

HPC_Joshua

by Joshua Burge

FILE	3742_JOSHUA_BURGE_HPC_JOSHUA_107698_1169299186.DOCX (205.52K)		
TIME SUBMITTED	16-MAY-2019 11:53PM (UTC+0100)	WORD COUNT	1319
SUBMISSION ID	107437252	CHARACTER COUNT	6299

Introduction:

Cryptography is a vast field spanning the encryption and decryption of either data traffic, such as internet traffic when exchanging financial details between one user and another as is typical when purchasing items from an online store or transferring funds between an online bank account to another, or the data itself which normally falls under trying to keep the data safe and secure from people who should not have access to it, such as medical information pertaining to patients that can only legally be seen by people whom the patient consents to having that information outside of the medical practitioners that require it to perform treatments for that patient.

The practice of encoding information in ways that require a decipher tool or key to understand what's being presented can be found all through-out history. In more modern times, computers typically handle the encrypting of data by applying methods such as SHA-256 and md5, those being the more commonly seen methods for general use outside of military and private security sectors.

One form of security measure that has become commonplace in recent years is the implementation of passwords, a series of characters that are required for authentication reasons to confirm that the person who is trying to access data is the person they say they are, and that they can view the data. Whilst modern practices of password encryption, creation and protection have become more complex they can still be defeated by simple methods.

One such method that exists but cannot be as widely deployed as it once was, is a brute force attack. Put simply, a brute force attack is attempting any and every combination of characters to try and guess the right set required to pass the password validation and gain access to the computer system or data. The most common way this is defeated is by limiting the number of attempts in a given time span or simply just blocking all future attempts for the user whose password is attempting to be guessed until a later time.

However, there currently exists a small niche area where a brute force attack on an encrypted password is still commonly performed which is in bitcoin mining, where a set number of cryptocurrency tokens are locked behind a hash of increasing complexity for every hash solved. The user runs a program that attempts to solve the hash by matching a specific value that acts as a key for the tokens and once the user does so, they gain the currency.

Due to the extensive amount of calculations that must be performed, the usage of graphics cards as the choice of hardware for solving the hashes has become more prevalent over the years compared to central processing units. This is due to a variety of reasons such as graphics cards being purpose built to perform calculations that are mathematical in nature, better parallelisation when performing this calculations and being able to work in tandem with up to 3 additional graphics cards depending on the hardware limitations with regards to the motherboard and power supply.

For the purposes of this project however, I will specifically be looking at how a current high-end central processing unit designed to handle a large amount of data at once handles a brute force attack on a simple password with no encryption. Then, I'll run the same test on a high-end graphics card to see what the difference is, if any, in the time to complete the attack.

The Implementation:

The general outline of what must happen is as follows;

- 1) A password must be set by the user or read from a file.
- 2) The password cannot be encrypted using any method.
- 3) The password is stored into an array of characters.
- 4) The algorithm used must identify the characters in the password in the correct order.
- 5) The algorithm cannot use any hints as to what the password may be.
- 6) The algorithm and password can only use normal keyboard ASCII characters, no special symbols.

To achieve this, the CPU side of the program was built first to act as a test bed to ensure that the logic and methods being used worked before trying to shift the implementation over to CUDA.

The specifications of the Test Machine are as follows;

- 1) The Central Processing Unit is 1x Ryzen Threadripper 2920X overclocked to 4.1Ghz
- 2) The Graphics Card is 1x Nvidia 2080, MSI Aorus variant with no further modification.
- 3) The Random Access Memory installed is 32gb in size, 2400Mhz in speed.
- 4) The storage used is 1x Samsung 970 EVO NVME SSD at 1TB.

The algorithm developed for the Ryzen worked as follows;

- 1) Take a password from the user or from a file.
- 2) It was then stored / read into a character array.
- 3) This array was then filtered down so that there were no null values.
- 4) The program would then begin to randomly choose the ASCII characters in the range 32 to 127 which are 97 of the characters found on a standard keyboard.
- 5) Each random pick was checked against the first character in the password.
- 6) If it was a match, the program moved onto the next character.
- 7) If it wasn't, the randomly chosen character was added to a vector.
- 8) The random selection process is repeated but only unique characters are tested and added to the vector.
- 9) The process continues until the password has been "guessed".

This implementation was decided upon for two reasons. One, by not loading the entire 97 character set in and reading through it in sequence it reduces the initial memory requirements and lowers the computational demand of the program as it's possible that the first character of the password could be guessed on the first character generated. Second, by ensuring that no characters can be generated and checked twice it reduces the time spent waiting for the program to sift through the currently generated characters as it doesn't check multiple times if the password "CAT" has the letter D in it five times.

For the 2080 side of the implementation, a similar method is employed;

- 1) Take the password the same way as the CPU side.
- 2) Set up the memory on the GPU to take the character inputs and two counts.
- 3) Increment one of the counts constantly until the INT value matches that of the character.
- 4) Return the character's INT value and translate it back into a Char.

Unfortunately, attempting to parallelise this provided extremely difficult as the intended method would require that each assigned block, blocks chosen as they have a higher upper limit

than threads, would have to pull a specific value from the password each without overlapping. I.E, each block had to work on a unique part of the password, such as Block 1 working on Character 1, Block 2 on Character 2 and so forth.

However, with the current implementation a patchwork form of testing can occur in so far as timing how long the CPU takes to solve one character in a password of one character length against that of the GPU. The results are below.

The Results and Conclusion:

Function Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time	Module Name
mainCRTStartup	1	99.99%	0.00%	32,510,505.00	470.00	RNT.exe
_set_common_main	1	99.99%	4.56%	32,510,035.00	1,483,190.00	RNT.exe
_set_common_main seh	1	95.42%	0.07%	31,026,712.00	21,441.00	RNT.exe
invoke_main	1	94.43%	0.02%	30,704,269.00	5,311.00	RNT.exe
main	1	94.42%	0.66%	30,698,958.00	215,496.00	RNT.exe
?7?5basic_ostringstream@DU?5char_traits@D@std...	52	43.96%	43.96%	274,891.00	274,891.00	MSVCP140D.dll
std::operator<< <std::char_traits<char>>	53	23.06%	0.40%	141,462.00	2,444.00	RNT.exe
std::operator<< <std::char_traits<char>>	27	11.67%	0.14%	140,527.00	1,702.00	RNT.exe
?putc@?5basic_streambuf@DU?5char_traits@...	27	11.14%	11.14%	134,094.00	134,094.00	MSVCP140D.dll
?putc@?5basic_streambuf@DU?5char_traits@...	53	21.76%	21.76%	133,465.00	133,465.00	MSVCP140D.dll
pre_c_initialization	1	0.38%	0.14%	123,300.00	46,363.00	RNT.exe
?7?5basic_ostringstream@DU?5char_traits@D@std...	27	7.80%	0.05%	93,923.00	594.00	MSVCP140D.dll
std::endl<char, std::char_traits<char>>	27	7.73%	0.03%	93,328.00	359.00	RNT.exe
?put@?5basic_ostringstream@DU?5char_traits@D@...	27	7.45%	7.45%	89,657.00	89,657.00	MSVCP140D.dll
std::vector<int, std::allocator<int>>::vector<int...	1	0.24%	0.02%	78,002.00	5,585.00	RNT.exe
std::_Vector_alloc<std::Vec_base_types<int, st...	1	0.22%	0.07%	72,417.00	22,482.00	RNT.exe
pre_cpp_initialization	1	0.22%	0.22%	70,520.00	70,520.00	RNT.exe
_set_narrow_argv_policy:configure_argv	1	0.18%	0.18%	58,515.00	58,515.00	RNT.exe
_stl__cudaRegisterAll	1	0.16%	0.14%	52,678.00	45,409.00	RNT.exe
std::_Vector_alloc<std::Vec_base_types<int, st...	1	0.15%	0.06%	48,812.00	18,783.00	RNT.exe
_set_initialize_crt	1	0.11%	0.02%	36,335.00	5,025.00	RNT.exe
_isa_available_init	1	0.10%	0.09%	31,310.00	38,750.00	RNT.exe
std::vector<char, std::allocator<char>>::push_b...	1	0.09%	0.01%	28,471.00	2,251.00	RNT.exe
std::vector<int, std::allocator<int>>::_Re...	9	0.72%	0.02%	26,177.00	759.00	RNT.exe
std::vector<int, std::allocator<int>>::_Re...	1	0.06%	0.00%	21,009.00	292.00	RNT.exe
std::vector<char, std::allocator<char>>::_Re...	1	0.06%	0.00%	20,561.00	607.00	RNT.exe
std::vector<int, std::allocator<int>>::_Re...	9	0.53%	0.05%	19,290.00	1,720.00	RNT.exe
std::vector<char, std::allocator<char>>::_Re...	1	0.06%	0.00%	18,453.00	1,488.00	RNT.exe
std::mersenne_twister_engine<int, 32, 624, 39...	1	0.06%	0.06%	18,158.00	18,158.00	RNT.exe
std::vector<int, std::allocator<int>>::push_b...	25	1.34%	0.08%	17,402.00	1,077.00	RNT.exe

Above are the results of the CPU test, ordered by the average elapsed inclusive time.

Function Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time	Module Name
mainCRTStartup	1	100.00%	0.00%	1,009,171,855.00	470.00	RNT.exe
__scrt_common_main	1	100.00%	0.01%	1,009,171,385.00	58,726.00	RNT.exe
__scrt_common_main...	1	99.99%	0.01%	1,009,112,491.00	61,109.00	RNT.exe
invoke_main	1	99.95%	0.00%	1,008,634,576.00	1,428.00	RNT.exe
main	1	99.95%	0.00%	1,008,633,148.00	24,984.00	RNT.exe
pBreak	1	99.93%	99.51%	1,008,450,605.00	1,004,195,010.00	RNT.exe
??6?basic_ostream@...	6	0.28%	0.28%	469,834.00	469,834.00	MSVCP140D.dll
pre_cpp_initialization	1	0.01%	0.01%	148,664.00	148,664.00	RNT.exe
pre_c_initialization	1	0.01%	0.00%	118,323.00	29,834.00	RNT.exe
std::operator<<<std::...	3	0.03%	0.00%	109,163.00	1,412.00	RNT.exe
?putc@?5basic_strea...	3	0.03%	0.03%	102,095.00	102,095.00	MSVCP140D.dll
??6?basic_ostream@...	6	0.06%	0.00%	98,473.00	715.00	MSVCP140D.dll
std::endl<char,std::ch...	6	0.06%	0.00%	97,758.00	4,494.00	RNT.exe
std::operator<<<std::...	3	0.03%	0.00%	95,565.00	753.00	RNT.exe
?putc@?5basic_strea...	3	0.03%	0.03%	92,295.00	92,295.00	MSVCP140D.dll
?put@?5basic_ostream...	6	0.05%	0.05%	88,515.00	88,515.00	MSVCP140D.dll
__sti__cudaRegisterAll	1	0.01%	0.01%	69,422.00	63,306.00	RNT.exe
std::vector<int,std::al...	1	0.01%	0.00%	67,622.00	580.00	RNT.exe
std::Vector_alloc<st...	1	0.01%	0.00%	67,042.00	38,897.00	RNT.exe
__scrt_narrow_argv_p...	1	0.01%	0.01%	54,339.00	54,339.00	RNT.exe
std::vector<char,std::...	1	0.00%	0.00%	43,031.00	1,587.00	RNT.exe
tst	3	0.01%	0.00%	38,459.00	346.00	RNT.exe
_device_stub_Z3tstP...	3	0.01%	0.00%	38,112.00	1,822.00	RNT.exe
std::Vector_alloc<st...	1	0.00%	0.00%	37,681.00	1,482.00	RNT.exe
std::Vector_alloc<st...	1	0.00%	0.00%	35,454.00	2,013.00	RNT.exe
__scrt_initialize_crt	1	0.00%	0.00%	35,378.00	17,135.00	RNT.exe
cudaLaunchCchar>	6	0.02%	0.02%	32,184.00	32,184.00	RNT.exe
pRand	3	0.01%	0.00%	30,575.00	218.00	RNT.exe
_device_stub_Z5pRa...	3	0.01%	0.00%	30,357.00	2,279.00	RNT.exe

And here are the results from the GPU finding a single character.

It's worth noting that it took the CPU 0.009 seconds to complete whilst the GPU took 0.288 seconds. This is almost certainly down to the setup time required by CUDA before the calculations can be done.

From this, it can be inferred that if the parallelisation were correctly implemented for the GPU, and if the password reached a enough length such as above 1000 characters, the GPU would significantly outperform the CPU as that is designed to work in a more serial manner. However, further development and testing would be required to confirm this hypothesis.

FINAL GRADE

GENERAL COMMENTS

Instructor

39/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

GRADING FORM: SMU GENERIC_GJ

JOSHUA BURGE

38.50

RESEARCH-INFORMED LITERATURE



You've clearly researched the area but there is no evidence here of your research (NO REFS INCLUDED!)

3

KNOWLEDGE AND UNDERSTANDING

Review and discussions limited. Several other



approaches could have been identified and reviewed.

7

ANALYSIS



Conclusion appropriate given limitations of GPU results. Other approaches could have been reviewed and then analysed. Algorithms for each implementation developed.

5

PRACTICAL APPLICATION AND DEP.



CPU timings generated -limited but correct. Impact of initialisation overhead identified on GPU. More work on GPU/CUDA is needed here.

21.5

SKILLS FOR PROFESSIONAL PRAC.



NO REFS! Structure in evidence. Generally this document requires further work.

2