Exploring a GPU-Based Brute Force Attack

A look at using massively parallel programming to perform a brute-force attack

---

John Rivera
Prof. Russ Miller

December 5, 2019

CSE 702 – Programming Massively Parallel Systems
SUNY The University at Buffalo

Programming the Nvidia GPU

- Nvidia GPUs use the SIMT (Single Instruction, Multiple Threads) architecture for parallel programming.
- CUDA, a proprietary extension to the C language developed by Nvidia, is the primary programming language for developing parallel applications on the GPU.
- A Nvidia GPU contains a number of cores. There are two kinds of cores: Streaming Multiprocessors (SMs) and CUDA cores.
- SMs are special cores that dispatches threads to the CUDA cores in an efficient manner. Each SM is responsible for a certain number of CUDA cores.

Programming the Nvidia GPU Cont'd

The CCR cluster's GPU Compute nodes feature the Nvidia Tesla V100 GPU, a member of the Volta family of Nvidia GPUs. Some quick facts:

Each Nvidia Tesla V100 GPU has:

- 80 Streaming Multiprocessors
- 64 CUDA cores per Streaming Multiprocessor
- 5,120 (80×64) CUDA cores
- 1,024 threads per block
- CUDA 7.0 platform support

CUDA

- CUDA is a deceptively simple extension to the C programming language.
- There are only two extensions to the base language: a declaration of where the function can be run; the GPU (‘kernel’), the CPU (‘host’) or both (‘global’); and special syntax for calling ‘kernel’ functions specifying the number of blocks and threads to run the function on.
- The most important parts of the CUDA API are functions for transferring the contents of system memory to GPU memory (and back) and a special struct which reveals which block and thread a ‘kernel’ function is running on.

Blocks and threads is an important concept to understand when programming in CUDA. It can be visualized as a grid:

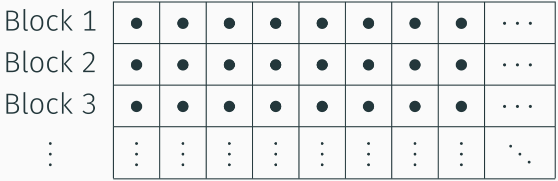


Figure 1: A grid in CUDA consists of blocks and threads.

CUDA Cont'd

The following is a simple CUDA program:

```
#include <stdio.h>

__global__ void hello_world() {
  printf("Hello, World!");
}

int main(void) {
  hello_world<<<1, 1>>>();
}
```

CUDA Cont'd

- The function marked with `__global__` can be executed on either the CPU or the GPU.
- The `<<<x, y>>>` syntax denotes both that the function should be executed on the GPU, and how many threads we want to run the function on; x denotes the number of blocks and y denotes the block size (i.e. the number of threads per block).
- In summary, the program executes the `hello_world()` function on one (1×1) thread on the GPU.

CUDA Cont'd

There are some considerations when programming in CUDA:

- The logic is more or less pure C; the programmer is responsible for thread synchronization, memory allocation, etc.
- Nvidia GPUs use the SIMT architecture; it works best with a single function running on many threads.
- CUDA only allows us to work with threads; it is not possible to ensure a $1:1$ mapping to the cores themselves. A CUDA program can only specify the number of threads to run a function on, and leave it up to the SMs to dispatch the threads to the CUDA cores as they see fit.

Cryptography

A deep dive into cryptography is out of the scope of this presentation. In essence, all we really need to know are:

- For simplicity, we are using symmetrical cryptography – that means we have the same key for both encryption and decryption.
- The ciphertext is the plaintext, encrypted.
- Key strength is generally defined in bits; 32-bit, 128-bit, etc.
- For the project, I'm using the RC4 algorithm. Do NOT use this in production code – vulnerabilities within the algorithm has been discovered a long time ago. RC4 is NOT secure, no matter how strong the key may be.

The Brute-Force Attack

- A brute-force attack is simple: try every possible key until we decrypt the ciphertext.
- This is where the importance of key strength comes in play. Say, we have a 16-bit key; we will need, in the worst case, $\Theta(2^{16})$ tries to crack a key. 128-bit key? $\Theta(2^{128})$ tries.
- Given enough time, ALL encryption algorithms are vulnerable to a brute force attack. All of them. This is why many algorithms add “busy work” to the decryption algorithm.

The Brute-Force Attack Cont'd

To generalize, the worst-case running time for a sequential brute force attack is:

$$\Theta(2^{cn})$$

where c is the time taken in “busy work“ and n is the size of the key in bits. We can see that this is an extremely fast-growing function.

This is essentially what is keeping us secure. The idea is that by the time a sufficiently strong key is cracked, either a) it is no longer relevant, or b) we are all long dead.

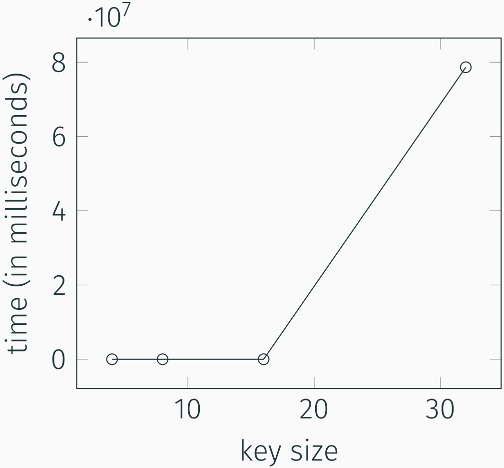
Results

I am using the insecure RC4 algorithm because unlike most algorithms, it allows for an arbitrary key size. This is useful for my experiment, where I can run an attack on a number of different key sizes.

Also, I am running the attack in its entirety – I do not stop when a key is found. This eliminates a degree of randomness in my results, to avoid a situation where the key is found relatively early, skewing the graph.

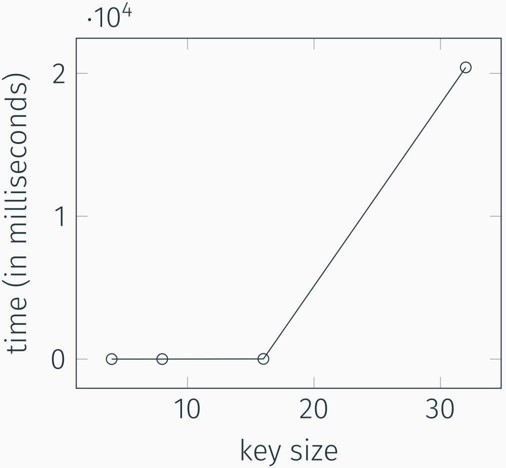
Results (Sequential)

n	time
4	0.3573
8	5.2981
16	1219.2634
32	78640400.8017
64	> 72 hours



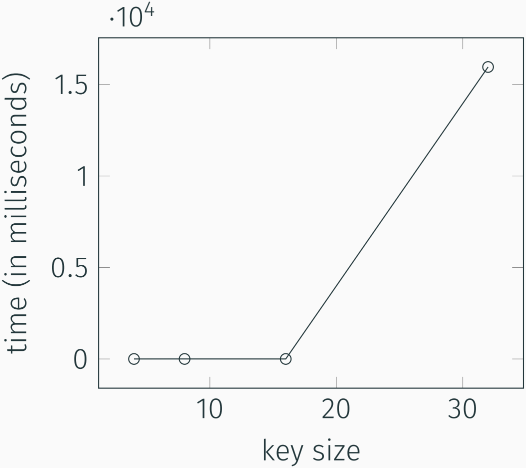
Results (Key Per Thread)

n	time
4	0.1065
8	1.3209
16	12.2685
32	20413.9160
64	> 72 hours



Results (5,120 Threads)

n	time
4	0.1065
8	1.3209
16	0.3522
32	15957.7314
64	> 72 hours



References

- https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
- https://devblogs.nvidia.com/even-easier-introduction-cuda/
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- https://en.wikipedia.org/wiki/RC4
- https://gist.github.com/rverton/a44fc8ca67ab9ec32089