Cracking DES-encrypted passwords using CUDA

Francesca Lanzillotta
E-mail address
francesca.lanzillotta@edu.unifi.it

Abstract

The goal of this project is to explore the advantages of using the CUDA programming model on data-parallel computation-intensive tasks.

1. Introduction

In this project we will simulate a brute force attack of DES-encrypted passwords using both a sequential and a parallel approach. To parallelize our code, we'll use the CUDA library in order to accelerate our computations through the GPU.

1.1. Data Encryption Standard (DES)

The Data Encryption Standard is a symmetric-key encryption algorithm developed in the early 1970s at IBM. DES is a block cipher algorithm that takes a 64-bit plain text and transforms it, using a 56-bit key, into a cipher text bitset of the same length. The same algorithm and key are used for encryption and decryption, with minor differences. The overall structure of the encryption process, shown in figure 1, can be summarized as:

1. Apply the Initial Permutation (IP) to the 64-bit plain text.
2. Split the block in two halves, labeled L (left) and R (right)
3. For round 1 to 16 repeat:
   (a) Apply the Feistel function to R, XOR the result with L
   (b) Swap R and the result of the previous step
4. Fuse back the two halves and apply the Final Permutation (FP) to the 64-bit block.

The Feistel function is the heart of the encryption algorithm. For each round, the function uses a specific 48-bit subkey, each one derived from the original 56-bit key using a well defined key schedule. The input 32-bit half-block is expanded to a 48-bit bitset and XORED with the proper subkey for the current round. The result of this is passed through the so-called *substitution* process: the block is divided into eight 6-bit pieces and processed by eight substitution boxes. Each of the eight S-boxes replaces its input bits with a 4-bit output according to a non-linear transformation, provided in the form of a lookup table. The eight 4-bit results are then combined and passed trough a final permutation P. The function' scheme is displayed in figure 2.

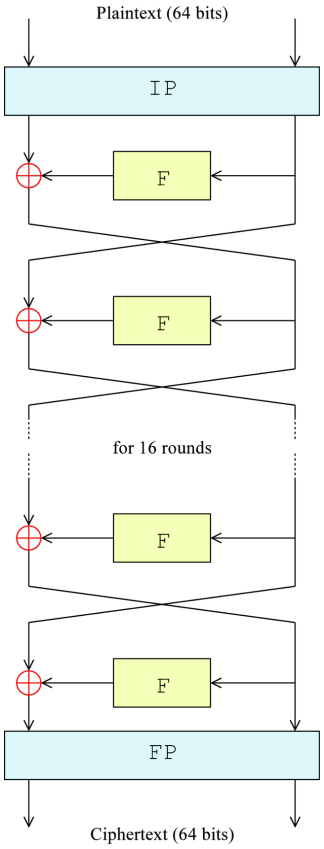

Figure 1. The overall DES structure

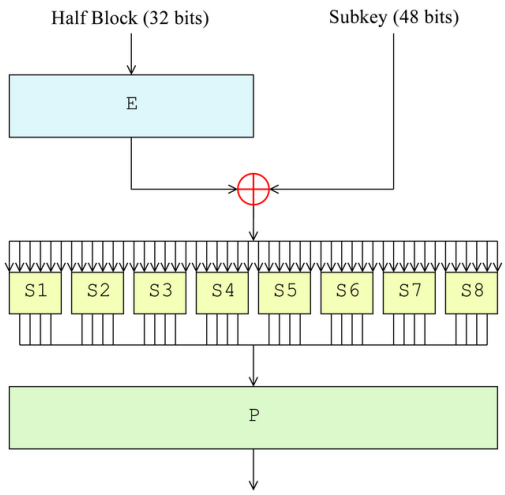

Figure 2. Feistel function structure

1.2. CUDA

The Compute Unified Device Architecture, commonly referred to as CUDA, is a parallel computing platform developed by NVIDIA that lets programmers develop high performance algorithms accelerated by thousands of parallel threads running on GPUs. Using CUDA we can take full advantage of the GPUs' capabilities for data-parallel computation-intensive tasks.
CUDA uses a heterogeneous model in which both the CPU and GPU are used: the host refers to the CPU and its memory, while the device refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches kernels, which are functions executed on the device. These kernels are executed by many GPU threads in parallel.

2. Simulating a brute force attack

The goal of this project is to understand the capabilities offered by the CUDA library by simulating a brute force attack with both a sequential and a parallel approach.
More specifically, the goal is to find a number of DES-encrypted passwords from a list of plain text passwords. The strategy used is:

- Randomly select a subset of passwords from the dictionary and encrypt them using DES.
- For all the encrypted passwords to crack, iterate over all the passwords in the dictionary until we find the element with the same encryption as the one we're looking for.

This kind of computation can be extremely costly using a sequential approach: it is a slowed down linear search problem over a large space. In listing 1 we can see how we collect the plain text passwords from the dictionary and create the vectors with the encrypted passwords to crack.

```
int N = 1000000;     // search space size
int nCrack = 10;     // number of passwords to crack
int nTests = 10;     // number of test repetitions

ifstream wordsFile(wordsPath);

// create a vector with all the words in the specified
     file

string pwd;
int pwdCount = 0;
auto *pwdList = new uint64_t [N];
while (getline(wordsFile, pwd) && pwdCount < N) {
    pwdList[pwdCount] = toUint64_T(pwd);
    pwdCount++;
}

wordsFile.close();

random_device rd;  // a seed source for the random
     number engine
mt19937 gen(rd()); // mersenne_twister_engine seeded
     with rd()
uniform_int_distribution<> distrib(0, N);

vector<uint64_t*> tests;
for(int idTest = 0; idTest < nTests; idTest++){
    auto test = new uint64_t[nCrack];
    for (int i = 0; i < nCrack; i++){
       test[i] = desEncrypt(key, pwdList[distrib(gen)]);
    }
    tests.push_back(test);
}
```
Listing 1. Passwords retrieval

2.1. Sequential approach

The sequential approach is pretty straightforward: for each of the passwords to crack, we iterate over all the plain text passwords in the list, we encrypt the current element and we check if the encryptions match. The implementation is shown in listing 2.

```
for (int i = 0; i < nCrack; i++){

    for (int j = 0; j < N; j++){

        if (pwdToCrack[i] == desEncrypt(key, pwdList[j])
     )
            break;
    }
}
```
Listing 2. Sequential implementation

2.2. Parallel approach

In order to parallelize our simulated attack with CUDA, we need to write both device code, to define the kernel function to execute, and host code, to allocate, initialize and transfer data to device memory, and to launch the kernel.
The main task of the `parallelCrack` function is to allocate memory on the device and copy data from host, as displayed in listing 3: first of all, we need to copy all the permutations' look-up tables from host memory to the device's constant memory. Constant memory is a read-only memory shared across all of the computing units. A big

advantage of of using this type of memory is that, besides being visible and accessible from all threads, its access cost is very low when all threads read the same data, as is the case of the look-up tables necessary for the encryption process. Then, we need to allocate and copy the data to the vectors for the plain text passwords and the encrypted passwords to crack. At last, we execute the kernel function with the appropriate number of blocks, given the block size.

```
#define BLOCK 64
#define HALF_BLOCK 32
#define ROUND_KEY 48
#define ROUNDS 16

__host__
bool * parallelCrack(uint64_t *pwdList, int N, uint64_t *pwdToCrack, int nCrack, uint64_t key, int blockSize){
    // copy all the permutations' look-up tables to constant memory
    cudaMemcpyToSymbol(d_initialPerm, initialPerm, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_finalPerm, finalPerm, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_expansion, expansion, sizeof(int) * ROUND_KEY);
    cudaMemcpyToSymbol(d_hS1, hS1, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_hS2, hS2, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_hS3, hS3, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_hS4, hS4, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_hS5, hS5, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_hS6, hS6, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_hS7, hS7, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_hS8, hS8, sizeof(int) * BLOCK);
    cudaMemcpyToSymbol(d_permutation, permutation, sizeof(int) * HALF_BLOCK);
    cudaMemcpyToSymbol(d_permutedChoice1, permutedChoice1, sizeof(int) * 56);
    cudaMemcpyToSymbol(d_permutedChoice2, permutedChoice2, sizeof(int) * ROUND_KEY);
    cudaMemcpyToSymbol(d_keyShiftArray, keyShiftArray, sizeof(int) * ROUNDS);

    // allocate memory and copy all the data to the device's memory
    uint64_t *d_pwdList, *d_pwdToCrack;

    cudaMalloc((void **)&d_pwdList, N * sizeof(uint64_t));
    cudaMemcpy(d_pwdList, pwdList, N*sizeof(uint64_t), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_pwdToCrack, nCrack * sizeof(uint64_t));
    cudaMemcpy(d_pwdToCrack, pwdToCrack, nCrack * sizeof(uint64_t), cudaMemcpyHostToDevice);

    bool *d_found;
    cudaMalloc((void **) &d_found, nCrack * sizeof(bool));
    cudaMemset(d_found, 0, nCrack * sizeof(bool));

    kernelCrack<<<(N + blockSize - 1) / blockSize, blockSize>>>(d_pwdList, N, d_pwdToCrack, nCrack, d_found, key);

    //copy the result from device to host
    bool *found = new bool[nCrack];
    cudaMemcpy(found, d_found, nCrack * sizeof(bool), cudaMemcpyDeviceToHost);

    // free device memory
    cudaFree(d_pwdList);
    cudaFree(d_pwdToCrack);
    cudaFree(d_found);

    return found;
```

Listing 3. Parallel implementation

In the kernel function, shown in listing 4, the strategy for the simulated attack has to be modified since we're using a CUDA programming model: the idea is to assign to each thread a plain text password, each thread will encrypt its password and then iterate over all the encrypted passwords to crack in order to check if the two encryptions match.

```
__global__
void kernelCrack(const uint64_t *pwdList, int nPwd, const uint64_t *pwdToCrack, int nCrack, bool *found, uint64_t key) {

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < nPwd){
        uint64_t e = d_desEncrypt(key, pwdList[tid]);
        for(int i = 0; i < nCrack; i++){
            if (!found[i] && e == pwdToCrack[i])
                found[i] = true;
        }
    }
}
```

Listing 4. Kernel function implementation

3. Experimental Results

All tests were performed on a Lenovo Legion 5 with an AMD Ryzen 7 5800H CPU and a NVIDIA GeForce RTX 3070 (8 GB) GPU.
Two kinds of tests were performed:

- The first tests aims to compare the completion times of the sequential and the parallel approach, considering increasing block sizes.
- In the second test we want to asses the performances of the two strategies considering an increasing number of passwords to crack.

All the passwords used in this project are contained in a generated dictionary of 1 million elements: each element is a word of 8 characters, selected from [a-zA-Z0-9.!].
Each experiment was repeated ten times in order to validate the data measured. All plots and tables contain the averages of these tests.

3.1. Increasing block size

The results of this test are displayed in figure 3: as expected, the completion time of the parallel version is significantly lower than the sequential one. From first column of table 1 we can also see that the best performance was achieved with a block size of 128 threads.

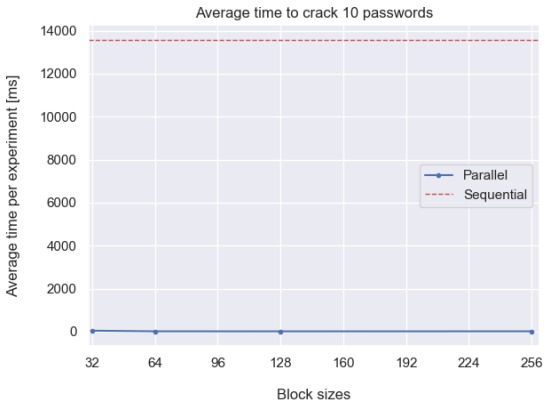

Figure 3. Comparison between the sequential and parallel approach

3.2. Increasing number of passwords

The results of this tests are summarized in table 1 and figure 4. The completion times reported in table 1 show that the parallel approach's performance increases along with the number of passwords to crack. It's even more very evident from figure 4, where we can see that the parallel approach has an almost 1800x speedup, when cracking 1000 passwords.

Block size	Number of passwords		
	10	100	1000
Sequential	13.6 s	148.2 s	14511.1 s
32	43.6 ms	113 ms	856.5 ms
64	14.5 ms	91 ms	844.5 ms
128	13.3 ms	88.6 ms	825.8 ms
256	14.2 ms	94.4 ms	906.3 ms

Table 1. Average times measured over ten experiments, increasing the number of passwords to crack

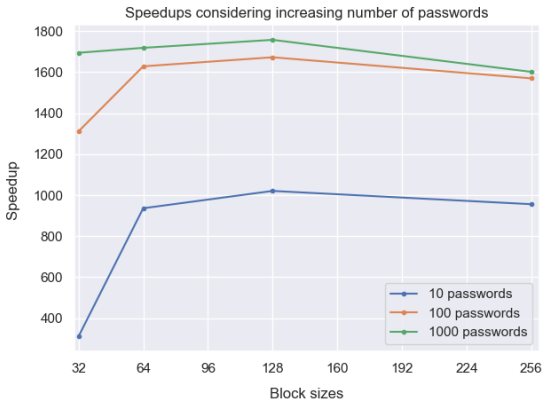

Figure 4. Speedup computed increasing the number of passwords

4. Conclusions

From all the tests conducted, it's quite evident that it's much more efficient to simulate a brute force attack on a list of DES-encrypted passwords using a parallel approach rather than a sequential one. This is due to the fact that a problem like this one lends itself easily to the CUDA programming model, that excels in data-parallel computation-intensive tasks.