# HW4
# Bitcoin Miner

Introduction to Parallel Computing
2022/04/26

# Spec

https://hackmd.io/@ipc22/hw4

# Introduction - What is a Miner?

- A miner's job is to find out a hash value, such that the block is secured
- More specifically, the primary task of a miner is to find out a proper value called "nonce", which is part of a block's header
- The value of nonce has to satisfy some requirements, including:
    - It has to be 32-bit long
    - It has to be less than a predefined value called "Target Difficulty"

# Introduction - What is a Block?

- A block consists of two parts: the block header and block body
- The block body contains a number of bitcoin transactions (or simply transactions) and coinbase transactions
    - Bitcoin transactions are the records of bitcoin transfers
    - Coinbase transactions records the bitcoins rewarded to the miner who finds the proper nonce for the block
- There are several fields included in the block header. We only cares about the following fields.
    - Nonce
    - The hash value of the previous block
    - Merkle root
    - Target difficulty (nBits)

# Introduction - Block Layout

- The layout of a bitcoin block looks like the table presented below.
    - The yellow part corresponds to the block header
    - The gray part corresponds to the block body
- A secure hash value is generated (on the right side) for the block header

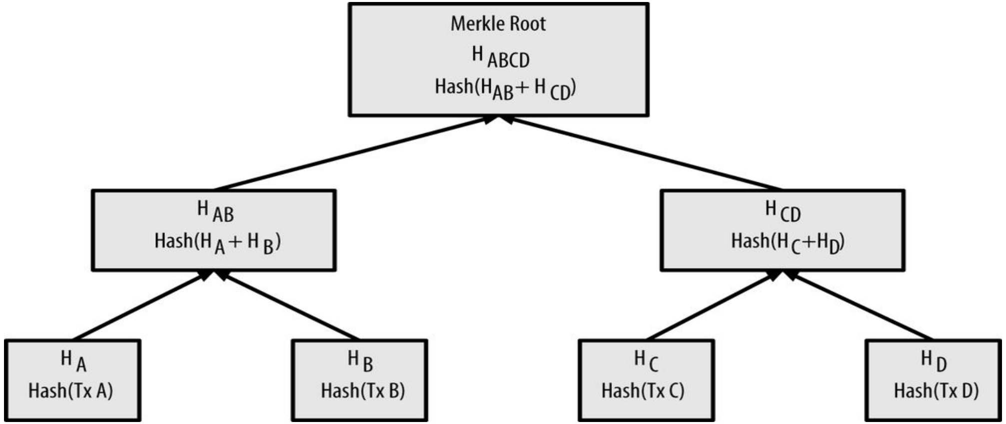
| | |
|---|---|
| version | 02000000 |
| previous block hash (reversed) | 17975b97c18ed1f7e255adf297599b55<br>330edab87803c81701000000000000000 |
| Merkle root (reversed) | 8a97295a2747b4f1a0b3948df3990344<br>c0e19fa6b2b92b3a19c8e6badc141787 |
| timestamp | 358b0553 |
| bits | 535f0119 |
| nonce | 48750833 |
| transaction count | 63 |
| coinbase transaction | |
| transaction | |
| ... | |

Block hash

0000000000000000<br>e067a478024addfe<br>cdc93628978aa52d<br>91fabd4292982a50

# Introduction - Merkle Tree Nodes

- We store the transactions in a data structure called “Merkle Tree”
  - The tree is constructed in a binary tree fashion
  - The root and intermediate nodes of the tree are generated from their children




$H_A$ = SHA256(SHA256(TxA))
$H_B$ = SHA256(SHA256(TxB))
$H_{AB}$ = SHA256(SHA256($H_A$+$H_B$))

Credit: https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch07.html

# Introduction - Double SHA-256

- Each parent node is generated by a double SHA-256 function
  - Each child node is encrypted by SHA-256 TWICE
  - The encrypted results (HA and HB) are summed together
  - The sum is then encrypted AGAIN by SHA-256 TWICE
- The procedure is repeated again and again, until the root node is reached
  - The root node is called the Merkle Root

$$H_A = \mathrm{SHA256}(\mathrm{SHA256}(\mathrm{TxA}))$$

$$\Downarrow$$

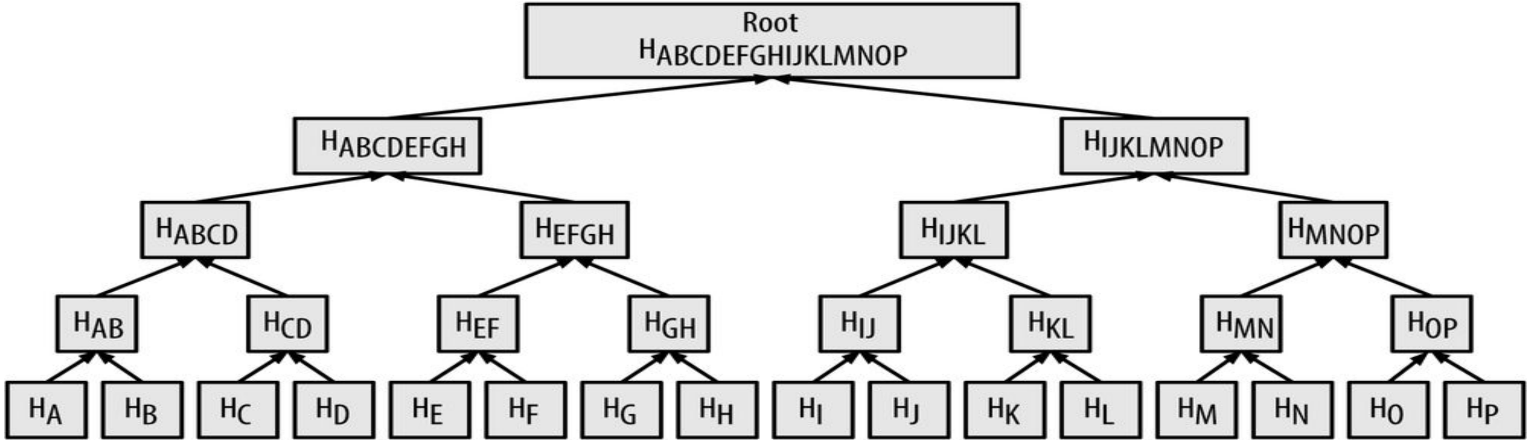
$$H_B = \mathrm{SHA256}(\mathrm{SHA256}(\mathrm{TxB}))$$

$$\Downarrow$$

$$H_{AB} = \mathrm{SHA256}(\mathrm{SHA256}(H_A+H_B))$$

# Introduction - Merkle Tree Structure

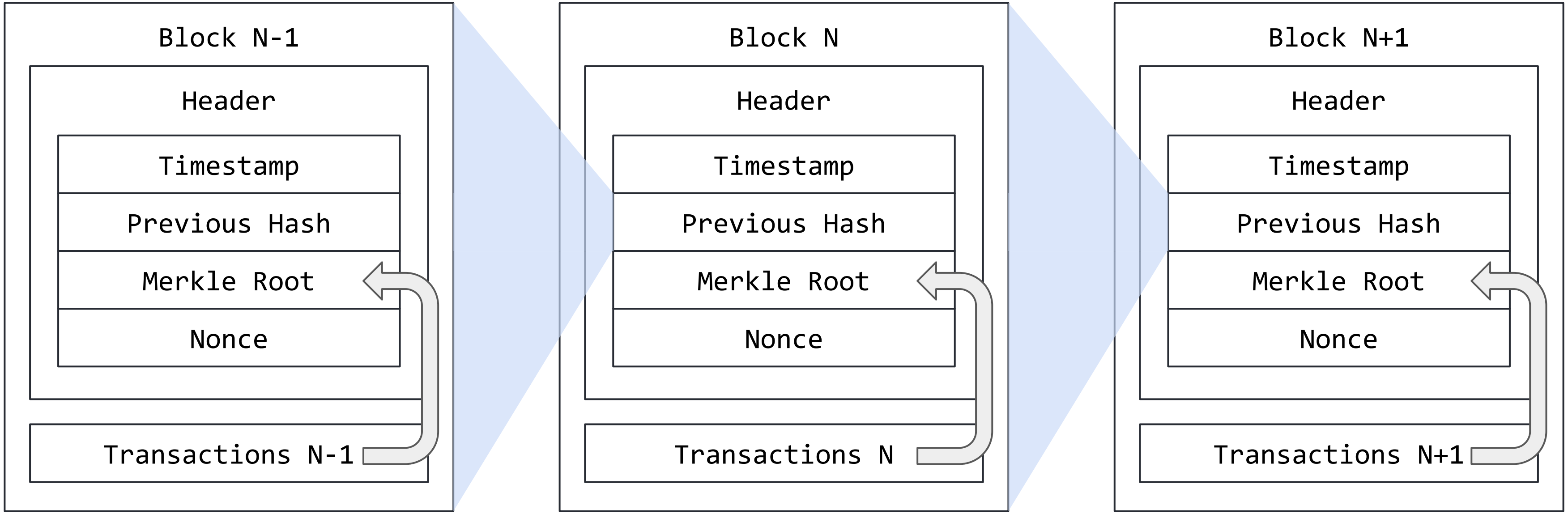
- A typically Merkle tree has the structure similar to the figure depicted below
  - Please note that the hash functions are double SHA-256




Credit: https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch07.html

# Introduction - Blockchain

- A blockchain is essentially a sequence of verified blocks
  - Each block contains the has value of its previous block's header
  - If something is modified, it will be immediately discovered



# Introduction - Verification of Transactions

- Who has to verify these transactions?
  - The nodes in the bitcoin network
- How do they verify the transactions?
  - The transactions from each node are broadcast to the entire network
  - Each node collects transactions by some algorithms defined by the bitcoin protocol.
  - The transactions are collected for block encryption
- How is the encrypted block added to the blockchain?
  - The newly encrypted block is broadcast to the entire network
  - The miners in the network verifies if the new block is correct and conforms to the bitcoin protocol (e.g., Whether block header's hash < Target Difficulty or not)

# Introduction - Target Difficulty

- The Target Difficulty is a value to determine how difficult it is to find out a proper hash value for the block header
- The Target Difficult conforms to the following rules
  - It is a 256-bit number, encoded by a field called “nbits” (in 4 bytes)
  - It is determined by the bitcoin network
  - It has n leading zeros (currently, n equals 76).
- The Target Difficulty is adjusted such that a new block is computed every 10 minutes (on average)
  - The value of n is determined by the computing power of the entire bitcoin network
  - It’s updated every 2,016 blocks (roughly every two weeks).
  - You may check current difficulty here: https://btc.com/stats/diff

# Introduction - Goal

- The primary goal of assignment 4 is to find out a proper hash value for a given block header
  - The hash value has to be less than the Target Difficulty
  - The time you spent on finding the hash value has to be accelerated
  - Use any techniques that you've learned in this class!

# Sequential Code

1. Read input
2. Calculate a merkle root from transactions
3. Decode find out the Target Difficulty
4. Find out a proper nonce, such that the hash value of the block header satisfies the requirement

```
for nonce = 0X00000000 to 0Xffffffff
    calculate a hash value
    if the hash value < Target Difficulty
        done
```

# Sequential Code - Header Extraction

- Step 1 : Read input
- There are several fields in the block header that you have to extract.
- Please familiar yourself with the fields extracted from the header.

```
// **** read data *****
char version[9];
char prevhash[65];
char ntime[9];
char nbits[9];
int tx;
char *raw_merkle_branch;
char **merkle_branch;

getline(version, 9, fin);
getline(prevhash, 65, fin);
getline(ntime, 9, fin);
getline(nbits, 9, fin);
fscanf(fin, "%d\n", &tx);
printf("start hashing")

raw_merkle_branch = new char [tx * 65];
merkle_branch = new char *[tx];
for(int i=0;i<tx;++i) {
   merkle_branch[i] = raw_merkle_branch + i * 65;
   getline(merkle_branch[i], 65, fin);
   merkle_branch[i][64] = '\0';
}
```

# Sequential Code - Header Fields

- A representative header fields of a block is depicted below
  - Please have a comparison with pages 6, 10, and 16
  - These information has to be extracted first before we carrying out the calculation
- `case01.in`

| | |
|---|---|
| `4` | Number of blocks |
| `20000000` | Version |
| `0000000000000000003348540cbfc68b70825e7abcd5a83a48a5f87fa7f1aace` | Previous block hash |
| `5ac22f8b` | Timestamp |
| `17502ab7` | Bits (packed difficulty) |
| `2094` | Number of transactions |
| `c6574adb277efbfb972658ab78b1277707a967076cfc90d6af800cd8a915396d` | |
| `c01c33240ba97fb6db2b98fbaf7e4211fe3b59585372994bdac1b96b1c9be0d3` | |
| `abce7b2b30ff62b4998864db6a1ea77db8eb33d3f6abe3f981e0d2802c999042` | |
| `fd463aca59df1302c8f08e8070b56bd64ebe0cf35ba68cd5b39d208a2ff50053` | Transactions |
| `4321318516cb6630e3b3caa29bd49227168a69d170e1110c220b3c30d15f913c` | |
| `f1f452e09dad935a67499a24e388d61fd7f0f38c97a2a92c73b2fd7019a85578` | |
| `dd24a8e3f419006cd2f7cb3336b605fa6c2898accecac3d0a844c2ae8f5f53d9` | |

# Sequential Code - Merkle Root

- Step 2 : Calculate a merkle root from transactions
  - The merle root is calculated by a function provided by the TA
  - You are encouraged to take a look of its implementation

```
// **** calculate merkle root ****

unsigned char merkle_root[32];
calc_merkle_root(merkle_root, tx, merkle_branch);

printf("merkle root(little): ");
print_hex(merkle_root, 32);
printf("\n");

printf("merkle root(big):    ");
print_hex_inverse(merkle_root, 32);
printf("\n");
```

# Sequential Code - Target Difficulty

- Step 3 : Decode find out the Target Difficulty
  - The is decoded and calculated from a number called bits
  - The decode algorithm is already implemented by us (shown below)
  - If you are interested in that algorithm, please take a look of this [website](website).

```
// ********** calculate target value *********
// calculate target value from encoded difficulty
which is encoded on "nbits"
unsigned int exp = block.nbits >> 24;
unsigned int mant = block.nbits & 0xffffff;
unsigned char target_hex[32] = {};

unsigned int shift = 8 * (exp - 3);
unsigned int sb = shift / 8;
unsigned int rb = shift % 8;
```

- Example:
  - 2022/04/14 Bits: `0x1709f8d9`
  - `Target = 0x09f8d9 * 2 ** (8 * (0x17 - 3))`
    `= 0x00000000000000000009d897000000000000000000000000000000000000000`

# Sequential Code - Nonce

- Step 4 : Find out a proper nonce, such that the hash value of the block header satisfies the requirement
  - Try if you can figure out a way to parallelize and accelerate the code

```
for(block.nonce=0x00000000; block.nonce<=0xffffffff;++block.nonce){
    //sha256d
    double_sha256(&sha256_ctx, (unsigned char*)&block, sizeof(block));
    if(block.nonce % 1000000 == 0) {
        printf("hash #%10u (big): ", block.nonce);
        print_hex_inverse(sha256_ctx.b, 32);
        printf("\n");
    }
    if(little_endian_bit_comparison(sha256_ctx.b, target_hex, 32) < 0) { // sha256_ctx < target_hex
        printf("Found Solution!!\n");
        printf("hash #%10u (big): ", block.nonce);
        print_hex_inverse(sha256_ctx.b, 32);
        printf("\n\n");
        break;
    }
}
```

# Sequential Code - SHA-256

- SHA-256 implementation for the sequential version can be found at:
  - sha256.h
  - sha256.cu
- If you are interested in the implantation details, please take a look at the following website
  - https://en.wikipedia.org/wiki/SHA-2

# Testcases

- We provide several testcases for you to test your program
- The execution time of the sequential code are given below:
  - case00.in : about 1 hr 15 min
  - case01.in : about 1 hr 15 min
  - case02.in : about 30 min
  - case03.in : about 11 min
- Please keep in mind that your goal is to parallelize the mining algorithm and accelerating it.
- It has to be faster than the sequential version.

# Grading

- Correctness (40%)
  - Please make sure to use GPU and verify your answer
- Performance (30%)
  - We will grade the performance of your assignments against the other students in the class
- Report (30%)
  - Please describe, in detail, how you parallelize your codes as well as your optimization methodology
- Advanced CUDA skills
  - You are welcome to use streaming, page-lock memory, asynchronous memory copy, or any other advanced skills
- Others
  - You will get credits if you are able to optimize the other parts of the source codes.  Please justify your solutions and provide a detailed comparison in your report

# Grading - Report

- In your report, please include the following parts
  - Your implementation
  - The parallelization and optimization techniques you used in your solution
  - Experiments of various combinations of the number of blocks & threads (at least 8 combinations) and plot them with the figures
  - Describe the details if you use advanced CUDA skills
  - If you optimize the other parts of your source codes, please demonstrate your experimental results. We REQUIRE you to justify your solutions so that we can give you credits.

# Submission

- Please submit your hw4 to eeclass
  - hw4.cu
  - sha256.cu
  - sha256.h
  - report.pdf
  - Makefile (optional)
- Please do not package them, directly upload the files to eeclass
- Please make sure your Makefile works properly and your program can run before submitting your assignment 4

# Reminder

- Because we are doing hash, there may be a situation multiple nonce can satisfies the requirement hash value < Target Difficulty
- We accept all nonce satisfies the requirement, so don’t worry if your nonce isn’t same as our provided solutions
- We ensure that all the testcases have more than one solution
- Your Makefile should build executable binary hw4

# Deadline

- The deadline of assignment 4 is 5/10 (Tue), 23:59
- Everyone is welcomed to ask questions on eeclass