Assignment 2
Parallelising hash-based proof-of-work system using CUDA

CS3210 – 2020/21 Semester 1

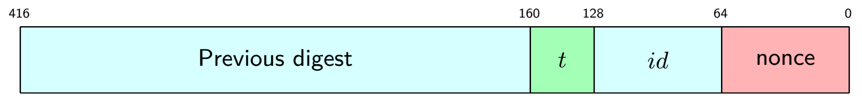
Learning Outcomes
This assignment provides you with the opportunity to demonstrate your understanding in building a parallel application using NVIDIA CUDA.

Problem Scenario: Bitcompute – A Proof-of-work-based System

In this assignment, you are required to implement a CUDA-based participant program that takes part in a proof-of-work-based system called **Bitcompute**. This system is created to validate transactions done with a particular digital currency, Bitcompute. For a transaction to be validated by the system, the transaction needs to be accompanied by a proof-of-work. Only once the proof-of work is validated, the system will validate and accept the transaction. This ensures that that the system will validate only transaction submitted with a proof-of-work. The proof-of-work deters service abuses (such as spam) by requiring some work from the service requester.

Proof-of-work challenge

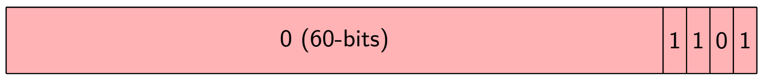
The Bitcompute system utilises a hashcash-based proof-of-work. This system accepts, as proof-of-work, a value X for which the digest of X (i.e. `SHA256`(X)) has a prefix that fulfills certain conditions (details in next section). A participant that needs to validate a transaction has to find such a value X. To check the digest for X, the participant should start by creating the value of X. X is 416-bits in length and is constructed as follows (most significant bit on the left):



where

- **previous digest** is the digest of the preceding transaction. This is provided by the system as input for computing a proof-of-work.
- t represents the UNIX timestamp (seconds since UNIX epoch), as an unsigned 32-bit number. The value of t used must be between the time the process of finding X is instantiated and completed (when proof-of-work is used).
- id refers to the transaction ID in `char` representation.
- **nonce** is an unsigned 64-bit number of your choice

For a transaction to be validated, the participant has to try out possible values of **nonce** till it finds a value X for which the prefix of the digest `SHA256`(X) has to be less than a specified target value n. This is based on the binary representation of the first 64-bits of the digest. For instance, a particular `SHA256`(X) has the following 64-bit prefix:



If the target value n is more than 13 (1101 in binary), this proof-of-work is accepted by the system.

Sample Example

A participant wishes to validate a transaction in the Bitcompute system. Here are some pertinent details:

- The digest of the previous transaction is
  `3210327c68bb9409c4aa5806a4c018e26dcd2ca599a5cbccfaf09c886f701b71`
  (in hexadecimal representation)
- The validation is made at around 7:00PM on 01/10/2020, which corresponds to the UNIX epoch of 1601555562
- The transaction ID is T3210_01 (ASCII Hex: `54333231305f3031`)
- The target value n of the system at that time is $1,073,741,824$

The participant needs to find a possible value for the nonce. Consider the candidate value 7618289129662652849 (in hex: $69b9999999c321b1$). We shall now check if the value of X created using this nonce is accepted. The value of X is crafted using the nonce and the other information given as follows:

`3210327c68bb9409c4aa5806a4c018e` `26dcd2ca599a5cbccfaf09c886f701b71`	`5f75cc6a`	`54333231305f3031`	`69b9999999c321b1`

Then, `SHA256(`X`)` (i.e. digest) is computed to be the following (first 32-bits separated):

`000000000be42789`	`b70bf62143896f7b3bc8784e31282cafa7389d9bbb7933ec`

As we can see, the first 64-bits of the digest, 000000000be42789, which is $199,501,705$ in decimal, is less than the target value of $1,073,741,824$. Thus, the value of X crafted would be accepted as the proof-of-work for the Bitcompute system.

Your Assignment: A Simple CUDA-based Participant Program for Bitcompute System

With your understanding of the Bitcompute system, you are required to write a participant program that finds a proof-of-work given an input. As the name **proof-of-work** suggests, there is a certain level of "computational effort" required to produce a digest that follows the pattern. In particular, this effort scales with the target value n that is set: the lower it is, the more difficult it becomes.

However, you would be able to reduce this effort by means of parallel computing and programming design. The only restriction on this assignment is that your submission must be a **CUDA-based parallel program**. That means you should demonstrate and showcase your knowledge of the various CUDA concepts (e.g. how threads are executed), in exploring improvements to your CUDA parallel implementation.

We have provided a CUDA-friendly implementation of SHA256 in `hash.cu`. Feel free to modify the file, **as long as the change does not affect the correctness of the hash function**. `hash.cu` should only contain SHA256-related code; the main implementation of the participant program should reside in a separate file.

Some of you might have taken computer security or cryptography related modules. SHA256 (the hash function used) is susceptible to certain attacks (e.g. length extension). Such attacks are not acceptable as performance improvements (it is doubtful that they are even useful). Document any changes you have made to `hash.cu` in your report.

Your parallel implementation should result in a meaningful speedup. The code should not take hours (or even more than 15 minutes) to produce a nonce. If this is the case, try a higher target value.

Inputs and Outputs

Your program should accept, as input, the following values in the following order:

- d – 256-bit digest of the preceding transaction (in hex format)
- tid – transaction ID as a string of exactly 8 characters
- n – the current target value that the system accepts as a valid "proof-of-work" (in decimal)

Sample Input

```
3210327c68bb9409c4aa5806a4c018e26dcd2ca599a5cbccfaf09c886f701b71
T3210_01
1073741824
```

The output from the program is as follows:

- `epoch` – the UNIX epoch utilized in the "proof" (in decimal)
- `nonce` – the nonce (in decimal)
- `digest` – resulting digest (i.e. `SHA256(`X`)`) (in hex format)

Sample Output

```
1601555562
7618289129662652849
000000000be42789b70bf62143896f7b3bc8784e31282cafa7389d9bbb7933ec
```

It is acceptable for the output from your program to be **non-deterministic** (i.e. different outputs across different runs).

Improvements and Analysis

While correctness is important in crafting a parallel program, the performance (for instance, speedup) is also an important component in a parallel implementation. After implementing the CUDA-based program, you should investigate various modifications of the code and how they affect different parallel performance metrics (e.g. speedup). These modifications include, but are not limited to:

- Different block and grid sizes. **Your implementation should work on varying grid and block sizes.**
- Different hardware/Compute Capability. You should run your program on at least two different GPUs (preferably with two different compute capabilities).
- Different data/task distribution methods

These investigations and their associated analyses will constitute a large part of your assignment grade. As a rule of thumb, **quality over quantity** (i.e. investigate meaningful modifications thoroughly, rather than to cover all possible modifications).

In order to collect useful data from your CUDA program, you may need to use `nvprof`, a profiler for CUDA. Refer to the Profiler Users Guide for more information.

What Can I Use? What Is Provided?

For this assignment, the following resources should be used to complete the assignment:

- `hash.cu` – contains a CUDA version of SHA256 (as explained earlier)
- **GPU Nodes in SoC Compute Cluster** – SSH into one of the following 10 nodes that have been reserved for you: `xgpc5-9`, `xgpf5-9`. You must use your SoC account to use these nodes. Furthermore, you must enable access to the SoC Compute Cluster from your MySoC Account page. If you are connecting from outside NUS, SSH first into the SoC network (sunfire.comp.nus.edu.sg). These nodes should be used for testing and measurement purposes. Take note of which node you are using for your testing.

Frequently Asked Questions (FAQ)

As per Assignment 1, this file contains FAQs received from students for this assignment. Please check this file first before asking a question.

If there are still any unanswered questions regarding the assignment, please post them on the LumiNUS forum or email Cristina (ccris@comp.nus.edu.sg).

Useful resources for Assignment 2

- CUDA Programming Guide
- CUDA `nvprof` Guide

FAQ

- FAQ file for Assignment 2

Admin Issues

You are allowed to work in groups of maximum two students for this assignment, and can discuss the assignment with others as necessary. **However, in the case of plagiarism (from other students or online sources), all parties involved will be severely penalized.**

The deadline for this assignment is **Mon, 19 October at 11am**.

Report

The submission should be accompanied by a report, containing **at least** the following information:

- An outline and brief explanation of your parallel strategy for calculating a valid digest
- Measurements (performance or otherwise), along with noteworthy observation and explanations
- Explain the modifications that you have made to your code and their impact on performance.

There is no minimum or maximum page length for the report. As mentioned earlier, be **comprehensive**, yet **concise**.

Deliverables

Submit your assignment to the folder **Assignment 2** under **LumiNUS Files** by **Mon, 19 October at 11am**. Submit **one ZIP archive per student** with your **student number** (`A0123456Z.zip` - if you worked by yourself, or `A0123456Z_A0173456T.zip` - if you worked with another student) containing:

1. CUDA code, including `hash.cu`
2. **Five (5)** sample input files with their corresponding outputs
3. `README` file, minimally with instructions on how to execute the code. You may substitute the `README` with a `Makefile`.
4. Report in PDF format (`a2_report.pdf`)

Note that for group submissions, only **the most recent submission** (from any group member) will be graded, and both students will receive the same grade.
The assignments weights a total of 10% of your CS3210 grade. Late submissions are accepted with a late penalty of about 5% of your assignment grade per day.