Parallel Computing Mid-Term
DES Decryption con C, Cuda & OpenMP

Edoardo D'Angelis
Università degli studi di Firenze
edoardo.dangelis@stud.unifi.it

Mirco Ceccarelli
Università degli studi di Firenze
mirco.ceccarelli@stud.unifi.it

Abstract

L'obiettivo di questo progetto è quello di decifrare una password di otto caratteri avendo a disposizione il suo valore di hash, generato tramite algoritmo DES (Data Encryption Standard), e supponendo di conoscere il salt con cui è stata cifrata. In questa relazione verranno presentati e confrontati tre approcci diversi: uno sequenziale (in C) e due paralleli (con CUDA e OpenMP).

1. Introduzione

In questo progetto vogliamo confrontare i tempi di esecuzione di un *attacco con dizionario* all'algoritmo di cifratura DES utilizzando tre metodologie diverse.

Con l'espressione *attacco con dizionario* si intende una tecnica di attacco alla sicurezza di un sistema, in cui si suppone di avere un dizionario di password in chiaro e fornendo una password cifrata, il programma dovrebbe scoprire se si trova nel dizionario, cifrando ad una ad una ogni parola per poi confrontare il risultato con l'hash della password.

1.1. DES

Il DES (Data Encryption Standard) è un algoritmo di cifratura basato su chiave simmetrica per cifrare e decifrare dati, risalente agli anni '70. Attualmente il DES è considerato insicuro perchè la chiave utilizzata per la cifratura è di soli 56 bit, il che lo rende facilmente vulnerabile agli attacchi. Negli ultimi anni tale algoritmo è stato sostituito dall'AES (Advanced Encryption Standard) [1].

1.2. Descrizione del dataset

Il dataset utilizzato per gli esperimenti è stato estratto da uno più grande contenente 10 milioni di password [2]. Con uno script Python sono state estratte solamente le password lunghe 8 caratteri e appartenenti al set [a-zA-z0-9]. Il dizionario finale contiene un milione e mezzo di password (file *txt*, 13MB di dimensione).

1.2.1 Scelta delle password da testare

Per un confronto più accurato sui tempi di esecuzione, anziché selezionare le password casualmente sono state scelte dieci password uniformemente distribuite lungo il dizionario, riportate di seguito: *'sirpizza', '3kingdom', 'tyleisha', 'marumari', 'giacomix', 'dbcookie', 'Yesssss', 'Mypaypa1', '6Melissa', '1Mazzola'*.

Per la valutazione dello speedup ottenuto con le due versioni parallele, invece, di queste dieci ne sono state selezionate tre (una all'inizio, *'3kingdom'*, una a metà, *'giacomix'*, e una alla fine, *'6Melissa'*).

1.3. Specifiche hardware

Per questi esperimenti è stata utilizzata un'istanza *EC2* di *Amazon Web Services* con le seguenti specifiche:

- Istanza *p2.xlarge* con accelerazione del calcolo
- Processore a 2,3 GHz (base) e 2,7 GHz (turbo) Intel Xeon E5-2686 v4
- GPU NVIDIA K80, ognuna con 2.496 core di elaborazione in parallelo
- 90 GiB di memoria CPU e 12 GiB di memoria GPU

2. Implementazione

Sono state eseguite tre implementazioni, una in maniera sequenziale in linguaggio C e due in maniera parallela rispettivamente con CUDA e OpenMP.

L'algoritmo DES non è stato implementato ma è stata usata la funzione *crypt* per il C e *crypt_r* per OpenMP, entrambe definite nella libreria *crypt* del C.

Per quanto riguarda CUDA, data l'assenza della libreria *crypt*, è stata utilizzata un'implementazione open-source del DES specifica per la piattaforma [3].

Come *salt*, per cifrare le password, è stata utilizzata una parola di due caratteri ("PC", Parallel Computing), in modo che *crypt* e *crypt_r* facciano uso dell'algoritmo DES e non MD5, come indicato nel manuale della libreria GNU. [4]

2.1. Versione Sequenziale in C

La versione sequenziale dell'algoritmo è piuttosto semplice. Dato in input il dizionario e la password cifrata da cercare, l'algoritmo scorre tutto il dizionario e ad ogni passo viene cifrata la parola corrente con DES. Se i due hash sono uguali, la password è stata trovata e il ciclo viene interrotto. Si itera fino alla fine del dizionario.

Per misurare il tempo di esecuzione è stata utilizzata la funzione *clock* della libreria standard del C *time* che restituisce il tempo di CPU espresso in unità *clock_t* (colpi di clock). Per ottenere il risultato in secondi si divide per la costante CLOCKS_PER_SEC.

```
const char* password = "3kingdom";
char* enc = strdup(crypt(password, SALT));

while ((getline(&current, &len, dict)) != -1) {
  char* curr_enc = strdup(crypt(current, SALT));
  if (strcmp(enc, curr_enc) == 0) {
    // Password trovata!
    break;
  }
}
```

1. Esempio di ricerca sequenziale nel dizionario

2.2. Versione parallela con CUDA

Data l'assenza di *crypt* su piattaforma CUDA, è stata utilizzata una libreria esterna open-source per la cifratura DES: *des-cuda* [3]. Questa espone un metodo *full_des_encode_block* che prende in input due valori di tipo *uint64_t* (interi a 64 bit, parola da cifrare e *salt*) e restituisce a sua volta un *uint64_t* corrispondente alla cifratura DES della parola con il *salt* fornito. Per adeguarsi, quindi, nella fase di preprocessamento tutte le parole del dizionario sono state convertite in interi a 64 bit.

Dopo aver allocato e copiato la memoria necessaria da *host* a *device*, si effettua il lancio del *kernel*, porzione di codice che viene eseguita in parallelo dai vari thread della GPU, con dimensione dei *blocchi* variabile e dimensione del *grid* dipendente dalla lunghezza del dizionario.

```
// Organizzazione blocchi e grid
dim3 blockDim(blockSize);
dim3 gridDim(DICTIONARY_SIZE/blockDim.x + 1);

// Lancio kernel
kernel<<<gridDim, blockDim>>>(device_dictionary, found);

// Copio risultato da device a host
CUDA_CHECK_RETURN(cudaMemcpy(psw_found, found,
                  sizeof(uint64_t),
                  cudaMemcpyDeviceToHost));

if (*psw_found) {
  // Password trovata!
}
```

2. Esempio di lancio del kernel in CUDA

Il dizionario di parole viene salvato in memoria globale, in modo che sia condiviso tra tutti i thread, e viene passato come input al *kernel*.

È stata utilizzata inoltre una variabile *found* di tipo *uint64_t*, inizializzata a zero, che viene assegnata nel *kernel* non appena un thread decifra correttamente la password. In questo modo, terminata l'esecuzione del *kernel*, viene copiata la porzione di memoria relativa a questa variabile da *device* a *host*, ed è quindi possibile sapere se la password è stata decifrata correttamente.

In questo caso non è stato necessario chiamare il metodo *cudaDeviceSynchronize()* dopo il lancio del kernel, solitamente utilizzato per attendere la terminazione di tutti i task della GPU, poichè la successiva operazione di copia di memoria (*cudaMemcpy* di tipo *cudaMemcpyDeviceToHost*) da sola garantisce che l'esecuzione avvenga in modo seriale, nonostante il lancio del kernel sia asincrono. In CUDA, infatti, tutte le operazioni che riguardano lo stesso *stream* sono sincrone [5].

Se la password è stata trovata da un thread, come per la versione sequenziale, si misura il tempo di esecuzione utilizzando la funzione *clock* della libreria standard *time* del C. Per ottenere il risultato in secondi si divide per la costante CLOCKS_PER_SEC.

Per quanto riguarda la memoria della GPU è stato scelto di salvare il *salt* e la password cifrata da cercare nella *device constant memory*, essendo entrambe informazioni costanti che non variano durante l'esecuzione del *kernel* e che richiedono la sola lettura.

Ogni thread della GPU esegue il controllo su una sola parola, in base all'indice calcolato a partire dalle sue coordinate (*blockIdx.x, blockDim.x* e *threadIdx.x*).

Viene anche eseguito un ulteriore controllo per assicurarsi che l'indice del thread entri nei limiti della dimensioni del dizionario, in modo da evitare *overflow* al contorno dell'ultimo blocco.

```
__constant__ uint64_t salt;
__constant__ uint64_t encryptedPassword;

__global__ void kernel(uint64_t *dict, uint64_t *found) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < DICTIONARY_SIZE) {
    uint64_t psw = dict[idx];
    uint64_t enc = full_des_encode_block(psw, salt);
    if (enc == encryptedPassword) {
      // Password trovata!
      *found = psw;
      return;
    }
  }
}
```

3. Esempio di kernel in CUDA

2.3. Versione parallela con OpenMP

Per quanto riguarda la versione parallela con OpenMP è stato scelto di utilizzare una classe, *Decrypter*.

Il costruttore prende in ingresso il nome del file *txt* relativo al dizionario e il *salt* da utilizzare per la cifratura. Seguendo l'idioma RAII (*Resource Acquisition Is Initialization*), il dizionario viene letto nel costruttore e ogni parola viene inserita in un vettore di stringhe, *fullDictionary*, variabile privata della classe.

All'interno di *Decrypter* è stato dichiarato anche un metodo, *setPassword*, che salva in una variabile privata la password cifrata tramite la funzione *crypt*.

La classe espone inoltre un metodo *decrypt*, che prende in ingresso il numero di thread da utilizzare nella direttiva di OpenMP *#pragma omp parallel num_threads*, ed esegue l'algoritmo principale. Si scorre quindi l'intero vettore di password utilizzando la direttiva *#pragma omp for* e, all'interno del ciclo, la parola corrente viene cifrata tramite la funzione *crypt_r*, versione *reentrant* di *crypt*.

Poichè i cicli di OpenMP non possono essere interrotti, è stata utilizzata una variabile booleana di tipo *volatile* (che garantisce la consistenza nella lettura del suo valore da parte di tutti i thread) per segnalare a tutti gli altri thread che la password è stata trovata, in modo che ciclino a vuoto senza eseguire alcuna istruzione finchè non terminano.

Il tempo di esecuzione è stato misurato utilizzando il metodo *now()* della classe *steady_clock* della libreria *std::chrono*.

```
double Decrypter::decrypt(int threads) {

  volatile bool found = false;
  auto start = chrono::steady_clock::now();

#pragma omp parallel num_threads(threads)
    {
  struct crypt_data data;
  data.initialized = 0;

#pragma omp for
  for (int i = 0; i < dict.size(); i++) {
    if (found) continue;
    char *current = crypt_r(dict[i], salt, &data);
    if (strcmp(current, encrypted) == 0) {
       // Password trovata!
       found = true;
    }
  }
    }

  if (found) {
    auto end = chrono::steady_clock::now();
    std::chrono::duration<double> s = end - start;
    return s.count();
  } else {
    return 0;
  }
}
```

4. Esempio di ricerca in parallelo con OpenMP

3. Esperimenti & Risultati

Per ogni esperimento sono state eseguite cinque iterazioni in modo da ottenere un misura più accurata dei risultati. Tutti i grafici presentati di seguito sono stati generati con uno script Python utilizzando la libreria *matplotlib*.

L'accuratezza dei risultati ottenuti è stata valutata tramite la metrica dello *speedup* (S), definito come il rapporto tra il tempo di esecuzione sequenziale (t_s) e il tempo di esecuzione parallelo (t_p): $S = \frac{t_s}{t_p}$.

3.1. Valutazione dello Speedup con CUDA

Date tre parole posizionate rispettivamente una all'inizio (*"3kingdom"*), una in mezzo (*"giacomix"*) e una alla fine (*"6Melissa"*) del dizionario, è stato misurato lo *speedup* al variare del numero di thread per blocco.

In particolare sono stati usati 8, 16, 32, 64, 128, 256, 512 thread per blocco (*block_size*), mentre il numero di blocchi per ogni *grid* è data dalla formula $\frac{dictionary\_size}{block\_size} + 1$, dove $dictionary\_size$ è il numero di parole presenti nel dizionario.

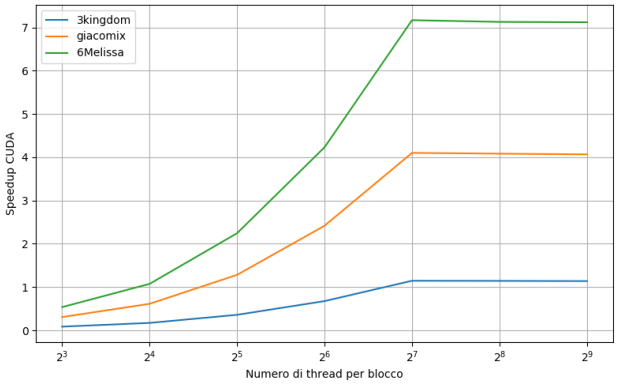

Figura 1. Speedup in CUDA

Come si può vedere dalla figura 1 lo *speedup* maggiore si ottiene con le parole posizionate in fondo al dizionario e il suo valore cresce all'aumentare del numero di thread per blocco, stabilizzandosi e rimanendo costante dopo 128 thread per blocco.

Come era verosimile aspettarsi, soltanto per parole all'inizio del dizionario risulta più efficiente la versione sequenziale ($speedup \leq 1$) soprattutto con numero limitato di thread per blocco.

3.2. Valutazione dello Speedup con OpenMP

Date tre parole posizionate rispettivamente una all'inizio (*"3kingdom"*), una in mezzo (*"giacomix"*) e una alla fine (*"6Melissa"*) del dizionario, è stato misurato lo *speedup* al variare del numero di thread di OpenMP. In particolare sono stati usati 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 thread.

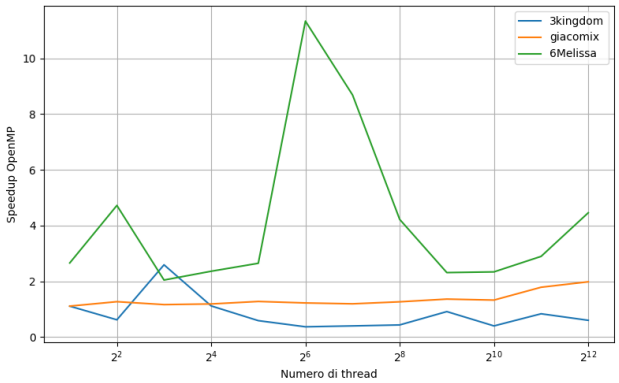

Figura 2. Speedup in OpenMP

Come si può vedere dalla figura 2, in generale lo *speedup* maggiore si ottiene con le parole posizionate in fondo al dizionario; in particolare con la parola *"6Melissa"* presa in esame è possibile ottenere uno *speedup* maggiore di 11 con 64 thread. Questo si tratta però chiaramente di un caso fortunato e totalmente dipendente dalla posizione che assume la parola nella divisione automatica in *chunk* da parte di OpenMP.

Per le parole all'inizio del dizionario, come ad esempio *"3kingdom"*, la parallelizzazione con OpenMP risulta invece poco efficiente anche all'aumentare del numero di thread in confronto alla versione sequenziale (infatti $speedup \leq 1$ quasi ovunque). Questo risultato può essere attribuito all'elevato *overhead* introdotto dalla gestione dei thread.

3.3. Confronto sui tempi di esecuzione

Per un'ulteriore valutazione sono state quindi scelte dieci parole uniformemente distribuite lungo il dizionario, in modo da poter confrontare direttamente i tre approcci diversi (C, CUDA e OpenMP) in alcuni casi particolari.

In CUDA è stato scelto il caso con 128 thread per blocco, mentre per OpenMP è stato scelto il caso con 32 thread.

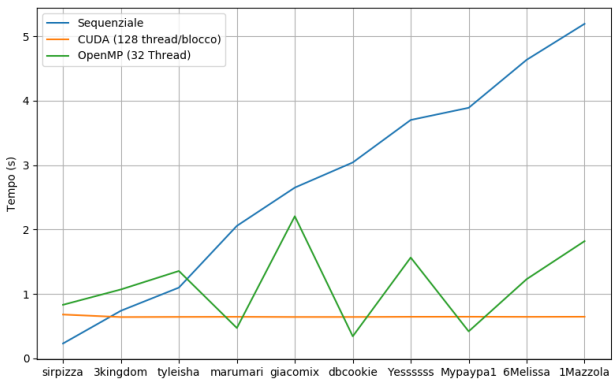

Figura 3. Confronto tempi di esecuzione

Come si può chiaramente notare dalla figura 3, da un certo punto in poi del dizionario ($\approx$ 25%) entrambi gli approcci paralleli risultano sempre più efficienti rispetto all'approccio sequenziale.

Come prevedibile, il tempo richiesto per la versione sequenziale aumenta linearmente con lo scorrere della posizione delle parole verso il fondo del dizionario.

In CUDA, indipendentemente dalla posizione nel dizionario della parola scelta, il tempo di esecuzione rimane costante (≈ 0.7 secondi).

Con OpenMP, invece, il tempo di esecuzione presenta delle oscillazioni, riuscendo in alcuni casi particolari ad ottenere tempi di esecuzioni inferiori rispetto a CUDA. Come detto prima, questi casi sono del tutto fortuiti in quanto dipendono dalla posizione della parola nel *chunk* di OpenMP.

4. Conclusione

Come era possibile aspettarsi il migliore approccio per un *attacco con dizionario* risulta essere quello parallelo.

Il problema principale delle versioni parallele potrebbe essere quindi quello della ricerca del numero ottimale dei thread affinché si ottenga lo *speedup* migliore.

Da una parte OpenMP risulta sicuramente più facile da implementare rispetto a CUDA, in quanto la parallelizzazione avviene in maniera del tutto automatica (anche se con poca flessibilità sulla scelta del modo in cui vengono divisi i dati).

Dall'altra parte l'utilizzo di CUDA risulta più macchinoso, richiede molte più attenzioni e soprattutto risorse (una o più GPU di calcolo), ma offre un tempo di esecuzione sempre costante e fino a sette volte più performante rispetto alla versione sequenziale.

Bibliografia

[1] "Wikipedia". Data encryption standard. https://it.wikipedia.org/wiki/Data_Encryption_Standard.

[2] "Daniel Miessler". Seclists. https://github.com/danielmiessler/SecLists/tree/master/Passwords.

[3] "Maciej Grzeszczak". Des-cuda. https://github.com/mgrzeszczak/des-cuda.

[4] "GNU C Library Manual". Encrypting passwords. https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_node/libc_650.html.

[5] "Steve Rennich". Cuda c/c++ streams and concurrency. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.