Storage Optimization [1]

1. No gas cost if we declare without initializing

Example:

```
contract GasOptimization {
   // Storage variable declaration (no initialization cost)
   uint256 public storedData;
```

2. Minimize on chain data : minimizing on chain data reduces gas costs
3. Variables globally declare kore reuse kora instead of new variables
4. Calculate everything in memory variable and then update storage variable
5. Variable packing (variable packing example.sol)

Example:

```
contract MyContract {
  uint32 x;  // Storage slot 0
  uint32 z;  // Storage slot 0
  uint256 y; // Storage slot 1
}
```

6. Dont initialize zero values when writing loops (dont use (uint256 index = 0;). Instead, use the uint256 index) (initialize&Reinitialize.sol)
7. Static values to constant using immutable
8. Event based data use caution cos no other contract can use this on chain.
9. Store Data in calldata Instead of Memory for Certain Function Parameters

Example:

Before:

```
function loop(uint[] memory arr) external pure returns (uint sum) {
   for (uint i = 0; i < arr.length; i++) {
      sum += arr[i];
   }
}
```

After:

```
function loop(uint[] calldata arr) external pure returns (uint sum) {
   for (uint i = 0; i < arr.length; i++) {
      sum += arr[i];
   }
}
```

10. Caching data that are used frequently saves gas (cold access vs warm access) [6]

Example:

```
uint256 length = 10;
```

```
function loop() {
    uint256 l = length;

    for (uint256 i = 0; i < l; i++) {
        // do something here
    }
}
```

Refunds: [1]

1. Freeing storage slots by setting unused slot to zero (refunds 15k gas) (refundsExample.sol)
2. Using self destruct (refunds 24k gas) limitation: The refund obtained from selfdestruct cannot exceed half of the gas used by the ongoing contract call. This limitation is in place to prevent abuse of the refund mechanism.

Example for 1 and 2:

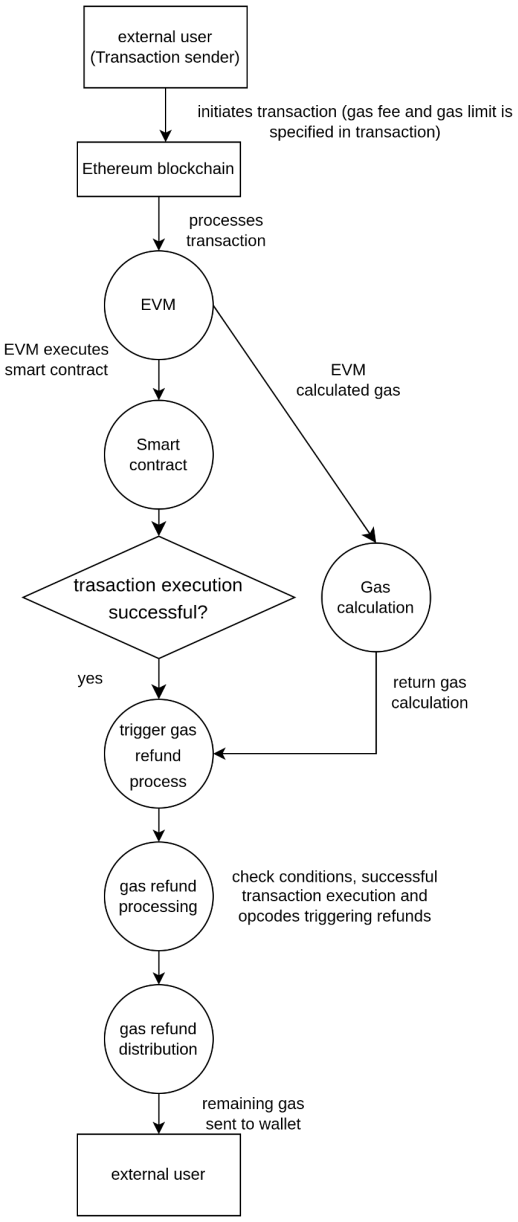
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract GasRefundExample {
  // Example storage variables
  uint256 public value1;
  uint256 public value2;

  // Function to update values and free storage slot
  function updateValuesAndFreeStorageSlot(uint256 _newValue1, uint256 _newValue2)
external {
      // Perform some operations with the values
      value1 = _newValue1;
      value2 = _newValue2;

      // Clear the storage slot by zeroing the variables
      // This refunds 15,000 gas
      assembly {
          sstore(value1.slot, 0)
          sstore(value2.slot, 0)
      }
  }

  // Function to selfdestruct and refund gas
  function destroyContract() external {
      // Ensure the refund doesn't surpass half the gas used
      require(gasleft() > gasleft() / 2, "Refund cannot surpass half the gas used");
```

```
        // Selfdestruct and refund 24,000 gas
        selfdestruct(payable(msg.sender));
    }
}
```

3. https://twitter.com/libevm/status/1468390867996086275?s=21 - using reinit, for testing in production. This mostly applied to MEV but if you are doing some cool factory-based programming it's worth trying out.
   Okay so as far as i understood - we can upgrade our contracts - proxy updates. But vanilla implementations arent gas efficient enough for using too many SLOADS, to get around that researches use a combination of SELFDESTRUCT and CREATE2 to achieve upgradable contract effect. Create2 allows to deploy contracts at predetermined addresses if they are empty, so first has to selfdestruct contract then creating with patched version. CREATE2 only recreates the same addresses if the salted bytecode is exactly the same, but the upgradibility feature is doab,e with a little bit of inline assembly and directional logic magic. This is mostly applied to MEV
4. Use ERC1167 To Deploy the same Contract many times

EIP1167 minimal proxy contract is a standardized, gas-efficient way to deploy a bunch of contract clones from a factory. EIP1167 not only minimizes length, but it is also literally a "minimal" proxy that does nothing but proxying. It minimizes trust. Unlike other upgradable proxy contracts that rely on the honesty of their administrator (who can change the implementation), the address in EIP1167 is hardcoded in bytecode and remain unchangeable

Refund how:
https://arxiv.org/pdf/2005.07908.pdf
https://docs.soliditylang.org/en/develop/introduction-to-smart-contracts.html#deactivate-and-self-destruct read this
https://fastercapital.com/content/Gas-Refunds--Uncovering-Gas-Refunds-and-Their-Role-in-Ethereum-s-Ecosystem.html have to read this too
How does Gas refunds work:
Gas refunds are an important feature of the Ethereum ecosystem, as they incentivize users to write more efficient and optimized smart contracts. [13]
The gas refund is only applied to successful transactions.
The maximum gas refund is half of the gas used.
if a transaction reverts/fails, the sender of the transaction will not receive a refund for any of the gas used. [11]
The refund is given in the same transaction, and it can't be stored. Also, the refund can never lower the transaction gas usage to under the basic cost of 21000. This is correct for any refund. [12]


external user
(Transaction sender)
initiates transaction (gas fee and gas limit is
specified in transaction)
Ethereum blockchain
processes
transaction
EVM
EVM executes
smart contract
EVM
calculated gas
Smart
contract
trasaction execution
successful?
Gas
calculation
yes
return gas
calculation
trigger gas
refund
process
gas refund
processing
check conditions, successful
transaction execution and
opcodes triggering refunds
gas refund
distribution
remaining gas
sent to wallet
external user

Data types and packing [1]

1. Use bytes32 whenever possible because it is the most optimized storage type.
2. If the length of bytes can be limited, use the lowest amount possible from bytes1 to bytes32.
3. Using bytes32 is cheaper than using string. (bytes32vsStringExample.sol and bytes32vsStringTest.js)
4. Variable packing occurs only in storage — memory and call data are not packed.
5. You will not save space trying to pack function arguments or local variables.
6. Storing small numbers in uint8 may not be cheaper. The EVM only works 32 bytes/256 bits at a time. This means that if you use uint8, EVM has to first convert it into uint256 to work with, and conversion costs extra gas. Storing a small number in a uint8 variable is not cheaper than storing it in uint256 coz the number in uint8 is padded with numbers to fill 32 bytes. [9] (uint8VSuint256.sol)

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.7;

contract withUint256 {
```

```
    uint256 a = 1;
    // 2246
    function getUint() public view returns (uint256) {
        return a;
    }
}

contract withUint128 {
    uint128 a = 1;
    // 2258
    function getUint() public view returns (uint128) {
        return a;
    }
}

contract withBoolean {
    bool a = true;
    // 2258
    function getBool() public view returns (bool) {
        return a;
    }
}
```

7. Pack structs as well
8. Use struct when dealing with different input arrays to enforce array length matching [7]

Example:

Before:

```
function vote(uint8[] calldata v, bytes[32] calldata r,  bytes[32] calldata s) public {
                    require(v.length == r.length == s.length, "not matching");
}
```

After:

```
struct Signature {
        uint8 v;
        bytes32 r;
        bytes32 s;
    }

function vote(Signature[] calldata sig) public {
    // no need for length check
}
```

Inheritance [1]

1. Inheritance is more gas efficient than composition
2. when we extend a contract, the variables in the child can be packed with the variables in the parent.
3. The order of variables is determined by C3 linearization, all you need to know is that child variables come after parent variables.

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract Parent {
   uint256 public parentVar;

   constructor(uint256 _parentVar) {
       parentVar = _parentVar;
   }
}

// Child contract inherits from Parent
```

```
contract Child is Parent {
  uint256 public childVar;

  constructor(uint256 _parentVar, uint256 _childVar) Parent(_parentVar) {
    childVar = _childVar;
  }
}

// Example of using the Child contract
contract Example {
  Child public myChild;

  constructor(uint256 _parentVar, uint256 _childVar) {
    // Creating an instance of Child initializes both parentVar and childVar
    myChild = new Child(_parentVar, _childVar);
  }

  function getParentVar() external view returns (uint256) {
    // Accessing parentVar from the Child contract
    return myChild.parentVar();
  }

  function getChildVar() external view returns (uint256) {
    // Accessing childVar from the Child contract
    return myChild.childVar();
  }
}
```

Memory vs storage [1]

1. **Use storage pointer**: Copying between the memory and storage will cost some gas, so don't copy arrays from storage to memory; use a storage pointer.

2. **Understand the complicated nature of memory**: The cost of memory is complicated. You “buy” it in chunks, the cost of which will go up quadratically after a while
3. Try adjusting the location of your variables by playing with the keywords “storage” and “memory”.
4. Writing data to storage is one of the most expensive operations in Solidity. Reduce gas costs by avoiding unnecessary writes. For example, move constant values to memory

Mapping vs Array

1. Opt for mapping over arrays for better gas savings (specially in cases of large dataset/requiring direct access instead of iteration, in case of requiring iteration in small datasets - use arrays)
2. We also use mapping to avoid looping, since looping costs a lot

Optimizing Variables

1. Optimize variable visibility: Avoid public variables; instead, use private visibility to save gas.
2. Efficient use of global variables: it is good to use global variables with private visibility as it saves gas
3. Events for data logging: Another strategy is to use events for data logging rather than storing data directly in the contract.

Example:

```
// Use events rather than storing data

  event ValueUpdated(uint256 newValue);
```

4. **Streamline return values**: A simple yet effective optimization technique is to name the return value in a function, eliminating the need for a separate local variable. (returnValueExample.sol)

   // Use return values efficiently

```
  function calculateProduct(uint256 a, uint256 b) external pure returns (uint256 product) {
      // Naming the return value directly
      product = a * b;
  }
```

5. **Direct updates to private variables**: Also, when updating private variables, do so directly and use events to log changes, avoiding unnecessary local variables. (DataLoggingExample.sol)

```
// Update private variable efficiently and emit an event
  function updatePrivateVar(uint256 newValue) external {
      // Assigning the value directly, no need for a local variable
      myPrivateVar = newValue;
      // Emit an event to log the update
      emit ValueUpdated(newValue);
  }

  // Retrieve the private variable
  function getPrivateVar() external view returns (uint256) {
      // Access the private variable directly
      return myPrivateVar;
  }
```

6. **Fixed vs dynamic variables and string:** Fixed sizes are always cheaper than dynamic ones. If we know size we should use fixed
7. **Dynamic** : it is best to structure our functions to be additive instead of subtractive. Extending an array costs constant gas, whereas truncating an array costs linear gas.

Example for 6 and 7:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

```
contract VariableOptimization {
    // Fixed size variable (cheaper than dynamic)
    uint256 public fixedVar;

    // Fixed-size memory array (saves gas)
    uint256[5] public fixedSizeArray;

    // Additive operation for dynamic array (saves gas)
    uint256[] public dynamicArray;

    // Additive function for dynamic array
    function addToDynamicArray(uint256 newValue) external {
        dynamicArray.push(newValue);
    }

    // Truncating function for dynamic array (more gas)
    function removeFromDynamicArray() external {
        require(dynamicArray.length > 0, "Array is empty");
        dynamicArray.pop();
    }

    // Fixed-size string using bytes32 (more gas-efficient for short strings)
    bytes32 public fixedSizeString;

    // Dynamic string (higher gas cost)
    string public dynamicString;

    // Set fixed-size string
    function setFixedSizeString(bytes32 _value) external {
        fixedSizeString = _value;
```

```
    }

    // Set dynamic string
    function setDynamicString(string memory _value) external {
        dynamicString = _value;
    }
}
```

8. Caching the length in for loops. (fixedvsVariableExample.sol)
9. The increment in the for loop's post condition can be made unchecked
10. ++i costs less gas compared to i++ or i += 1 (OperatorExample.sol)

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.7;

contract IncDecV1 {
    uint256 private _num = 5;
    // Function Execution Cost = 26,401
    function inc() external {
        _num += 1;
    }
}

contract IncDecV2 {
    uint256 private _num = 5;
    // Function Execution Cost = 26,388
    function inc() external {
        _num = _num + 1;
    }
}
```

```
contract IncDecV3 {
    uint256 private _num = 5;
    // Function Execution Cost = 26,343
    function inc() external {
        _num++;
    }
}

contract IncDecV4 {
    uint256 private _num = 5;
    // Function Execution Cost = 26,337
    function inc() external {
        ++_num;
    }
}
```

11. Don’t initialize Zero Values - when writing a for loop instead of writing uint256 index = 0; instead write uint256 index; as being a uin256 it will be 0 by default so you can save some gas by avoiding initialization (initialize&Reinitialize.sol)
12. Try to avoid unbounded loops [5]
13. To sum up, the best gas-optimized loop will be:

```
uint length = arr.length;
for (uint i; i < length;) {
  unchecked { ++i; }
}
```

[4]

no declaring i=0, no i++ in the condition, length pre read, using unchecked, using ++i, instead of i++ or i =i+1

(loopingExample.sol)

14. > is cheaper than > =: This is due to some supplementary checks (ISZERO, 3 gas)). [4]

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.7;

contract GreaterThan {
    // Function Execution Cost = 21,451
    function checkArray(uint256 x) external returns (bool) {
        return x > 15;
    }
}

contract GreaterThanOrEqualTo {
    // Function Execution Cost = 21,454
    function checkArray(uint256 x) external returns (bool) {
        return x >= 15;
    }
}
```

15. > 0 is cheaper than != 0 sometimes [4]
16. Use Shift Right/Left instead of Division/Multiplication if possible [4] [pure e korle ki hoy?]
17. In a variable, the cost of setting non zero to non zero value < cost of setting zero to non zero value, and setting to zero actually refunds, so we should always start from 1, or iterate from n to 0, instead of the other way around. (initialize&Reinitialize.sol)

Function optimization: [1]

1. Use external functions whenever possible instead of public ones. [the public visibility modifier is equivalent to using the external and internal visibility modifier, meaning both public and external can be called from outside of your contract, which requires more gas.]
2. **Optimize public variables** - it costs more (creates getter function implicitly)
3. Ordering of functions: placing often called functions earlier in the contract

4. **Parameter optimization:** Reducing the number of parameters in a function can save gas, as larger input data increases the gas cost due to more data being stored in memory.
5. **Use of payable functions**: Payable functions can be slightly more gas-efficient than non-payable ones. This is because the compiler doesn't need to check for the transfer of Ether in payable functions.
6. **Replacing Modifiers with Functions**: Solidity modifiers can increase the code size. Sometimes, implementing the logic of a modifier as a function can reduce the overall contract size and save gas.'

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;



contract FunctionOptimization {
  address public owner; // Reduce public variables

  constructor() {
    owner = msg.sender;
  }

  // Use external most of the time whenever possible
  function getDataExternal() external view returns (uint256) {
    return 42;
  }

  // Often-called function placed earlier
  function frequentlyCalledFunction() external pure returns (string memory) {
    return "Hello, frequently called!";
  }
```

```
  // Reduce parameters if possible
  function calculateSum(uint256 a, uint256 b) external pure returns (uint256 sum) {
      sum = a + b;
  }

  // Payable function saves some gas compared to non-payable functions
  function receiveEther() external payable {
      // Additional logic can be added
  }

  // Modifier implemented as a function to reduce code size
  function onlyOwner() internal view {
      require(msg.sender == owner, "Not the owner");
  }

  // Example function using the modifier
  function updateOwner() external {
      onlyOwner();
      // Additional logic for owner-only functionality
  }
}
```

7. **Batching Operations:** Batching operations enables developers to batch actions by passing dynamically sized arrays that can execute the same functionality in a single transaction, rather than requiring the same method several times with different values. [2]

Example:

Before:

```
function doSomething(uint256 x, uint256 y, uint256 z) public {
                require msg.sender == registeredUser
                ....
```

```
}
```

After:

```
function batchDoSomething(uint256[] x, uint256[] y, uint256[] z) public {
                require msg.sender == registeredUser
   loop
                        _doSomething(x[i], y[i], z[i])
}
function doSomething(uint256 x, uint256 y, uint256 z) public {
                require msg.sender == registeredUser
                _doSomething(x, y, z);
}
function _doSomething(uint256 x, uint256 y, uint256 z) internal {
                ....
}
```

8. Reorder statements to shift cheaper ops first [3]

Example:

Before:

```
case 1: 100 gas
case 2: 50 gas

if(case1 && case2) revert
if(case1 || case2) revert
```

After:

```
if(case2 && case1) revert
if(case2 || case1) revert
```

9. Precompute Off-Chain When Possible [3]
10. View functions cost gas when called within a transaction
11. Order cheap functions before : f(x) is cheap and g(y) is expensive. The ordering should be for example : f(x) || g(y), f(x) && g(y) [5]
12. In the same way, AND wont check the second function if the first one is false, OR wont check the second function if the first one is true, thus saving gas, so order appropriately
13. `RevertEarly` is likely to be more gas-efficient in scenarios where the `require` condition is expected to fail frequently, as it avoids unnecessary state changes when the condition is not met. [6]
14. Using double require instead of AND [6]

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.7;

contract withOperator {
    // gas cost = 21510
    function check(uint256 x) external returns (uint256) {
        require(x > 0 && x < 10);
        return x;
    }
}

contract withDoubleRequire {
    // gas cost = 21502
    function check(uint256 x) external returns (uint256) {
        require(x > 0);
        require(x < 10);
        return x;
```

```
    }
}
```

Things we can do:

1. Using indexed events: Including a mechanism to keep track of a smart contract's activity after it is deployed is helpful in reducing overall gas. While looking at all of the contract's transactions is one way to keep track of the activity, because message calls between contracts are not recorded on the blockchain, that approach might not be sufficient. [2]

Example:
Before:

```
event Withdraw(uint256, address);

function withdraw(uint256 amount) public {
    emit Withdraw(amount, msg.sender);
}
```

After:

```
event Withdraw(uint256 indexed, address indexed);

function withdraw(uint256 amount) public {
    emit Withdraw(amount, msg.sender);
}
```

2. Using events over logs: For debugging, events are a cheaper way to record data than logs
3. Turning on Solidity Compiler Optimizer: reduces CDC and FCC. CDC decreases with fewer runs, but as we increased the number of runs, the CDC increased while the FCC decreased. [6]

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.16;

/*
 * NO OPTIMIZATION: CDC = 74395, FCC = 21162
 * OPTIMATION ENABLED
 * RUNS SET TO 0:       CDC = 72661, FCC = 21138
```

```
 * RUNS SET TO 200:     CDC = 72661, FCC = 21138
 * RUNS SET TO 1000:    CDC = 72661, FCC = 21138
 * RUNS SET TO 100000:  CDC = 72661, FCC = 21138
 */
contract OptimzationExample1 {
    function transferEthers() public payable {}
}

/*
 * NO OPTIMIZATION: CDC = 282513, FCC = 25680
 * OPTIMATION ENABLED
 * RUNS SET TO 0:         CDC = 180298, FCC = 46143
 * RUNS SET TO 200:       CDC = 180298, FCC = 46143
 * RUNS SET TO 1000:      CDC = 188192, FCC = 24799
 * RUNS SET TO 10000:     CDC = 198760, FCC = 24799
 * RUNS SET TO 100000:    CDC = 198748, FCC = 24799
 * RUNS SET TO 10000000:  CDC = 198760, FCC = 24799
 */
contract OptimzationExample2 {
    uint256 private _fixer;
    uint256 private _lastCalculatedValue;

    constructor(uint256 fixer) {
        _fixer = fixer;
    }

    function getValue(uint256 num1, uint256 num2) public returns (uint256) {
        uint256 fixer = _fixer;
        require(num1 < num2, "num2 must be greater than num1");
        require(
            (num1 + num2) > fixer,
            "sum of num1 and num2 must be greater than fixer"
        );
        uint256 sum = num1 + num2;
        uint256 fixedSum = sum * fixer;
        _lastCalculatedValue = fixedSum;
        return fixedSum;
    }
}
```

4. Enable the ABI Encoder v2: The ABI encoder translates data types into the format stored on the blockchain. Enabling the newer v2 encoder saves gas

Example:

```
pragma solidity ^0.8.0;

// Enable ABI encoder v2
pragma abicoder v2;

contract MyContract {
  //...
}
```

5. Use Clones for Cheap Contract Deployments

Here's an example:

```
function _executeTransfer(address _owner, uint256 _idx) internal {
        (bytes32 salt, ) = precompute(_owner, _idx);
        new FlashEscrow{salt: salt}( //gas: deployment can cost less through clones
            nftAddress,
            _encodeFlashEscrowPayload(_idx)      );
             }
```

There's a way to save a significant amount of gas on deployment using Clones: [OpenZeppelin video](#) [4]

6. Remove dead code
7. Use different versions of solidity to see which costs less
8. EXTCODESIZE is quite expensive, this is used for calls between contracts,
    a. Minimizing External Calls
    b. Batching Operations: Instead of making multiple individual calls to external contracts, consider batching operations when possible.
    c. Local Data Storage: If data from an external contract is needed frequently, consider storing a copy of that data locally within the contract to avoid the need for repeated calls to the external contract. [may increase storage costs]

9. **Unchecked Blocks:** if we know that arithmetic operations won't result in underflow or overflow, we can use unchecked blocks. (CheckedExample.sol and CheckedvsUncheckedTesting.js)

Example:

```
function loop(uint256 length) public {
        for (uint256 i = 0; i < length; ) {
            // do something
            unchecked {
                i++;
            }
        }
}
```

10. **Use custom errors to save deployment and runtime costs in case of revert**
Instead of using strings for error messages (e.g., require(msg.sender == owner, “unauthorized”)), you can use custom errors to reduce both deployment and runtime gas costs. In addition, they are very convenient as you can easily pass dynamic information to them. [7]
Before:

```
function add(uint256 _amount) public {
    require(msg.sender == owner, "unauthorized");

    total += _amount;
}
```

After:

```
error Unauthorized(address caller);

function add(uint256 _amount) public {
    if (msg.sender != owner)
        revert Unauthorized(msg.sender);

    total += _amount;
}
```

11. Optimize for frequent events
    a. Common cases >> rare cases
    b. Runtime cost >> deployment cost
    c. User interaction >> admin interaction [7]
12. Using safemath library: only required for solidity versions before 0.8 [8]
https://www.npmjs.com/package/hardhat-gas-reporter

Compiler based optimization

The Solidity compiler involves optimizations at three different levels (in order of execution):

- Optimizations during code generation based on a direct analysis of Solidity code.
- Optimizing transformations on the Yul IR code.
- Optimizations at the opcode level. [10]

Yul related jinish: https://github.com/harendra-shakya/solidity-gas-optimization

Sources:

1. https://hacken.io/discover/solidity-gas-optimization/
2. https://www.alchemy.com/overviews/solidity-gas-optimization
3. https://www.infuy.com/blog/7-simple-ways-to-optimize-gas-in-solidity-smart-contracts/
4. https://betterprogramming.pub/solidity-gas-optimizations-and-tricks-2bcee0f9f1f2
5. https://github.com/harendra-shakya/solidity-gas-optimization
6. https://coinsbench.com/comprehensive-guide-tips-and-tricks-for-gas-optimization-in-solidity-5380db734404
7. https://0xmacro.com/blog/solidity-gas-optimizations-cheat-sheet/
8. https://moralis.io/gas-optimizations-in-solidity-top-tips/
9. https://blockchain.oodles.io/dev-blog/solidity-gas-optimization-tips/
10. https://docs.soliditylang.org/en/latest/internals/optimizer.html
11. https://medium.com/@Lordumf/the-ethereum-blockchain-is-a-decentralized-platform-that-runs-smart-contracts-applications-that-5d90415dce81
12. https://ethereum.stackexchange.com/questions/92965/how-are-gas-refunds-payed
13. https://fastercapital.com/content/Gas-Refunds--Uncovering-Gas-Refunds-and-Their-Role-in-Ethereum-s-Ecosystem.html