Solidity gas optimization

This repository contains the solution to a solidity optimization problem. The baseline contract to optimize is given in `contracts/MBTestBaseline.sol` and the goal is to reduce gas costs on the EVM.

Several solutions with different tradeoffs are proposed. They are available in the `contracts/MBTestOptimized*.sol` files. These solutions are presented later in this document.

Note: This document is available in compiled **PDF format** in the README.pdf file.

Install

```
yarn install
```

Tests

You can run the tests with these commands:

```
yarn hardhat run scripts/test-v1.js
yarn hardhat run scripts/test-v2.js
yarn hardhat run scripts/test-v3.js
yarn hardhat run scripts/test-v4.js
```

I suggest to write the test output to a file:

```
yarn hardhat run scripts/test-v1.js > test-v1.txt
```

And then to tail the file for visual feedback (either after having put the process in background or using another terminal window):

```
tail -F test-v1.txt
```

Inspecting for errors can be done easily using grep:

```
grep "Error with" test-v1.txt
```

Problem statement

The contract `MBTestBaseline.sol` computes the formula:

$$\left(1+\frac{1}{x}\right)^{\frac{a}{b}}$$

Using the Maclaurin Binomial Series:

$$(1+y)^{\alpha}=\sum_{k=0}\binom{\alpha}{k}y^{k}=1+\frac{\alpha}{1!}y+\frac{\alpha(\alpha-1)}{2!}y^{2}+\frac{\alpha(\alpha-1)(\alpha-2)}{3!}y^{3}+...$$

Optimize the code to reduce gas costs.

The full problem statement can be found in the problem-statement.pdf file.

Thoughts & solution

First, we note that the implicit constraint `a <= b` is enforced by the baseline implementation since line 106 of `MBTestBaseline.sol` contains `n.sub(1).mul(b).sub(a)` which is executed at least once for `n = 2` yielding `b.sub(a)` which will trigger `SafeMath: substraction overflow` as soon as `a > b`. The assumption that `a <= b` is also used in the baseline implementation to compute the `sign` variable.

A word on methodology

There is a wide array of factors that impact gas costs: address of the contract, ordering of the functions within a solidity contract, and many more. On top of these, most node implement estimateGas in an approximate manner that is not as reliable as executing the transaction and looking at the gas it used. Therefore, the most correct way to perform gas comparison is to start two exact clones of the same blockchain and deploy the baseline and optimized contracts at the same address.

In this exercise, we use the provided hardhat framework which does not allow this degree of precision. Given the nature of the task, this is good enough. In order to prevent gas costs difference due to methods ordering within a solidity contract, we always deploy two contracts (the baseline and the optimized variant) instead of deploying a single contract with both baseline and optimized methods.

As a general rule, optimizations should always start with work that has the most impact, and then progressively dive deeper and deeper into the minor changes. There is a law of diminishing returns at play here, both in terms of time spent optimizing and in terms of readability costs. Indeed, in solidity, optimized code (with or without assembly) is often less readable and duplicated.

Our approach is as follows:

1. Make sure the overall algorithm is optimal before attempting low level optimizations
2. Then, optimize what costs more first:
    1. Storage first
    2. Calldata second
    3. Memory third
    4. And then minor code tweaks with regards to opcodes costs

Last but not least, always compile with **optimizations enabled** to create valid benchmarks.

Improving the overall algorithm first

Looking at the provided baseline contract, we can see that some values are recomputed several times among the different calls to `getCoefficient(i, x, a, b)`. The first step to optimize gas costs is therefore to remove this duplicated work. At the same time, we can try to express the Maclaurin binomial series in a way that further reduces gas costs.

Gas costs for opcodes on the EVM can be found in Appendix G and H of the Yellow Paper and Geth implementation. Exact costs may change with time so benchmarking is always necessary for a high degree of precision.

Of interest here are the costs for `ADD/SUB` (verylow: 3 gas) and `MUL/DIV` (low: 5 gas). So one `ADD/SUB` is preferred to one `MUL/DIV`, but one `MUL/DIV` is preferred to two or more `ADD/SUB`. In addition, each additional word of memory costs 3 gas and usual memory manipulation operations also cost 3 gas. It is interesting to note that unlike for traditional GPU and CPU, on the EVM the opcodes `MUL` and `DIV` cost the same amount of gas.

Reworking the Maclaurin binomial series on paper, we find:

$$\left(1+\frac{1}{x}\right)^{\frac{a}{b}} = 1 + \frac{a}{bx} + \frac{a}{bx}\frac{(a-b)}{2bx} + \frac{a}{bx}\frac{(a-b)}{2bx}\frac{(a-2b)}{3bx} + ...$$

Where at each step, the additional colored fraction can be computed from the previous one by substracting b in the numerator and adding bx in the denominator:

$$\text{next}\left(\frac{(a-b)}{2bx}\right) = \frac{(a-b)-b}{2bx+bx} = \frac{(a-2b)}{3bx}$$

We use the following vocabulary: the series is a sum of **terms** and each term is a product of **factors**, as shown on the equation below.

$$\left(1+\frac{1}{x}\right)^{\frac{a}{b}} = 1 + \frac{a}{bx} + \frac{a}{bx}\frac{(a-b)}{2bx} + \underbrace{\frac{a}{bx}\frac{(a-b)}{2bx}\underbrace{\frac{(a-2b)}{3bx}}_{factor}}_{term} + ...$$

Turning this equation into reccurence relations suitable for implementation yields the following, which can be implemented using 4 variables:

```
(Implementend in Optimized v2)

total = sum(term(i) for i <= precision)

term(i) = term(i-1) * factor_numerator(i) / factor_denominator(i)
```

```
factor_numerator(i) = factor_numerator(i-1) - b
factor_denominator(i) = factor_denominator(i-1) + bx
```

However, in the baseline implementation, each term is computed by first computing its numerator, then its denominator and then performing a single division. Since the ordering of multiplications and divisions matters for numerical accuracy, we have implemented this approach in the Optimized v1 contract. The recurrence relations are as follow, and require an additional variable:

```
(Implemented in Optimized v1)

total = sum(term(i) for i <= precision)

*term(i) = term_denominator(i) / term_numerator(i)
term_numerator(i) = term_numerator(i-1) * factor_numerator(i)
term_denominator(i) = term_denominator(i-1) * factor_denominator(i)

factor_numerator(i) = factor_numerator(i-1) - b
factor_denominator(i) = factor_denominator(i-1) + bx

The star * denotes a notation shortcut and not a real variable.
```

Comparison of v1 and v2

The Optimized v1 contract keeps the same ordering of multiplication and division as the baseline contract and therefore produces the exact same results. When compared to the baseline implementation, gas savings increase with the precision and can be as high as 80% for a precision of 19. A Precision of 20 or more triggers an overflow in both the baseline implementation and the v1 contract.

The Optimized v2 contract, on the other hand changes the ordering of multiplication and division when compared to the baseline contract. Doing so saves (1) a multiplication and (2) a variable, further improving the gas savings. By dividing before multiplying, the v2 contract is less sensitive to overflows and allows for a greater precision, above 20.

Because changing the order of the numerical operations leads to different rounding schemes, the v2 contract does not output the exact same results as the baseline contract. Keeping in mind that the baseline contract approximates a mathematical formula, this is not an issue provided the difference is near zero. In practice, the observed difference in output between the baseline and the v2 contract is less than 5 units. Compared to the decimals values of 1000000000000000000, this difference is negligible.

Results

The maximal precision that we can reach depends on the values of the other parameters as they influence how fast the numerator and denominator grow. Using the following values:

```
a = 250
b = 365
invRate = 2
decimals = 1000000000000000000
```

The results when comparing the Optimized v1 contract to the baseline contract are as follow:

```
Batch 1: v1 vs baseline:
 Precision = 1 savings:  -2 %  |  -617 gas units
 Precision = 2 savings:  -3 %  |  -874 gas units
 Precision = 3 savings:  -9 %  |  -2462 gas units
 Precision = 4 savings:  -18 %  |  -5648 gas units
 Precision = 5 savings:  -26 %  |  -9414 gas units
 Precision = 6 savings:  -34 %  |  -14038 gas units
 Precision = 7 savings:  -41 %  |  -19544 gas units
 Precision = 8 savings:  -47 %  |  -25914 gas units
 Precision = 9 savings:  -53 %  |  -33170 gas units
 Precision = 10 savings:  -58 %  |  -41291 gas units
 Precision = 11 savings:  -62 %  |  -50304 gas units
 Precision = 12 savings:  -65 %  |  -60187 gas units
 Precision = 13 savings:  -68 %  |  -70968 gas units
 Precision = 14 savings:  -71 %  |  -82623 gas units
 Precision = 15 savings:  -73 %  |  -95182 gas units
 Precision = 16 savings:  -75 %  |  -108620 gas units
 Precision = 17 savings:  -77 %  |  -122973 gas units
 Precision = 18 savings:  -79 %  |  -138209 gas units
```

In other words, on this specific set of parameters, the savings for a precision of 18 are 79% when compared to **baseline**. For higher precisions, both contracts trigger a multiplication overflow.

Using the same parameters, the results of comparing the Optimized v2 contract to the Optimized v1 contract are as follow:

```
 Precision = 1 savings:  0 %  |  0 gas units
 Precision = 2 savings:  0 %  |  0 gas units
 Precision = 3 savings:  0 %  |  0 gas units
 Precision = 4 savings:  0 %  |  -143 gas units
 Precision = 5 savings:  -1 %  |  -273 gas units | delta = 1
 Precision = 6 savings:  -1 %  |  -403 gas units
 Precision = 7 savings:  -1 %  |  -533 gas units
 Precision = 8 savings:  -2 %  |  -663 gas units | delta = 1
```

```
Precision = 9 savings:  -2 %  |  -793 gas units
Precision = 10 savings:  -3 %  |  -923 gas units
Precision = 11 savings:  -3 %  |  -1053 gas units
Precision = 12 savings:  -3 %  |  -1183 gas units | delta = 1
Precision = 13 savings:  -4 %  |  -1313 gas units | delta = 1
Precision = 14 savings:  -4 %  |  -1443 gas units | delta = 1
Precision = 15 savings:  -4 %  |  -1573 gas units | delta = 1
Precision = 16 savings:  -4 %  |  -1703 gas units | delta = 1
Precision = 17 savings:  -5 %  |  -1833 gas units | delta = 1
Precision = 18 savings:  -5 %  |  -1963 gas units | delta = 2
```

In other words, on this specific set of parameters, the savings for a precision of 18 are improved by a further 5% when compared to **Optimized v1**. For higher precisions, the Optimized v1 contract triggers a multiplication overflow but the Optimized v2 contract does not and can compute results with a precision as high as 30 (Maybe even more, I have not tested beyond 30).

The additional delta value at the end of each lines indicates the difference in output. Here the maximal observed delta is 2 which is negligible when compared to the decimals of 1000000000000000000.

Real accuracy

Since the baseline contract and the Optimized v2 contract do not output the same results, it is interesting to compare their output to the true value (which was computed using wolfram alpha). The table below gives the number of correct decimals for each precision levels, using the same parameters as before. Once the precision is high enough, the baseline contract triggers an overflow.

Precision	Baseline (correct decimals)	Optimized v2 (correct decimals)
4	3	3
10	6	6
16	8	8
18	9	9
21	Overflow	10
26	Overflow	11
27	Overflow	12

Hence, this table shows that Optimized v2 allows for a greater accuracy than

the baseline contract.

Compressing data

From the Yellow paper, we can see that:

- an additional word in storage costs 20000 gas,
- an additional word in calldata costs 2176 gas and
- an additional word in memory costs 3 gas.

Therefore, optimizing storage and calldata use is important. Here the baseline contract does not use storage, but we can save some gas by compressing the parameters sent to the contract to reduce calldata usage.

For the sake of demonstration, we will assume that `k` fits in a `uint128` and that the other arguments (`x`, `a`, `b`, `precision`) each fit into an `uint16`. These assumptions seems reasonable: for instance, they impose the constaint `a,b <= 65535` which is ok since we expect `b=365` and `a<=b`.

With these constaint, we can encode the arguments into a single `uint256`, thus saving words on the calldata. The encoding/decoding is shown below:

Encoding in javascript:

```
const args = BigNumber.from(0)
    .mul(two.pow(128))
    .add(k)
    .mul(two.pow(16))
    .add(x)
    .mul(two.pow(16))
    .add(a)
    .mul(two.pow(16))
    .add(b)
    .mul(two.pow(16))
    .add(precision);
```

Decoding in solidity:

```
uint256 prec = args & uint16(int16(-1));
args >>= 16;
uint256 b = args & uint16(int16(-1));
args >>= 16;
uint256 a = args & uint16(int16(-1));
args >>= 16;
uint256 x = args & uint16(int16(-1));
args >>= 16;
uint256 k = args & uint128(int128(-1));
```

Looking at the optimized-IR YUL code generated by the solidity compiler for the decoding of our variable shows that the compiler inlined our bit shifts operations.

```
let value := calldataload(4)
let _2 := 0xffff
let vloc := fun_maclaurinBinomial_192(
    and(shr(64, value), 0xffffffffffffffffffffffffffffffff),
    and(shr(48, value), _2),
    and(shr(32, value), _2),
    and(shr(0x10, value), _2),
    and(value, _2))
```

Gas savings: This optimization allows to save roughly 1% when compared to Optimized v2.

Solidity 0.8

In solidity 0.8 and above, checked arithmetic is enabled by default so the SafeMath library is not needed. Also, newer versions of solidity have better optimizations implemented into the compiler. Migrating the v2 code to solidity 0.8.10 further reduces the gas costs by up to 10%. As shown in the test run below:

In the tests output block below, each line shows the id of the testcase, the precision used and the savings of Optimized v4 (using solidity 0.8.10) when compared to Optimized v3. Note that both contracts output the exact same results.

```
Testcase 110 (prec=1) -EXACT- savings:  0 %  |  32 gas units
Testcase 111 (prec=3) -EXACT- savings:  -3 %  |  -752 gas units
Testcase 112 (prec=5) -EXACT- savings:  -3 %  |  -979 gas units
Testcase 113 (prec=7) -EXACT- savings:  -4 %  |  -1283 gas units
Testcase 114 (prec=9) -EXACT- savings:  -5 %  |  -1586 gas units
Testcase 115 (prec=11) -EXACT- savings:  -6 %  |  -1890 gas units
Testcase 116 (prec=13) -EXACT- savings:  -7 %  |  -2194 gas units
Testcase 117 (prec=15) -EXACT- savings:  -7 %  |  -2498 gas units
Testcase 118 (prec=17) -EXACT- savings:  -8 %  |  -2802 gas units
Testcase 119 (prec=19) -EXACT- savings:  -8 %  |  -3107 gas units
Testcase 120 (prec=21) -EXACT- savings:  -9 %  |  -3411 gas units
```

When compared to baseline directly, this v4 improves the gas costs by up to 83%, as shown in the test output below.

```
Testcase 77 (prec=1) -EXACT- savings:  -4 %  |  -1066 gas units
Testcase 78 (prec=3) -EXACT- savings:  -13 %  |  -3695 gas units
Testcase 79 (prec=5) -EXACT- savings:  -31 %  |  -11146 gas units
Testcase 80 (prec=7) -EXACT- savings:  -46 %  |  -21840 gas units
Testcase 81 (prec=9) -EXACT- savings:  -57 %  |  -36029 gas units
Testcase 82 (prec=11) EPSILON savings:  -66 %  |  -53727 gas units | delta = 1
Testcase 83 (prec=13) EPSILON savings:  -72 %  |  -74955 gas units | delta = 2
Testcase 84 (prec=15) EPSILON savings:  -77 %  |  -99733 gas units | delta = 2
Testcase 85 (prec=17) EPSILON savings:  -80 %  |  -128088 gas units | delta = 2
```

```
Testcase 86 (prec=19) EPSILON savings:  -83 %  |  -160046 gas units | delta = 2
```

Other ways to improve the code

- It is not excluded that another mathematical formula than the Maclaurin binomial series would allow for more optimized computations.
- The versions implemented so far use safe arithmetic for all operations. A finer analysis might reveal that some checks can safely be bypassed.
- Since solidity 0.8.10 implements checked arithmetic on signed integer, using int256 instead of uint256 would allow to remove the manual computation of the `sign` variable and enable further gas savings.
- Looking further into the bytecode generated by the solidity compiler would give an outline of the optimizations that direct use of assembly allows.

A word on assembly

With regards to gas costs, inline assembly in solidity is mostly useful to bypass some solidity conventions (smaller functions selectors, unchecked array accesses, smaller memory layout) which are not of major importance given the nature of this contract.

In any case, relying on assembly code here would only yield marginal benefits compared to the 80%+ savings outlined above. There is always a tradeoff between gas savings, human time spent coding and readability of the contract.

Conclusion

We have explored 4 optimized versions, each improving upon the previous one.

The first version used an optimized formula to improve the gas costs by ~80% when compared to baseline. The second version changed operations ordering to remove a variable and further improve the gas costs by 5% when compared to v1. The third version implemented custom arguments packing and improved the gas costs by 1% when compared to v2. The fourth version used the latest solidity version to improve the gas costs by a further 9% when compared to v3. Overall, version 4 improved the gas costs by 83% when compared to baseline directly.

We have checked numerical accuracy of the results against wolfram alpha and showed that versions 2 and above allow for a greater precision than the baseline.