Decriptazione password

Brute Force con Dizionario in CUDA

Liuzzo Antonino Mauro

Introduzione

Hash e Brute Force

- Obiettivo: Decifrare una password di 8 caratteri avendo a disposizione il suo valore di hash generato tramite l'algoritmo DES. La password contiene lettere (maiuscole/minuscole) e numeri.
- Una funzione crittografica di hash converte dati di lunghezza arbitraria in una stringa di testo fissa, ossia il valore di hash. Tale funzione è unidirezionale, ossia è molto difficile se non impossibile invertirla.
- Avendo un valore di hash, per scoprire la stringa che lo ha generato, un possibile approccio consiste nel provare tutte le possibili combinazioni dei dati in ingresso e vedere quale da il risultato desiderato. Questo approccio viene definito Attacco a Forza Bruta, o Brute Force.
- Con il Brute Force, prima o poi, si ottiene sempre la soluzione.
- Fattore positivo: Con il Brute Force, prima o poi, si ottiene sempre la soluzione.
- Fattore negativo: Con il Brute Force, prima o poi, si ottiene sempre la soluzione.
- Quando il numero di caratteri della password in ingresso e la dimensione del charset è elevato, il numero di possibili combinazione è elevatissimo. Trovare la giusta soluzione potrebbe richiedere moltissimo tempo.

Introduzione

Brute Force con Dizionario

- Gli utenti utilizzano molto spesso come password delle parole di senso compiuto, e molto spesso utilizzano come termini parole a loro familiari.

- È possibile sfruttare tale aspetto creando un dizionario contenente delle parole la cui la probabilità che esse siano la soluzione corretta è piuttosto elevata. La creazione di un buon dizionario richiede una buona conoscenza della vittima dell'attacco.

- In questo caso è stato creato un dizionario piuttosto generico contente:
  - Parole comuni di 8 lettere in inglese (e.g. password).
  - Parole comuni di 8 lettere in inglese con iniziale in maiuscolo (e.g. Password).
  - Leet delle due categorie precedenti (e.g. p4ssw0rd).
  - Composizioni di 8 numeri che costituiscono una data (e.g. 23011990).

- In totale il dizionario contiene 285465 parole

Introduzione

Alcune Considerazioni

- La cifratura della password avviene tramite algoritmo DES. La funzione utilizzata prende in input due valori di tipo uint64_t, pertanto avviene la conversione delle parole presenti nel dizionario e quelle generate.

```
__host__ __device__ uint64_t full_des_encode_block(uint64_t key, uint64_t block)
```

- La password viene cifrata utilizzando come chiave la password stessa. Non viene utilizzato nessun salt

Versione Sequenziale

Brute Force con Dizionario

- La versione sequenziale è pressochè banale:
  - Controllo sull'input.
  - Conversione dell'inpunt in utf64_t e cifratura dell'input.
  - Apertura del file dizionario.
  - Per ogni parola del dizionario:
    - Cifratura della parola e verifica degli hash
    - Eventuale report risultato (se si è trovata la soluzione).
  - Generazione tramite for annidati di tutte le combinazioni possibili.
  - Per ogni possibile combinazione:
    - Cifratura della parola e verifica degli hash
    - Eventuale report risultato (se si è trovata la soluzione)

Versione Parallela

Brute Force con Dizionario

- La versione parallela è stata implementata tramite CUDA. CUDA ci permette di utilizzare moltissimi thread. Ciò ci consente di utilizzare un thread per ogni combinazione possibile.
- Sono introdotti alcuni overhead dovuti alla gestione della memoria della GPU
- Sono stati progettati due kernel diversi, uno per la ricerca tramite dizionario e uno per il brute force vero e proprio

Versione Parallela

Flusso di Esecuzione

- Controllo sull'input, conversione in *uint64_t* e cifratura
- Import del dizionario, conversione in *uint64_t* e cifratura
- Preparazione delle variabili per il Dictionary Kernel
- Lancio del Dictionary Kernel e stampa del risultato
- Se la Parola è stata trovata:
  - Liberazione delle variabili delle variabili
- Se la Parola non è stata trovata:
  - Liberazione delle variabili utilizzate dal Dictionary Kernel e non usate dal Matrix Kernel
  - Preparazione Dati per il Matrix Kernel
  - Lancio dei Brute Kernel (in un ciclo for) e stampa del risultato
  - Nel caso peggiore, vengono lanciati 2^{64}/(brute_blocks*brute_threads) Brute Kernel diversi, ognuno con un diverso valore di offset.

Versione Parallela

Dictionary Kernel

- **__global__ void dict_kernel**(uint64_t *dictionary, uint64_t *result)
- Input: dizionario e variabile per scrivere il risultato
- Il dizionario viene salvato nella global memory, dato che ogni parola viene utilizzata una sola volta
- L'hash da decifrare viene salvato nella constant memory e poi caricato nella shared memory, per una maggiore efficienza.

Versione Parallela

Dictionary Kernel

- **__global__ void dict_kernel**(uint64_t *dictionary, uint64_t *result)
- Il kernel è progettato per blocchi di thread monodimensionali
- Le coordinate del thread vengono utilizzate per determinare quale parola del dizionario verrà provata dal thread.
- Vengono utilizzati blocchi da 512 thread: Il numero di blocchi viene calcolato in base alle dimensioni del dizionario
- Viene inserito un controllo per verificare se il thread entra nei limiti delle dimensioni del dizionario (condizioni al contorno per thread dell'ultimo blocco)

Versione Parallela

Brute Kernel

- `__global__ void brute_kernel(uint64_t *result, int offset)`
- Input: variabile per scrivere il risultato e offset
- L'hash da decifrare viene salvato nella constant memory e poi caricato nella shared memory, per una maggiore efficienza.
- L'offset viene utilizzato in modo che kernel diversi provino parole diverse (e.g. il thread 0 del blocco 0 ha un indice pari a 0x3030303030303030, il thread 0 del blocco 1 ha un indice pari a 0x3030303030343030)

Versione Parallela

Brute Kernel

- **__global__ void brute_kernel**(uint64_t *result, int offset)
- Anche in questo caso, ad ogni thread viene “assegnata” una parola da provare.
- Ad ogni parola di 8 lettere corrisponde un intero senza segno di 64 bit. Ogni thread possiede un indice univoco che lo identifica (ottenuto tramite *blockIdx.x*, *blockDim.x* e *threadIdx.x*).
- L’indice viene traslato di 0x3030303030303030 in modo che il thread 0 del blocco 0 “provi” l’equivalente della parola “00000000”. In questo modo si evita l’utilizzo di alcuni thread “inutili” (e.g il numero 0x0000000000000000 non corrisponde ad una parola).
- Si utilizzano grid 1D formati da 512 blocchi 1D, a loro volta costituiti da 512 thread (valori impostati empiricamente).

Versione Parallela

Brute Kernel

indice	0	…	**4612879247298593283**	…	2^{64}	…..
Indice traslato(hex)	3030303030303030		70346D6C6F313233		N/A	.N/A
parola	.00000000		p4mlo123		N/A	.N/A

Confronto Tempi di Esecuzione

Hardware Utilizzato

- CPU: Intel i7 8550U 4 core 1.8 GHz (Frequenza Base)
- 8 GB RAM DDR4 2400 MHz
- GPU: NVIDIA GeForce MX150 4GB DRAM, 384 CUDA Core, Compute Cap. 6.1

Confronto Tempi di Esecuzione

Esempi presenti Nel dizionario

Password	Seq Time(s)	Parallel Time(s)
augurers	0,062	0,692
carolled	0,162	0,692
fondness	0,404	0,693
password	0,738	0,694
Furthest	1,626	0,718
Ravening	2,030	0,694
h3m4t1ns	2,881	0,692
s4Vt01rs	3,303	0,688
02121996	5,427	0,725
10021920	6,937	0,716

Nella versione sequenziale il tempo dipende dalla posizione della parola nel dizionario. In casi fortunati il tempo di esecuzione è minore della versione parallela.

Nella versione parallela il tempo di esecuzione è quasi costante (differenza tra max e min di 0,037 s) per ogni parola del dizionario.

Confronto Tempi di Esecuzione

Esempi non presenti Nel dizionario

Password	Seq Time(s)	Parallel Time(s)
00000000	6,724	1,160
000000z0	7,636	1,207
00000z00	13,900	9,934
00000z0z	16,050	9,914
00000zA0	16,701	9,930
00000zAz	17,335	9,948
00000za0	17,473	9,914
00000zaz	17,846	9,941
00000zz0	17,958	9,965
00000zzz	18,651	9,925

Nella versione parallela il tempo di esecuzione dipende principalmente dal numero di kernel eseguiti.