PC-2023/24 DES - Password Decryption

Alessandro Filino
E-mail address
alessandro.filino@edu.unifi.it

Abstract

Questo progetto si focalizza sull'implementazione di un attacco di tipo brute force per la ricerca di una password cifrata con DES. Viene quindi proposta una soluzione sequenziale in C++, una parallela seguendo il framework OpenMP ed una con CUDA .

1. Introduzione

Il Data Encryption Standard (DES), è un algoritmo di crittografia a blocchi tra i più diffusi e influenti nel campo della sicurezza informatica. Creato da IBM negli anni 70', DES è stato adottato come standard federale negli Stati Uniti nel 1977 ed è rimasto in uso fino all'inizio degli anni 2000.

1.1. Framework e Linguaggi di Programmazione

Lo sviluppo si è inizialmente concentrato nell'implementazione dello standard DES ricostruendo i passaggi fondamentali che l'algoritmo segue. Questo è stato fatto seguendo un approccio sequenziale attraverso il linguaggio C/C++.
Successivamente ci siamo concentrati su una implementazione parallela secondo le direttive poste da OpenMP e da CUDA.
OpenMP, acronimo di Open Multi-Processing, è una specifica di programmazione parallela che consente agli sviluppatori di scrivere codice che sfrutta efficacemente le architetture multi-core. L'approccio di OpenMP si basa sull'aggiunta di direttive ***pragma*** al codice sorgente, che indicano al compilatore come suddividere e distribuire il lavoro in base ai thread disponibili.
Il vantaggio di utilizzare framework come OpenMP è che non richiedono la ristrutturazione completa del codice sequenziale di partenza.
CUDA invece è un metodo di programmazione messo a disposizione da NVIDIA che consente l'esecuzione di algoritmi in maniera parallela su GPU. È basato su C e C++ e richiede la ristrutturazione del codice consentendo allo sviluppatore un pieno controllo dell' hardware a disposizione.

1.2. Strumenti di sviluppo

Per la realizzazione del progetto sono stati utilizzati i seguenti tool in fase di sviluppo:

- **CLion**: IDE sviluppato da JetBrains[4], progettato principalmente per lo sviluppo con linguaggi C e C++. CLion offre tools molto efficienti e già integrati a supporto dello sviluppatore come la gestione dei file CMake che semplificano notevolmente la configurazione dei progetti.
- **Google Tests**: Framework per lo Unit Test di linguaggi C e C++. È sviluppato e utilizzato principalmente da Google per testare il proprio software, ma è disponibile anche come risorsa open-source per tutti gli sviluppatori[2]. Google Test fornisce un insieme di strumenti e direttive per scrivere, organizzare e eseguire test in modo efficace e robusto. Nel nostro caso specifico sarà utilizzato per verificare che i risultati prodotti

da ogni funzione del nostro sistema produca i risultati aspettati se prese singolarmente.

2. Data Encryption Standard (DES)

Il Data Encryption Standard (DES) è un algoritmo di crittografia a blocchi simmetrici sviluppato negli anni 70' da IBM [7] e successivamente adottato come standard dal governo degli Stati Uniti per la sicurezza dei dati fino agli anni 2000.
L'algoritmo DES opera su blocchi di dati a 64 bit ossia, dato un testo composto da al più 64 bit (8 simboli ASCII) produce, utilizzando una chiave di crittografia a 56 bit, un testo cifrato di nuovo a 64 bit.
L'operazione di cifratura si compone di varie fasi che includono una serie di operazioni tra cui una permutazione iniziale, una fase di cifratura che segue il cifrario di Feistel [1] e una permutazione finale.
Operando attraverso l'uso di bit, per rendere efficienti le operazioni da svolgere nell'algoritmo, per l'implementazione sequenziale e parallela tramite openMP si è scelto di utilizzare la classe *std::bitset* presente in C++ adattando dinamicamente la propria dimensione attraverso l'uso dei *template*. Per CUDA invece i kernel sono basati su C e sono quindi stati utilizzati puntatori a char o ad interi (in base al tipo di rappresentazione richiesta).

2.1. Come generare le 16 Sub Keys

Per prima cosa nell'algoritmo è necessario creare 16 sub-keys [3] da 48 bit ciascuna a partire dalla chiave iniziale scelta.
Data una chiave principale composta da 8 caratteri scelti tra lettere o numeri, essa viene rappresentata in binario e quindi trasformata in una sequenza di 0 e 1 lunga 64 bit.
Applichiamo quindi una prima permutazione seguendo la tabella **parity_drop_table** e passando così dai 64 bit iniziali a 56 bit.

Table 1. Parity Drop Table

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Successivamente il risultato prodotto sarà suddiviso in due sottoblocchi ciascuno da 28 bit denominati **Left** e **Right**.
Le due parti seguono quindi 16 iterazioni in cui ad ogni ciclo, per ottenere una sub key:

- Si eseguono delle rotazioni circolari di una o due posizioni in base al contenuto della tabella **shift_table** nella posizione corrispondente all'indice di ciclo che stiamo seguendo
- Si ricompone una sequenza a 56 bit (ossia left + right dopo la fase di shifting)
- Si esegue una permutazione tra la stringa ottenuta e la tabella **key_compression_table** ottenendo così una delle 16 sub keys a 48 bit

Table 2. Key Compression Table

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

2.2. Cifrare un testo in chiaro con DES

Per cifrare un testo in chiaro a 64 bit, date le 16 sottochiavi ottenute dalla chiave iniziale scelta, è necessario:

- Eseguire una permutazione dei 64 bit iniziali con la tabella **initial_permutation_table**. Il risultato prodotto sarà sempre una stringa a 64 bit
- Dividiamo la stringa ottenuta in due sottoblocchi ciascuno da 32 bit denominati come **Left** e **Right**
- Eseguiamo i 16 round di cifratura ossia applichiamo l'algoritmo di ***Feistel Horst***.

Table 3. Initial Permutation Table

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Questo prevede diverse fasi:

1. **Espansione**: Il blocco destro (**Right**) a 32 bit viene espanso fino a 48 bit eseguendo una permutazione con la tabella **expansion_permutation_table**
2. **XOR Operation**: Il risultato ottenuto viene messo in XOR con la sottochiave chiave i-esima a 48 bit ottenuta precedentemente
3. **Sostituzione**: I 48 bit risultanti dall'operazione XOR sono divisi in otto gruppi di 6 bit ciascuno che determinano l'indice riga-colonna a cui accedere tra le 8 tavole di sostituzione (**s_box_table**) disponibili. Questa operazione sostituisce quindi il gruppo da 6 bit iniziale con 4 nuovi bit ossia producendo una nuova stringa finale a 32 bit

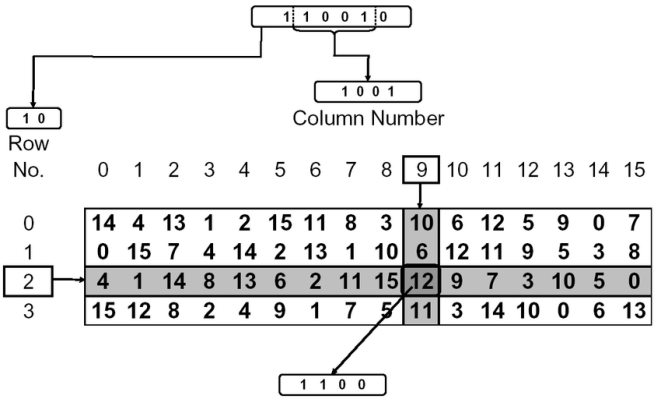

Figure 1. S-Box Example

4. **P-BOX Substitution**: Il risultato della fase di sostituzione viene quindi permutato con la tabella **straight_permutation_table** così da ottenere una nuova stringa a 32 bit
5. **XOR Operation**: Il risultato ottenuto viene messo in XOR con il blocco sinistro (**Left**)
6. **Swapping**: Si esegue lo swap tra il blocco destro e il blocco sinistro

- Completati i 16 round, il blocco destro e sinistro comporranno un'unica stringa a 64 bit. A questa andremo ad effettuare un'ultima permutazione seguendo la tabella **final_permutation_table**.

Table 4. Final Permutation Table

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Abbiamo così ottenuto la stringa cifrata a 64 bit.

3. Brute Force Attack

L'idea alla base per poter trovare una password cifrata conoscendo la propria lunghezza, il set di caratteri di cui si compone e la chiave di cifratura DES utilizzata, è stata quella di eseguire un attacco a forza bruta.
Supponiamo che gli 8 caratteri di cui la password si compone siano stati scelti nel seguente dataset:

Table 5. Allowed Characters

a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p
q	r	s	t	u	v	w	x
y	z	A	B	C	D	E	F
G	H	I	J	K	L	M	N
O	P	Q	R	S	T	U	V
W	X	Y	Z	1	2	3	4
5	6	7	8	9	0	.	/

L'attacco brute force implementato segue i seguenti passaggi:

- Componiamo password a 8 caratteri in ordine sequenziale a partire dalla stringa

"*aaaaaaaa*" fino a raggiungere l'ultima possibile "*////////*":

Iteration	Password generated
1	aaaaaaaa
2	baaaaaaa
3	caaaaaaa
...	...
i-1	2.W.caaa
i	2/W.caaa
...	...
64^8	////////

- Cifriamo la password attraverso l'algoritmo DES utilizzando la chiave nota
- Verifichiamo se la sequenza a 64 bit ottenuta corrisponde alla sequenza che stiamo cercando

Possiamo comprendere che l'elemento fondamentale per trovare la password è l'**iterazione** che stiamo eseguendo.
Risulta quindi fondamentale mantenere una struttura di generazione delle password comune per poter confrontare il caso sequenziale dai due casi paralleli.
Per fare questo, possiamo scegliere due diversi approcci:

- Dividere le combinazioni in *n* blocchi (dove *n* = n. threads a disposizione) ed assegnare ad ogni thread la generazione delle porzioni di password corrispondenti ad inizio e fine blocco assegnato:
  es. Ho un dataset di 281.474.976.710.656 passwords e 8 threads a disposizione:

Thread	start_index_password	end_index_password
0	0	35.184.372.088.831
1	35.184.372.088.832	70.368.744.177.663
2	70.368.744.177.664	105.553.116.266.495
3	105.553.116.266.496	140.737.488.355.327
4	140.737.488.355.328	175.921.860.444.159
5	175.921.860.444.160	211.106.232.532.991
6	211.106.232.532.992	246.290.604.621.823
7	246.290.604.621.824	281.474.976.710.655

  Questa è un'ottima strategia da utilizzare in quanto ogni thread esamina una porzione dell'intero dataset in diversi punti. Questo metodo però è difficilmente paragonabile con un approccio sequenziale nel quale la ricerca partirà sempre dall'indice 0 e che, nel caso peggiore, richiederà di esplorare tutte le 281.474.976.710.656 iterazioni.
- Esplorare le combinazioni di *n* in *n* (dove n. = n. thread a disposizione). L'approccio è simile al caso precedente ma anziché suddividere il dataset in blocchi, lo dividiamo in iterazioni.
  Es. Ho un dataset di 281.474.976.710.656 passwords e 8 threads (thread_number = 8) a disposizione. Fino a quando non ho raggiunto l'ultima password, ogni thread, genererà le stringhe a partire dal proprio indice di base e le successive *i + thread_number*:

Listing 1. generate_passwords

```
for (int i = thread_number; i <
    n_passwords; i += thread_number)
{
   password = generatePassword(i);
}
```

  Si è scelto quindi di operare con questo tipo di approccio poiché risulta più sensata la comparazione dei tempi di esecuzione tra il caso sequenziale e quelli paralleli nonché segue in maniera più appropriata la logica di un attacco brute force con generazione di password contigue.

Per fare ciò in **openMP**, l'istruzione da usare per suggerire al compilatore come distribuire i blocchi di password è :

```
#pragma omp for schedule(static,
    number_threads)
```

Nella Sez. 4 è stato paragonata l'efficienza tra questa istruzione openMP ed il caso in cui non venga specificato il secondo argomento nell'opzione schedule (*number_threads*).
In **CUDA** la stessa cosa può essere fatta indicando direttamente nel kernel le operazioni da far eseguire ad ogni thread (come visto prima in Listing 1).

Un ulteriore precisazione va fatta sulla **condizione di stop**: Poiché stiamo lavorando con

sistemi multi-thread, utilizzare una flag non è sufficiente ad interrompere il funzionamento di tutti i thread nel momento esatto in cui uno di essi soddisfi la condizione di ricerca. L'utilizzo di una variabile di questo tipo richiederebbe sincronizzazione ed un controllo che dovrebbe essere fatto o ad inizio ciclo di ogni thread o alla fine:

Listing 2. is_password_found

```
bool is_found = False; #variabile shared
for (int i = 0; i < n_passwords; i +=
    thread_number) {
        password = generatePassword(i);
        ..
        if(is_found) {
                return;
        }
}
```

Questo introdurrebbe un delay sempre più grande all'aumentare del numero di thread in esecuzione: Più il numero di thread sarà grande e maggiore sarà il tempo che dovrò attendere prima che tutti si sincronizzino tra loro.
Poiché in un sistema brute-force, l'obiettivo è trovare una password target nel minor tempo possibile, si è scelto di interrompere l'esecuzione parallela in maniera imminente.
In **openMP** questo si fa introducendo la direttiva:

```
#pragma omp cancel for
```

Per funzionare, questo richiederà che la variabile di ambiente **OMP_CANCELLATION** sia impostata a **True** prima dell'inizio dell' esecuzione del blocco parallelo e per evitare di specificarlo manualmente, questa viene impostata inizialmente all'inizio dell'esecuzione del programma:

Listing 3. Activate OMP Cancellation

```
char *hasCancel =
    getenv("OMP_CANCELLATION");
    if (hasCancel == nullptr) {
        setenv("OMP_CANCELLATION", "true",
            1);
    }
```

In questo modo indichiamo al compilatore di segnalare a tutti i thread, a prescindere dal loro stato di esecuzione, di interrompersi nel punto in cui è posto la direttiva:

```
#pragma omp cancellation point for
```

Il risultato finale sarà del tipo:

Listing 4. openMP_stop_condition

```
#pragma omp for schedule(static,
    number_threads)
for (long i = 0; i < n_passwords; i++) {
   generate_password(
   password_generate, password_length, i);
        ...
        if (cipher_password_target ==
            cipher_password_generate) {
                ...
                #pragma omp cancel for
        }
    #pragma omp cancellation point for
 }
```

In **CUDA** non esiste un metodo standard e sopratutto efficiente per interrompere in maniera imminente l'esecuzione di tutti i thread. Per questo motivo è stata utilizzata l'istruzione:

```
__trap();
```

Questa genererà un'interruzione causando l'arresto immediato del sistema.

Listing 5. cuda_stop_condition

```
for (long i = index; i < n_passwords; i +=
    threads_number) {
  generate_password(...);
  ...
        if(isBinaryEqual(cipher_current_password,
            cipher_password_target,
            blockSize)){
                ...
                __trap();
        }
 }
```

N.B: L'utilizzo dell'istruzione `__trap()` combinata alla funzione **cudaGetLastError()** per il debug, ritornerà errore. Essendo consapevoli di come stiamo utilizzando questa istruzione, possiamo tralasciare l'errore sollevato.

4. Test And Results

Andiamo adesso ad analizzare i risultati ottenuti dalle varie esecuzioni.
Per prima cosa analizziamo il caso C++ e OpenMP ricordando che la macchina con cui stiamo eseguendo i test è un `Macbook Pro 14, M1 Pro` quindi con un processore ARM a 8 core divisi in: 6 performance core, 2 efficenty Core.
Tutti i test effettuati sono stati fatti utilizzando come **chiave di cifratura DES** la stringa: `A4rT9v.w` e la **password cercata** è: `2/W.caaa`.

Per prima cosa analizziamo il grafico in cui valutiamo il tempo di ricerca della password necessario tra un esecuzione sequenziale e i due approcci paralleli con **openMP** valutando le differenze specificando o meno il parametro `number_threads` nell'istruzione:

```
#pragma omp for schedule(static,
    number_threads)
```

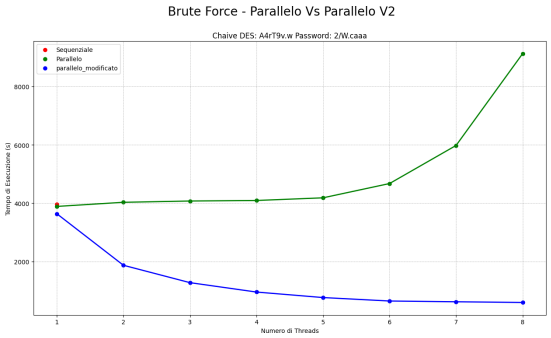

Figure 2. Tempo di ricerca variando la distribuzione delle password tra i vari thread

Come possiamo vedere dalla figura, aumentando il numero di threads senza specificare come frammentare le password (non utilizzare il parametro `number_threads`) aumenta drasticamente il tempo di esecuzione (linea verde).
Al contrario, seguendo l'approccio descritto nel capitolo 3 (linea blu), il tempo di ricerca necessario diminuisce drasticamente all'aumentare del numero di core.
Valutiamo quindi lo **speed-up** tra il tempo necessario nel metodo sequenziale e il metodo parallelo al variare dei thread in uso.

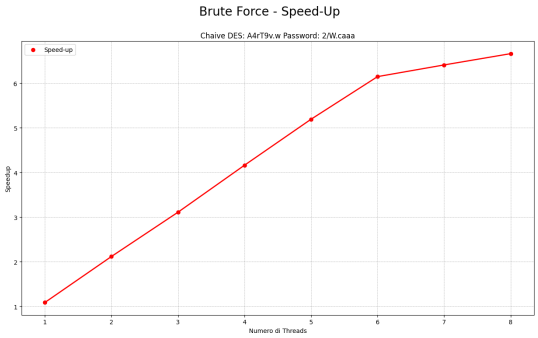

Figure 3. Speed-up sequenziale / parallelo openMP con 8 thread

Come possiamo notare dal grafico in Fig. 3 l'andamento iniziale è costante all'aumentare del numero di thread per poi subire un rallentamento superato l'uso di 6 threads.

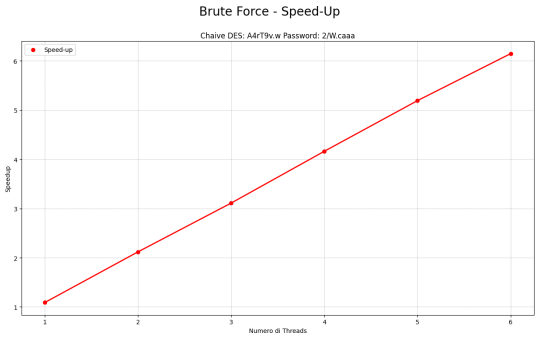

Figure 4. Speed-up con 6 thread

Questo mostra infatti come il processore abbia 6 core ad alte prestazioni e 2 che puntano all'efficienza.
Aumentando ancora il numero di thread superando il numero di core disponibili, possiamo notare un ulteriore flessione dovuta alle risorse comunque richieste dal S.O durante l'esecuzione dei test (fig. 5).

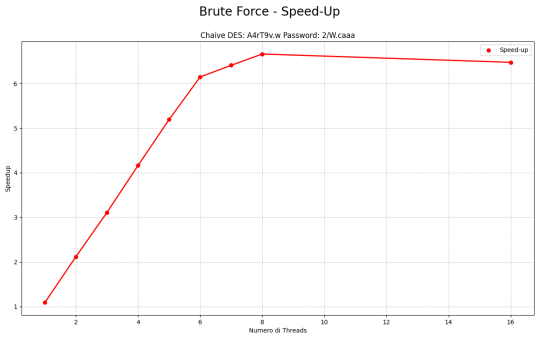

Figure 5. Speed-up con 16 thread

Passiamo ora all'uso di **CUDA** per risolvere il medesimo problema e analizziamone i risultati ottenuti.
La GPU utilizzata in questo caso è una `NVIDIA RTX A2000` [5] scheda con 12 GB di memoria DDR6, ammette fino a 1024 thread per block ed un massimo di 2.147.483.647 blocchi (**getGPUProperties**(*gpuID*)).
Per poter confrontare il precedente approccio con la parallelizzazione in GPU si è scelto di mantenere come elemento variabile il **numero totale di thread** da utilizzare per ogni esecuzione.
Per come è strutturato CUDA questo non sarebbe l'ottimale poiché rischiamo di non avere warp densi e ben organizzati.
Per ottimizzare la gestione dei blocchi, prima di eseguire il kernel il numero di thread viene organizzato in maniera efficiente in questo modo:

- Il numero di thread totale scelto è inferiore a 32: siamo nel caso peggiore possibile. In questo caso userò un solo blocco con all'interno un numero di thread pari a quelli scelti
- Il numero inserito è un multiplo di 256, 128 o 32: In questo caso il numero di blocchi in uso sarà uguale al risultato della divisione tra il numero di thread scelto e uno tra questi casi (favorendo l'uso del multiplo più alto). Ogni blocco conterrà quindi il multiplo di riferimento
- Il numero inserito non rientra nei casi precedenti: In questo caso cercherò di ottimizzare l'esecuzione approssimando il numero inserito ad uno dei multipli di riferimento indicati al primo punto e successivamente operando come descritto precedentemente

I multipli 32, 128 e 256 sono stati scelti come candidati valutando l' **occupancy** ottenuta per il problema in questione sapendo che stiamo utilizzando **52 registri** (valutazione fatta profilando l'esecuzione attraverso il tool NVIDIA Visual Profiler [6]):

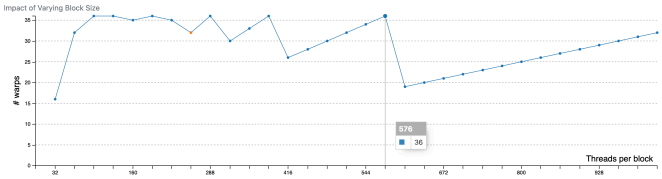

Figure 6. CUDA occupancy calculator

In Fig. 7 riportiamo quindi l'andamento delle performance al variare del numero totale di thread scelti.

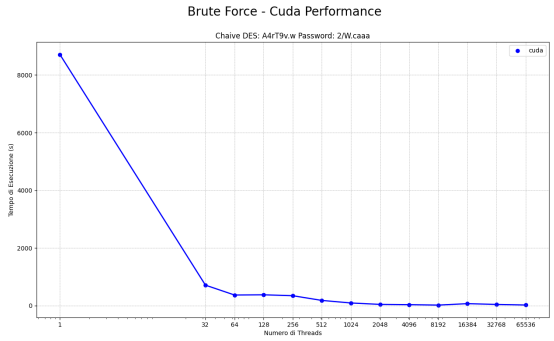

Figure 7. CUDA Performance

Come possiamo vedere troviamo che il caso migliore è ottenuto utilizzando 8192 thread divisi in 32 blocchi da 256 threads ciascuno e che il tempo di esecuzione totale richiesto per completare il problema è di $\sim$17s contro i $\sim$595s ($\sim$ 10 min) di openMP.
Possiamo inoltre valutare lo speedUP che otteniamo tra l'esecuzione sequenziale iniziale e l'utilizzo di CUDA.

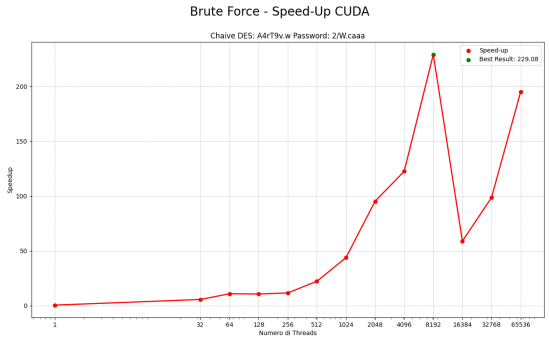

Figure 8. CUDA SpeedUP

Riportiamo infine in Fig. 9 un confronto tra i vari speedUP sottolineando ancora la vertiginosa differenza tra un approccio CPU centrico ed uno GPU.

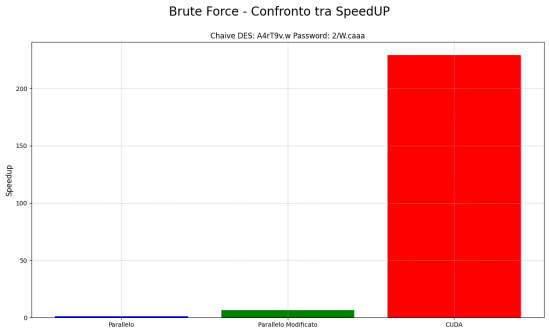

Figure 9. Confronto SpeedUP generale tra i vari metodi di parallelizzazione

N.B: Analizzando meglio il calcolo dell'occupancy otteniamo i seguenti risultati:

Block Size	Occupancy
32	0,33 %
128	0,75 %
256	0,66 %
384	0,75 %
576	0,75 %

Sarebbe quindi lecito aspettarsi che con un block-size da 576 threads avremmo dovuto ottenere risultati migliori rispetto ad uno con 256.
Questo non accade nemmeno aumentando o diminuendo il numero di threads totali distribuiti ossia aumentando o diminuendo il numero di blocchi in uso.

Table 6. Execution Time results al variare della Block Size

Block Size	Occupancy	Total Threads	Execution Time
32	0,33 %	8192	**23.99**
128	0,75 %	8192	**29.06**
256	0,66 %	8192	**17.30**
384	0,75 %	8064	**18.11**
384	0,75 %	8448	**17.41**
384	0,75 %	8832	**27.65**
576	0,75 %	7488	**58.80**
576	0,75 %	8064	**31.13**
576	0,75 %	8640	**48.94**

Questo risultato evidenzia come così posto, il problema non stia effettivamente sfruttando a pieno le performance che la GPU potrebbe offrire nonostante questa esegua un carico del 100% (cmd: `$ nvidia-smi`). Questo risultato infatti è dovuto al metodo con cui abbiamo distribuito il lavoro nei vari threads senza lasciare che fosse la GPU a svolgere questo compito in autonomia. Per risolvere questo problema e quindi ottenere migliori performance dovremmo rimuovere il ciclo `for` all'interno del kernel CUDA cambiando però la logica che abbiamo supposto nella Sez. 3.

References

[1] U. degli studi di Roma. Cifrario di feistel. http://ricerca.mat.uniroma3.it/users/merola/critto09/deshandout.pdf.

[2] Google. Google Tests. https://github.com/google/googletest.

[3] J. O. Grabbe. The des algotithm. https://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm.

[4] JetBrains. CLion. https://www.jetbrains.com.

[5] NVIDIA. Nvidia rtx a2000, scheda grafica a2000 12 gb. https://www.nvidia.com/it-it/design-visualization/rtx-a2000/.

[6] NVIDIA. Nvidia visual profiler. https://developer.nvidia.com/nvidia-visual-profiler.

[7] Wikipedia. Data Encryption Standard. https://it.wikipedia.org/wiki/Data_Encryption_Standard.