Weekly Report - Learning CUDA to Understand the Code

MSc Project
Runchao Han

December 28, 2017

1 Progress of the Last Week

1. Successfully imported the Ethminer project to Nvidia Visual Profiler
2. Learned about CUDA basic principles and programming
3. Compared two Ethash implementations in Ethminer
4. Got more accurate data

2 Imporing Ethminer to Nvidia Visual Profiler

The problem I met last two weeks is that I installed two CUDA versions, and the version in `/bin` is the false one.

After I substituted related binary files with correct ones, the importing succeeded, as shown in Fig. 1.

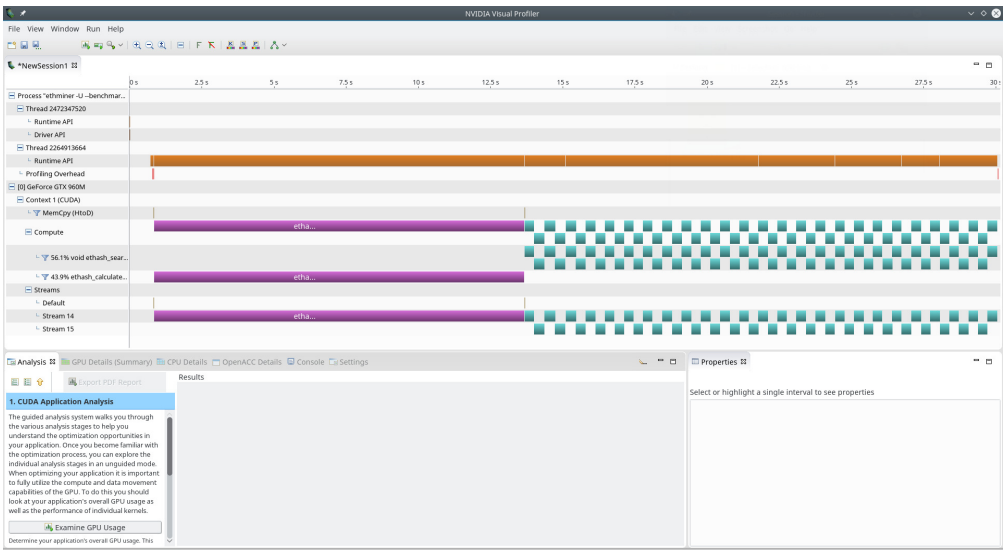

Figure 1: Importing the Ethminer to Nvidia Visual Profiler

However, this tool is unsuitable for profiling a small part of the code in a big project as far as I know. Before I isolate the critical code from the project, I will still use the timestamp approach.

3 CUDA Basic Principles

This two week I spent most time on learning GPU and CUDA, because my hardware knowledge is still rudimentary.

3.1 An Introduction to GPU

Graphical Processing Unit (GPU) was designed for computer graphic applications at first, focusing on parallel scientific computing. The comparison between CPU and GPU is shown in Fig. 2.

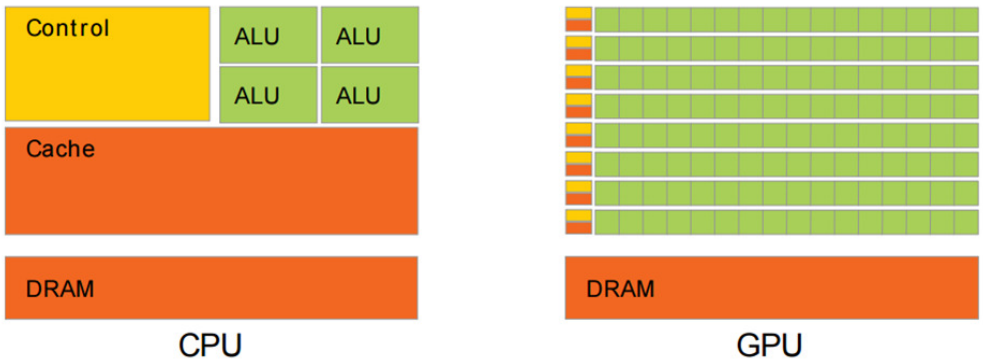

Figure 2: The architecture comparison between CPU and GPU

It is shown that GPU contains much more ALUs which execute computations. Moreover, state-of-the-art GPUs contain a large number of cores so suitable for parallel computing.

3.2 The Lifecycle of a CUDA Program

CUDA was introduced in previous reports, so this section focuses on the execution of a CUDA program.

Fig. 3 shows the logical view of CPU, GPU, main memory and GPU memory.

A CUDA function is called a "Kernel function".

Typically, a CUDA program is executed by following steps:

1. Launch the binary file
2. Load data to the main memory
3. Allocate space to data structures in the Graphics Card memory
4. Copy data from the main memory to the Graphics Card memory
5. Invoke the kernel function executed by GPU
6. Copy the result from the Graphics Card memory to the main memory

3.3 Progammers' Perspective: Grid, Block and Thread

The concepts of grid, block and thread are on the program level to provide support of organising threads for programmers, as shown in Fig. 4.

A kernel function initialises a grid containing multiple blocks, while a block contains multiple threads. Both blocks and threads are organised as a 3D array.

3.4 Hardware's Perspective: SP, SM and Warp

SP means the `Streaming Processor`, while SM means the `Streaming Multiprocessor`, the relationship between which is shown in Fig. 5.

A SP is a core executing instructions, also called CUDA core. SM is a big core consisting of multiple SPs and other resources like registers and warp schedulers, which can be treated as a CPU with multiple cores.

Warp is the basic unit of scheduling containing 32 threads, where 32 threads are executing the same instruction (SIMT). A warp should take a SM to execute, so warps take a SM in turn.

3.5 Basic APIs

3.5.1 Function Type Specifiers

Function type specifiers specify the execution place of a function, which are:

- *__device__*: The function is invoked and executed on GPU.
- *__glocal__*: The function is invoked by CPU and executed on GPU, so called "Kernel function".
- *__host__*: The function is invoked and executed on CPU. This is the default option where the function is a traditional C function.

3.5.2 Variable Type Specifiers

Variable type specifiers specify the place of variable storage, which are:

- *__device__*: The data is stored in the Graphical Card memory which every thread can access.
- *__shared__*: The data is stored in the shared memory in GPU. Only threads in the block can access it.
- *__constant__*: The data is stored in the constant memory which every thread can access.

3.5.3 Existing Variables

5 built-in variables in CUDA are used for getting information about the grid, block and thread indices, which are:

- gridDim: A struct with (x, y, z), which specifies the size of the 3D grid.
- blockDim: A struct with (x, y, z), which specifies the size of the 3D block.

- blockIdx: A struct with (x, y, z), which specifies the index of the block of the current thread in the grid
- threadIdx: A struct with (x, y, z), which specifies the index of the thread of the current thread in the block
- warpSize: The size of the warp. (Compute Capability¿1.0?32:24)

3.6 The New Shuffle APIs

In the newest CUDA version 9.0, the Shuffle APIs are created. $\_\_shfl()$ is used for exchanging variables among threads in a warp. Compared to the conventional shared memory, Shuffle can do the same thing with the better efficiency. More information can be found at [1].

Ethash is implemented by both ways: the shared memory and the Shuffle. I just have figured out CUDA basics so I only tested the Shuffle implementation.

4 A More Detailed Profiling on Ethash (Shuffled)

This time I did a detailed profiling, but still haven't figured out the implementation because of CUDA (Now it is much better). I uploaded my data and my Jupyter notebook to Github.[2]

However I haven't draw a stack graph, because the data is still coarse grained and some parts take time much longer than other parts, making the graph hard to read.

5 Miscellaneous

6 Next Week's Plan

1. Figure out the Ethash code and comment on every line
2. Output an analysis of the Ethash source code

[1] https://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec4.pdf
[2] https://github.com/SebastianElvis/MScProject/tree/master/tests/Ethash_tests

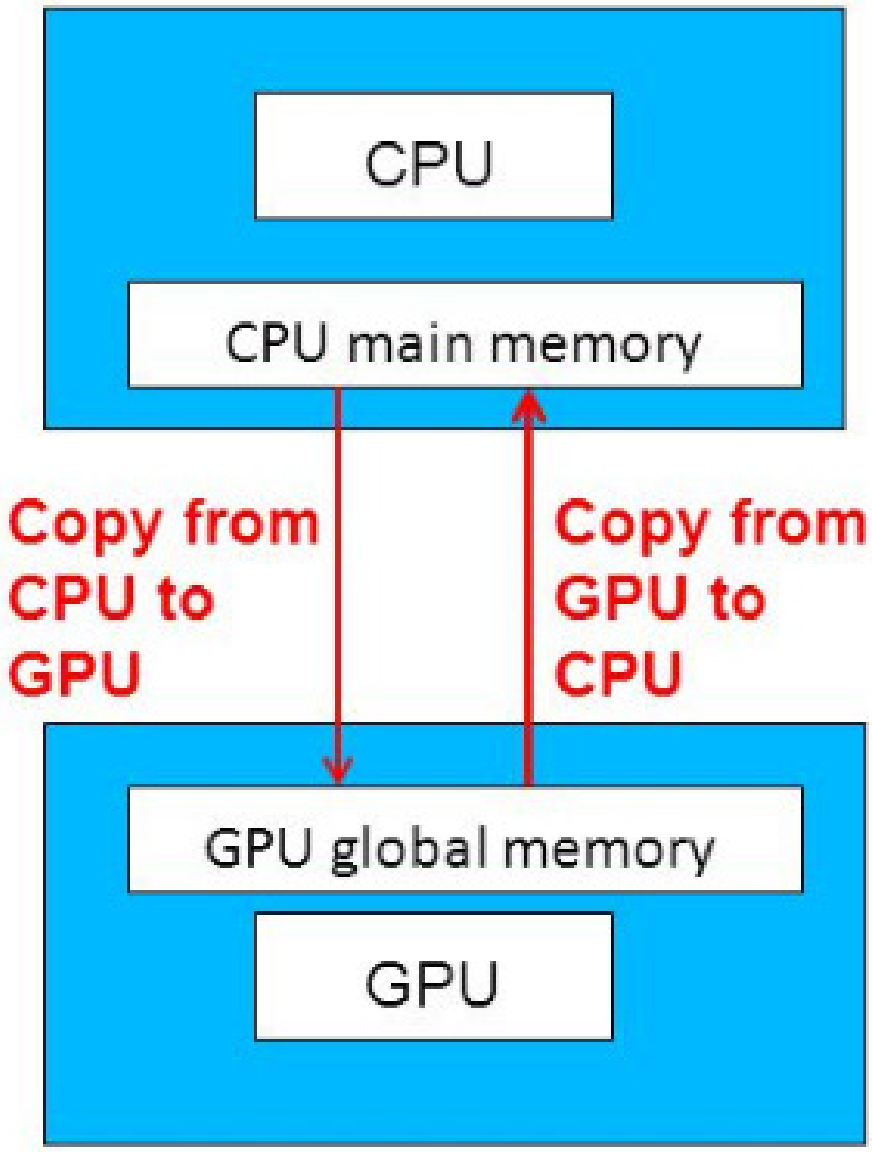

Figure 3: The architecture comparison between CPU and GPU

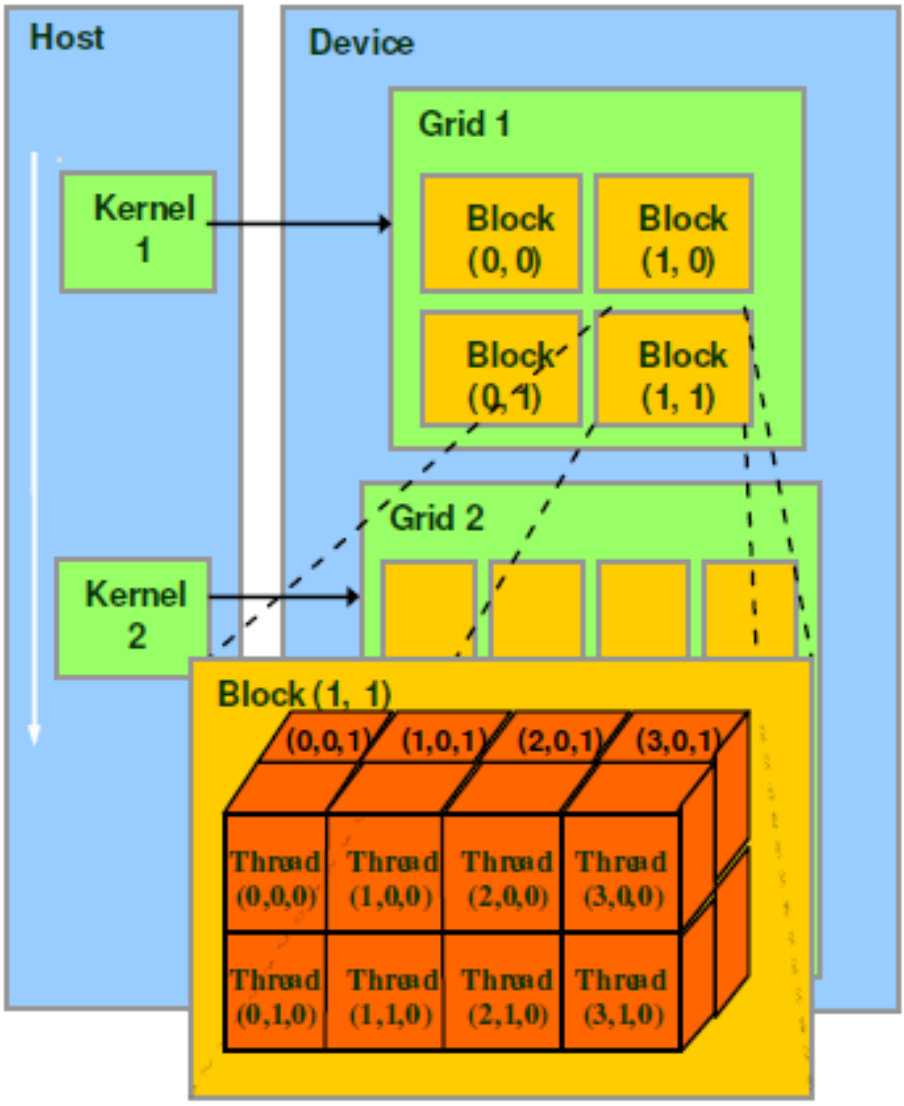

Figure 4: The programmers' perspective of a CUDA program.

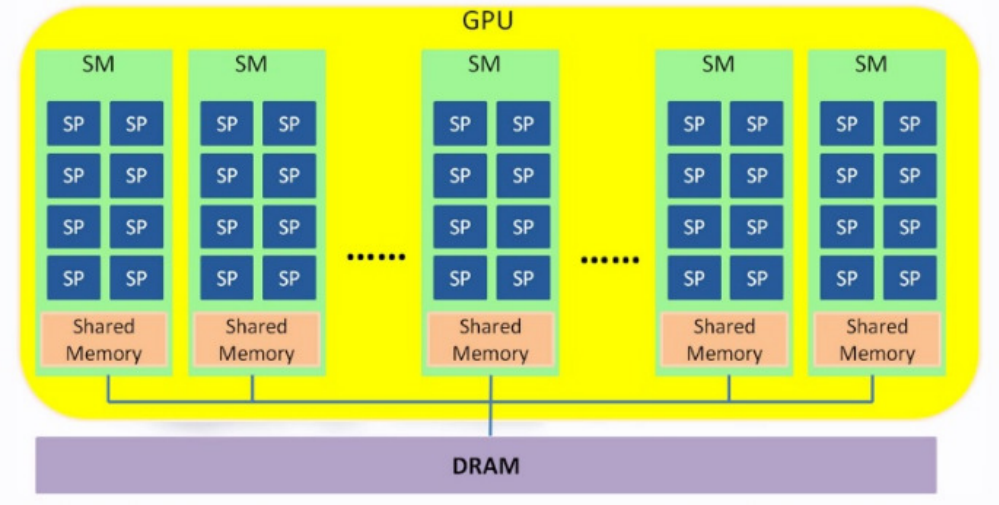

Figure 5: The relationship between SP and SM