Documentazione

NOTA TECNICA
Per una corretta compilazione del programma abilitare il flag “separate compilation” nella sezione build > settings del progetto.

REPORT
Redazione di un report sintetico su problema affrontato e strategie implementative.

Introduzione
Il problema affrontato riguardava l'implementazione di un algoritmo parallelo che permettesse di generare password sicure. È stato scelto **PBKDF2** (Password-Based Key Derivation Function 2), algoritmo utilizzato per la generazione di derived keys crittograficamente sicure e resistenti ad attacchi quali dictionary attack o brute force attack.
Il problema nell'utilizzo delle password così come vengono scelte dagli utenti come chiavi crittografiche risiede nel fatto che le stringhe inserite spesso hanno poca entropia (lunghezza ridotta, poco utilizzo di numeri o caratteri speciali ecc) e non risultano quindi adatte per essere utilizzate direttamente. Quello che si può fare però è partire da una di queste password e tramite un algoritmo PBKDF2 generare una derived key sicura e robusta da poter utilizzare.

Funzionamento del algoritmo
Viene applicata una pseudo-random function alla password alla quale è stato concatenato un sale (sequenza di bit randomica per aumentare l'entropia del input) e viene ripetuto questo processo molte volte; da letteratura almeno 1000 iterazioni.
Gli elementi da tenere in considerazione sono la **password** da utilizzare come base di partenza, la **lunghezza della derived key** (DK) che da costruire e quante **iterazioni** far compiere al nostro algoritmo.
Importante la scelta della **pseudo-random function** (PRF) con cui andremo a fare i calcoli degli hash. La funzione che si va a scegliere deve essere computazionalmente lenta, in quanto questa caratteristica rende estremamente difficoltoso per l'attaccante effettuare attacchi di forza bruta in quanto impiegherebbero troppo tempo per computare. Nel nostro caso è stata scelta *hmac_sha1*.

Breve descrizione del algoritmo
La derived key viene generata come segue:
DK = PBKDF2(PRF, Password, Salt, c, dkLen)
PRF: funzione di hashing che andremo ad utilizzare (noi abbiamo usato *hmac_sha1)*.
Password: stringa in input con poca entropia dalla quale partiamo per generare una chiave crittograficamente sicura.
Salt: è una sequenza di Byte randomica non necessariamente segreta utilizzata per aggiungere entropia alla password.
c: numero di iterazioni che andremo a fare (come detto sopra almeno 1000).
dkLen: lunghezza in Byte della chiave derivata che vogliamo generare

Una derived key viene composta da tanti blocchi di Byte che vengono **computati singolarmente** e concatenati fra loro. La dimensione di questi blocchi è data dalla lunghezza del output della PRF scelta.
DK = T1 || T2 || ... || Tdklen/hlen
Se vogliamo una DK lunga 64 Byte e la PRF genera un output di 20 Byte avremo 4 blocchi distinti

T_1, T_2, T_3, T_4 di 20 Byte l'uno che andremo a concatenare per formare la sequenza finale. Di questa sequenza prenderemo i primi 64 Byte su 80 totali generati in quanto la lunghezza richiesta della DK era di 64 Byte.
Ogni singolo blocchetto viene calcolato separatamente:

```
Ti = U1 xor U2 xor … xor Uc
```

Dove:

```
U1 = PRF(Password, Salt || i);      //Questo è il punto in cui viene utilizzato il sale per aggiungere entropia
U2 = PRF(Password, U1);             //Il valore computato al passo precedente viene usato al posto del sale nel passo corrente
U3 = PRF(Password, U2);
. . .
Uc = PRF(Password, Uc-1);
```

Soluzioni implementate

Per rendere questa problematica più interessante dal punto di vista della parallelizzazione dei task abbiamo deciso di non limitarci a generare una singola derived key ma di generare un numero arbitrario di derived key scelto dall'utente. In questo modo è stato possibile ragionare sia su come parallelizzare il calcolo dei singoli blocchi T_i all'interno di una DK sia su come parallelizzare la generazione di più derived keys.
In primo luogo sono state implementate delle soluzioni naive e solo in un momento successivo si è pensato ad una soluzione più efficiente andando a ragionare sulla natura del problema per migliorarne l'efficienza.

Presentiamo le soluzioni implementate, le considerazioni sulle prestazioni verranno fatte nel paragrafo Profiling.

Soluzione 1: chiamata alla funzione execution1(...)
L'idea è stata quella di lanciare tanti kernel quante sono le DK che l'utente vuole generare. Ogni kernel si occuperà di calcolare una ed una sola DK. Il numero di thread che ogni kernel necessita dipenderà dalla dkLen (lunghezza in Byte della chiave derivata).

Esempio con numeri bassi
DK richieste: 256
Lunghezza DK: 128 Byte
Lunghezza output PRF: 16 Byte
Vengono lanciati 256 kernel, ognuno dei quali utilizza 8 (128/16 = 8) thread per portare a termine il lavoro. Tutti gli altri thread eventualmente lanciati saranno idle; ogni thread non idle si occuperà di generare uno degli 8 blocchetti T_i lunghi 16 Byte in questo caso.

Soluzione 2: chiamata alla funzione execution2(...)
Stesso identico approccio della soluzione 1 ma gli N kernel vengono lanciati su N stream differenti. Il pensiero è stato sfruttare il grado di parallelizzazione che gli stream offrono fra la esecuzione del kernel e i trasferimenti di memoria.

Soluzione 3: chiamata alla funzione execution3(...)
Da una analisi più approfondito di quale era il problema da ingegnerizzare è stato notato che tutti i thread indipendentemente dal kernel (o stream) di appartenenza svolgono lo stesso lavoro ovvero calcolare un singolo blocco di Byte T_i iterando tante volte quante richieste dall'utente. Si è quindi compresa l'inutilità di lanciare tanti kernel quante erano le chiavi da produrre e si è deciso di concentrare il lavoro in un unico kernel. Il numero di thread necessari per portare a termine il lavoro sarà dato dal numero di blocchetti T_i necessari per generare tutte le chiavi richieste.

Esempio banale con numeri bassi
DK richieste: 256
Lunghezza DK: 128 Byte
Lunghezza output PRF: 16 Byte
Bytes totali: 256 * 128 = 32768
Blocchetti T_i da generare: 32768 / 16 = 2048
Verranno serviranno 2048 thread al singolo kernel per completare il lavoro. Come già detto ogni thread calcolerà un blocchetto T_i.

Soluzione 4: chiamata alla funzione execution4(...)
Stesso approccio della soluzione 3 ma provando a suddividere il lavoro totale su un numero limitato di stream per vedere se si potesse avere una qualche sorta di guadagno in termini di efficienza.

CODIFICA
La scrittura di un codice chiaro e ben documentato

CUDA APPLICATION

Autori: Tagliabue Luca, Marco Predari Luglio 2017

TEST

Una fase di test in cui si mostra la correttezza del codice su istanze benchmark significative

Prima di parlare di correttezza descriviamo la logica con cui viene costruito il sale nel nostro algoritmo. Nelle 4 soluzioni parallele implementate la lunghezza del sale è di 20Byte e viene calcolato come segue:

Stringa “Salt”	Random float	DK_LEN (long)	**Totale**
4Byte	4Byte	8Byte	16Byte su 20

Nelle caso sequenziale il sale ha sempre lunghezza 20Byte ed è composto da:

Stringa “Salt”	Random int	DK_LEN (long)	**Totale**
4Byte	4Byte	8Byte	16Byte su 20

I restanti 4Byte del sale sono occupati da zeri.

Per la generazione del random float nel caso parallelo abbiamo utilizzato la libreria cuRand di Nvidia, mentre nel caso sequenziale per la generazione del random int abbiamo utilizzato la funzione rand di C. Nell'utilizzo della cuRand di Nvidia due dei valori che entrano in gioco sono il seed e il sequence number. Come seed abbiamo utilizzato l'id del thread all'interno della griglia (idx) mentre come sequence number il kernel id (valore proveniente dall'host). All'interno del singolo thread in computazioni distinte il valore del sale varia solamente in base alla lunghezza della source key in input (in quanto il primo thread del primo kernel avrà sempre idx = 0 e kernel_id = 0). Quindi esecuzioni differenti che richiedono chiavi della stessa lunghezza e stesso numero di iterazioni generano valori di sale uguali e di conseguenza derived keys uguali. Questa scelta è voluta in quanto facilità l'analisi e il debug dell'applicazione.
Se ci fosse la necessità di generare chiavi distinte per quanto riguarda esecuzioni differenti con gli stessi argomenti in ingresso sarebbe sufficiente:
Sequenziale: all'inizio dell'algoritmo inizializzare il seed della srand.
Parallelo: sostituire il valore del kernel id con un valore generato lato host dalla funzione srand.

È stata analizzata la correttezza delle funzioni *hmac_sha1* utilizzate nel nostro algoritmo. Preso il sale (calcolato come descritto precedentemente) e la source key scelta dall'utente, è stato verificato che i valori prodotti dalle PRF *hmac_sha1* da noi utilizzate corrispondessero a quelli generati dalla funzione equivalente della libreria OpenSSl:
echo -n <source_key> | openssl dgst -sha1 -hmac <salt>
Per quanto riguarda la funzione hmac_sha1 utilizzata lato kernel, è stato tenuto come rifermento il paper di ricerca “Using CUDA for Exhaustive Password Recovery” di “Tore K. Frederiksen.”
http://daimi.au.dk/~jot2re/cuda/
Per quanta riguarda la funzione hmac_sha1 utilizzata lato host, è stato tenuto come riferimento l'algoritmo implementato dalla libreria openource “freeradius” di Apple.
https://opensource.apple.com/source/freeradius/freeradius-11/

Per dimostrare la reale robustezza e correttezza del intero algoritmo avremmo avuto bisogno di un attaccante che partendo dalla chiave derivata tentasse di risalire alla source key in input. Lato nostro, ricordando la scelta voluta di non generare un sale completamente randomico, possiamo solo dimostrare la corretta generazione delle chiavi verificando che con lo stesso sale, con la stessa source key e con lo stesso numero di iterazioni otteniamo gli stessi output in seguito ad esecuzioni distinte. Possiamo quindi affermare che il processo implementato è deterministico.

Nella directory test sono presenti dei report ridotti utili per farsi un'idea di come si presenta una chiave derivata generata.

PROFILING

Analisi delle prestazioni (speedup e profiling in genere) ricavate dall'algoritmo parallelo rispetto a quello sequenziale

Spiegazione argomenti in ingresso all'eseguibile

./EXE_NAME <Bx> <source_key> <iterations> <len_derived_keys> <num_derived_keys> <PRINT_KEY> <INFO> <SLOW_EXECUTION>

<Bx>: thread x blocco, deve essere una potenza di due
<source_key>: password da cui partiamo a generare le DK crittograficamente più robusta
<iterations>: numero di iterazioni che andremo a computare
<len_derived_keys>: lunghezza delle DK, deve essere una potenza di due
<num_derived_keys>: numero di DK richieste, deve essere una potenza di due
<PRINT_KEY>: flag booleano, se settato vengono stampate tutte le chiavi generate
<INFO>: flag boolean, se settato rende l'esecuzione più verbosa
<SLOW_EXECUTION>: flag boolean, se non settato esegue solamente la soluzione 3 e la soluzione sequenziale ignorando le altre.

Introduzione

Le esecuzioni che presenteremo in questa analisi sono sia esecuzioni sensate dove viene generato un numero consistente di chiavi con una lunghezza richiesta ragionevole, sia esecuzioni dove la lunghezza delle derived key è molto molto elevata in modo da evidenziare come l'approccio parallelo e quello sequenziale differiscono al variare di quel parametro. Nel dettaglio verranno analizzati due casi d'uso.
Precisazione: la PRF da noi utilizzata è hmac_sha1 che genera un output di 20 Byte.

Analisi delle soluzioni parallele

Premessa: Le considerazioni che seguono fanno riferimento ad esecuzioni che richiedono il calcolo di almeno 2048 Byte in totale, per esempio almeno 32 chiavi da 64 Byte oppure almeno 64 chiavi da 32 Byte, oppure una qualsiasi altra configurazione per la quale *numero_dk x lunghezza_dk > 2048*. Per parametri minori, considerando 1000 iterazioni (numero consigliato dalla letteratura sul argomento), il tempo della computazione sequenziale è ~0,5 secondi, quindi un valore molto piccolo non interessante da trattare od ottimizzare. Inoltre con tempistiche così basse i tempi di setup di cuda dominerebbero ed il sequenziale vincerebbe sempre.
Empiricamente è stato osservato che la **soluzione 3** è sempre la soluzione di gran lunga più efficiente, per questo motivo sarà presa come riferimento per il confronto con il caso sequenziale.
Analizziamo nel dettaglio quali problemi hanno gli altri tre approcci.
Caso 1: Full report /profiling/profiling_soluzione_1.txt **esecuzione:** ./EXE_NAME 128 foo 1000 256 512 0 0 1
Caso 2: Full report /profiling/profiling_soluzione_2.txt **esecuzione:** ./EXE_NAME 128 foo 1000 65536 8 0 0 1

Soluzione 1 kernel function pbkdf2

CASO 1 Generiamo 512 chiavi lunghe 256 Byte

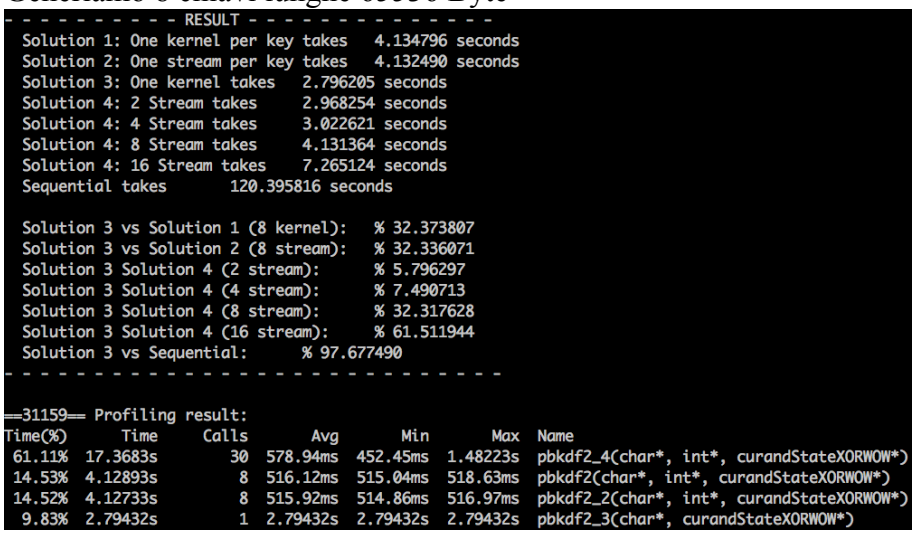
```
- - - - - - - - - - RESULT - - - - - - - - - - - - - -
  Solution 1: One kernel per key takes   219.482312 seconds
  Solution 2: One stream per key takes   219.474732 seconds
  Solution 3: One kernel takes   0.771075 seconds
  Solution 4: 2 Stream takes     1.035992 seconds
  Solution 4: 4 Stream takes     1.822700 seconds
  Solution 4: 8 Stream takes     3.615464 seconds
  Solution 4: 16 Stream takes    7.239672 seconds
  Sequential takes       30.475926 seconds

  Solution 3 vs Solution 1 (512 kernel):         % 99.648685
  Solution 3 vs Solution 2 (512 stream):         % 99.648673
  Solution 3 Solution 4 (2 stream):      % 25.571329
  Solution 3 Solution 4 (4 stream):      % 57.696000
  Solution 3 Solution 4 (8 stream):      % 78.672861
  Solution 3 Solution 4 (16 stream):     % 89.349310
  Solution 3 vs Sequential:      % 97.469888
- - - - - - - - - - - - - - - - - - - - - - - - - - - -

==29810== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 48.41%  219.442s       512  428.60ms  425.48ms  432.26ms  pbkdf2(char*, int*, curandStateXORWOW*)
 48.40%  219.428s       512  428.57ms  425.46ms  432.23ms  pbkdf2_2(char*, int*, curandStateXORWOW*)
  3.02%  13.6971s        30  456.57ms  450.35ms  516.12ms  pbkdf2_4(char*, int*, curandStateXORWOW*)
  0.17%  769.50ms         1  769.50ms  769.50ms  769.50ms  pbkdf2_3(char*, curandStateXORWOW*)
```

Vediamo dall'immagine che questa soluzione è la più lenta, anche più di quella sequenziale, a pari merito con la soluzione 2.

- Kernel : 512
- Thread x kernel: 128 (13 attivi, 115 idle)
- Totale Byte di tutte le chiavi: 131.072
- Tempo: ~219 secondi

CASO 2

Generiamo 8 chiavi lunghe 65536 Byte



- Kernel: 8
- Thread x kernel: 3328 (3.277 attivi, 51 idle)
- Totale Byte di tutte le chiavi: 524.288 (più del triplo della esecuzione precedente)
- Tempo: ~4.1 secondi. (meno di un decimo della esecuzione precedente)

Soluzione lenta in alcune casistiche ed estremamente lenta in altre (a volte anche più lenta della soluzione sequenziale). (**Caso 1)** Questo accoppiamento *1 DK → 1 kernel* comporta una sequenzializzazione del lavoro ed un overhead notevole per la gestione di tutti i kernel, soprattutto quando questi iniziano ad essere nell'ordine delle centinaia o delle migliaia. Il calcolo per la generazione della seconda DK può sostanzialmente iniziare solamente dopo che il calcolo per la prima è terminato.

Altro problema è lo spreco di risorse. Capita spesso di avere una grossa quantità di thread idle all'interno dei kernel. Se per esempio la richiesta è 512 chiavi lunghe 256 Byte ed il valore di thread per blocco è impostato a 128, vengono eseguiti 512 kernel ai quali vengono assegnati 128 thread di cui solamente 13 (ceil(256/20)) utilizzati effettivamente.

(**Caso 2**) Questa soluzione ha una buona efficienza quando vengono istanziati pochi kernel con tantissimi thread che lavorano parallelamente ed indipendentemente per calcolare i vari blocchetti T_i della chiave. Questo si verifica a fronte di richieste di generazione di poche chiavi estremamente lunghe.

Soluzione 2 kernel function pbkdf2_2

Aver spostato la computazione su più stream non nulli non ha portato nessun tipo di vantaggio. I tempi di esecuzione non si discostano dalla soluzione 1 indipendentemente dai parametri di input. Aver sovrapposto i trasferimenti di memoria alla esecuzione dei kernel non ha avuto impatto sulle prestazioni in quanto anche a fronte di richieste di chiavi molto lunghe, 65536 Byte, il trasferimento di memoria impiega un tempo irrisorio.

Un possibile guadagno concreto rispetto alla soluzione 1 ci sarebbe stato se l'hardware a disposizione avesse avuto la tecnologia Hyper Q, ovvero la possibilità di usufruire di 32 code di lavoro al posto della singola coda a nostra disposizione. In questo modo si sarebbero potuti far lavorare concorrentemente fino a 32 kernel per volta.

Soluzione 4 kernel function pbkdf2_4

CASO 1 Generiamo 512 chiavi lunghe 256 Byte

```
- - - - - - - - - - RESULT - - - - - - - - - - - - - -
  Solution 1: One kernel per key takes   219.482312 seconds
  Solution 2: One stream per key takes   219.474732 seconds
  Solution 3: One kernel takes   0.771075 seconds
  Solution 4: 2 Stream takes     1.035992 seconds
  Solution 4: 4 Stream takes     1.822700 seconds
  Solution 4: 8 Stream takes     3.615464 seconds
  Solution 4: 16 Stream takes    7.239672 seconds
  Sequential takes       30.475926 seconds

  Solution 3 vs Solution 1 (512 kernel):         % 99.648685
  Solution 3 vs Solution 2 (512 stream):         % 99.648673
  Solution 3 Solution 4 (2 stream):      % 25.571329
  Solution 3 Solution 4 (4 stream):      % 57.696000
  Solution 3 Solution 4 (8 stream):      % 78.672861
  Solution 3 Solution 4 (16 stream):     % 89.349310
  Solution 3 vs Sequential:      % 97.469888
- - - - - - - - - - - - - - - - - - - - - - - - - - - -

==29810== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 48.41%  219.442s       512  428.60ms  425.48ms  432.26ms  pbkdf2(char*, int*, curandStateXORWOW*)
 48.40%  219.428s       512  428.57ms  425.46ms  432.23ms  pbkdf2_2(char*, int*, curandStateXORWOW*)
  3.02%  13.6971s        30  456.57ms  450.35ms  516.12ms  pbkdf2_4(char*, int*, curandStateXORWOW*)
  0.17%  769.50ms         1  769.50ms  769.50ms  769.50ms  pbkdf2_3(char*, curandStateXORWOW*)
```

- Stream : 2_1, 4_2, 8_3, 16_4
- Thread per stream:
  - 3328 (3277 attivi, 51 idle)$_1$
  - 1664 (1639 attivi, 25 idle)$_2$
  - 896 (820 attivi, 76 idle)$_3$
  - 512 (410 attivi, 102 idle)$_4$
- Totale Byte di tutte le chiavi: 131.072
- Tempo: ~1.0$_1$, ~1.8$_2$, ~3.6$_3$, ~7.2$_4$ secondi

CASO 2 Generiamo 8 chiavi lunghe 65536 Byte

```
- - - - - - - - - - RESULT - - - - - - - - - - - - - -
  Solution 1: One kernel per key takes   4.134796 seconds
  Solution 2: One stream per key takes   4.132490 seconds
  Solution 3: One kernel takes   2.796205 seconds
  Solution 4: 2 Stream takes     2.968254 seconds
  Solution 4: 4 Stream takes     3.022621 seconds
  Solution 4: 8 Stream takes     4.131364 seconds
  Solution 4: 16 Stream takes    7.265124 seconds
  Sequential takes       120.395816 seconds

  Solution 3 vs Solution 1 (8 kernel):   % 32.373807
  Solution 3 vs Solution 2 (8 stream):   % 32.336071
  Solution 3 Solution 4 (2 stream):      % 5.796297
  Solution 3 Solution 4 (4 stream):      % 7.490713
  Solution 3 Solution 4 (8 stream):      % 32.317628
  Solution 3 Solution 4 (16 stream):     % 61.511944
  Solution 3 vs Sequential:      % 97.677490
- - - - - - - - - - - - - - - - - - - - - - - - - - - -

==31159== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 61.11%  17.3683s        30  578.94ms  452.45ms  1.48223s  pbkdf2_4(char*, int*, curandStateXORWOW*)
 14.53%  4.12893s         8  516.12ms  515.04ms  518.63ms  pbkdf2(char*, int*, curandStateXORWOW*)
 14.52%  4.12733s         8  515.92ms  514.86ms  516.97ms  pbkdf2_2(char*, int*, curandStateXORWOW*)
  9.83%  2.79432s         1  2.79432s  2.79432s  2.79432s  pbkdf2_3(char*, curandStateXORWOW*)
```

- Stream : 2_1, 4_2, 8_3, 16_4
- Thread per stream:
  - 13184 (13108 attivi, 76 idle)$_1$,
  - 6655 (6554 attivi, 102 idle)$_2$,
  - 3328 (3277 attivi, 51 idle)$_3$,
  - 1664 (1639 attivi, 25 idle)$_4$
- Totale Byte di tutte le chiavi: 524.288
- Tempo: ~2.9_1, ~3.0_2, ~4.1_3, ~7.2_4 secondi

Si è deciso di suddividere il lavoro prima fra 2, poi 4, poi 8 e poi 16 stream. L'idea è avendo trovato una soluzione efficiente, la numero 3 che verrà discussa nel ultimo paragrafo insieme a quella sequenziale, che lavora con un unico kernel, vedere se splittando il lavoro su pochi stream questo potesse essere d'aiuto a migliorare ulteriormente le performance. Avendo come sopra descritto un'unica coda di lavoro questa operazione non porta nessun tipo di guadagno, anzi rispetto alla soluzione 3 che usa un unico kernel introduce un gradiente di sequenzializzazione. (**Caso 1**)Da notare però come questo approccio lavori meglio delle prime due soluzioni quando il numero di chiavi è alto in quanto vengono comunque lanciati un numero limitato di kernel ognuno con tanti thread che lavora in parallelo indipendentemente.

Soluzione 3 e considerazioni rispetto al sequenziale

CASO 1 Generiamo 512 chiavi lunghe 256 Byte

```
- - - - - - - - - - RESULT - - - - - - - - - - - - - -
  Solution 1: One kernel per key takes   219.482312 seconds
  Solution 2: One stream per key takes   219.474732 seconds
  Solution 3: One kernel takes   0.771075 seconds
  Solution 4: 2 Stream takes     1.035992 seconds
  Solution 4: 4 Stream takes     1.822700 seconds
  Solution 4: 8 Stream takes     3.615464 seconds
  Solution 4: 16 Stream takes    7.239672 seconds
  Sequential takes       30.475926 seconds

  Solution 3 vs Solution 1 (512 kernel):         % 99.648685
  Solution 3 vs Solution 2 (512 stream):         % 99.648673
  Solution 3 Solution 4 (2 stream):      % 25.571329
  Solution 3 Solution 4 (4 stream):      % 57.696000
  Solution 3 Solution 4 (8 stream):      % 78.672861
  Solution 3 Solution 4 (16 stream):     % 89.349310
  Solution 3 vs Sequential:      % 97.469888
- - - - - - - - - - - - - - - - - - - - - - - - - - - -

==29810== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 48.41%  219.442s       512  428.60ms  425.48ms  432.26ms  pbkdf2(char*, int*, curandStateXORWOW*)
 48.40%  219.428s       512  428.57ms  425.46ms  432.23ms  pbkdf2_2(char*, int*, curandStateXORWOW*)
  3.02%  13.6971s        30  456.57ms  450.35ms  516.12ms  pbkdf2_4(char*, int*, curandStateXORWOW*)
  0.17%  769.50ms         1  769.50ms  769.50ms  769.50ms  pbkdf2_3(char*, curandStateXORWOW*)
```

- Kernel: 1
- Thread per stream: 6656 (6554 attivi, 102 idle)
- Totale Byte di tutte le chiavi: 131.072
- Tempo: ~0,7secondi

CASO 2
Generiamo 8 chiavi lunghe 65536 Byte

```
- - - - - - - - - - - RESULT - - - - - - - - - - - - - - -
  Solution 1: One kernel per key takes   4.134796 seconds
  Solution 2: One stream per key takes   4.132490 seconds
  Solution 3: One kernel takes   2.796205 seconds
  Solution 4: 2 Stream takes     2.968254 seconds
  Solution 4: 4 Stream takes     3.022621 seconds
  Solution 4: 8 Stream takes     4.131364 seconds
  Solution 4: 16 Stream takes    7.265124 seconds
  Sequential takes       120.395816 seconds

  Solution 3 vs Solution 1 (8 kernel):   % 32.373807
  Solution 3 vs Solution 2 (8 stream):   % 32.336071
  Solution 3 Solution 4 (2 stream):      % 5.796297
  Solution 3 Solution 4 (4 stream):      % 7.490713
  Solution 3 Solution 4 (8 stream):      % 32.317628
  Solution 3 Solution 4 (16 stream):     % 61.511944
  Solution 3 vs Sequential:      % 97.677490
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -

==31159== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 61.11%  17.3683s        30  578.94ms  452.45ms  1.48223s  pbkdf2_4(char*, int*, curandStateXORWOW*)
 14.53%  4.12893s         8  516.12ms  515.04ms  518.63ms  pbkdf2(char*, int*, curandStateXORWOW*)
 14.52%  4.12733s         8  515.92ms  514.86ms  516.97ms  pbkdf2_2(char*, int*, curandStateXORWOW*)
  9.83%  2.79432s         1  2.79432s  2.79432s  2.79432s  pbkdf2_3(char*, curandStateXORWOW*)
```

- Kernel: 1
- Thread per stream: 26240 (26215 attivi, 25 idle)
- Totale Byte di tutte le chiavi: 524.288
- Tempo: ~2.8secondi

Questa soluzione calcola semplicemente quanti byte in totale devono essere generati tenendo conto che vanno prodotte un certo numero di DK con una certa lunghezza e lancia un kernel nel quale ogni thread si occupa di calcolare 20 di questi Byte totali (ricordiamo che il numero 20 è dato dalla dimensione del output della PRF che abbiamo utilizzato). Lato host viene fatta la suddivisione di tutto quello che è l'informazione contenuta nella global memory al fine di salvare correttamente i valori delle chiavi generate nelle variabili di output.
La soluzione sequenziale è composta da tre cicli innestati. Partendo dall'esterno il primo ciclo itera sul numero di chiavi da generare. Per ciascuna chiave si itera sul numero di blocchi in cui è divisa. Infine per ciascun blocco si calcola il sale (come spiegato nel paragrafo testing) per poi applicare la PRF tante volte quante sono le iterazioni scelte. Il totale di iterazioni di questo algoritmo è dato da :
num_derived_keys * ceil(dkLen/PRF_len) * iterations
Sia nei due use case mostrati, sia nella maggior parte delle computazioni non banali, il guadagno di tempo rispetto al caso sequenziale è del ~97%.