.53e+04
rsa	2.68e+03	2.68e+03
[app]	2.68e+03	2.68e+03
[bench]	5.33e+03	9.73e+03

In both columns of results, the `pi` results have not been presented, as they cannot be compared to anything in Ruby, yet. In the second column, we also reduce the test by not including the `multiply` and `divide` tests that Ruby's Bignum algorithms cannot handle.

These results have been included to primarily show the results of two tests that Ruby's Bignum is unable to compute: `multiply(16777216, 262144)` and `divide(16777216, 262144)`. Whereas GMP can multiply two 128-bit numbers together more than 45 million times per second, and even two 2097152-bit numbers more than 40 times per second, it can only multiply a 16777216-bit and a 262144-bit number about 10 times per second.

At the same time, pure GMP works hard to divide one huge number by another: it can divide an 8192-bit by a 32-bit number more than 700,000 times per second, but only divide a 16777216-bit by a 262144-bit number about 5 times per second.

One can also get a grasp of how why the *geometric* mean is important when computing the scores for, say, the `multiply` or the `divide` program. Removing the two slowest test cases from the `multiply` set raises the geometric mean from about 21,500 to about 55,800 multiplications per second. An arithmetic mean would produce scores that might be difficult to compare side-by-side.

Ultimately, the reduced test cases change the overall benchmark score from about 5000 to about 10000. This shows why, ultimately, none of the test scores here should be compared with scores from the original, full gmpbench 0.2 suite. All of the scores analyzed in this document can only be used to compare *some* other scores also analyzed in this document.

GMP v gmp gem v Ruby Bignum

Now that we have all of the required reduced test results, and the known limitations of Ruby's Bignum and the gmp gem's binary operators, we can do a proper comparison between raw GMP, the gmp gem, and Ruby's Bignum. First, a table with some summarized results, and no direct comparisons:

Program	GMP	gmp gem	Ruby Bignum
multiply	5.58e+04	2.17e+04	4.89e+03