INF-3201 Parallel Programming – GPGPU

Enrico Tedeschi
ete011@post.uit.no

I. Introduction

The goal of this assignment is to parallelize the given piece of code with **CUDA/OpenCL** and multiprocessing (combined).

A. Requirements

- Parallelize an encrypt/decrypt code given
- Use multiple cores on the CPU, and GPU, at the same time
- The CPU parallel implementation can use shared memory or message passing models
- Parallelize it with CUDA/OpenCL
- Analyse the solution using the Nvidia Visual Profiler (if using CUDA)

II. Technical Background

– Concurrency and parallelism concepts
– Parallel programming concepts
– Basic programming approach
– Notion of GPU programming approach
– Knowledge of C language
– Knowledge of Python language
– Notion of design pattern principles
– Theory about software engineering
– Knowledge of git to manage the software versions

III. Analysis

A. Precode

To solve the problem a deep analysis of the precode is required. The given sequential code is written in *Python* and it consists of three classes: *precode.py*, *decrypt.py* and *encrypt.py*. The precode class is the one which does all the encryption and decryption work while the other two manage the input file to encrypt/decrypt and which kind of password to give. Since the decryption consists in a brute-force guessing password method, it has a terrible computational time and with a password of three characters it already runs out of memory.

The Fig 1 shows how the encryption works. It takes the message as input and it generates from it a *numpy array*. The array is made of 32 bits, 8 to contain the shuffled message and the other 24 tell about the right location of the shuffled message. To generate the first encrypted output the plaintext is enciphered using the key(which is an md5 of the current password) and the initial Instance Vector. The generated output becomes the new IV in the next iteration.

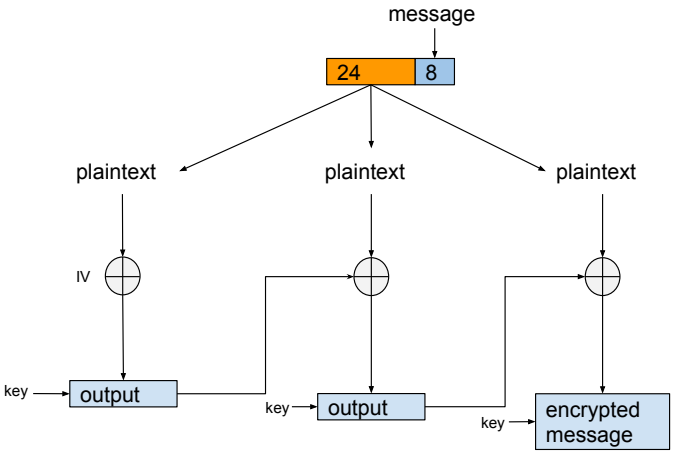

Fig. 1. XTEA and CBC encryption scheme

$$C_i = E_k(P_i \oplus C_{i-1}), C_0 = IV$$

In the formula P is the plaintext and E_k is the encryption method using the key [1].
The decryption works in the same way. The program tries to decrypt the message every time with a different key, and if in the text obtained it occurs the *known_text*, then the current key is the right password.

B. Profiling

The profiling of the precode was done on the *ifilab110* machine and it shows that the most expensive function in matter of time computation is *decipher()*, since the total execution time with a one character password is 15 seconds.

```
1 ncalls  tottime  filename:lineno(function)
2 55000   13.794   precode.py:215(decipher)
```

C. Programming in C CUDA

A good analysis of the C CUDA language is needed before starting the implementation, especially if you're new in GPU's programming. An important thing to consider is how the cuda kernel works with the variables. Every variable created in the CPU which is supposed to be used on the GPU must be allocated first in the GPU's memory, then initialized with a copy *from host to device*; once the GPU is done the variable space must be deallocated and it returns free and the output variables have to be copied *from device to host*. To simplify this process and to get an easy access to the CUDA API, **PyCuda** has been used. With it is possible to avoid the memory allocation part and manage directly the input and output variables. The function in the cuda kernel shouldn't have any return value since each variable is passed using the **HtoD** or **DtoH** copy.
To rewrite the code from *Python* to *C cuda* a deep analysis on the variable type has been done. These are the main types used in Python with the respective value in C cuda:

Python	**C cuda**
long	unsigned long
int	unsigned int
numpy.ndarray	unsigned int *
numpy.uint32	unsigned int

With GPUs a relevant number of threads could be used in parallel. The structure of an nVidia GPU includes a grid where an uni-dimensional, bi-dimensional or three-dimensional block list could be declared. Each block could contain as well a 1D, 2D or 3D thread list, with a maximum number of 1024 threads per block.

D. Possible Optimizations

For every key guess the message is split in 1000 parts. The decipher function is executed for every piece of the message; consider that, the first implementation of the parallel version could be done parallelizing, for each key, the decipher for each part of the message (Fig 2).

The speedup wouldn't probably be that much by parallelizing only on the message length, since the most expensive part in matter of computation time to deal with is the huge amount of possible key combination. Good could be hence, to get the threads work at the same time on different keys (Fig 3).

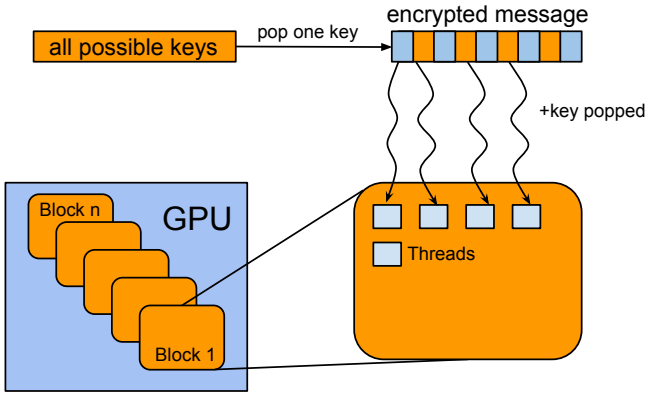

Fig. 2. Possible 'easy' parallel solution

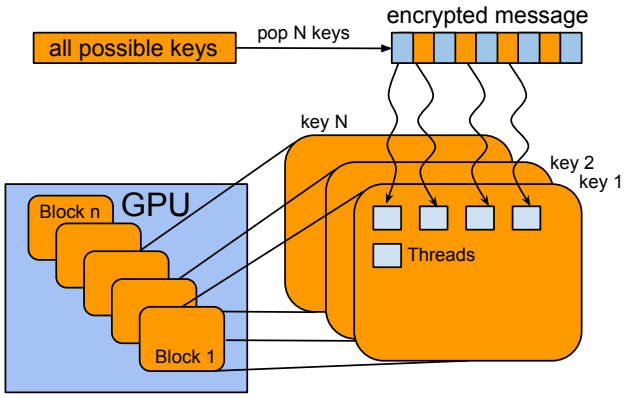

Fig. 3. An harder way of getting a parallel solution

IV. Implementation

A. Environment

The code has been developed using JetBrains CLion 1.1 on Windows 10 and the git control of version has been done on an Ubuntu Virtual Machine. The compilation and the execution of the code has been made on a dedicated machine in *ifilab*. That due to a lack of NVidia graphic unit in my local computer.
To synchronize the ifilab machine and the local computer and to keep trace of all the changes in the code, a git repository was created and the git command on linux were used to commit and to push/pull data from repositories.
The benchmarking and the test were made on the *ifi-lab110* machine. The connection with the remote machine was established by using PuTTY on Windows.

B. Parallelization

V. Result and Benchmarking

A. Speedup

$$S(p) = \frac{t_s}{t_p}$$

where t_s is the sequential time and t_p is the parallel time using p processors.

B. Efficiency

$$S(p) = \frac{t_s}{t_p * p}$$

where p is the number of processes used. So the question which efficiency answers is: *"How good is the parallelization using p number of cores?"*.

C. Scaling

VI. Discussion

VII. Conclusion

References

[1] Wikipedia - cbc. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_Block_Chaining_.28CBC.29, 2015.