平行程式設計 hw4 report

資工四 B10502136 邱垣盛

1 CUDA Implementation

The parallel implementation refactor the sequential design by moving the computationally expensive nonce search from the CPU to the GPU. The host is responsible for initial setup, including reading block data, calculating Merkle root, and pre-calculate the initial SHA-256 mid-state from the static parts of the block header. Then these data is copied to the GPU memory.

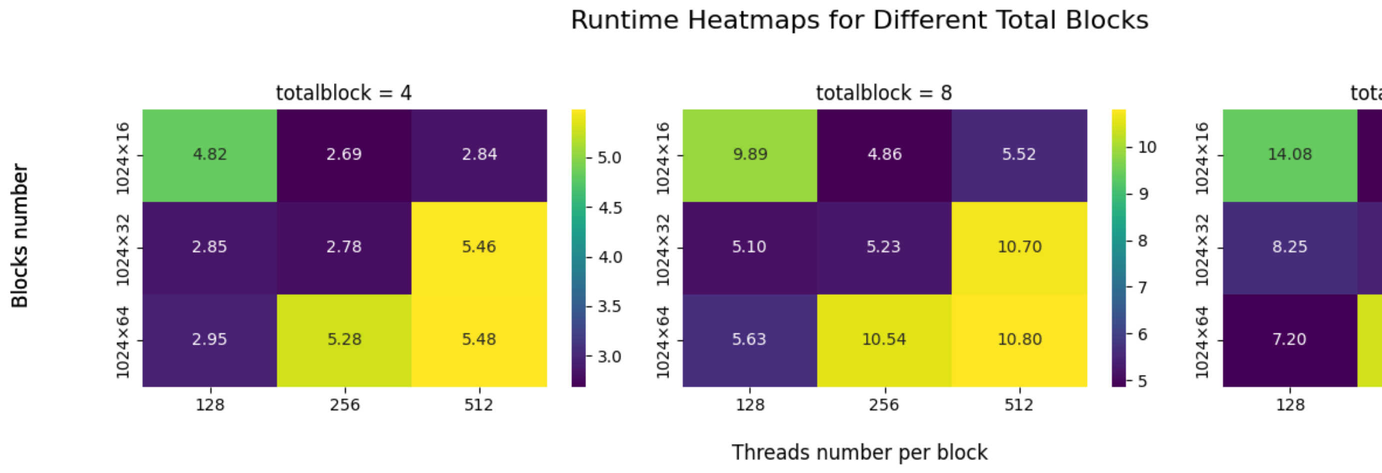
The new design divides the labor: the host CPU is now responsible only for setup, which includes reading the block data, calculating the Merkle root, and most importantly, pre-computing the initial SHA-256 "mid-state" from the constant parts of the block header. This mid-state is then transferred to the GPU, which launches a massive grid of thousands of threads. Each GPU thread independently and concurrently tests a portion of the nonce space, starting its hash calculation from the shared mid-state. This parallelizes the search, allowing millions of nonces to be checked simultaneously instead of sequentially, with the host simply polling for the solution found by the first successful thread.

A single, large CUDA kernel is then launched, assigning thousands of threads to concurrently search the entire 32-bit nonce space. Each thread efficiently checks a portion of the nonces using a grid-stride loop, beginning its hash calculation from the pre-computed mid-state to avoid redundant work. When any thread finds a valid nonce that meets the target difficulty, it atomically writes this solution to a found flag in global memory. The host CPU simply polls this flag, and as soon as a solution is written, the program can report the result and terminate.

2 Optimization Skills

Normal parallelization: Move the massive brute-force for loop in `solve()` to check all possible nonce from CPU to GPU, to leverage its thousands of cores.

Mid-state precomputation: In the sequential miner, in each nonce iteration, we have to fill in 80 bytes of block header then do SHA-256 computation twice. However, the first 64 bytes, which is version, prehash, and the first 28 bytes of merkle-root, are fixed, hence we don't need to overwrite them in every iteration. Instead, we could pre-caclculate a mid-state of these 64 bytes on CPU once, and handle the rest 16 bytes changeable data in GPU kernel, which eliminates the time on computing the first round of SHA-256.



Grid-Stride loop: In GPU kernel, each threads use `nonce\_ll += step\_stride` to loop, where `step\_stride` equals the number of threads in the grid times the number of nonce a thread handles consecutively (`STEP=4`). Also, this skill makes the implementation scalable on different hardware constraints.

Shared memory optimization: The mid-state and the second chunk of block data are constant for all threads within a block, and reading them from global memory in every loop iteration would be a major bottleneck. Therefore, I move them to `__shared__` memory, so all threads in the block could access it faster.

SHA-256 improvement: Originally a 64-word array `w` is used for message schedule, resulting in high register pressure and likely spills to slow local memory. To keep the schedule in registers, I use a 16-word circular buffer and fuse the message schedule calculation (`s0` and `s1` operations) directly into the main compression loop. Also, there is `if` case in `sha256_transform`, which might cause branch divergence, potentially avoid more aggresive optimization from the compiler, so I split it into two for loops, one for i=0 to 15 and the other for i=16 to 63, eliminating `if` case. Also, for these two for loops, I use `#pragma unroll` to unroll them, eliminating loop branch overhead.

Constant memory: 64 SHA-256 constants value are stored in `__constant__` memory `d_k`, which is cached and optimized for broadcast-style reads, where all threads in a warp read the same address. On the other hand, for CPU access there is an identical array `k` stored in `const` memory. By this design, both CPU and GPU access gain better performance.

3 Experiments

I tested 9 combinations in total, from blocks number $(1024 * 16, 1024 * 32, 1024 * 64)$ and threads number $(128, 256, 512)$, tested on 3 different `totalblock` $(4, 8, 12)$.

Here is the heatmap for the test result, we could observe that blocks number $= 1024 * 32$ and threads number $= 256$ generate best performance in different `totalblock`.

grid size/block size	128	256	512
1024 * 16	4.82/9.89/14.08	2.69/4.86/7.15	2.84/5.52/8.1
1024 * 32	2.85/5.1/8.25	2.78/5.23/7.99	5.46/10.7/15.07
1024 * 64	2.95/5.63/7.2	5.28/10.54/15.57	5.48/10.8/16.23

4 Advanced CUDA skills

Atomic operation for reporting: When a thread finds a valid nonce, it must write the value back to global memory so that the host can see it. Consider case that multiple threads find valid nonces at the same time, normal write might cause race condition. `atomicExch` guarantee only one thread could access `d\_found\_nonce` at a time, avoiding corruption.

Warp level step: After checking a fix number (16384) of nonce, a Warp would check if any threads found the answer nonce, if it finds the shared variable is modified, all threads would exit kernel immediately, avoiding unnecessary computation.

5 Feedback

I think the algorithm used in this homework is much simpler than the previous ones, making it easier to optimize parallelization, since I could have better understanding the core logic and adjust the parallelization method accordingly.