UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing
DES Decryption

Academic Year 2019-2020

Paolo Le Piane

Manuel Natale Sapia

Abstract: *L'obiettivo di questo progetto è l'implementazione di un framework per effettuare un attacco a dizionario al fine di determinare una password target di otto caratteri presente in una lista di password tutte cifrate con l'algoritmo crittografico DES. Per lo scopo sono state realizzate tre versioni del framework, di cui una sequenziale e due parallele, utilizzando rispettivamente il linguaggio C , i thread di C e CUDA. Infine è stato effettuato un confronto tra queste tre varianti in termini di tempistiche di esecuzione e di speed up.*

1. Introduzione

Un attacco a dizionario è una tecnica di attacco alla sicurezza di un sistema informatico mirata a "rompere" un meccanismo di autenticazione, provando a decifrare un codice cifrato tra un gran numero di possibilità. In pratica si tenta di accedere a dati protetti da password tramite una serie continuativa e sistematica di tentativi di inserimento della password basandosi su uno o più dizionari di riferimento che contengono le varie password da provare.[1]

In questa implementazione, il dizionario è rappresentato da un file di testo composto da una serie di password di 8 caratteri alfanumerici, uno per ogni riga del file.

Queste password vengono criptate utilizzando l'algoritmo di cifratura DES [2] (Data Encryption Standard) a chiave simmetrica con chiave a 64 bit.

Nella prima parte del nostro lavoro è stato implementato il framework in tre versioni: una versione sequenziale utilizzando il linguaggio C, una versione parallela utilizzando sempre il linguaggio C e la libreria Pthread[3] e un'ultima versione utilizzando CUDA[4] sfruttando la GPU per migliorare le tempistiche di esecuzione.

La seconda parte del nostro elaborato invece si è concentrata sulla valutazione e comparazione delle prestazioni dei vari framework andando a confrontare i tempi di esecuzione e il relativo speed up, variando il numero di thread utilizzati e la posizione della password da cercare, a parità di dimensione del dizionario.

A causa dell'incompatibilità della libreria *crypt[5]* con l'ambiente Windows, le varie versioni del framework sono state testate in ambiente Linux, in particolare utilizzando il S.O. Ubuntu.

Le prime due versioni sono state testate su una macchina virtuale con queste specifiche:

- Processore Intel Core i7-6700HQ da 2.60GHz con turbo boost fino a 3.5GHz
- GPU NVIDIA GeForce GTX 970M con 1280 core e 3GB di memoria dedicata
- 16GB di memoria RAM (di cui 8GB usabili sulla macchina virtuale)

La versione in CUDA invece è stata testata su una macchina remota con queste specifiche:

- Processore Intel Core i7-860 da 2.80GHz com turbo boost fino a 3.5GHz
- GPU NVIDIA GeForce GTX 980 con 2048 core e 4GB di memoria dedicata
- 15GB di memoria RAM

2. Implementazione

2.1. Versione sequenziale: C

La versione sequenziale è stata sviluppata utilizzando il linguaggio C ed è composta da un unico metodo *findPassword()* incaricato della lettura del dizionario e della ricerca della password nel dizionario.

Nello specifico, dopo aver scelto la password da ricercare e settato un salt fisso, il metodo procede a cifrare la password con l'algoritmo di cifratura DES utilizzando il metodo *crypt()* della libreria *crypt* a cui vengono passati come parametro la password e il salt.

Ottenuta la password target cifrata, il metodo *findPassword()* procede a leggere il file di testo contenente la lista delle password da cercare riga per riga. Da ogni riga viene rimosso il ritorno a capo ("\n") in modo tale da avere solo i caratteri che compongono la password, quindi questa viene cifrata sempre utilizzando il metodo *crypt()* con lo stesso salt utilizzato per cifrate la password target.

Infine il metodo *findPassword()* va a confrontare la password appena cifrata con la password target cifrata e in caso di riscontro positivo stampa una serie di messaggi informativi e il tempo di esecuzione, andando in fine a libera la memoria. In caso di riscontro negativo invece continua la sua esecuzione andando ad effettuare la stessa procedura per tutte le password del dizionario e se questa non viene trovata quando l'ultima password viene processata, ritornerà un messaggio di riscontro negativo e libererà la memoria.

```
to_find = strdup(crypt(pass, salt));
//apertura file
while(fgets(file, 10, fp) != NULL){
      word = strtok(file, "\n");
      encrypt = strdup(crypt(word, salt));
      if (strcmp(to_find, encrypt) == 0) {
       //stampa esito positivo e tempo di esecuzione
       //libera memoria
}
//libera memoria e ritorna esito negativo
```

1.Esempio funzionamento framework sequenziale

2.2. Versione parallela: C con Pthread

La prima versione parallela del framework è stata implementata sempre nel linguaggio C ma utilizzando la libreria *Pthread* per la gestione dei thread.

L'idea di base di questa versione è quella di memorizzare il dizionario in una certa struttura dati e di partizionarlo in un numero di blocchi pari al numero di thread selezionati per l'esecuzione, in modo tale che ogni thread lavori sulla propria porzione del dizionario.

A differenza di quanto fatto nella versione sequenziale, in questo caso la lettura del dizionario non è vincolata al metodo che effettua i confronti, infatti è presente un metodo *myReadFile()* che si occupa di leggere ogni riga del file e memorizzare le varie password in un array di stringhe, la cui dimensione viene incrementata e allocata dinamicamente ad ogni nuovo inserimento.

Il metodo incaricato della gestione dei thread è *findPasswordPar()* nel quale si calcola la dimensione dei blocchi, si fanno partire i vari thread in un metodo *compute()* che riceve come parametro l'indice di quel thread e infine si raccolgono i risultati. Questo indice sarà fondamentale nella gestione del blocco di password di cui ogni thread è responsabile. Le variabili su cui lavorano tutti i thread sono definite come variabili

```
//apertura file
stringList = (char**)malloc(sizeof(char*));
    int index = 0;
    while(fgets(file, 10, fp) != NULL){
        word = strtok(file, "\n");
        size++;
        stringList = realloc(stringList, size * sizeof(char*));
        stringList[size - 1] = (char *) malloc(10 * sizeof(char));
        strcpy(stringList[index], word);
        index++;
    }
```

2. Estratto del metodo myReadFile() con allocazione dinamica dell'array di stringhe

globali, quindi accessibili ogni thread. In particolare, in questo metodo la variabile *final_result* viene inizializzata con una stringa contenente un messaggio "password non trovata" che, se non verrà modificata da alcun thread, vorrà dire che la password non sarà stata trovata.

```
block_size = (size / NUM_THREADS) + 1;
final_result = (char*)malloc(40*sizeof(char));
strcpy(final_result,"ERROR: PASSWORD NOT FOUND");
…
for(long in = 0; in < NUM_THREADS; in++) {
        pthread_create(&thread_list[in], NULL, compute, (void*)in);
}
for(long thread = 0; thread < NUM_THREADS; thread++) {
       pthread_join(thread_list[thread], (void*)&result);
    }
…
if(strcmp(result,"ERROR: PASSWORD NOT FOUND") != 0)
        printf("Time elapsed: %.2f ms\n", time_elapsed * 1000);
```

3. Estratto del metodo findPasswordPar()

La ricerca vera e propria della password viene effettuata all'interno del metodo *compute()* nel quale ogni thread effettua i vari confronti solo nel proprio blocco di password. Questo è stato implementato con un ciclo nel quale l'indice i parte da 0 sulla dimensione del blocco, e ad ogni iterazione si ha che l'indice della password da cercare è dato dall'indice i, a cui viene sommato il prodotto tra la dimensione del blocco e l'indice del thread. Il confronto è stato implementato similmente a quanto fatto per la versione sequenziale ma in questo caso una variabile binaria *pass_found* è responsabile di fermare la ricerca da parte degli altri thread quando la password viene individuata.

Si noti che in questa versione la cifratura della password target e di quelle del dizionario viene effettuata con il metodo *crypt_r()* che è la versione rientrante del metodo *crypt()* e che permette l'accesso concorrente all'area di memoria del dizionario. Questo metodo per funzionare correttamente ha anche bisogno di una struct di tipo *crypt_data,* la cui variabile *inizialized* viene settata a 0 da ogni thread per assicurare che il metodo lavori su dati consistenti e sia possibile effettuare iterazioni multiple.

```
struct crypt_data data;
data.initialized = 0;
to_find = strdup(crypt_r(pass, salt,&data));
for(int i = 0; i < block_size; i++){
       index = (block_size * th_index) + i;
       if(index >= size || pass_found == 1)
             //libera memoria e ritorna final_result
       …
       if (strcmp(to_find, encrypt) == 0) {
              //stampa messaggi informativi e libera memoria
              //aggiorna pass_found e final_result
              //ritorna final_result
```

4. Estratto del metodo compute()

2.3. Versione parallela: CUDA

La seconda versione parallela è stata implementata in CUDA, architettura hardware per l'elaborazione parallela sulle GPU creata da Nvidia. Il linguaggio utilizzato è stato C++.

Elemento principale dell'implementazione è stato l'utilizzo della libreria Thrust per CUDA, che garantisce alte performance nella programmazione parallela, facilità il passaggio da GPU e CPU e viceversa e nasconde i metodi di allocazione della memoria in CUDA. In un *thrust host_vector* è stata memorizzata la lista di password da analizzare, partendo da dimensione uno per poi aumentare la dimensione tramite la funzione *resize()* fornita da Thrust. Il salt è rappresentato da una stringa di due caratteri "F4" ed è stato scelto fisso.

Per effettuare l'encryption delle password è stata utilizzata una libreria per la cifratura DES chiamata des-cuda, dove è presente una funzione *full_des_encode_block()* che prende in ingresso la password da cifrare e il salt, entrambi di tipo intero a 64 bit, per poi fornire in output la password cifrata sempre *uint64_t*. Per convertire password e salt da stringhe a *uint64_t* è stata adoperata una funzione *str2uint64()*.

```
int index = 0; //index used to resize the host list
	ifstream inputfile("password_dictionaryNew.txt");
	if (inputfile.is_open()) {
		string password;
		while (getline(inputfile, password)) {
			host_list[index] =
str2uint64(password.c_str());
			index++;
			host_list.resize(index+1);
		}
```

5. Lettura file di testo con resize della host_list

Per effettuare le analisi di questa versione parallela, la dimensione dei blocchi è stata variata da 2 a 256 thread, mentre la grid è stata adattata in base alla dimensione della lista di password.

La lista di password allocata in un Thrust *host_vector* è stata copiata in un Thrust *device_vector* ma, per poter passare quest'ultimo al CUDA kernel è stato necessario trasformare il Thrust *device_vector* in una struct formata da un puntatore al *device_vector* e un indice di dimensione. Questo è stato implementato nella funzione *convertToKernel()*.

```
template <typename T>

KernelArray<T> convertToKernel(thrust::device_vector<T>&
dVec) {

	KernelArray<T> kArray;

	kArray._array = thrust::raw_pointer_cast(&dVec[0]);

	kArray._size = (int) dVec.size();

	return kArray;

}
```

6. Funzione per convertire il device_vector in una struct KernelArray

Al Kernel, al momento della sua invocazione, vengono passati il Thrust *device_vector* in forma di struct, il salt per cifrare le password della lista, la password obiettivo, una variabile *foundD* che indica se la password è stata trovata e *resultD* che conterrà il risultato trovato.

Nel Kernel viene calcolato un valore *stride*, blockDim.x * gridDim.x, che rappresenta il numero totale di thread lanciati all'invocazione del Kernel. Si osserva che tutti i thread processano i primi blockDim.x * gridDim.x elementi alla volta, fino al completamento dell'analisi di tutta la lista di password in ingresso.

In caso di password trovata, vengono aggiornati i valori *foundD* e *resultD*.

```
__global__ void decrypt_kernel(KernelArray<uint64_t> device_list, uint64_t u_salt,
uint64_t crypt_pass, int *foundD, uint64_t *resultD) {
	uint64_t crypt;
	int i = threadIdx.x + blockIdx.x * blockDim.x;
	int stride = blockDim.x * gridDim.x;
	while (i < device_list._size) {
		crypt = full_des_encode_block(device_list._array[i], u_salt);
		if (crypt_pass == crypt) {
			*foundD = 1;
			*resultD = crypt;
			return;
		}
		i += stride;
	}
}
```

7. CUDA Kernel

In fine i valori utilizzati e aggiornati nel Kernel vengono salvati nella memoria dell' Host tramite la funzione *cudaMemcpy()* e la lista di password trasferita dalla device memory alla host memory grazie alla funzione Thrust *copy()*.

Il tempo di esecuzione della decryption viene calcolato tramite la funzione *clock()* della libreria time.

3. Dataset

Come dataset è stata utilizzata una lista di password di otto caratteri, formata da lettere maiuscole e minuscole e numeri. Tutte le password sono state elaborate e trasformate in una lunghezza di otto caratteri grazie ad uno script in Python che, prendendo in ingresso una lista di password già esistente, ha eliminato le password più lunghe di otto caratteri e aggiunto lettere e numeri casuali a quelle con un numero di caratteri inferiori alla soglia. Il file di password finale è risultato di quasi 9 Mb.

4. Esperimenti e risultati

La valutazione delle prestazioni e quindi dei tempi di esecuzione di ogni versione del framework è stata effettuata mediante tre esperimenti. Nei primi due sono state scelte tre password in posizioni specifiche e fatto variare il numero di thread mentre nel terzo è stato fissato il numero di thread e scelto le password in modo uniforme all'interno del dizionario. Per rendere più accurata la valutazione, ogni fase sperimentale è stata eseguita tre volte e fatta la media dei tempi rilevati.

I grafici in seguito presentati sono stati tutti generati attraverso Excel e il file relativo viene fornito insieme al resto della documentazione.

Il confronto tra le tempistiche delle varie versioni nel primo esperimento è stato valutato mediante la metrica dello *speed up (S)* la quale è definita come il rapporto del tempo di esecuzione del sequenziale diviso per il tempo di esecuzione del parallelo. Nel secondo esperimento invece la valutazione è stata effettuata solo in base ai tempi di esecuzione.

4.1. Valutazione prestazioni in C con Pthread

In questo primo esperimento sono state scelte tre password ("Manuel96", "Paololep", "carlo666") posizionate rispettivamente all'inizio, al centro e alla fine del dizionario e fatto variare il numero di thread, in particolare usando 2, 4, 8, 16, 32, 64, 128, 256 thread. È stato poi calcolato lo speed up di ognuno di questi risultati rispetto ai tempi di esecuzione del programma sequenziale.

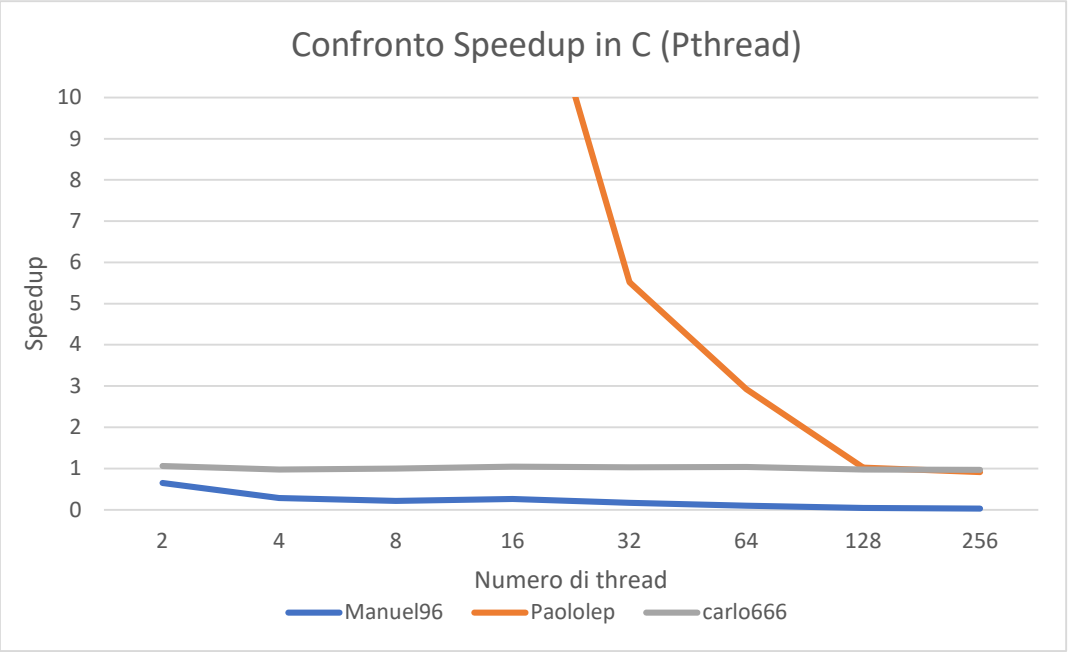

Figura 1 Confronto Speed up in C (Pthread)

Come si può notare dal grafico, non ci sono particolari miglioramenti nelle prestazioni quando il framework deve scorrere tutto il dizionario, c'è un degradamento delle prestazioni quando la password si trova in testa al dizionario, mentre per password posizionate al centro lo speed up è decisamente alto. Il valore dello speed up infatti con 2 thread raggiunge addirittura circa 170 con una differenza considerevole tra il tempo di esecuzione del sequenziale (2029 ms) e del parallelo (12 ms), con 4 thread è pari a 96 e con 8 thread è pari a 34. Lo speed up continua ad essere inversamente proporzionale al numero di thread fino ad assestarsi quando questi diventano maggiori o uguali a 128. Per questo motivo il grafico è stato troncato ad un valore pari a 10 sull'asse dello speed up in quanto altrimenti non si sarebbe potuto averne una visione globale.

Questi alti valori di speed up però sono totalmente dipendenti dalla posizione della password all'interno del dizionario e, in particolare, dalla posizione che essa assume all'interno del blocco di dati di cui ogni thread è responsabile.

Un'ulteriore valutazione non documentata è stata effettuata nella fase iniziale del progetto, nel quale erano state sviluppate due versioni del framework. La prima utilizzando Pthread e la seconda utilizzando OpenMP ma basandosi entrambi sull'idea della creazione di blocchi di dati su cui ogni thread andava a lavorare, gestita autonomamente in OpenMP. Dunque, in quella valutazione preliminare OpenMP aveva mostrato prestazioni nettamente migliori di Pthread a parità di numero di thread e dimensioni del dizionario. Ciò fa pensare che OpenMP oltre ad essere un modo semplice per parallelizzare un codice è anche molto più efficiente rispetto a Pthread, il quale probabilmente presenta dei thread overhead più elevati.

4.2. Valutazione prestazioni in CUDA

Anche in questo secondo esperimento sono state scelte tre password (“Manuel96”, “Paololep”, “carlo666”) posizionate rispettivamente all’inizio, al centro e alla fine del dizionario ed è stato calcolato lo speed up al variare del numero di thread per blocco (block_size). In particolare, sono stati usati 2, 4, 8, 16, 32, 64, 128, 256 thread per blocco mentre il numero di blocchi per ogni grid è dato da 1 sommato al rapporto fra la dimensione del dizionario e la block_size È stato poi calcolato lo speed up di ognuno di questi risultati rispetto ai tempi di esecuzione del programma sequenziale.

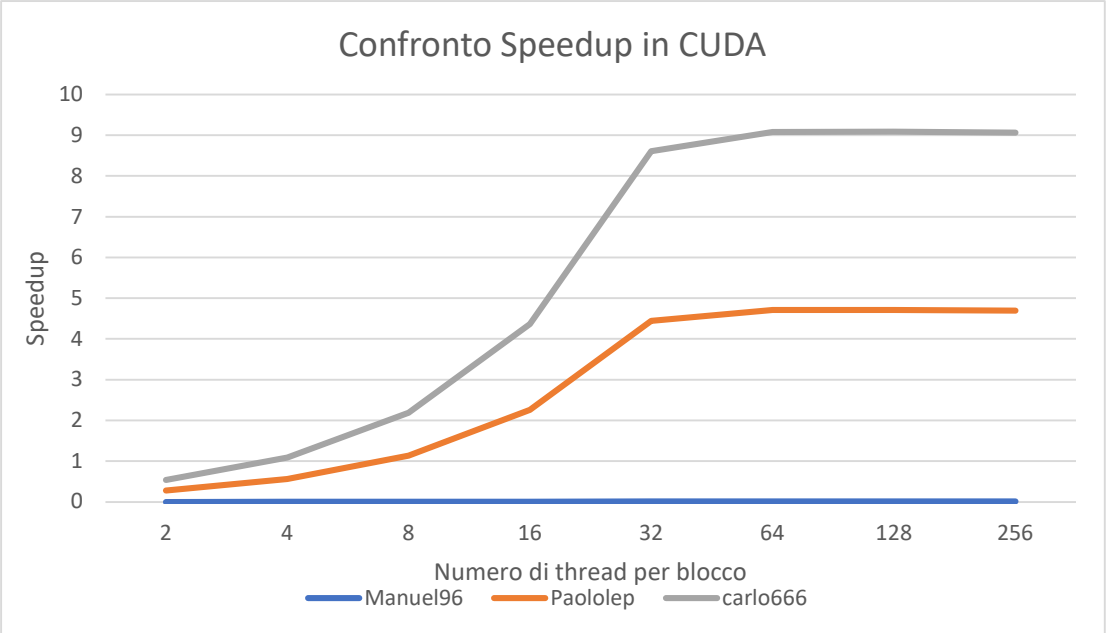

Figura 2 Confronto Speed up in CUDA

Da questo grafico si possono fare alcune considerazioni, la prima in assoluto è rispetto alla password in testa e si nota come le prestazioni siano degradate rispetto alla versione sequenziale avendo dei valori di speed up molto minori di 1.

Per la password al centro invece si può notare che lo speed up parte da un valore minore di uno ma, quando la block size supera il valore 8, si ha un miglioramento delle prestazioni che si va a stabilizzare quando questa supera il valore 32 con un tempo di esecuzione di 430 ms, cioè circa 4 volte e mezzo più veloce della versione sequenziale.

Nella ricerca della password in coda al dizionario invece il miglioramento si vede prima, infatti il valore di speed up supera l’1 quando la block size diventa pari a 4 e continua a crescere fino a stabilizzarsi quando questa supera il valore 32 con un tempo di esecuzione di 430ms, cioè circa 9 volte più veloce della versione sequenziale.

4.3. Confronto tempistiche di esecuzione a parità di parametri

In quest'ultimo esperimento sono state scelte 10 password distribuite uniformemente all'interno del dizionario fissando il numero di thread in C a 16 e la

block size in CUDA a 128. Questo in modo tale da avere un'ulteriore idea sui tempi di esecuzione, data la dipendenza dei risultati dalla posizione della password nel

dizionario come visto in Pthread. Le password scelte sono state ("freese93", "feelin74", "RaZeRis6", "pan20139", "cork1084", "38993414", "ybrf1721",

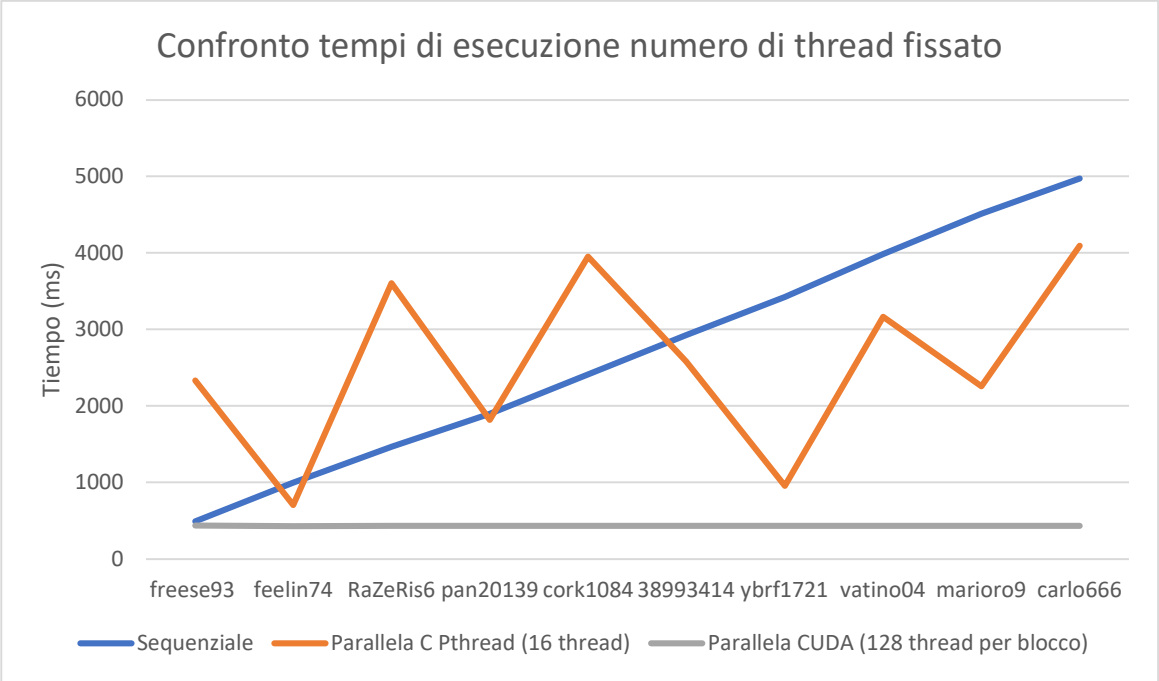

Figura 3 Confronto tempi di esecuzione numero di thread fissato

"vatino04", "marioro9", "carlo666").

Si può notare ancora una volta la natura aleatoria dei risultati generati dalla soluzione parallela con Pthread ma si vede anche che dopo una certa soglia tutte le versioni risultano migliorative rispetto a quella sequenziale il cui tempo di esecuzione, ovviamente, aumenta linearmente allo scorrere del dizionario.

La versione con CUDA offre invece le tempistiche migliori che rimangono inoltre costanti indipendentemente dalla posizione che assume nel dizionario la password desiderata, con dei valori circa pari a 430ms.

5. Conclusioni

Dopo aver confrontato le varie versioni, la soluzione migliore per un attacco a dizionario risulta quella parallela, ed in particolare utilizzando CUDA si riescono a raggiungere sul lungo termine risultati impossibili da raggiungere con altri approcci paralleli basati su CPU.

Inoltre, in questo studio la soluzione basata su Pthread non ha prodotto grandi migliorie sul lungo temine rispetto a quelle che avrebbe potuto produrre l'utilizzo di OpenMP. Infatti, Pthread offre prestazioni migliori della versione sequenziale solo in caso di posizione "fortunata" nel dizionario.

In fine in CUDA le prestazioni sono risultate eccellenti sia per quanto riguarda i tempi di esecuzione che lo speed up, improbabili da raggiungere su CPU.

Bibliografia

[1] https://it.wikipedia.org/wiki/Attacco_a_dizionario

[2] https://it.wikipedia.org/wiki/Data_Encryption_Standard

[3] https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html

[4] https://it.wikipedia.org/wiki/CUDA

[5] https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_node/libc_650.html