110062118
賴姿妘

HW4 Bitcoin Miner

A. Implementation

✧ CPU and GPU task distribution

In the assignment, I let CPU responsible for calculating Merkle root and other preprocess essential for finding nonce. This task, including I/O, by observation, doesn't take lot of time to finish. Therefore, I didn't put effort on using stream or asynchronous method to increase throughput or applying OpenMP to accelerate CPU task.

As for GPU, each thread is responsible for validating certain nonce. Once a thread finds the nonce, the other threads will terminate as fast as possible. The corresponding kernel is launch with **number_of_blocks = 16777216 and threads_per_block = 256,** which is the sweet spot after several trial.

✧ Sequential Code Optimization

I replace most of '*', '/' or '%' with "<<", ">>", and '&', though there is no significant improve.

✧ Sha256 for GPU

In the original sequential version, there is no specific function for GPU to do sha256 calculation. Thus, I duplicated and multiplied necessary functions to enable GPU to run sha256.

Essential array k is set in constant memory to improve efficiency.

```
__constant__ unsigned int k[64] = {
    0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,0x923f82a4,0xab1c5ed5,
    0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,0x72be5d74,0x80deb1fe,0x9bdc06a7,0xc19bf174,
    0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0x76f988da,
    0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,0x06ca6351,0x14292967,
    0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,0x650a7354,0x766a0abb,0x81c2c92e,0x92722c85,
    0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624,0xf40e3585,0x106aa070,
    0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a,0x5b9cca4f,0x682e6ff3,
    0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,0x90befffa,0xa4506ceb,0xbef9a3f7,0xc67178f2
};
```

✧ Communication between CPU and GPU

Before GPU finds nonce, the program will malloc space for target value and result on GPU, and then copy target to the device. Finishing the task, CPU can get result by copy result in GPU. It may

110062118
賴姿妘

```
unsigned char *d_target_hex;
cudaMalloc(&d_target_hex, 32);
cudaMemcpy(d_target_hex, target_hex, 32, cudaMemcpyHostToDevice);

unsigned int *d_result;
cudaMalloc(&d_result, sizeof(unsigned int));

unsigned int start_nonce = 0;
unsigned int step = 1;
unsigned int total =  4294967296;
unsigned int blocks = 16777216;
unsigned int threads_per_block = 256;
//{4 38.39} --> {4 37.53}



reset_found_kernel<<<1, 1>>>();
cudaDeviceSynchronize();
nonce_search_kernel<<<blocks, threads_per_block>>>(block, d_target_hex, start_nonce, step, d_result);

unsigned int nonce;
cudaMemcpy(&nonce, d_result, sizeof(unsigned int), cudaMemcpyDeviceToHost);
```

✧ nonce_search_kernel

Before this kernel launched, a kernel **reset_found_kernel()**, to reset the found signal, which indicates whether nonce has been found, to false.

The whole nonce_search_kernel is shown below. It will first calculate the index depends on block information and thread id. The first thread of a block will first copy found signal to shared found signal. If the nonce is found, the thread just returns. This cache-like approach limits the times to access global variable. When a thread found the answer, both global found signal and shared found signal will be set to true and the nonce value will be passed to result.

Under this structure, two thread found the answer may complete result at the same time or some threads may not terminate immediately after one of the other threads found the nonce. However, the first is not a big deal since both answers are fine and since the found signal can only be set from false to true, there is no racing condition occurs. As for the second doubt, compared with the overhead results from strictly state synchronization, redundant time for termination is more acceptable.

The block data will be copied to local variable to achieve efficiency.

110062118
賴姿妘

```cpp
__global__ void nonce_search_kernel(HashBlock block, unsigned char *target_hex, unsigned int start_nonce, unsigned int step, unsigned int *result)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int nonce = start_nonce + idx * step;
    __shared__ bool s_found  ;

    if(threadIdx.x==0){
        s_found = d_found;
    }


    if(s_found)
        return;

    HashBlock local_block = block;
    local_block.nonce = nonce;

    SHA256 sha256_ctx;
    double_sha256_kernel(&sha256_ctx, (unsigned char*)&local_block, sizeof(local_block));

    if (little_endian_bit_comparison_kernel(sha256_ctx.b, target_hex, 32) < 0) {
        printf("find!!!!!!!!!!!!\n");
        *result = nonce;
        s_found  = true;
        d_found  = true;
    }
}
```

✧ Optimization

I use **#pragma unroll on every for loop.** A silly baby step results in significant improvement. Before the modification, it takes about 36 seconds to finish four testcases, while 10 seconds are necessary with unrolled loop. It really amazes me. This is because unrolling reduces if-else branch conditionals and enables the GPU to utilize coalesced memory access.

```cpp
#pragma unroll
```

B. Experiment

I experimented with different combinations of blocks and threads. Based on the experiments, having 256 threads per block is the best choice.