CS3210 Assignment 2 Report

Tan Xin You A0135812L

November 19, 2018

1 Implementation

1.1 `ProofOfWorkGenerator` Class

The main class that will calculate a valid digest from the parameters it was initialized with: `std::string prevDigest`, `std::string id`, `ulong target`.

Upon calling one of its `generate` function, `ProofOfWorkGenerator` calls `generateKernel` function to generate the valid digest.

1.2 `generateKernel` function

`generateKernel`, as shown in Figure 1, uses the following arguments:

`templateX` A pointer to the template for the calculating a valid digest. Basically, from the 415$^{\text{th}}$ bit to the 64$^{\text{th}}$ bit of X, proof of work, as described in the assignment (see Figure 2 for reference). This pointer is initialized with the `cudaMallocManaged` in the unified memory where any processor in the system can access, similar to defining a variable as a `__managed__`, allowing both device and host code to access this variable.

`nonce` A pointer to another unified memory created by `cudaMallocManaged` where a valid nonce will be stored in when found.

`digest` A `cudaMallocManaged` created pointer to an unified memory to store the digest of the proof-of-work with the stored `nonce`.

`target` The specified target, as described in the assignment.

`found` A `cudaMallocManaged` created pointer to an unified memory to store a `int` (but used like a boolean because `atomicCAS` does not allow `bool`) which indicates true when a valid nonce is found. A Cuda's `atomicCAS` function is used to check and toggle this value to prevent synchronization issues between threads, as shown in Figure 3. This way only one thread is able to modify the memory for `nonce` and `digest` in each execution.

```
__global__ void generateKernel(const uint8_t* templateX, ullong*
    nonce, uint8_t* digest, ulong target, int* found)
```

Figure 1: `generateKernel` function in `ProofOfWorkGenerator.cu`

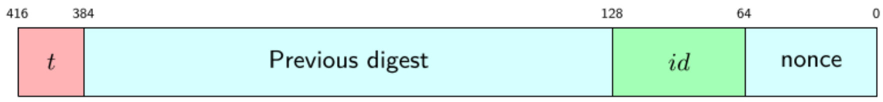

Figure 2: Screenshot of proof of work from assignment.

```
if(verified && !atomicCAS(found, false, verified))
{
    atomicExch(nonce, i);
    for(unsigned j=0; j<DIGEST_SIZE_IN_BYTES;++j)
    {
        digest[DIGEST_SIZE_IN_BYTES - j - 1] = hash[j];
    }
    return;
}
```

Figure 3: Update `nonce` code in `generateKernel` from `ProofOfWorkGenerator.cu`

Each thread will execute `generateKernel` in these steps:

1. Calculates the number of values to try as nonce.
2. Calculates its own thread id, unique throughout the system.
3. Copy `templateX` from the unified memory.
4. Try the range of values iteratively until `found` is `true`
   (a) Calculate the hash using the given `sha256` function
   (b) Verify if the first 64 bits of the hash is below `target`
   (c) If below target, it is a valid nonce. Store the found nonce and set `found` to be `true`.

2 Assumptions

2.1 Given SHA256 code accepts proof-of-work in Big-Endian

At least this is my understanding from what I read from the code. I used little-endian in my code mostly in the beginning as it was easier for me to code but realized later that the given `sha256` code uses big-endian. There wasn't much of an issue but the reader should not be shocked by the usage of little-endian in parts of the code.

2.2 Max dimensionality of grid and block configuration

I assumed 2D to be the maximum dimensionality to be used. I felt there is little use in using 3D configuration as my code does not require communication between threads, thus no need for complex configuration to aid communication.

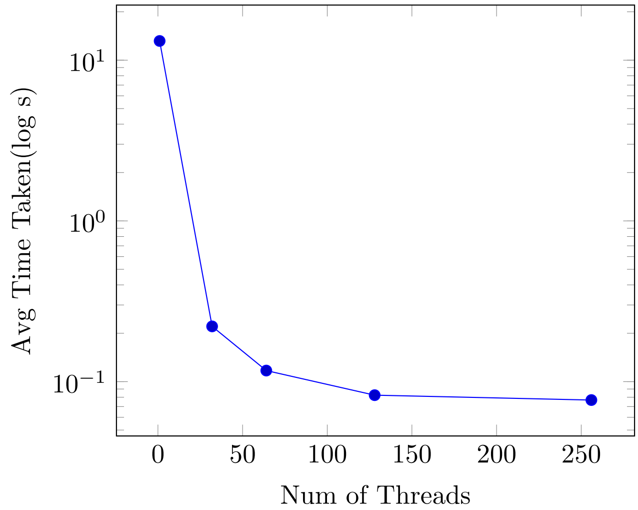

Figure 4: Time Taken to find valid nonce for `1.in` on the jetson machine.

2.3 Using different epoch on every trial

Due to the requirement, in order to compute a proof-of-work, the epoch used must be different on every trial. This creates more uncertainty in the experiments as some epoch may cause the proof-of-work to require a nonce lower in order of the range which a thread tries. However, we assume that running many enough trials per configuration would smoothen out this effect.

3 Results

3.1 Experiment set up (for result reproduction)

Experiments to get measurement were done on the compute cluster machines(xgpd0) and the jetson machine in the lab (jetsontx2-03). One can run `make benchmark` to run the same experiments. Input files used in the experiments are:

`1.in` As per the example in assignment. $target = 2^{48}$.

`2.in` Same digest as `1.in`, $target = 2^{40}$.

`3.in` Same digest as `1.in`, $target = 2^{32}$.

`4.in` Digest: a6d569d489eca7f807e2edad9876473b918694ef68b5125585f5a9d667224033, $target = 2^{48}$

`5.in` Same digest as `4.in`, $target = 2^{40}$

The grid configuration used is 64x1 and block configuration used are 1x1, 32x1, 64x1, 128x1, 256x1. Each of the input files will be run using all of these configurations except for 1x1 which I conducted experiments only for `1.in` and `2.in` as it takes too long to run.

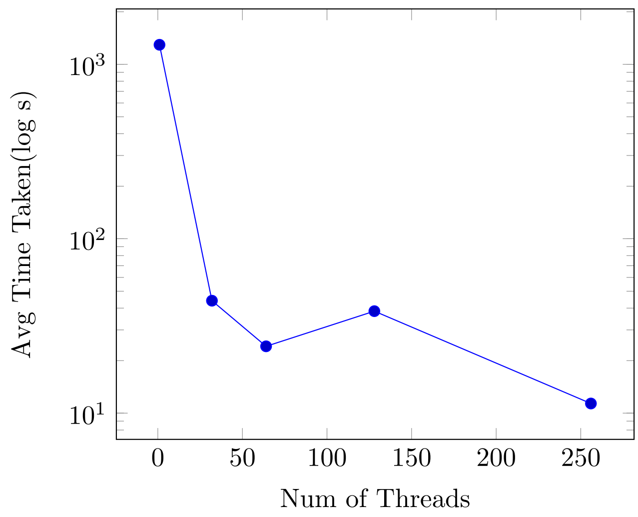

Figure 5: Time Taken to find valid nonce for `2.in` on the jetson machine.

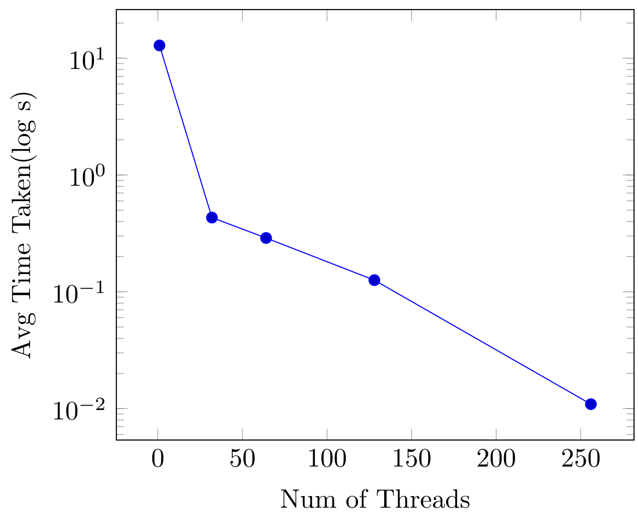

Figure 6: Time Taken to find valid nonce for `1.in` on the xgpd machine.

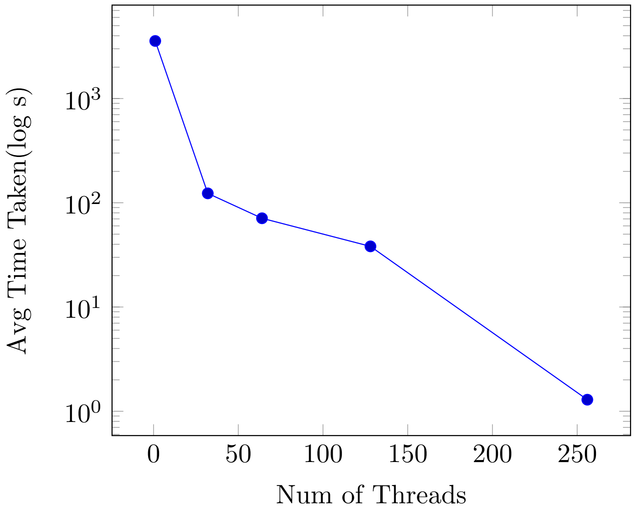

Figure 7: Time Taken to find valid nonce for `2.in` on the xgpd machine.

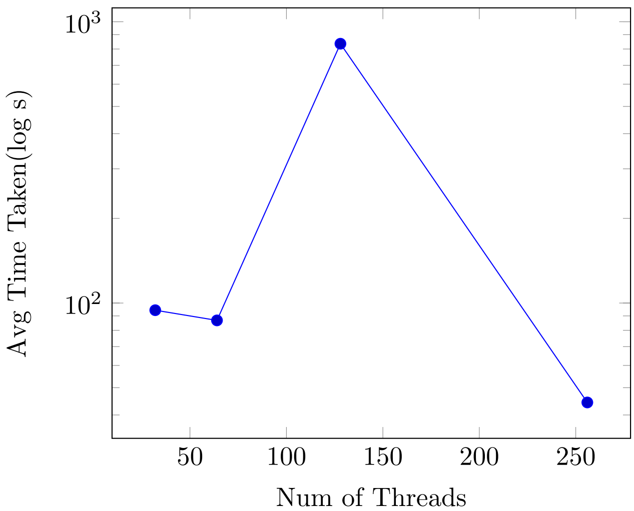

Figure 8: Time Taken to find valid nonce for `3.in` on the xgpd machine.

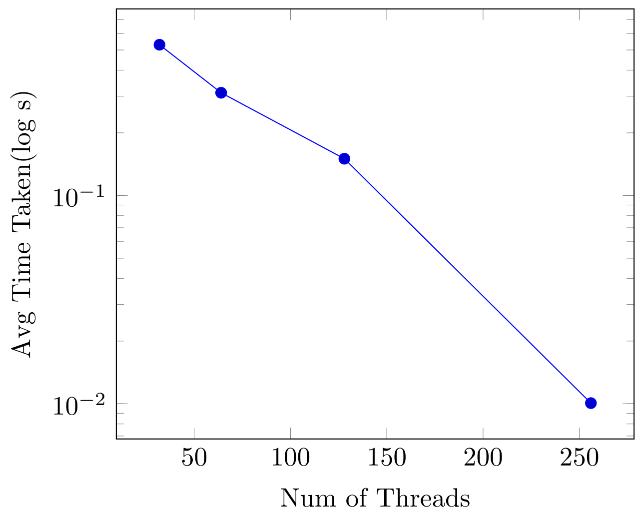

Figure 9: Time Taken to find valid nonce for `4.in` on the xgpd machine.

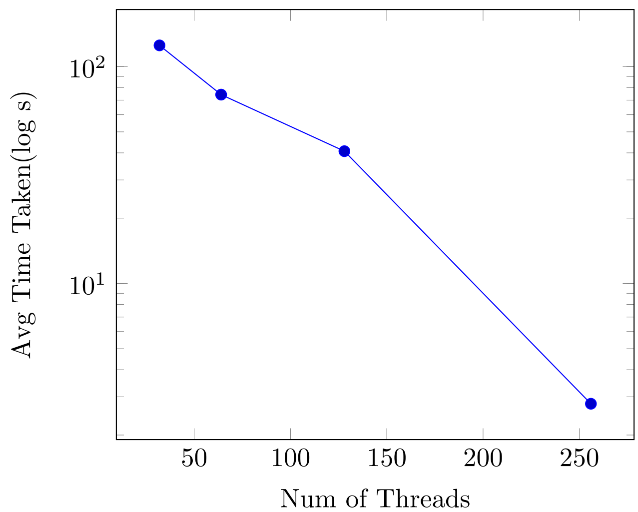

Figure 10: Time Taken to find valid nonce for `5.in` on the xgpd machine.

Input	Num of Threads	Avg Time Taken(s)	Max Time Taken(s)	Min Time Taken(s)
1	1	13.169	30.457	1.643
1	32	0.221	0.395	0.119
1	64	0.117	0.215	0.030
1	128	0.082	0.152	0.013
1	256	0.077	0.234	0.005
2	1	1293.854	2359.570	148.075
2	32	44.115	125.946	0.613
2	64	24.156	41.587	1.922
2	128	38.430	128.273	1.015
2	256	11.353	22.836	1.604

Table 1: Measurements taken from experiments ran on jetson machine.

Input	Num of Threads	Avg Time Taken(s)	Max Time Taken(s)	Min Time Taken(s)
1	1	12.856	17.806	8.627
1	32	0.432	0.551	0.347
1	64	0.289	0.448	0.193
1	128	0.126	0.168	0.105
1	256	0.011	0.038	0.002
2	1	3566.627	5056.150	2491.340
2	32	123.103	199.371	87.614
2	64	71.015	84.997	53.711
2	128	38.182	65.237	31.374
2	256	1.290	11.760	0.024
3	32	94.306	266.079	10.865
3	64	86.758	180.735	1.736
3	128	835.417	6568.860	13.180
3	256	44.314	88.777	11.281
4	32	0.530	0.842	0.316
4	64	0.312	0.384	0.203
4	128	0.150	0.200	0.123
4	256	0.010	0.026	0.001
5	32	125.194	174.482	75.851
5	64	74.229	97.105	41.562
5	128	40.720	69.115	25.442
5	256	2.788	21.084	0.091

Table 2: Measurements taken from experiments ran on xgpd machine.

3.2 Observations

Generally, as number of threads used increases, we are able to reduce the time taken to find a valid nonce, as seen in Figure 4. The same trend can be observed in the xgpd machine as seen in Figure 6.

Additionally, results are much more unstable as less threads are used. From Table 2, we can see that there is greater variance in the time taken by observing the difference between *Max Time Taken(s)* and *Min Time Taken(s)*, this shows that using different epoch across the different trials has a great effect, and thus one has to be careful when comparing such results. The extent of this effect can be seen in Figure 5, using 128 threads per block used has a larger *Avg Time Taken(s)* compared to using 1, 32 or 64 threads. However, this anomaly is caused by a single trial out of the ten, which happen to have a valid nonce ranked higher in order of the threads' search range.

4 Modifications

4.1 Using busy wait instead of `CudaDeviceSynchronize`

My initial design requires the CPU thread to wait for the termination of all threads via the use of `CudaDeviceSynchronize`. I realize that this becomes counter-productive when I try to scale using more blocks and threads, see Figure 11. The `__host__` code will have to wait for more threads to terminate as it scales resulting in time wasted before the program outputs the result. I circumvent this by using a busy wait strategy in the `generateBusyWait`, see Figure 12. The result is an average of $10 - 15\%$ speedup in large enough grid and block configuration. One might think that the speedup shouldn't have been that significant, as most threads should immediately terminate when it `return` after reading `found` as `true`. However, when using big enough configuration, there is a lot of context switching involved when each new block runs, I suspect this is the reason for the significant speedup. Until Cuda develops a better way to terminate all running threads and/or stop them before they get run, this workaround seems to be the only way to avoid waiting for complete termination of the threads. However, resources will still be used for these threads' initialization, one can imagine a worse situation if the kernel function used requires more registers, or if each thread initialization involves more memory copying work.

4.2 Using thread id to calculate its search range

I made a mistake of having the CPU thread calculate the range each thread has to search within and then sending those values to each of these threads. On hindsight, that was a simple solution as I do not have to deal with the calculation within the GPU threads. This was a huge mistake. Not only did that code waste time computing each of the threads' work, time was wasted communicating those values to the threads too. Not to mention the amount of memory being copied from device to device then to each of the thread's memory stack.

```
void ProofOfWorkGenerator::generateCudaDeviceSynchronize()
{
    dim3 gridDim(this->gridDimX, this->gridDimY);
    dim3 blockDim(this->blockDimX, this->blockDimY);

    generateKernel<<<gridDim, blockDim>>>(this->templateX, this->
    nonce, this->digest, this->target, this->found);
    cudaDeviceSynchronize();

    if(*(this->found) == 1)
    {
        cerr << "Found " << this->getNonce() << endl;
    } else
    {
        cerr << "Failed\n";
    }
    check_cuda_errors();
}
```

Figure 11: `generateCudaDeviceSynchronize` function in texttttProofOfWork-Generator.cu

4.3 Using randomizer to achieve stable results

In an attempt to achieve a better expected performance, I used `cuRAND` a cuda library that allows usage of randomizer in the Cuda kernel. Instead of iteratively trying every value in its search range, a thread would randomly pick a value from its search range to try as depicted in Figure 13. Using this function, did result in slightly better expected result with lower variance but due to it picking a random value to try, it could pick the same value and thus not try all the value in its search range, causing the program to be have a less likelihood to find a valid nonce. In order to avoid this, one can increase the number of times to sample and try, but this is not within the scope of this paper.

5 Discussion

It seems OOP design does not suit the Cuda API that well (yet...) due to the need to use memory modifiers/qualifiers such as `__device__` and `__global__`. I was able to work around by dynamically creating memory using `CudaMallocManaged`, but this would have been unnecessary if `__managed__` is used instead, which in this assignment one could have as all sizes are determined.

```
void ProofOfWorkGenerator::generateBusyWait()
{
    dim3 gridDim(this->gridDimX, this->gridDimY);
    dim3 blockDim(this->blockDimX, this->blockDimY);

    generateKernel<<<gridDim, blockDim>>>(this->templateX, this->
    nonce, this->digest, this->target, this->found);
    while(*(this->found) != 1);
    cerr << "Found " << this->getNonce() << endl;
    check_cuda_errors();
}
```

Figure 12: `generateBusyWait` function in textttProofOfWorkGenerator.cu

```
double randDouble = curand_uniform_double(my_curandstate);
randDouble *= (stop - start+0.999999);
randDouble += start;
ullong randUllong = (ullong)truncf(randDouble);

ullong_to_uint8_big_endian(candidateX + len, randUllong);
```

Figure 13: Main difference between `generateKernelWithRand` function and `generateKernel` function in `ProofOfWorkGenerator.cu`