CS190: Blockchain Programming and Applications

Lecture 7: Gas Optimization and Testing

Yanju Chen

Gas is the unit that measures how much computational work an operation costs in a blockchain system.

- HW1 is due today (Oct 20)
- HW2 will be out soon today (Oct 20)
- Slides and code of this lecture is available on course website.

UCSB

Why Optimize Gas?

Lower User Cost

During the ERC-404 frenzy, average Ethereum gas spiked to ~70 gwei (~$60/tx) on Feb 9, 2024, pricing out small users.

That meant simple transfers or mints became unaffordable.

Better UX under Stress

In a historic liquidation event on Oct 17, 2025, Solana stayed fast/cheap while Ethereum struggled, showing efficiency matters for user experience.

Efficient code scales better under pressure and provides smoother user experience.

Stay Competitive

Ethereum's Dencun upgrade on Mar 13, 2024 added EIP-4844 "blobs," cutting many L2 fees dramatically (reports of ~90–99% reductions).

Optimizing gas helps blockchains and apps stay competitive in a multi-chain world.

Network Sustainability

Best-practice guides on Jan 3, 2025 stress minimizing storage writes/packing to curb state bloat and keep nodes affordable.

Less wasteful storage = smaller global state = cheaper and easier node operation.

Measuring Gas Consumption in Foundry

```
function test_gas_endWithoutCleanup() public {
    uint256 gasStart = gasleft();
    auction.endWithoutCleanup();
    uint256 gasUsed = gasStart - gasleft();
    emit log_named_uint("Gas used (no cleanup)", gasUsed);
}
```

Gas Metering

Uses **gasleft()** to record the remaining gas before and after a code segment.

Results can vary slightly (be imprecise) due to compiler optimizations, internal calls, or logging overhead.

Gas Reports

Automatically collects and summarizes gas usage for ***entire*** functions or tests, giving a higher-level overview without manual instrumentation.

```
forge test --match-test "testGas_OpenAccount_Log" --gas-report
```

select target test **show gas report**

src/Bank.sol:BankOriginal Contract					
Deployment Cost	Deployment Size				
718613	3124				
Function Name	Min	Avg	Median	Max	# Calls
openAccount	66613	66613	66613	66613	1

src/BankOptimized.sol:BankOptimized Contract					
Deployment Cost	Deployment Size				
718613	3124				
Function Name	Min	Avg	Median	Max	# Calls
openAccount	66613	66613	66613	66613	1

Gas Refund (I)

DEMO RefundAuction.sol

```
/// @notice End auction without cleanup (no refund)
function endWithoutCleanup() external {
    require(msg.sender == admin);
    require(!ended);
    ended = true;
    // do nothing: leaves bids nonzero
}
```

Refund: No

```
/// @notice End auction with cleanup (refund happens)
function endWithCleanup() external {
    require(msg.sender == admin);
    require(!ended);
    ended = true;
    for (uint256 i = 0; i < participants.length; i++) {
        delete bids[participants[i]]; // nonzero -> zero => refund
    }
}
```

Refund: Yes

- Ends auction but keeps all bids in storage
- Ends auction and deletes each bid

When would gas refund be triggered?

A gas refund is triggered when a transaction frees storage *by setting non-zero values back to zero*, such as using **delete** or **clearing mappings or arrays**.

Gas Refund (II)

Do the following statements trigger gas refund?

```
delete myMapping[user]; // key exists      // Delete mapping entry → refund
myArray.pop(); // when arr[last] != 0      // Remove last array element → refund (if nonzero)
myVar = 0;                                 // Overwrite nonzero storage slot → refund
delete myStruct.value;                     // Clear struct field → refund
```

```
uint x = 0; x = 1;                         // non-zero←zero write (costly) — NO refund
delete temp;                               // deleting a memory var — NO refund (memory ≠ storage)
bytes memory b; delete b;                  // memory cleanup — NO refund
arr.pop(); // when arr[last] == 0          // popping a zero value — NO refund (no nonzero→zero)
delete myMapping[user]; // unset key       // deleting a key that was already zero — NO refund
delete arr;                                // clears length only; slots not zeroed — NO refund
emit Event(v);                             // logs/events do not affect storage — NO refund
selfdestruct(payable(owner));              // NO refund post EIP-3529; also deprecated
```

Saving Gas with Struct Packing (I)

DEMO StructPackingDemo.sol

1 slot = 32 bytes
1 byte = 8 bits
uintXX, XX means #bits
bytesXX, XX means #bytes

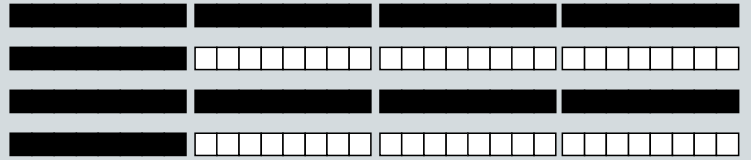
Ordering struct variables so that multiple **smaller** types (like **uint8**, **bool**, **uint16**) fit into a single **32-byte storage slot**, reducing gas usage.

1 byte **1 slot = 32 bytes** (EVM reads and writes storage in 32-byte slots at a time.)

Data Type	Bytes	Packable	Notes
bool, uint8, int8, small enum	1	Yes	Enums default to uint8 unless specified
uint16, int16	2	Yes	
uint32, int32	4	Yes	
uint64, int64	8	Yes	
uint128, int128	16	Yes	
address	20	Yes	Can share slot with other small fields
bytes1 .. bytes31	1–31	Yes	Fixed-size bytes (except bytes32)
uint256, int256	32	No	Occupies a full slot
bytes32	32	No	Occupies a full slot
string, bytes (dynamic)	–	No	Dynamic; stored via separate slot/keccak pointer
mapping	–	No	Not stored inline; uses hashed slot per key
dynamic array (T[])	–	No	Length in slot; elements in separate area

```
struct UserBad {
    uint256 a;
    uint64  x;
    uint256 b;
    uint64  y;
}
```

Slot 0: full
Slot 1: 1/4
Slot 2: full
Slot 3: 1/4



Data Layout

Can we optimize this?

Saving Gas with Struct Packing (II)

DEMO StructPackingDemo.sol

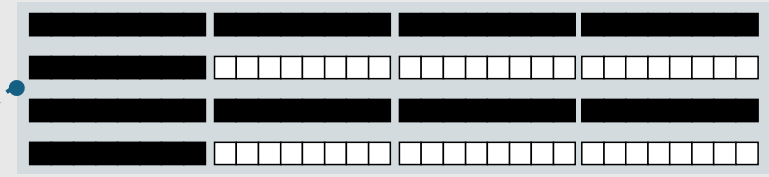
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract StructPackingDemo {
    struct UserBad {
        uint256 a;
        uint64  x;
        uint256 b;
        uint64  y;
    }

    struct UserGood {
        uint256 a;
        uint256 b;
        uint64  x;
        uint64  y;
    }

    mapping(uint256 => UserBad)  public badUsers;
    mapping(uint256 => UserGood) public goodUsers;
}
```

Slot 0: full
Slot 1: 1/4
Slot 2: full
Slot 3: 1/4



Unoptimized Layout (4 slots)

Slot 0: full
Slot 1: full
Slot 2: 1/4
Slot 2: 1/4

Optimized Layout (3 slots)

Is this packing scheme always the best?

```
contract LayoutPlayOriginal {
    struct S {
        uint128 a;
        uint64 b;
        uint64 c;
        uint8 t;
        uint64 d;
    }

    mapping(uint256 => S) private book;

    function _bumpABT(
        uint256 id,
        uint128 da,
        uint64 db
    ) internal {
        S storage x = book[id];
        x.a += da;
        x.b += db;
        x.t ^= 1;
    }
}
```

Slot 0: 1/2
Slot 0: 1/4
Slot 0: 1/4
Slot 1: 1/32
Slot 1: 1/4

Both slots are read. The seemingly best packing scheme is not saving gas.

Best Practices on Layout for Lower Gas (I)

DEMO SmallIntAndBoolDemo.sol

Use smallest integer types

Reduces storage slot usage; allows struct packing; avoids excessive fragmentation or overflow risk

```
struct BigNums {
    uint256 a;
    uint256 b;
    uint256 c;
}
```

Unoptimized Layout (3 slots)

```
struct SmallPacked {
    uint128 a;
    uint64  b;
    uint64  c;
}
```

Optimized Layout (1 slots)

```
struct FlagsBool {
    uint128 a;
    bool    f1;
    bool    f2;
    uint64  b;
}

struct FlagsU8 {
    uint128 a;
    uint8   f1;
    uint8   f2;
    uint64  b;
}
```

Replace bool with uint8

Bool adds **extra masking and validation logic** at the EVM level, while **uint8** is written directly *without* additional checks

Best Practices on Layout for Lower Gas (II)

DEMO ConstImmutableDemo.sol

Use constant for fixed values

Constants are hardcoded into bytecode at compile time

Use immutable for deployment-time constants

Stored at deploy time, no SLOAD at runtime

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract ConstImmutableDemo {
    // compile-time constant (baked into bytecode)
    uint256 public constant FEE_CONST = 10;

    // set-once at deployment time
    uint256 public immutable FEE_IMM;

    // regular storage slot
    uint256 public feeStorage = 10;

    constructor(uint256 imm) {
        FEE_IMM = imm; // e.g., pass 10 at deploy
    }

    // ...
}
```

What's the difference?

- **Constant** is a value that never changes and is written directly into the code
- **Immutable** is a value **you choose** once when you deploy the contract and it can't change afterward.

Best Practices on Layout for Lower Gas (III)

DEMO AvoidDynamicArrayDemo.sol

Avoid dynamic storage arrays

Expansion and indexing are expensive; use mapping or fixed arrays instead

```
contract AvoidDynamicArrayDemo {
    uint256[] public arr;

    mapping(uint256 => uint256) public map;
    uint256 public mCount;

    // ...
}
```

Dynamic; Expensive

Fixed; Cheap

```
function bumpArray(uint256 n) external {
    uint256 len = arr.length;
    if (n > len) n = len;
    for (uint256 i = 0; i < n; i++) {
        arr[i] = arr[i] + 1; // SSTORE nonzero->nonzero
    }
}
```

Extra Arithmetic & Bound Checks

```
function bumpMap(uint256 n) external {
    if (n > mCount) n = mCount;
    for (uint256 i = 0; i < n; i++) {
        map[i] = map[i] + 1; // SSTORE nonzero->nonzero
    }
}
```

Direct Key-Slot Hash

```
function clearArrayByDeletingElements() external {
    uint256 len = arr.length;
    for (uint256 i = 0; i < len; i++) {
        delete arr[i];
    }
    delete arr;
}

function clearArrayLengthOnly() external {
    delete arr;
}

function clearMapByResetCount() external {
    mCount = 0;
}
```

O(n)

O(1)