Weekly Report - A rough Profiling Attempt to Ethash

MSc Project
Runchao Han

December 15, 2017

1 Progress of the Last Week

1. Did a rough profiling to Ethash by adding timestamps
2. Learned about Nvidia GPU and CUDA programming
3. Learned about profiling CUDA programs by `nvprof` and `Nvidia Visual Profiler`

2 Introduction to CUDA profilers

To profile CUDA programs, tools are involved. After searching for approaches, I found the only two profiling tools for CUDA programs:

- `nvprof`[1]. This is a command line tool.
- `Nvidia Visual Profiler`[2]. This is an Eclipse-based profiling tool which wraps `nvprof`.

A rough profiling was done by `nvprof`, by which a statistic was outputed, shown in Fig. 1.

```
==13285== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   92.09%  12.4960s        16  781.00ms  774.86ms  801.24ms  ethash_calculate_dag_item(unsigned int)
                    7.89%  1.07077s         9  118.97ms  117.53ms  122.84ms  void ethash_search<unsigned int=4>(unsigned int volatile *, unsigned long)
                    0.02%  2.1957ms         7  313.67us     576ns  2.1912ms  [CUDA memcpy HtoD]
      API calls:   99.25%  12.5002s        17  735.31ms  14.055us  801.39ms  cudaDeviceSynchronize
                    0.68%  85.936ms         1  85.936ms  85.936ms  85.936ms  cudaDeviceSetCacheConfig
                    0.02%  2.2206ms         1  2.2206ms  2.2206ms  2.2206ms  cudaMemcpy
                    0.02%  2.1484ms        25  85.934us  44.026us  174.14us  cudaLaunch
                    0.01%  1.1894ms         2  594.69us  140.81us  1.0486ms  cudaMalloc
                    0.00%  546.85us         2  273.42us  259.82us  287.03us  cudaGetDeviceProperties
                    0.00%  519.02us         2  259.51us  8.7850us  510.23us  cudaHostAlloc
                    0.00%  479.18us        94  5.0970us     257ns  203.27us  cuDeviceGetAttribute
                    0.00%  311.31us         1  311.31us  311.31us  311.31us  cudaDeviceReset
                    0.00%  175.22us         1  175.22us  175.22us  175.22us  cuDeviceTotalMem
                    0.00%  136.82us         6  22.804us  4.5900us  75.945us  cudaMemcpyToSymbol
                    0.00%  107.75us        25  4.3090us  1.2800us  19.649us  cudaConfigureCall
                    0.00%  58.341us         1  58.341us  58.341us  58.341us  cuDeviceGetName
                    0.00%  46.110us        34  1.3560us     452ns  2.4720us  cudaSetupArgument
                    0.00%  40.829us         2  20.414us  12.307us  28.522us  cudaStreamCreate
                    0.00%  12.052us         1  12.052us  12.052us  12.052us  cudaSetDevice
                    0.00%  10.801us        10  1.0800us     759ns  2.2330us  cudaGetLastError
                    0.00%  7.4830us         1  7.4830us  7.4830us  7.4830us  cudaProfilerStart
                    0.00%  5.6550us         4  1.4130us     838ns  1.7840us  cudaGetDeviceCount
                    0.00%  3.8980us         3  1.2990us     465ns  2.6790us  cuDeviceGetCount
                    0.00%  1.8660us         1  1.8660us  1.8660us  1.8660us  cudaSetDeviceFlags
                    0.00%  1.8350us         2     917ns     362ns  1.4730us  cuDeviceGet
```

Figure 1: The `nvprof` result of *run_ethash_search*().

The result shows that *ethash_calculate_dag_item*() costs most of the time, which generates the whole 1GB DAG by a seed. However, the DAG generation can be pre-computed or copied from

[1] http://docs.nvidia.com/cuda/profiler-users-guide/index.html
[2] http://www.sie.es/wp-content/uploads/2015/12/cuda-profiling-tools.pdf

others, which makes optimising this process meaningless. The optimisation target is the mining process, which is the $compute\_hash()$ function in the code.

Currently, I have not succeeded in importing the project into `Nvidia Visual Profiler`. This is the main target next week.

3 Profiling $compute\_hash()$ Function

Due to the limitation of the command line, it is hard to get statistics of different steps which are set by myself. Therefore, I set timestamps manually in the code.

The steps of conducting the profiling are listed below:

1. Set the CUDA mining function involves only one block and the block contains only one thread.
    - A CUDA Kernel function takes a grid (fixed)
    - A grid takes several blocks (modifiable)
    - A block takes several threads (modifiable)
2. Add timestamps in the code. The code involves steps listed below:
    (a) State initialisation
    (b) An iteration of 4 (t)
        i. An iteration of 4 (ti1)
        ii. An iteration of 16 (ti2)
        iii. An iteration of 4 (ti3)

The time of initialisation and three inner iterations of a single outer iteration is recorded by timestamps. (A problem is that if I record the whole outer iteration execution time, the long long int will even be overflowed. I will think about solving this next week.) The output is shown in Fig. 2.

The unit of values is the time of the GPU clock. The result indicates that the second inner iteration ti2 takes most of the time. However, it is uncertain that if different single outer iterations make inner iterations take different time. Further research is needed, which is the next week's plan.

It is noted that the implementation uses the most up-to-date CUDA APIs, like $\_\_shfl()$ and $\_\_shfl\_sync()$. Further research is needed to figure out these APIs.

4 Miscellaneous

- I decided to use Eclipse CDT to conduct experiments, which I found very convenient.
- I have learned basic CUDA programming by official documentations and examples. Currently I have basic understanding on the Ethminer code.

5 Next Week's Plan

1. Further profile the Ethash algorithm
2. Import the Ethminer to `Nvidia Visual Profiler` and get a better profiling result
3. Produce the Stack Graph if possible

No reference this week because the work in this week is about profiling the source code, which is fairly an engineering problem.

```
nonce-> 3128178979190234009;t1 -> 8652.000000
nonce-> 3128178979190234009;ti1 -> 1519.000000
nonce-> 3128178979190234009;ti2 -> 56217.000000
nonce-> 3128178979190234009;ti3 -> 680.000000

nonce-> 3128178979191282585;t1 -> 8878.000000
nonce-> 3128178979191282585;ti1 -> 1153.000000
nonce-> 3128178979191282585;ti2 -> 56133.000000
nonce-> 3128178979191282585;ti3 -> 633.000000

nonce-> 3128178979192331161;t1 -> 8903.000000
nonce-> 3128178979192331161;ti1 -> 1401.000000
nonce-> 3128178979192331161;ti2 -> 56043.000000
nonce-> 3128178979192331161;ti3 -> 743.000000

nonce-> 3128178979193379737;t1 -> 8654.000000
nonce-> 3128178979193379737;ti1 -> 1521.000000
nonce-> 3128178979193379737;ti2 -> 56320.000000
nonce-> 3128178979193379737;ti3 -> 681.000000

nonce-> 3128178979194428313;t1 -> 8661.000000
nonce-> 3128178979194428313;ti1 -> 1387.000000
nonce-> 3128178979194428313;ti2 -> 56803.000000
nonce-> 3128178979194428313;ti3 -> 684.000000

nonce-> 3128178979195476889;t1 -> 8817.000000
nonce-> 3128178979195476889;ti1 -> 1491.000000
nonce-> 3128178979195476889;ti2 -> 56320.000000
nonce-> 3128178979195476889;ti3 -> 938.000000

nonce-> 3128178979196525465;t1 -> 8652.000000
nonce-> 3128178979196525465;ti1 -> 1517.000000
nonce-> 3128178979196525465;ti2 -> 56451.000000
nonce-> 3128178979196525465;ti3 -> 771.000000
```

Figure 2: Profiling *compute_hash*() by adding timestamps.