TDT4200 PS4b - CUDA Password Cracker

Håvard Ose Nordstrand, Zawadi Svela
Anne C. Elster

Deadline: Fri October 23rd, 2020 at 22:00 on BlackBoard

Attention!

All work has to be done **individually** without consulting others except via the Forum on BB and contact with the TA. You are also **required to test the code on the class machines (Selbu or Cybele PCs).** If your code runs on your own machine, but not on the class machine during TA testing, you will not get the credit for a working code.

PLEASE TEST CODE ON ONLY ONE GPU ON THE CLASS MACHINES!

1 Introduction

In this assignment you will again parallelize password cracker, this time using CUDA. Background info on the password cracker itself can be found in PS1. One significant difference from the PS1 version is that a provided CUDA implementation of `SHA512Crypt()` will be used instead of the `crypt()` function from the GNU C Library, as CUDA devices can't use host/system libraries. This also means that this version only supports SHA512-based password hashes.

2 Requirements

- Linux (e.g. Ubuntu)
- C++ toolkit. Oon Ubuntu, install using:
  `sudo apt update && sudo apt install -y build-essential`
- NVIDIA CUDA-capable GPU (see `https://developer.nvidia.com/cuda-gpus`)
- NVIDIA CUDA toolkit (see `https://developer.nvidia.com/cuda-downloads`)

3 Provided Files

- This assignment paper.
- A password cracker implemented partially in CUDA C/C++. CUDA C/C++ uses extension **'.cu'** for source files and **'.cuh'** for header files. C++ uses extension **'.cpp'** for source files and **'.hpp'** for header files.
- A library consisting of the C++ reference implementation of SHA512Crypt (ref: https://www.akkadia.org/drepper/SHA-crypt.txt) which has been translated to CUDA C/C++, including reimplementations of certain string functions it uses.
- A set of shadow files containing passwords to be cracked. Their names tells you what kinds of passwords they are. Each line is an entry for a user, where the second field is the encrypted password field. The **'passwords.txt'** file contains the original passwords for each variation, so you can check that your results are correct.
- A set of dictionary files containing symbols (words or characters) to try while cracking. Each line is a different symbol, which may be combined into multi-symbol strings according to the length and separator parameters provided to the application.

4 A Note About C++

You may notice that this assignment uses C++ instead of C when looking at the makefile and file endings. This is due to the fact that modern versions of CUDA C/C++ mainly support only C++. However, as this course generally uses C and as C++ generally supports C code, you shouldn't notice any differences from pure C and CUDA C. While pure C++ and CUDA C++ support many features that pure C and CUDA C does not, we will not be using them.

5 Usage

The password cracker generates passwords to compare against by giving it a dictionary of symbols, a max length of how many symbols passwords may contain and a separator to separate the symbols. To test for random alphanumerical passwords, you would give it a dictionary containing only single numbers and letters ('dict/alnum.txt'), no separator (the default) and a max length equal to the number of characters. To test for space-separated words, you would give it a dictionary of words ('dict/12dicts/2of5core.txt'), space as the separator and the number of words ad the max length. To test for common passwords, simply give it a dictionary of common passwords ('seclists/10-million-password-list-top-10000.txt') and length 1 (the default).

This simple password cracker does not support features like adding numbers at the end (like 'password2') or mutating letters (like 'P422w0Rd'), which is common for real passwords.

Usage:

- Build application (using `mnvcc`): `make build`
- Remove temporary files (including binary): `make clean`
- Run the password cracker:
  `./crack [-q] [-v] -i <shadow-file> -d <dict-file>`
  `[-l <max-length>] [-s <separator>] [-o <result-file>]`
  Options:
  - `-q`: Don't print results to console.
  - `-v`: Print all password probes tried to console.
  - `-i <shadow-file>` (required): The shadow file with password hashes to crack.
  - `-d <dict-file>` (required): The dictionary file to use, consisting of one symbol per line.
  - `-l <max-length>`: The maximum number of symbols to combine to password probes. Defaults to 1.
  - `-s <separator>`: The text string to use to separate symbols when max length is more than one. Defaults to the empty string. Note: Space must be quoted, like `-s " "`.
  - `-o <result-file>`: The file to write results to.

  Examples:

- Crack SHA512, 4 digits (fast):
  `./crack -i data/shadow/sha512-4num -d data/dict/num.txt -l 4`
- Crack SHA512, 2 alphanumeric characters (fast):
  `./crack -i data/shadow/sha512-2alnum -d data/dict/alnum.txt -l 2`
- Crack SHA512, 1 simple word (fast):
  `./crack -i data/shadow/sha512-1word -d data/dict/12dicts/2of5core.txt`
- Crack SHA512, 2 simple words, space-separated (slow):
  `./crack -i data/shadow/sha512-2word -d data/dict/12dicts/2of5core.txt -l2 -s" "`
- Crack SHA512, 1 common password (fast):
  `./crack -i data/shadow/sha512-common`
  `-d data/dict/seclists/10-million-password-list-top-10000.txt`

6 Memory Checking and Profiling

To check for CUDA memory access violations, run the application using `cuda-memcheck`. This will print to the console if it detects any violations.

`cuda-memcheck` example:

`cuda-memcheck --leak-check full ./crack [arguments]`

You can profile the CUDA application using `nvprof`.

`nvprof` example to show which CUDA calls and kernels takes the longest to run: `sudo nvprof ./crack [arguments]`

`nvprof` example to show some specific set of interesting metrics (slow):

```
sudo nvprof --metrics "eligible_warps_per_cycle,achieved_occupancy,sm_efficiency,
alu_fu_utilization,dram_utilization,inst_replay_overhead,gst_transactions_per_request,
l2_utilization,gst_requested_throughput,flop_count_dp,gld_transactions_per_request,
global_cache_replay_overhead,flop_dp_efficiency,gld_efficiency,gld_throughput,
l2_write_throughput,l2_read_throughput,branch_efficiency,local_memory_overhead"
./crack [arguments]}
```

7 Example Output

Command:

```
./crack -i data/shadow/sha512-1word -d data/dict/12dicts/2of5core.txt
```

Output:

```
Chosen CUDA grid size: 16
Chosen CUDA block size: 128
Max symbols: 1
Symbol separator: ""
Shadow file: data/shadow/sha512-1word
Dictionary file: data/dict/12dicts/2of5core.txt
Read 4690 words from dictionary file.
CUDA device count: 1 (max 256 allowed)
CUDA device #0:
        Name: GeForce GTX 1070 Ti
        Compute capability: 6.1
        Multiprocessors: 19
        Warp size: 32
        Global memory: 7.9GiB bytes
        Per-block shared memory: 48.0kiB
        Per-block registers: 65536

Entries:
user="daemon", alg="UNKNOWN" status="SKIP" duration="0.000000s" attempts="0"
attempts_per_second="-nan" password=""
[...]
```

```
user="user1", alg="SHA512" status="SUCCESS" duration="0.186744s" attempts="635"
attempts_per_second="3400.380410" password="chair"
user="user2", alg="SHA512" status="SUCCESS" duration="0.415834s" attempts="2313"
attempts_per_second="5562.309940" password="lamp"
user="user3", alg="SHA512" status="SUCCESS" duration="0.160152s" attempts="1812"
attempts_per_second="11314.233870" password="glass"
user="user4", alg="SHA512" status="SUCCESS" duration="0.162333s" attempts="1660"
attempts_per_second="10225.900176" password="floor"

Overview:
Total duration: 0.925s
Total attempts: 6420
Total attempts per second: 6940.065
Skipped: 18
Successful: 4
Failed: 0
```

8 Task

The goal of this assignment is to parallelize the application using CUDA. The task description has been loosely split into a series of steps to guide you along the way. You will only need to modify **'main.cu'** for this assignment. If you've modified any other files for some reason, please mention it in the delivery.

1. Look through the makefile to see how the CUDA application is built.
2. Look through the source code to get an overview of how it's structured.
3. To keep track of and make it easy to change the number of CUDA blocks in a grid and the number of threads in a block, define two global constants `GRID_SIZE` and `BLOCK_SIZE`. You can define a constant `TOTAL_THREAD_COUNT` as the product of the two former constants as well, to keep track of the total number of threads used.
4. In `init_cuda()`, get the CUDA device count using `cudaGetDeviceCount()` and make sure it's at least one. We will be using CUDA device #0, in case you need to specify it later.
5. In `init_cuda()`, get the CUDA device properties for the CUDA device using `cudaGetDeviceProperties()` so we can print some useful info about the device.
6. In `init_cuda()`, look at how errors are detected and printed for this application. We could alternatively check the return status of every single CUDA call with the help of C macros, but we won't bother with that. You can copy-paste this code into later parts of the program where it seems reasonable to check for errors.

7. In `crack`, allocate one pair of arrays for holding jobs and results in host memory (using `malloc()`), and one pair in device memory (using `cudaMalloc()`). The four arrays will be used by the individual threads, which tells you how large the arrays should be. Make sure to free the arrays at the end of the function (using `free` and `cudaFree()`).

8. Inside the `crack()` loop, fill the host job array with new jobs (fill in the missing details for the `prepare_job()` loop).

9. Inside the `crack` loop, copy the job array from the host to the device (using `cudaMemcpy()`).

10. Turn the `crack_job()` function into a CUDA kernel (using `__global__`).

11. Inside the `crack` loop, call the newly created kernel with the correct grid and block sizes (using the `<<<>>>` syntax), passing the pointers to the device arrays as arguments.

12. Inside the `crack()` loop, copy the result array from the device to the host.

13. Inside the `crack()` loop, fill in the missing variables for handling the results in the for loop.

14. In `crack_job()`, point the `job` and `result` pointers to the correct element of the job and result arrays. Use the `blockIdx` and `threadIdx` tuples in order to find a unique index into the array for each thread. (Hint: `blockIdx.x * BLOCK_SIZE + threadIdx.x`)

15. Build and run it with any of the SHA512 shadow files.

16. To help you locate CUDA memory access violations, run the application with `cuda-memcheck` as described above.

17. Make sure it returns the correct results (as can be seen in 'shadow/passwords.txt').

18. To get some insight into how it performs, profile it using `nvprof` as described above.

9 Deliverables

A single compressed folder (ZIP-file) containing all the code. The code must be runnable using CUDA and the hashing part must happen in device code using the provided translated SHA512Crypt implementation. It does not need to include the shadow files, dictionary files or any libraries. Make sure to run `make clean` to remove the binary, object files and other temporary files before zipping the folder. The solution must not make use of any new libraries.