Progetto di Piattaforme Software per la Rete Mod. 2

di Marco Fumagalli

Implementazione e parallelizzazione dell'AES su GPU
con linguaggio OpenCL

1. Algoritmo AES

Il progetto si basa sull'implementazione, in ambiente Linux, dell'algoritmo dell'**Advanced Encryption Standard** (**AES**), conosciuto anche come **Rijndael**, di cui più propriamente ne è una specifica implementazione modificata; è un algoritmo di cifratura a blocchi utilizzato come standard dal governo Americano.
Infatti, a differenza dell'algoritmo originario, i blocchi cifrati sono di dimensione fissa (128 bit) ed utilizzano tre tipi di chiavi (128, 192, 256 bit).
AES opera utilizzando matrici di 4×4 byte chiamate stati (*states*). Quando l'algoritmo ha blocchi di 128 bit in input, la matrice State ha 4 righe e 4 colonne; se il numero di blocchi in input diventa di 32 bit più lungo, viene aggiunta una colonna allo State, e così via fino a 256 bit. In pratica, si divide il numero di bit del blocco in input per 32 e il quoziente specifica il numero di colonne.
C'è un passaggio iniziale:

1. **AddRoundKey** – Ogni byte della tabella viene combinato con la chiave di sessione, la chiave di sessione viene calcolata dal gestore di chiavi.

Successivamente per cifrare sono previsti diversi round o cicli di processamento: ogni round (fase) dell'AES (eccetto l'ultimo) consiste dei seguenti quattro passaggi:

1. **SubBytes** – Sostituzione non lineare di tutti i byte che vengono rimpiazzati secondo una specifica tabella.
2. **ShiftRows** – Spostamento dei byte di un certo numero di posizioni dipendente dalla riga di appartenenza
3. **MixColumns** – Combinazione dei byte con un'operazione lineare, i byte vengono trattati una colonna per volta.
4. **AddRoundKey** – Ogni byte della tabella viene combinato con la chiave di sessione; la chiave di sessione viene calcolata dal gestore delle chiavi.

Il numero di round o cicli di processamento/elaborazione crittografica dei quattro passaggi precedenti è 10, 12 o 14, in base alla lunghezza della chiave, ma sempre con l'ultimo round che salta il passaggio MixColumns.
L'implementazione di questo algoritmo è leggermente diversa, infatti i quattro passaggi vengono compattati in uno unico grazie all'utilizzo di tabelle di ricerca che riescono ad unire tutte le operazioni sopra elencate.

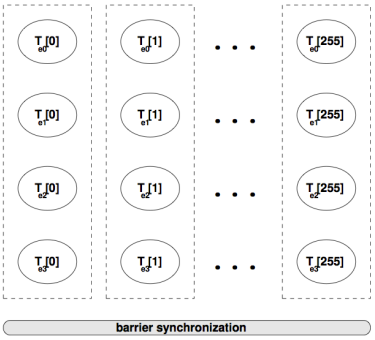
$$T_0[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \end{bmatrix} \quad T_1[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \\ S[a_{i,j}] \end{bmatrix}$$

$$T_2[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \\ S[a_{i,j}] \end{bmatrix} \quad T_3[a_{i,j}] = \begin{bmatrix} S[a_{i,j}] \\ S[a_{i,j}] \\ S[a_{i,j}] \bullet 03 \\ S[a_{i,j}] \bullet 02 \end{bmatrix}$$

Questo permette di suddividere gli stati in quattro, eseguire le operazioni in parallelo ed infine riunire il tutto per ottenere i dati criptati.

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}] \oplus k_j$$

Per far sì che questo accada in tempi brevi, viene utilizzata la potenza della GPU attraverso OpenCL.



2. OpenCL

OpenCL è una libreria basata sul linguaggio C che può essere utilizzata soprattutto sulla GPU per effettuare operazioni in parallelo.
Essa può essere catalogata nell'ambito del **GPGPU** (**G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits).
Lo scopo di questi studi e quindi della creazione di questa libreria, è la possibilità di sfruttare le potenzialità di calcolo che la GPU, non solo per realizzare immagini tridimensionali ma anche per altri calcoli non inerenti alla grafica, come nel mio caso la criptografia.
L'utilizzo di OpenCL consiste nella creazione di due tipologie di file: un host file ed un kernel file.

2.1 Host File

Il primo file (host) in c++, viene eseguito dall'host (CPU) che inizializza le variabili, costruisce il contesto e l'ambiente per il device (GPU).
Come prima cosa bisogna preparare la piattaforma per lanciare il programma sul device, poi recuperare il device da utilizzare ed infine creare il contesto di comunicazione e la coda di comandi.
Tutto questo avviene attraverso i comandi seguenti:

- `clGetPlatformIDs();`
- `clGetDeviceIDs();`
- `clCreateContext();`
- `clCreateCommandQueue();`

Teoricamente, i dati da criptare vengono recuperati da file, convertiti nel formato adeguato e messi a disposizione al device attraverso dei buffer; ne segue un esempio:

- `clCreateBuffer();`

Successivamente viene creato e costruito il programma che verrà poi eseguito sul device, con i seguenti comandi:

- `clCreateProgramWithSource();`
- `clBuildProgram();`

Direttamente dall'host c'è la possibilità di creare il "kernel" che viene eseguito dal programma partendo da una funzione che può essere specificata via codice con la parola chiave __kernel, come se fosse il main:

- `clCreateKernel();`

Infatti, successivamente vengono passati uno ad uno i paramenti definiti in questa funzioni.
Dato che la GPU può essere suddivisa in vari gruppi, ognuno dei quali può avere diversi thread, come ultima operazione di inizializzazione vengono definiti quanti gruppi e thread devono essere lanciati in parallelo:

- `clGetKernelWorkGroupInfo();`
- `clEnqueueNDRangeKernel();`

Infine, dopo che il programma ha terminato l'eseguzione, i dati sono recuperati dal device, sempre attraverso i buffer.

- `clEnqueueReadBuffer();`

2.2 Kernel File

Il secondo file (kernel) è di formato .cl, viene trasformato in un programma autonomo e compilato a runtime direttamente dall'host ed eseguito sul device in parallelo in diversi thread in cui l'unica differenza tra loro sono i dati da elaborare.
Inizialmente viene calcolata la chiave iniziale con una funzione ad hoc che implementa il primo passaggio AddRoundKey, e successivamente si inizia con la parallellizzazione.

- `aes_set_key(&ctx, key, size_key_d);`

Ad ogni thread, in base al proprio id, è assegnata una colonna della matrice di stato, nella quale sono stati trasformati nel corretto formato ed inseriti i dati in input.
Una volta settato tutto a dovere si può iniziare ad effettuare la cifratura vera e propria.
Per ogni round, i quattro thread criptano la propria parte di dati in parallelo, si sincronizzano e riuniscono i dati che poi saranno utilizzati nel round successivo.

```
for(int i=0; i<ctx->nr; i++)
{
	if(idx < 4)
	{
		switch(idx)
		{
			case 0:
				aes_fround(Y0, X0, X1, X2, X3);
				break;
			case 1:
				aes_fround(Y1, X0, X1, X2, X3);
				break;
			case 2:
				aes_fround(Y2, X0, X1, X2, X3);
				break;
			case 3:
				aes_fround(Y3, X0, X1, X2, X3);
				break;
		}

		barrier(CLK_GLOBAL_MEM_FENCE);
		X0 = Y0;
		X1 = Y1;
		X2 = Y2;
		X3 = Y3;

	}
}
```

Il numero dei round dipende direttamente della lunghezza della chiave e possono essere 10, 12 o 14 cicli.
Per quanto riguarda l'ultimo round, viene eseguito sempre in parallelo ma all'esterno del ciclo perchè segue un processo differente.

```
switch(idx)
{
        case 0:
                X0 = RK[0] ^ ( FSb[ (uint8) ( Y0 >> 24 ) ] << 24 ) ^
                ( FSb[ (uint8) ( Y1 >> 16 ) ] << 16 ) ^
                ( FSb[ (uint8) ( Y2 >>  8 ) ] <<  8 ) ^
                ( FSb[ (uint8) ( Y3       ) ]       );
                PUT_UINT32( X0, output,  0 );
                break;
        case 1:
                X1 = RK[1] ^ ( FSb[ (uint8) ( Y1 >> 24 ) ] << 24 ) ^
                ( FSb[ (uint8) ( Y2 >> 16 ) ] << 16 ) ^
                ( FSb[ (uint8) ( Y3 >>  8 ) ] <<  8 ) ^
                ( FSb[ (uint8) ( Y0       ) ]       );
                PUT_UINT32( X1, output,  4 );
                break;
        case 2:
                X2 = RK[2] ^ ( FSb[ (uint8) ( Y2 >> 24 ) ] << 24 ) ^
                ( FSb[ (uint8) ( Y3 >> 16 ) ] << 16 ) ^
                ( FSb[ (uint8) ( Y0 >>  8 ) ] <<  8 ) ^
                ( FSb[ (uint8) ( Y1       ) ]       );
                PUT_UINT32( X2, output,  8 );
                break;
        case 3:
                X3 = RK[3] ^ ( FSb[ (uint8) ( Y3 >> 24 ) ] << 24 ) ^
                ( FSb[ (uint8) ( Y0 >> 16 ) ] << 16 ) ^
                ( FSb[ (uint8) ( Y1 >>  8 ) ] <<  8 ) ^
                ( FSb[ (uint8) ( Y2       ) ]       );
                PUT_UINT32( X3, output, 12 );
                break;
}

barrier(CLK_GLOBAL_MEM_FENCE);
```

In conclusione i dati elaborati e criptati vengono riconvertiti nel formato originale e restituiti all'host attraverso i buffer che sono indicati con la parola chiave __global.

3. Istruzioni per l'esecuzione

Il progetto è situato su un server fornito dal Politecnico, che lavora in ambiente Linux e possiede una GPU NVIDIA.
Il codice sorgente è situato nella directory:

```
/home/fumagalli/source/progetto/src/aesProgetto
```

Ed è composto da due file:

- aes_host_file.cpp
- aes_kernel_file.cl

Per la compilazione è stato utilizzato un Makefile fornito da NVIDIA, insieme a degli esempi, perchè ci sono state diverse complicazione con il collegamento alle librerie di OpenCL.
Per questo motivo l'eseguibile viene creato nella seguente directory:

```
/home/fumagalli/source/OpenCL/bin/linux/release
```

Una volta eseguito, il programma chiede di inserire i dati da criptare, in un formato di 16 caratteri numerici, e successivamente chiede di inserire la lunghezza della chiave con cui si vuole utilizzare l'AES e verrà utilizzata quella di default composta da tutti 0.
Finito di processare i dati il programma restituisce il risultato cifrato a video.

4. Bibliografia

1. Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, Geraldo Pelosi *Design of a Parallel AES for Graphics Hardware using the CUDA framework*
2. Joppe W. Bos, Dag Arne Osvik, Deian Stefan, *Fast Implementations of AES on Various Platforms*
3. Wikipedia, *Advanced Encryption Standard*
4. Wikipedia, ***GPGPU*** *(**G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits)*
5. NVIDIA, *OpenCL Programming Guide for the Cuda Architecture*
6. NVIDIA, *OpenCL BestPracticesGuide*
7. NVIDIA*, OpenCL_JumpStart_Guide*
8. NVIDIA*, OpenCL_Getting_Started_Linux*