Gas Optimisation

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming. - Donald Knuth

Optimisation Process

Be clear what / when / where you are optimising for (deployment / runtime)
Decide on an acceptable level of performance

Security is #1 concern !!!

1.Make the code correct
2. Prove code correctness with unit tests
3. Measure the performance (context is important here)
4. Pick the change that will have the most impact
5. Make the code changes
6. Go to 1

It is important to measure the performance, don’t just guess
Making trade offs according to the application context is important\

Gas Optimisation areas in solidity

- Storage
- Variables
- Functions
- Loops

Storage

Saving one slot that is a word of 256 bits to Storage (SSTORE) is 20,000 gas when you initially set it from zero to non-zero.
5,000 gas is spent when an already used Storage slot is rewritten.
Reading a Storage slot using SLOAD takes 200 gas.

Storage variable declaration doesn’t cost anything, as there’s no initialization.

Use writing and reading Storage variables carefully.
In most cases doing preliminary calculations in local variables and storing the final value to Storage is more cost-effective than constantly updating a Storage variable

Refunds

Free Storage slots by zeroing corresponding variables as soon as you don't need them anymore. This will refund 15000 of gas. The fact gas is refunded is even used in a project that aims to "tokenize" the gas by occupying slots at moments of relatively low gas price and freeing the slots when gas price is high. This gas difference can be then consumed by any smart contract decreasing the total costs of a call.

Thoroughly removing a contract (employing the SELFDESTRUCT opcode) rebates 24,000 gas. The completion of the contract execution is the only time when refunds occur, thus contracts are unable to pay for themselves. In addition, a refund must not surpass half the gas that the ongoing contract call uses.

Data Types and Packing

- Use bytes32 whenever possible, because it is the most optimized storage type.
- Type bytes should be used over byte[].
- If the length of bytes can be limited, use the lowest amount possible from bytes1 to bytes32.

Strings

Using bytes32 is cheaper than using the string type.

Packing variables

Pack several blocks of information into one Storage slot if they are smaller than a word of 32 bytes. This can give significant savings. Specifically if according to the application logic, they are usually updated and accessed together. For example, a structure of 2 uint128 can be stored in one slot in a mapping instead of storing them separately.

For example take this code

```
Struct Data {
uint64 a;
uint64 b;
uint128 c;
uint256 d;
}

Data public data

constructor (uint64 _a,uint64 _b, uint128 _c, uint256 _d) public {
    Data.a = _a;
    Data.b = _b;
    Data.c = _c;
    Data.d = _d;
}
```

The SStore instruction is performed twice, once to store a,b,c and a second time to store d (the solc optimiser is able to work out this optimisation)

```
[         a       ] [        b        ] [                  c
[8 bytes / 64 bits] [8 bytes / 64 bits] [        16 bytes / 128 bits

[                                      d
[                             32 bytes / 256 bits
```

Question

So generally is modifying a uint8 cheaper than modifying a uint256 ? ….

no…
storing a small number in a uint8 variable is not cheaper than storing it into a uint256 variable,
because for storing, any smaller data is padded with zeros to fill the 32 bytes, requiring additional operations from the EVM and additional gas.

Inheritance

When we extend a contract, the variables in the child can be packed with the variables in the parent.
The order of variables is determined by C3 linearization. For most applications, all you need to know is that child variables come after parent variables.

Memory versus Storage

Memory is generally cheaper than storage but

Copying between the memory and storage will cost some gas, so don't copy arrays from storage to memory, use a storage pointer
(but beware of subtle bugs when doing this, a Defi project was recently rekt by this)

Obviously some data needs to persist between function calls
The cost of memory is … complicated, you "buy" it in chunks, the cost of which will go up quadratically after a while.

Try adjusting the location of your variables playing with the keywords "storage" and "memory". Depending on the size and number of copying operations between Storage and Memory, switching to memory may or may not give improvements. All this is because of varying memory costs. So optimising here is not that obvious, and every case has to be considered individually.

Variables

- Use events rather than storing data
- Avoid public variables
- Inefficient use of memory arrays
- It may be good to avoid using storage, by employing memory arrays. If the size of the array is exactly known, fixed size memory arrays can be used to save gas.
- Inefficient use of global variables
- Inefficient use of return values
- A simple optimization in Solidity consists of naming the return value of a function. It is not needed to create a local variable then.

Mapping vs. Array

Most of the time it will be better to use a mapping instead of an array because of its cheaper operations.
However, an array can be the correct choice when using smaller data types. Array elements are packed like other storage variables and the reduced storage space can outweigh the cost of an array's more expensive operations. This is most useful when working with large arrays.

Functions

Calling functions is relatively cheap (it is just a jump instruction), but it can degrade the compiler's attempts at storage optimisation.

Memory, calldata and function parameters

Storing the input parameters in memory costs gas. For all public functions, the input parameters are copied to memory automatically.
If a function is only called externally, it should be explicitly marked as external, in a way that these parameters are not stored into memory but are read from call data directly.
This can save gas when the function input parameters are huge.

Function order

See Function order article (https://medium.com/joyso/solidity-how-does-function-name-affect-gas-consumption-in-smart-contract-47d270d8ac92)

Each position will have an extra 22 gas, so

- Reduce public variables
- Put often called functions earlier

Tool to optimise function name (https://emn178.github.io/solidity-optimize-name/)

Compress Input Data

See the example in Compress Input Data Article (https://medium.com/joyso/solidity-compress-input-in-smart-contract-d9a0eee30fe0)

they go from these function parameters

- uint256 amountSell,
- uint256 amountBuy,
- address tokenSell,
- address tokenBuy,
- address user,
- uint256 nonce,
- uint256 gasFee,
- uint256 takerFee,
- uint256 makerFee,
- uint256 joyPrice,
- bool isBuy,
- uint8 v,
- byte32 r,
- byte32 s

to these

- uint256 amountSell,
- uint256 amountBuy,

- uint256 data,
- uint256 gasFee,
- byte32 r,
- byte32 s

without losing functionality by packing many of the parameters in the data field

View Functions

You are not paying for view functions that aren't transactions. But this doesn't mean they aren't consuming gas, they do. It is just that it is free when executed on the local EVM. However, if a view function is called in a transaction, all the gas matters.

Loops

Due to the expensive SLOAD and SSTORE opcodes, managing a variable in storage is much more expensive than managing variables in memory. For this reason, storage variables should not be used in loops.

For example

```
uint num = 0;
function expensiveLoop(uint x) public {
  for(uint i = 0; i < x; i++) {
    num += 1;
  }
}
```

do this instead

```
uint num = 0;
function lessExpensiveLoop(uint x) public {
  uint temp = num;
  for(uint i = 0; i < x; i++) {
    temp += 1;
  }
  num = temp;
}
```

- Optimise loops to minimise the number and cost of instructions within the loop.
- Take unnecessary values out of the loop
- Predict values if possible
- Reduce the number of iterations by for example breaking out of loop as soon as possible

- Try to avoid unbounded loops

Miscellaneous Optimisations

- Remove dead code
- Opaque predicate
- Use optimization and set the counter to high values or leave the default 200. Setting it to 1 can be useful in a rare case when it's important to optimize contract deployment, but not subsequent functions call.
- Use Libraries (wisely)
  When a public function of a library is called, the bytecode of that function is not made part of a client contract. Thus, complex logic should be put in libraries for keeping the contract size small. But there is a cost for calling the library function.
- Require and Assert
  Use "require" for all runtime conditions validations that can't be prevalidated on the compile time. And "assert" should be used only for static validations that normally never fail in a properly functioning code. A failing "assert" consumes all the gas available to the call, while "require" doesn't consume any.
- EXTCODESIZE is quite expensive, this is used for calls between contracts, The only option we see to optimize the code in this regard is minimizing the number of calls to other contracts and libraries.
- Hash functions
  - keccak256: 30 gas + 6 gas for each word of input data
  - sha256: 60 gas + 12 gas for each word of input data
  - ripemd160: 600 gas + 120 gas for each word of input data
  - So if you don't have specific reasons to select another hash function, just use keccak256
- Short Circuiting

For 2 functions as follows
f(x) is low cost
g(y) is expensive

Ordering should go

f(x) || g(y)
f(x) && g(y)

Events

Here’s the formula for a LOG gas cost:

k + unindexedBytes * a + indexedTopics * b
where

k = 375
a = 8
b = 375

(Note also that if you use a bigger than 256 bit type for an indexed event topic, like bytes[1000] or something, then you still only pay 375, because in this case, only the Keccak hash of the value is actually indexed.)

More Advanced Techniques

Compressing Variables using Assembly Code

In general, we can compress the variables so that fewer SSTORE operations are performed. Such compression can be done manually, The following code shows how to compress 4 uint64 variables into a 256 but memory slot

```
function encode(uint64 _a,uint64 _b,uint64 _c,uint64 _d)
internal pure returns (bytes32 x){
assembly {

let y:= 0;
mstore(0x20, _d)
mstore(0x18, _c)
mstore(0x10, _b)
mstore(0x8, _a)
x:= mload(0x20)
}
}
```

Using Merkle Proofs to reduce storage load

A Merkle tree can be used to prove the validity of a large amount of data using a small amount of data
For an example of this see Tornado Cash and
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.2.0/contracts/cryptography/MerkleProof.sol
(https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.2.0/contracts/cryptography/MerkleProof.sol)

Tools and Measurement

Tools such as Remix and Truffle will give you an idea of gas costs
You can also use eth-gas-reporter with truffle

Tenderly : https://tenderly.co/ (https://tenderly.co/)

Web3 :
web3.eth.estimateGas(callObject [, callback])
can estimate the gas required for a transaction

Gas Optimisation Audit from Open Zeppelin

https://blog.openzeppelin.com/compound-gas-optimizations-audit/
(https://blog.openzeppelin.com/compound-gas-optimizations-audit/)