DES Decryption

Paolo Le Piane
Manuel Natale Sapia

Obiettivo

- Implementazione di un framework per effettuare un attacco a dizionario per determinare una password target in una lista di password cifrate con DES

Svolgimento

- Tre versioni di implementazione:
  - versione sequenziale in C
  - versione parallela in C
  - versione parallela in CUDA
- Risultati e analisi:
  - analisi speed up
  - analisi tempi di esecuzione

Dataset:

- Lista di password di otto caratteri, formate da lettere e numeri. Dimensione file di 9 MB

Specifiche PC utilizzato:

Per l'incompatibilità della libreria *crypt* con l'ambiente Windows, le versioni del framework sono state implementate in ambiente Linux con S.O. Ubuntu.

- **Macchina virtuale:**
  - 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 (8GB sulla macchina virtuale)

- **Macchina remota (CUDA):**
  - 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

Versione Sequenziale in C

- Unico metodo *findPassword() per lettura file e ricerca password.*
- Cifratura DES con metodo *crypt()* e salt fissato.
- Lettura da file riga per riga, una password per ogni riga.

```
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
```

Versione Parallela in C con Pthread

- Dizionario memorizzato con allocazione dinamica della memoria attraverso il metodo *myReadFile()*
- Ogni thread lavora sulla sua porzione di dizionario.
- Metodo incaricato alla gestione dei thread è *findPasswordPar(),* il quale li avvia e attende la loro terminazione (*join).*

```
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);
```

- Il metodo *compute()* è responsabile della ricerca della password.
- La variabile *th_index* permette ad ogni thread di lavorare nel proprio chunk di password.
- La variabile *pass_found* ferma la ricerca da parte degli altri thread.
- Cifratura password usando *crypt_r()* e una struct di tipo *crypt_data*

```
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
```

Versione Parallela in CUDA

- Linguaggio C++
- Utilizzo della libreria Thrust in CUDA (thrust host_vector e thrust device_vector)
- Utilizzo della funzione *full_des_encode_block()* per effettuare la des encryption della password
- Utilizzo della funzione *str2uint64()* per convertire un input da string a uint64_t

- Nel Kernel tutti i thread processano i primi blockDim.x * gridDim.x elementi alla volta, fino al completamento dell'analisi della lista di password.

```
__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;
...
```

- Funzione *convertToKernel()* utilizzata per trasformare il device_vector in una struct da poter passare al Kernel.

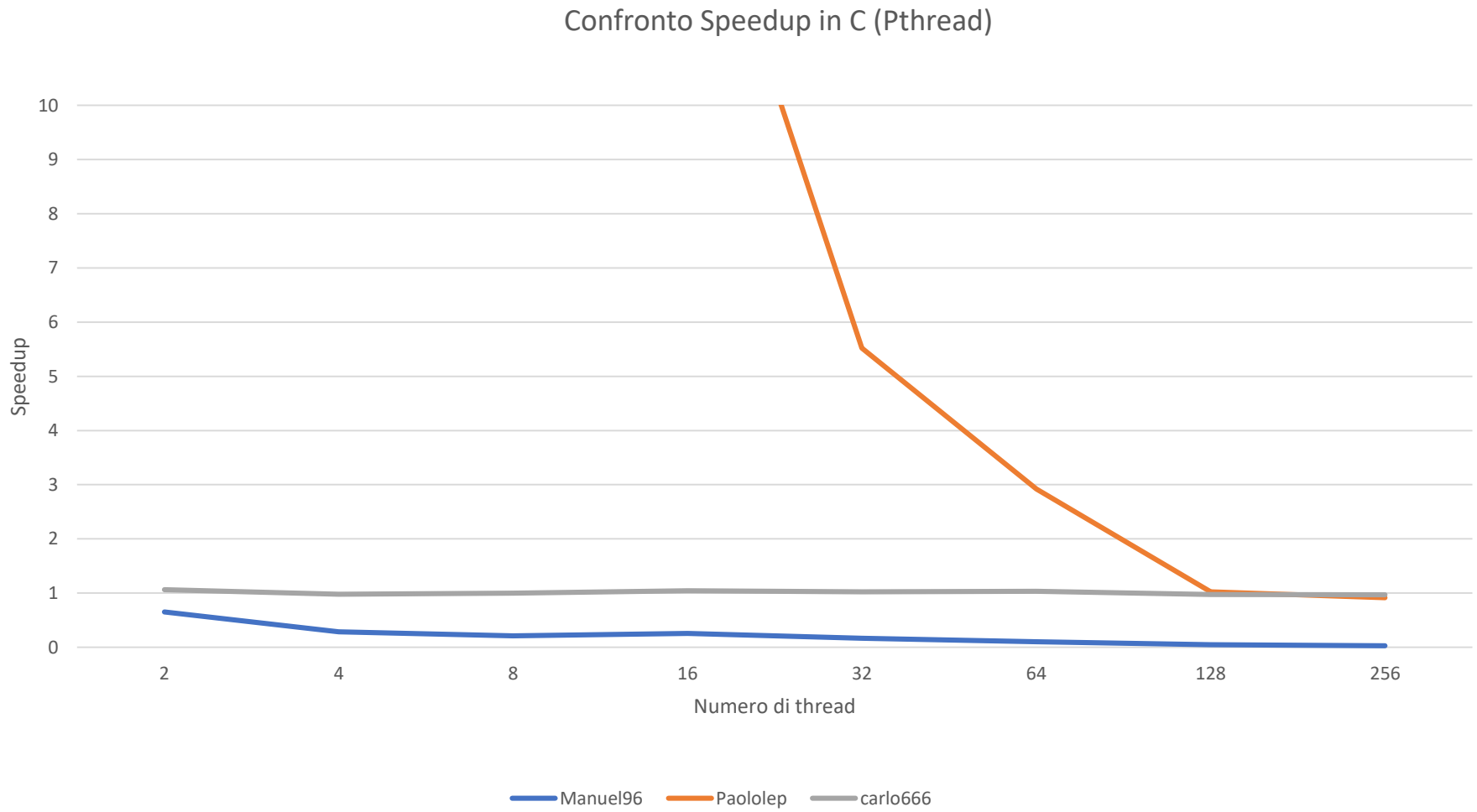
```
clock_t time_start = clock();

decrypt_kernel<<<gridDim,
blockDim>>>(convertToKernel(device_list), u_salt, crypt_pass,
foundD, resultD );

cudaMemcpy(foundH, foundD, sizeof(int),
cudaMemcpyDeviceToHost);

cudaMemcpy(resultH, resultD, sizeof(uint64_t),
cudaMemcpyDeviceToHost);
if (*foundH == 1) {
   clock_t time_end = clock();
   ...
}

//data from device to host
thrust::copy(device_list.begin(), device_list.end(),
host_list.begin());
...
```

Esperimenti e Risultati

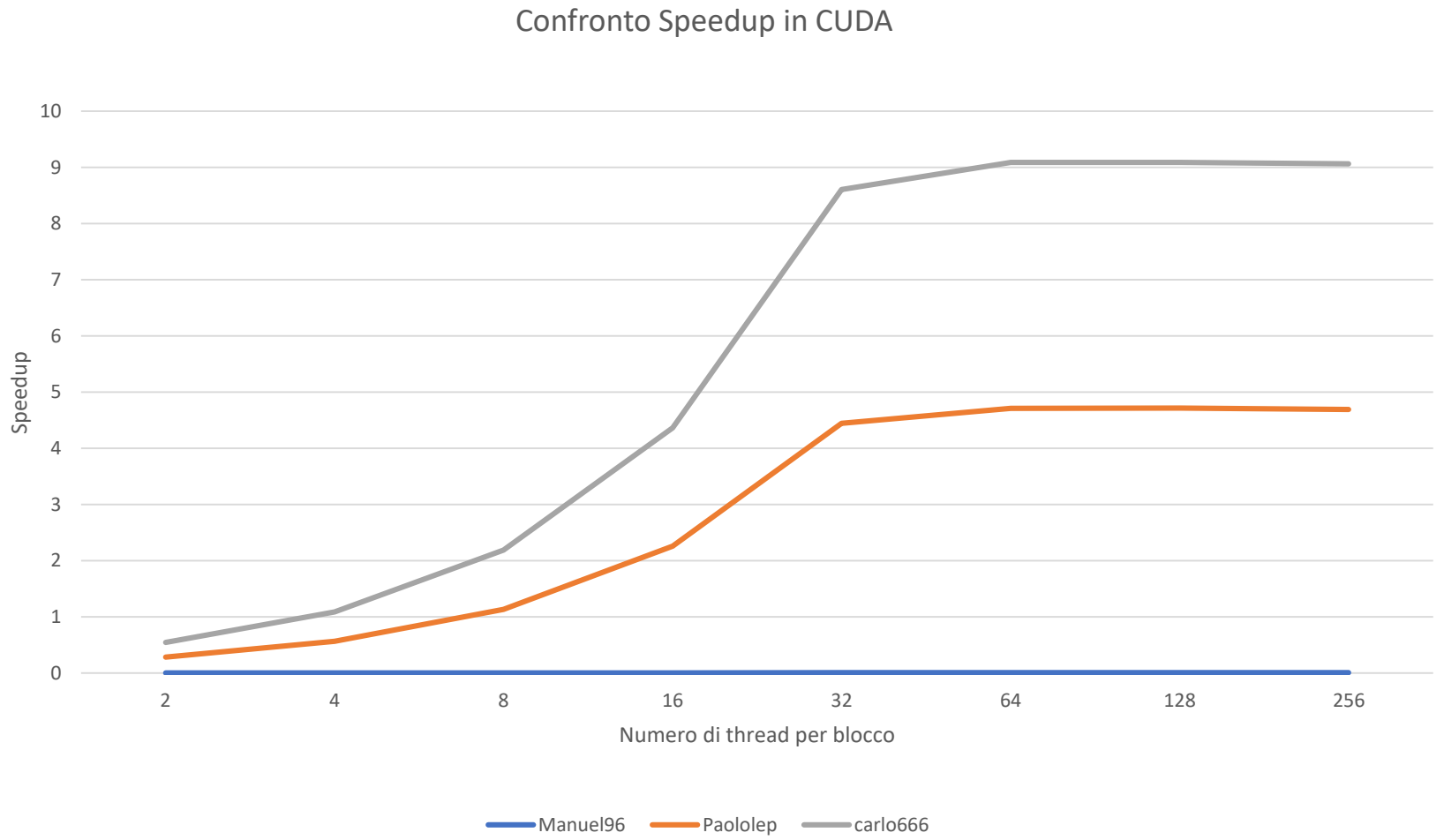
- **Valutazione e confronto speed up:** scelte tre password posizionate all'inizio, al centro e alla fine del dizionario e fatto variare il numero di thread, utilizzando 2, 4, 8, 16, 32, 64, 128, 256 thread.

- **Valutazione e confronto tempi di esecuzione:** scelte dieci password distribuite uniformemente all'interno del dizionario fissando il numero di thread in C a 16 e la block size in CUDA a 128.

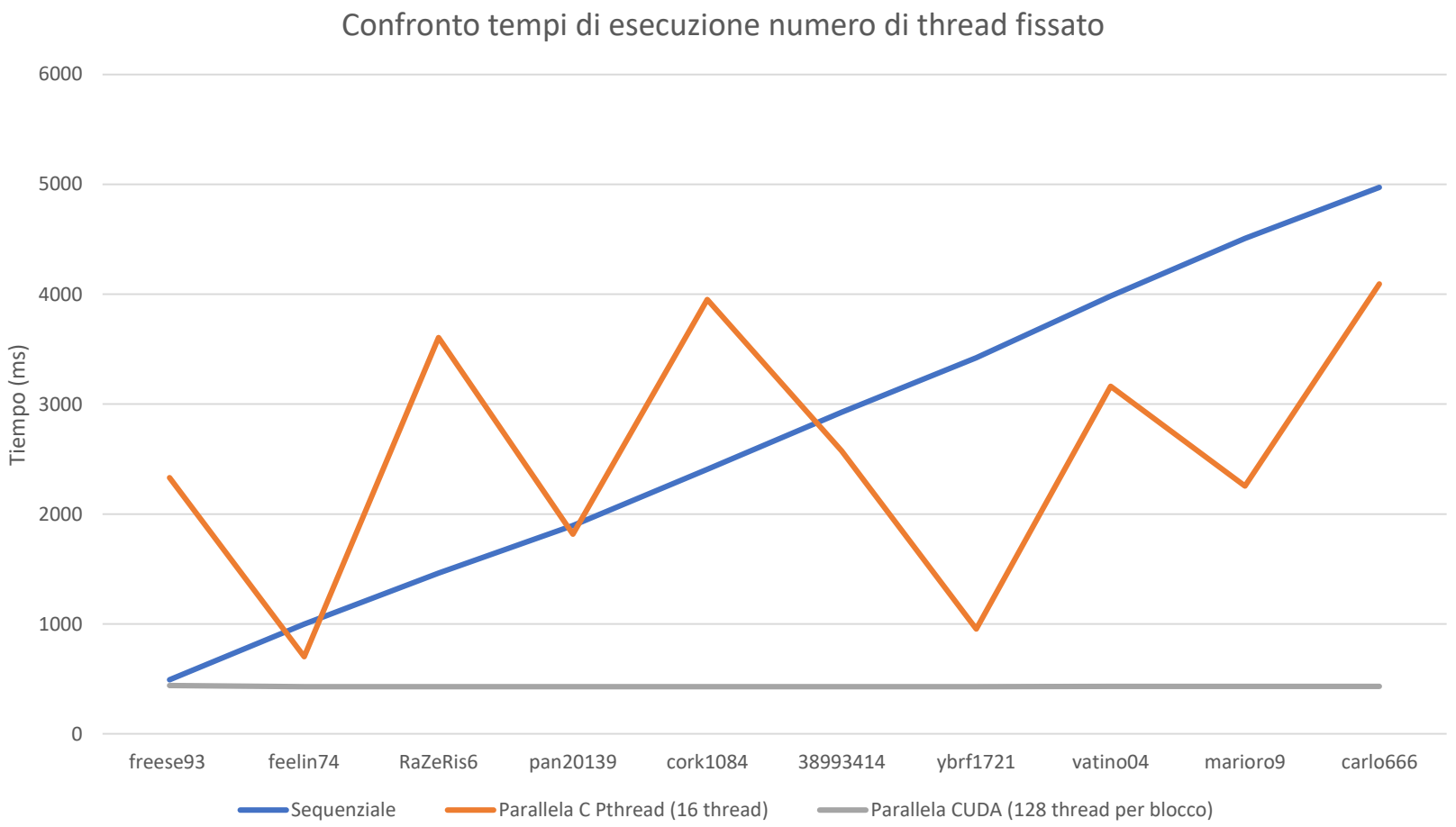
Confronto Speedup in C (Pthread)



Confronto Speedup in CUDA



Confronto tempi di esecuzione



Conclusione

In questo progetto abbiamo:

- Implementato un framework per effettuare un attacco a dizionario cifrato con DES
- Sviluppato in una versione sequenziale (C) e due parallele (C Pthread e CUDA)
- Effettuato analisi sullo speed up e sulle tempistiche di esecuzione

Grazie dell'attenzione