Università degli Studi di Perugia

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

PAES

Analisi e sviluppo di un'implementazione parallela dell'AES
per architetture eterogenee multi/many-core

Studente

Paolo Bernardi

Relatore

Osvaldo Gervasi

Anno Accademico 2008/2009

Indice

Introduzione

PAES è un'implementazione parallela dell'Advanced Encryption Standard (AES) in grado di sfruttare sia le Central Processing Unit (CPU) che le Graphics Processing Unit (GPU) per eseguire i calcoli. Grazie allo standard Open Computing Language (OpenCL), PAES funziona in modo quasi completamente trasparente su tutte e due le tipologie di dispositivi.

Lo scopo principale di questo progetto è esplorare l'uso di OpenCL e ricavare osservazioni utili per la sua applicazione a progetti futuri. In questo senso, PAES è un contributo allo sviluppo del ben più ampio esperimento Many-core Computing for future Gravitational Observatories (MaCGO) dell'Istituto Nazionale di Fisica Nucleare (INFN). In particolare è stato scelto di implementare AES perché quest'ultimo è uno standard[1] ampiamente utilizzato; si tratta quindi un ottimo banco di prova per OpenCL e il lavoro svolto potrà essere reimpiegato in qualsiasi progetto futuro che richieda l'uso della crittografia.

L'elaborato è diviso in due sezioni: una panoramica del contesto tecnologico e scientifico in cui PAES è collocato, seguita dalla descrizione del lavoro sperimentale che è stato svolto.

Il primo capitolo capitolo è una sintesi dello stato attuale delle architetture multi/many-core. In particolare si parlerà delle più recenti evoluzioni delle CPU nonché delle GPU.

Nel secondo capitolo viene presentato OpenCL. Poiché PAES è basato su OpenCL, la presentazione comprende tutti gli aspetti principali dello standard e, dove necessario, è piuttosto approfondita.

Il terzo capitolo descrive lo standard AES; la descrizione è necessariamente approfondita per quanto concerne gli aspetti necessari alla sua implementazione.

Il quarto capitolo parla dell'analisi e della progettazione di PAES. Il problema da risolvere è molto ben definito, quindi l'accento è posto sull'architettura del software, con particolare riguardo per la struttura del parallelismo.

Il quinto capitolo è incentrato sull'implementazione di PAES; l'uso di OpenCL ha imposto la suddivisione del programma in due parti, una che prepara l'ambiente di esecuzione e l'altra che esegue effettivamente l'algoritmo AES; in particolare, quest'ultima parte del programma può anche essere eseguita su GPU.

Il sesto capitolo riassume le principali osservazioni sulla metodologia usata per lo sviluppo di PAES. Dove possibile tali osservazioni sono generalizzate al fine di renderle utili per futuri progetti basati su OpenCL, di qualsiasi tipo essi siano.

Nel settimo capitolo sono riportate alcune misure delle prestazioni di PAES, comparate con un'implementazione seriale dell'AES. Si vedrà come effettivamente l'uso delle GPU sia un'ottima alternativa per programmi altamente parallelizzabili.

Parte I

Contesto

1 Architetture multi/many-core

The increased power of the hardware [...] made solutions feasible that the programmer had not dared to dream about a few years before.

E. W. Dijkstra

Nel 1965 Gordon Moore, cofondatore della Intel Corporation, predisse che la densità dei transistor nei circuiti integrati sarebbe raddoppiata circa ogni due anni[2]. Quest'affermazione è diventata nota come "Legge di Moore" e fino ad oggi ha descritto con buona approssimazione l'evoluzione dei microprocessori (vedi figura 1.1). Tra le sue molteplici formulazioni la più conosciuta recita: "le prestazioni dei microprocessori raddoppieranno circa ogni due anni". Tuttavia la crescita del livello di integrazione dei transistor nelle CPU è destinata a fermarsi in pochi anni. Perciò, per continuare a incrementare almeno le prestazioni delle CPU, i produttori hanno deciso di adottare architetture multi-core: si tratta di singoli chip contenenti due o più processori (core).

D'altro canto, i produttori di GPU hanno adottato anch'essi da tempo delle architetture basate su molteplici unità di calcolo parallele. Infatti le operazioni più pesanti che avvengono nell'ambito della grafica tridimensionale sono di tipo Single Instruction, Multiple Data (SIMD) e un elevato numero di core

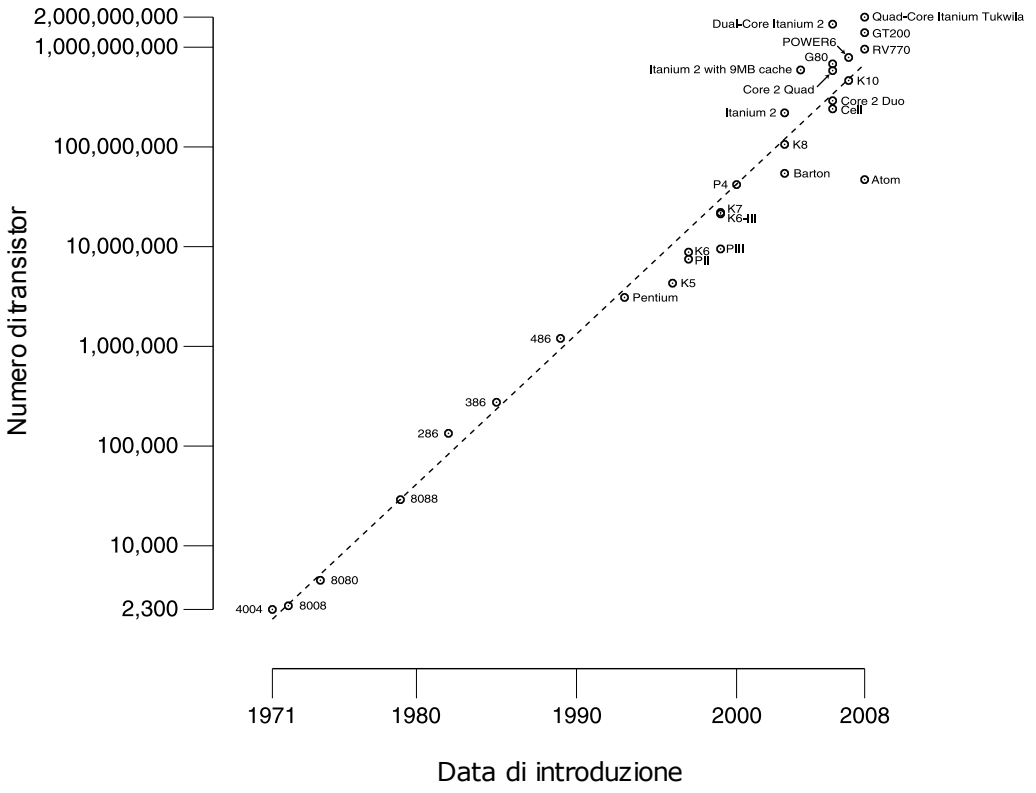

Figura 1.1: Numero di transistor per microprocessore dal 1971 al 2008 (fonte: Wikipedia).

è fondamentale per la loro velocizzazione. Poiché attualmente le GPU hanno un numero di core superiore a quello delle CPU questo tipo di architettura viene definito many-core.

Nei computer odierni sono quasi sempre presenti unità di calcolo molteplici ed eterogenee, ciascuna formata da un numero variabile di core. Lato software diventa quindi fondamentale adottare paradigmi, tecniche e strumenti di programmazione parallela che riescano a sfruttare al meglio le risorse computazionali disponibili. In effetti, secondo la Legge di Amdahl, il limite

superiore all'incremento di prestazioni di un software che sfrutta più risorse di calcolo contemporaneamente è proporzionale alla frazione dello stesso che può essere parallelizzata.

Nella pratica però la maggior parte dei programmi per scopi non scientifici non è stata scritta per fare uso di architetture parallele. Ciò avviene per diversi motivi, tra cui:

1. la programmazione parallela è molto più complessa di quella seriale;
2. gli strumenti per la programmazione parallela sono più complicati rispetto a quelli per la programmazione seriale.

Per supportare la diffusione del proprio hardware per il calcolo parallelo i produttori stanno cercando di fornire un'infrastruttura software appropriata: ad esempio la NVIDIA Corporation fornisce un Software Development Kit (SDK) per la scrittura di programmi in grado di sfruttare le sue GPU basate sulla Compute Unified Device Architecture (CUDA) per calcoli non necessariamente legati alla grafica. Questa soluzione consente di sfruttare soltanto le GPU della NVIDIA Corporation, tralasciando quelle della concorrenza e le CPU in genere; il supporto di quest'ultime è delegato ai comuni strumenti di programmazione.

La tendenza a sfruttare le GPU per calcoli non strettamente correlati ai loro compiti originari viene spesso indicata con l'espressione General-Purpose computation on Graphics Processing Units (GPGPU).

Anche l'Università degli Studi di Perugia si sta muovendo nell'ambito GPGPU. Si è recentemente costituito un gruppo di lavoro che unisce ricercatori di varie discipline che hanno deciso di effettuare un'azione sinergica al fine di creare un contesto locale che offra la possibilità di realizzare progetti importanti su architetture hardware innovative. Il gruppo di lavoro coinvolge i Dipartimenti di Fisica, Chimica e Matematica e Informatica ed è coinvolto su diversi fronti e progetti. In particolare presso il Dipartimento di Fisica e l'INFN si sta

portando avanti il progetto MaCGO, il quale si propone di sfruttare in modo ottimale i sistemi multi/many-core per calcoli nell'ambito dell'astronomia delle onde gravitazionali[1].

[1]Questa branca emergente dell'astronomia studia corpi celesti come buchi neri o stelle di neutroni grazie alla misurazione delle onde gravitazionali.

2 OpenCL

The limits of your language are the limits of your world.

L. Wittgenstein

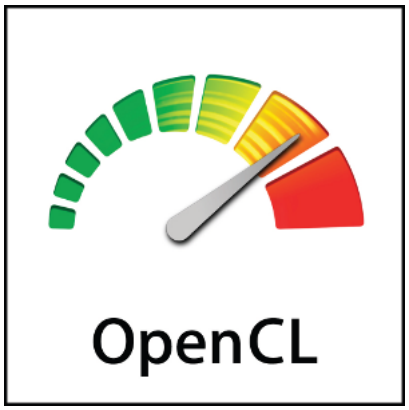

Figura 2.1: Il logo di OpenCL.

OpenCL è uno standard creato dal Kronos Group che definisce un ambiente di programmazione parallela multipiattaforma in grado di sfruttare con un'interfaccia uniforme CPU, GPU e altri dispositivi di calcolo[3]. Il Kronos Group comprende organizzazioni come AMD, NVIDIA, Intel, Apple e Los Alamos National Laboratory.

OpenCL definisce un'Application Programming Interface (API) per gestire le computazioni sui diversi dispositivi di calcolo e un linguaggio per la scrittura dei programmi che dovranno essere eseguiti sugli stessi.

Per poter comprendere e usare efficacemente OpenCL è necessario guardarlo da diversi punti di vista:

1. modello della piattaforma;
2. modello di esecuzione;
3. modello della memoria;
4. modello di programmazione.

2.1 Modello della piattaforma

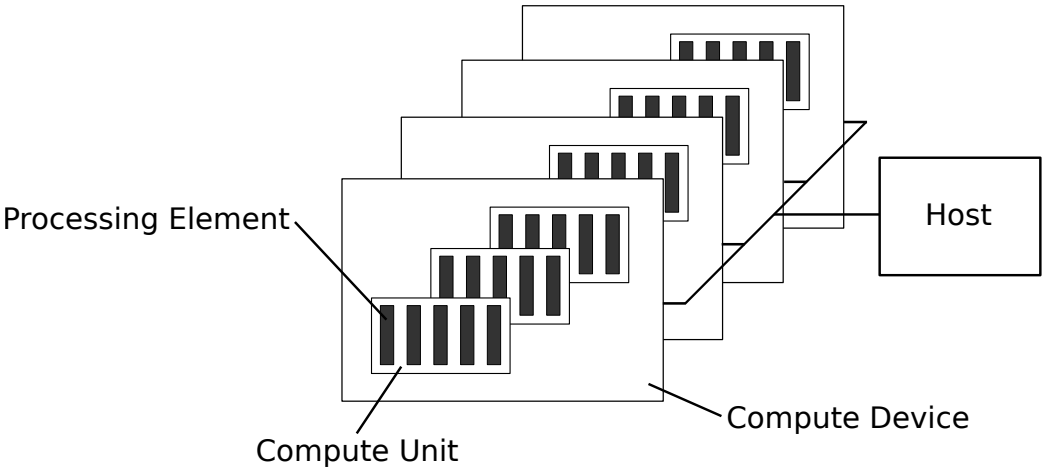

Figura 2.2: Il modello della piattaforma di OpenCL.

La figura 2.2 rappresenta il modello della piattaforma di OpenCL. In particolare, in tale modello viene definito un host (normalmente un comune computer) connesso a uno o più device (di solito CPU o GPU). Un device contiene a sua volta una o più Compute Unit (CU); una CU è ulteriormente divisa in

Processing Element (PE), ovvero i dispositivi che eseguono effettivamente i calcoli.

Il flusso d'esecuzione di un'applicazione OpenCL inizia nell'host; da qui l'applicazione invierà dei comandi affinché vengano eseguiti dei calcoli nei PE di un particolare device. I PE contenuti all'interno della medesima CU possono eseguire un singolo flusso di istruzioni secondo uno di questi modelli:

Single Instruction, Multiple Data: i PE eseguono lo stesso programma su dati diversi condividendo un medesimo program counter;

Single Program, Multiple Data: i PE eseguono lo stesso programma su dati diversi, ciascuno il proprio program counter indipendente da quello degli altri.

2.2 Modello di esecuzione

Il flusso di esecuzione di un programma basato su OpenCL è suddiviso in due parti:

1. un programma eseguito nell'host;
2. una o più funzioni eseguite dei device, dette kernel.

Il programma eseguito nell'host prepara l'esecuzione dei vari kernel, definendo i device da utilizzare, il tipo di parallelismo, i dati in input e quelli in output.

Il nodo principale del modello di esecuzione OpenCL è definito da come vengono eseguiti i kernel. Quando il programma nell'host richiede l'esecuzione di un kernel viene definito uno spazio degli indici e viene eseguita un'istanza del kernel per ogni punto di tale spazio. Ciascuna istanza viene definita con il nome di work-item ed è identificata proprio dal punto nello spazio degli indici cui è associata: questo identificatore è anche detto global ID. Ciascun work-item esegue il codice dello stesso kernel ma il flusso d'esecuzione può

non essere sincronizzato a causa di costrutti condizionali o cicli con condizioni variabili.

I work-item sono raggruppati in work-group; questi rappresentano una decomposizione a grana più grossa dello spazio degli indici. Ciascun work-group è identificato univocamente con un work-group ID e i suoi work-item hanno ciascuno un local ID, ovvero un identificatore univoco soltanto all'interno del work-group stesso.

Da quanto detto negli ultimi due paragrafi si evince che un work-item può essere identificato univocamente in due modi:

1. dal proprio global ID;
2. da una combinazione del suo local ID e dell'identificatore del work-group cui esso appartiene.

Con riferimento al modello della piattaforma, ciascun work-item di un dato work-group viene eseguito in modo concorrente in un PE di una singola CU.

Lo spazio degli indici può essere multidimensionale. La versione 1.0 della specifica OpenCL consente di avere spazi di una, due o tre dimensioni[3]; questo limite è riconducibile al compito principale delle GPU, l'elaborazione di grafica tridimensionale. Lo spazio degli indici di OpenCL viene chiamato N-Dimensional Range (NDRange) ed è implementato come vettore di lunghezza N in cui ciascuna componente specifica l'estensione della dimensione corrispondente. Ne consegue che anche i global ID, i local ID e i work-group ID sono dei vettori di N elementi. Seguendo la convenzione tipica del linguaggio C, ciascuna componente di un global ID può assumere un valore che va da 0 all'estensione della dimensione corrispondente meno uno. Lo stesso vale per local ID e work-group ID, considerando le opportune limitazioni superiori.

Un esempio bidimensionale sulla relazione tra global ID, work-group ID e local ID è visibile in figura 2.3. Sono noti lo spazio degli indici globale (G_x, G_y) e la dimensione di ciascun work-group (S_x, S_y), dai quali si possono ricavare

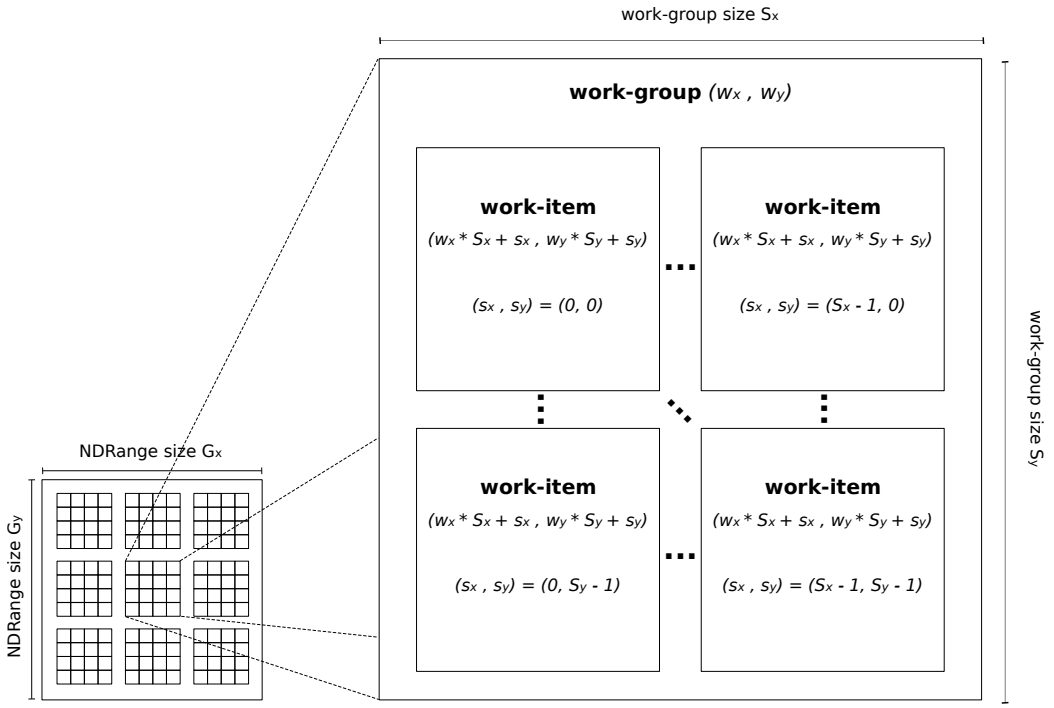

Figura 2.3: Un esempio di spazio degli indici bidimensionale; si noti la relazione tra global ID, work-group ID e local ID.

diverse informazioni. Ad esempio, il numero totale di work-item è dato dal prodotto $G_x G_y$ e il numero di work-group è pari a $\frac{G_x}{S_x}\frac{G_y}{S_y}$. Introducendo anche il local ID di un work-item, (s_x, s_y), nonché il suo work-group ID (w_x, w_y), si può anche mostrare la relazione che lega global ID, work-group ID e local ID:

$$(g_x, g_y) = (w_x S_x + s_x, w_y S_y + s_y)$$

2.2.1 Code di comandi e contesti

Un kernel OpenCL viene eseguito in un determinato contesto; sotto questa definizione sono raggruppate le seguenti risorse:

Device: come già detto, si tratta dei dispositivi fisici su cui andranno in esecuzione i kernel OpenCL;

Kernel: le funzioni OpenCL eseguite nei device;

Program Object: il codice sorgente e quello macchina dei programmi che implementano i kernel;

Memory Object: servono a condividere dati tra il programma host e i kernel in esecuzione sui device.

Grazie all'API OpenCL il programma in esecuzione nell'host deve prima definire un contesto, quindi creare una coda di comandi che può contenere le seguenti tipologie di istruzioni:

1. esecuzione di un kernel;
2. lettura e scrittura della memoria dei device;
3. sincronizzazione tra comandi in coda.

I comandi presenti nella coda vengono poi schedulati per l'esecuzione nel device prescelto. A causa di ciò l'esecuzione dei comandi da parte dell'host nei dispositivi OpenCL è generalmente asincrona, tuttavia esistono delle apposite funzioni per la sincronizzazione host/device. Ciascun device esegue i comandi inviatigli mediante le code nell'ordine in cui li riceve; esistono tuttavia due modalità di esecuzione:

1. in ordine, ovvero ciascun comando viene eseguito completamente prima che inizi il successivo;
2. fuori ordine, ovvero lo scheduler del device non attende il completamento di un comando per lanciare il successivo.

I comandi riguardanti l'esecuzione di un kernel oppure le operazioni sulla memoria possono essere associati a degli eventi; questi possono essere usati per coordinare l'esecuzione dei vari comandi o per rilevare il tempo impiegato per il loro completamento.

2.3 Modello della memoria

OpenCL definisce quattro tipi di memoria:

Global Memory: è leggibile e scrivibile da tutti i work-item di qualsiasi work-group nonché dal programma in esecuzione nell'host; potrebbe essere dotata di cache.

Constant Memory: è solo leggibile dai work-item mentre il programma in esecuzione nell'host può anche scrivervi; potrebbe essere dotata di cache[1].

Local Memory: è locale a un singolo work-group, quindi può essere usata per variabili che sono condivise da tutti i work-item dello stesso work-group. Nel caso di device di tipo GPU è solitamente una memoria dedicata, mentre per le CPU viene fatta corrispondere a una regione della Global Memory.

Private Memory: ciascun work-item ha una propria porzione di memoria privata, distinta da quella degli altri, nella quale può leggere e scrivere a piacimento.

Nella tabella 2.1 sono riassunte le modalità di accesso alle varie tipologie di memoria da parte di host e device, mentre in figura 2.4 è rappresentata graficamente la relazione tra il modello della memoria e quello della piattaforma.

Il programma in esecuzione nell'host usa la consueta API OpenCL per creare dei Memory Object che fungeranno da interfaccia verso la Global Memory e per mettere in una coda dei comandi per effettuare operazioni di lettura e scrittura su tali oggetti.

[1] Secondo quanto riportato in [4] tutte le schede video della NVIDIA Corporation hanno una Constant Memory dotata di cache.

	Global	**Constant**	**Local**	**Private**
Host	lettura scrittura	lettura scrittura	non accessibile	non accessibile
Device	lettura scrittura	lettura	lettura scrittura	lettura scrittura

Tabella 2.1: Modalità di accesso alla memoria OpenCL da parte di host e device.

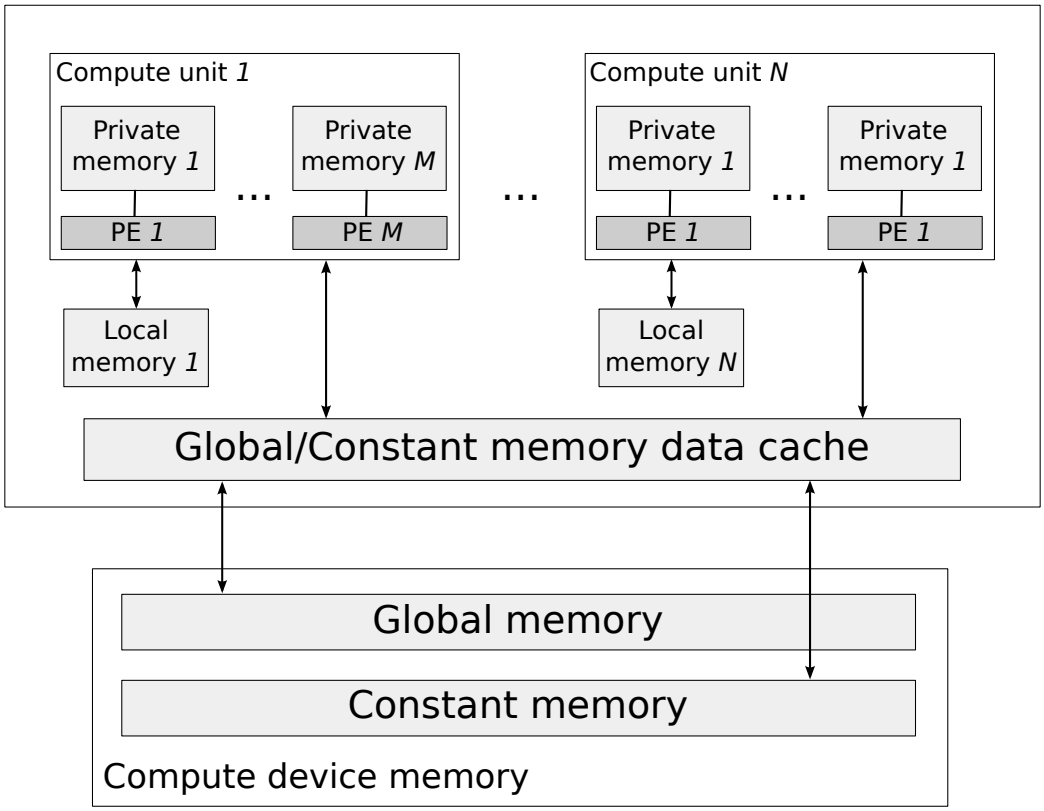

Figura 2.4: Relazione tra i modelli OpenCL della piattaforma e della memoria.

Lettura e scrittura dalla memoria dell'host a quella del device possono avvenire in due modi: copiando esplicitamente i dati o creando delle corrispondenze tra regioni di memoria dell'host e del device (in inglese questa operazione viene definita mapping).

Per copiare i dati esplicitamente il programma in esecuzione sull'host mette in coda dei comandi per il trasferimento dei dati; come già detto, questi comandi possono essere eseguiti sia in modo bloccante che in modo non bloccante.

Per finire, una nota molto importante sul modello di memoria OpenCL, riportata direttamente dalla specifica OpenCL [3]:

> *"OpenCL uses a relaxed consistency memory model; i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times"*

Le uniche garanzie che si hanno a proposito della memoria sui device sono:

1. la Private Memory è coerente rispetto alle operazioni di lettura e scrittura all'interno di un singolo work-item;
2. la Local Memory è coerente per tutti i work-item dello stesso work-group;
3. la Global Memory è coerente per tutti i work-item dello stesso work-group ma non ci sono garanzie circa la coerenza tra work-item di work-group diversi che stanno eseguendo il medesimo kernel.

Qualora il programmatore necessiti di un tipo di coerenza non fornito direttamente da OpenCL dovrà usare le funzioni di sincronizzazione per ottenere il risultato desiderato.

2.4 Modello di programmazione

OpenCL supporta tre modelli per la programmazione parallela:

1. parallelismo rispetto ai dati;
2. parallelismo rispetto ai task;
3. un ibrido dei due modelli precedenti.

Tra questi il modello maggiormente supportato è sicuramente il parallelismo rispetto ai dati.

2.4.1 Parallelismo rispetto ai dati

In questo caso la computazione viene definita in termini di istruzioni applicate a più elementi di un Memory Object. Lo spazio degli indici, definito nella sezione 2.2, viene usato per stabilire il numero di work-item e la loro corrispondenza alle diverse parti dei dati in memoria; ciascun work-item potrà poi operare parallelamente rispetto agli altri sulla sua sezione dei dati, a seconda delle unità di calcolo effettivamente disponibili nel device.

OpenCL fornisce al programmatore due modi di suddividere i dati tra i vari work-item:

1. specificando sia il numero totale di work-item che la loro suddivisione in work-group;
2. specificando soltanto il numero totale di work-item, lasciando che sia l'implementazione di OpenCL a gestire la loro ripartizione in work-group.

Come già detto nella sezione 2.1 il parallelismo rispetto ai dati può essere sia di tipo SIMD che di tipo Single Program, Multiple Data (SPMD).

2.4.2 Parallelismo rispetto ai task

In questo caso vengono eseguite diverse istanze di un kernel indipendentemente da qualsiasi spazio degli indici. Questo tipo di astrazione potrebbe essere

paragonato all'esecuzione di un kernel con un singolo work-group contenente a sua volta un unico work-item.

Per ottenere questo tipo di parallelismo vengono generalmente sfruttate le code dei comandi, inserendo in esse più istanze di kernel.

2.4.3 Sincronizzazione

OpenCL definisce due tipologie di sincronizzazione:

1. tra work-item del medesimo work-group;
2. tra comandi messi in coda in una o più code associate a un singolo contesto.

Non esiste invece alcun meccanismo di sincronizzazione tra work-group.

La sincronizzazione tra work-item dello stesso work-group viene ottenuta mediante una barriera a livello di work-group: ciascun work-item deve eseguire la barriera e attendere che tutti gli altri lo abbiano fatto a loro volta.

La sincronizzazione tra i comandi delle code può essere ottenuta in due modi:

1. mediante barriere a livello di coda, che però garantiscono il sincronismo soltanto per i comandi contenuti all'interno della medesima coda;
2. mediante l'attesa rispetto a un evento (vedi la sottosezione 2.2.1); questo metodo funziona anche tra code diverse.

2.5 Estensioni del linguaggio C per i kernel OpenCL

Per la creazione dei kernel, OpenCL definisce un linguaggio basato sullo standard ISO/IEC 9899:1999, conosciuto informalmente come C99, con diverse

estensioni per la programmazione parallela. Le seguenti sono particolarmente degne di attenzione:

1. tipi di dato vettoriali di lunghezza 2, 4, 8 o 16 che consentono di effettuare operazioni sui singoli elementi in parallelo qualora il device lo supporti;
2. tipi di dato per la rappresentazione di immagini a due o tre dimensioni;
3. controllo accurato delle operazioni di conversione tra tipi di dato;
4. conformità dei tipi di dato in virgola mobile allo standard IEEE-754;
5. modificatori aggiuntivi per variabili e argomenti di funzione al fine di specificare l'area di memoria in cui sono allocati (`__global`, `__constant`, `__local` e `__private`).

La definizione delle funzioni che implementano i kernel, la cui esecuzione sul device è evocabile dal programma sull'host, deve essere preceduta dal modificatore `__kernel`.

D'altro canto il C di OpenCL presenta anche diverse limitazioni, anche rispetto al C99, tra cui:

1. gli argomenti di tipo puntatore che compaiono in una funzione kernel devono essere preceduti dal qualificatore `__global`, `__constant` o `__local`;
2. gli argomenti di tipo puntatore di una funzione kernel non possono avere più di un livello di indirezione (nessun puntatore a puntatore, quindi);
3. un puntatore definito come `__global`, `__constant` o `__local` può essere assegnato soltanto a un altro puntatore definito con il medesimo qualificatore;
4. non è possibile usare puntatori a funzioni;

5. la maggior parte degli header della libreria standard C99 non può essere usata;
6. non sono supportati i qualificatori `extern`, `static` e `auto`;
7. non è possibile usare la ricorsione;
8. le funzioni kernel devono sempre essere di tipo `void`;
9. non è possibile scrivere in aree di memoria rappresentate da un puntatore di un tipo di dato la cui dimensione è inferiore a 32 bit[2].

2.6 Implementazioni

Lo standard OpenCL è stato implementato da diversi produttori, tra cui:

NVIDIA, per le proprie GPU[3];

AMD, sia per le proprie GPU che per le principali CPU[4];

IBM, per i suoi processori con architettura POWER[5].

Ciascuna implementazione fornisce i seguenti componenti:

1. driver per l'uso dei dispositivi di calcolo da parte del programma in esecuzione nell'host;
2. implementazione dell'API OpenCL;
3. compilatore per kernel OpenCL.

[2] In realtà è possibile fare ciò tramite delle estensioni, ma le implementazioni OpenCL non sono tenute a supportarle.

[3] http://www.nvidia.com/object/cuda_opencl.html

[4] http://ati.amd.com/technology/streamcomputing/opencl.html

[5] http://www.alphaworks.ibm.com/tech/opencl

3 Advanced Encryption Standard

There are two types of encryption: one that will prevent your sister from reading your diary and one that will prevent your government.

B. Schneier

L'AES è l'attuale standard per la crittografia simmetrica del governo statunitense poiché è stato dichiarato Federal Information Processing Standard (FIPS) dal National Institute of Standards and Technology (NIST) dopo un approfondito processo di selezione[1].

AES è basato sull'algoritmo per la cifratura simmetrica Rijndael[1] ideato dai crittografi belgi Joan Daemen and Vincent Rijmen; Rijndael è stato scelto tra le diverse proposte perché rappresenta un buon compromesso tra sicurezza e velocità di elaborazione, sia nelle sue implementazioni software che in quelle hardware.[5].

AES lavora su blocchi di dati di 128 bit e consente di usare chiavi di tre lunghezze: 128, 192 o 256 bit. La sua struttura è quella di un algoritmo

[1] Il nome Rijndael è una combinazione dei nomi fiamminghi dei suoi autori; si pronuncia "rèin-daal".

Lunghezza chiave	Numero round
128 bit	10
192 bit	12
256 bit	14

Tabella 3.1: Numero di round in relazione alla lunghezza della chiave AES.

```
State = input

AddRoundKey(State, RoundKey[0])

for r = 1 to rounds-1
    SubBytes(State)
    ShiftRows(State)
    MixColumns(State)
    AddRoundKey(State, RoundKey[r])
end

SubBytes(State)
ShiftRows(State)
AddRoundKey(State, RoundKey[rounds])

output = State
```

Codice 3.1: *Procedura di cifratura AES.*

crittografico iterativo; il numero di round dipende dalla lunghezza della chiave (vedi tabella 3.1).

Da un punto di vista molto astratto, l'algoritmo di cifratura AES può essere descritto dal codice 3.1.

La variabile *input* rappresenta il blocco di dati in chiaro di 128 bit mentre *output* è il corrispondente blocco cifrato. Le quattro procedure di base, *AddRoundKey*, *SubBytes*, *ShiftRows* e *MixColumns* sono tutte operazioni che modificano la variabile *State*.

```
State = input

AddRoundKey(State, RoundKey[rounds])

for r = rounds-1 to 1
    InvShiftRows(State)
    InvSubBytes(State)
    AddRoundKey(State, RoundKey[r])
    MixColumns(State)
end

InvShiftRows(State)
InvSubBytes(State)
AddRoundKey(State, RoundKey[0])

output = State
```

Codice 3.2: *Procedura di decifratura AES.*

Per ragioni che saranno chiare tra poco, conviene pensare alla variabile *State*, di 128 bit ovvero 16 byte, come a una matrice quadrata di lato 4 byte.

L'algoritmo di decifratura, descritto nel codice di Figura 3.2, utilizza le medesime funzioni o le loro inverse; ovviamente in questo caso *input* è il blocco di dati cifrato e *output* il corrispondente blocco in chiaro.

3.1 ShiftRows

La funzione *ShiftRows* (figura 3.1) trasforma la matrice *State* in questo modo:

1. la prima riga rimane invariata;
2. la seconda riga viene ruotata a sinistra di una posizione;
3. la terza riga viene ruotata a sinistra di due posizioni;

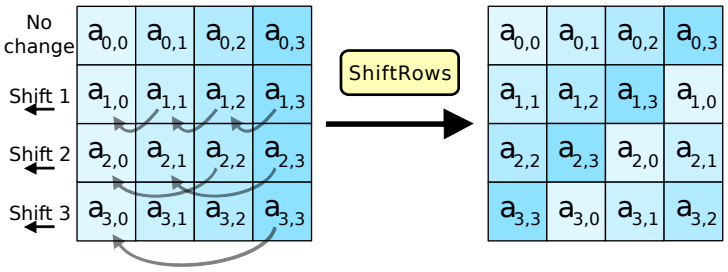

Figura 3.1: L'operazione ShiftRows di AES.

4. la quarta riga viene ruotata a sinistra di tre posizioni;

La funzione inversa opera sulle stesse righe le medesime rotazioni ma in senso opposto, verso destra.

3.2 SubBytes

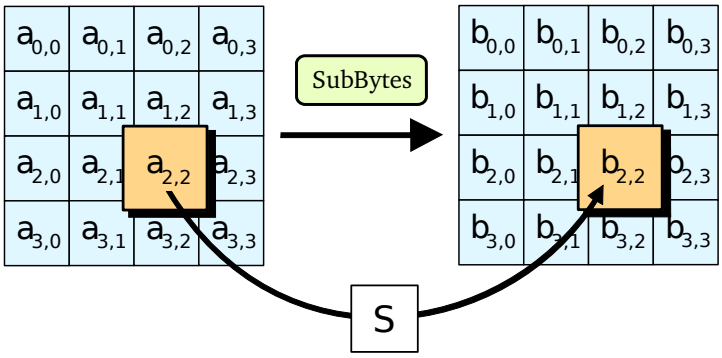

Figura 3.2: L'operazione SubBytes di AES.

Dal punto di vista implementativo l'operazione di *SubBytes* (figura 3.2) non è altro che una classica Substitution box (S-box), ovvero una sostituzione di tutti i byte della matrice *State* con i corrispondenti valori di una tabella di 256 elementi; in realtà tale sostituzione può essere definita algebricamente.

La formulazione algebrica della S-box di AES sfrutta il campo finito (o campo di Galois) basato sul seguente polinomio con valori in Z_2:

$$x^8 + x^4 + x^3 + x + 1$$

La S-box di AES può essere sostituita concettualmente dall'operazione di inverso moltiplicativo nel campo finito di cui sopra, abbinata con una trasformazione affine. Questa particolare S-box garantisce la non linearità ed è priva di punti fissi, rendendola immune da una classe di attacchi basata proprio su queste caratteristiche[6].

La S-box inversa è banalmente definita a partire da quella appena descritta.

3.3 MixColumns

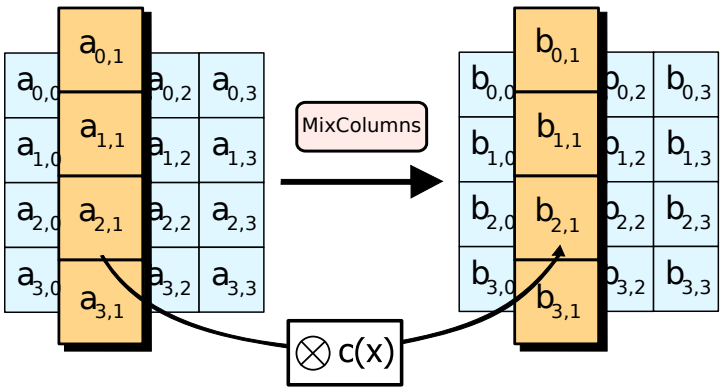

Figura 3.3: L'operazione MixColumns di AES.

La funzione *MixColumns* (figura 3.3) opera sulle quattro colonne della matrice *State*. Ogni colonna viene rimpiazzata dal risultato della sua moltiplicazione per una particolare matrice; le operazioni si svolgono nel campo finito descritto nella sezione 3.2.

3.4 AddRoundKey

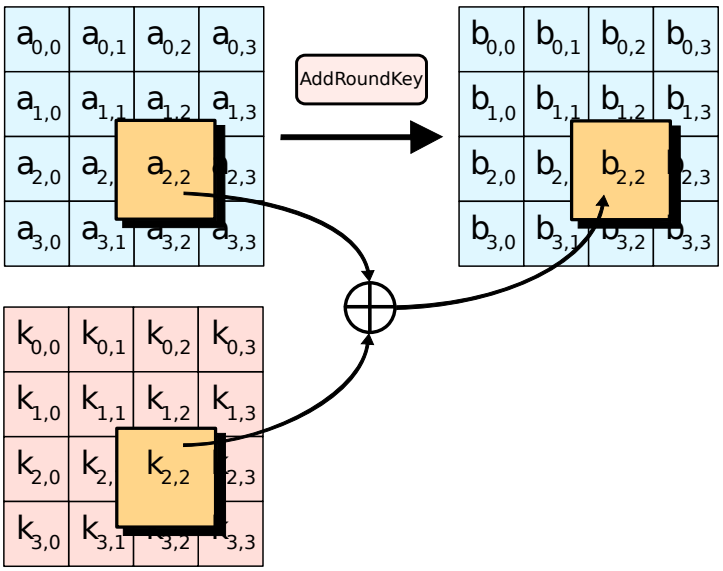

Figura 3.4: L'operazione AddRoundKey di AES.

La funzione *AddRoundKey* (figura 3.4) consiste in un semplice XOR della matrice *State* con la sottochiave del round corrente. La generazione delle sottochiavi, tante quante il numero di round più uno, è compito della procedura di espansione della chiave descritta nella sezione 3.5. In questo caso la funzione coincide con la sua inversa, quello che cambia all'atto della decifratura è l'ordine con cui vengono applicate le varie sottochiavi.

3.5 Espansione della chiave

Il processo di espansione della chiave coinvolge la S-box descritta nella sezione 3.2 e due ulteriori funzioni:

Rotate: ruota di 1 byte verso sinistra una parola di 4 byte.

Rcon: esegue l'elevamento a potenza di un byte; sia il byte che le operazioni vengono considerati nel campo finito descritto nella sezione 3.2.

Bisogna inoltre definire tre valori:

Nk: la dimensione in bit della chiave divisa per 32;

Nb: dimensione in byte della chiave;

Nr: il numero di round (vedi tabella 3.1).

L'algoritmo completo per l'espansione della chiave è riportato nel codice di Figura 3.3.

3.6 Cenni sulla sicurezza dell'AES

Già durante la gara indetta dal NIST, AES era stato violato rispettivamente fino a 7 round per chiavi a 128 bit, 8 per quelle a 192 bit e 9 per quelle a 256 bit[7].

Nell 2009 ci sono stati due attacchi importanti di tipo "related-key"; in questi casi lo svolgimento dell'attacco richiede l'osservazione del cifrario in azione con diverse chiavi sconosciute ma legate tra loro da una relazione matematica nota. Biryukov-Khovratovich-Nikolic hanno portato un attacco di complessità 2^{119} per chiavi di 256 bit e 192 bit[8]; Biryukov, Dunkelman, Keller, Khovratovich, e Shamir hanno elaborato un attacco di complessità 2^{39} una versione a 9 round di AES con chiave a 128 bit sfruttando soltanto due chiavi correlate[9].

```
key = {la chiave, un array di byte}
w = {l'insieme delle sottochiavi, un array di byte}
temp = {array di 4 byte}
i = 0

while i < Nk
    w[i] = [key[4 * i],
            key[4 * i + 1],
            key[4 * i + 2],
            key[4 * i + 3]]
    i = i + 1
end

i = Nk

while i < Nb * (Nr + 1)
    temp = w[i - 1]

    if i mod Nk = 0
        temp = SubWord(RotWord(temp)) xor Rcon[i / Nk]
    else
        if Nk > 6 and i mod Nk = 4
            temp = SubWord(temp)
        end

    w[i] = w[i - Nk] xor temp
    i = i + 1
end
```

Codice 3.3: *Procedura di espansione della chiave AES.*

Parte II

Fase sperimentale

4 Analisi e progettazione

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it.

E. W. Dijkstra

4.1 Struttura del parallelismo

Poiché AES è un cifrario a blocchi, è estremamente naturale applicarvi un parallelismo sui dati che sfrutti proprio la suddivisione dei dati di input in blocchi.

PAES può istanziare un qualsiasi numero di work-item (*global_size*) e ricevere in ingresso un file formato da qualsiasi numero di blocchi. Confrontando queste due variabili possiamo avere tre situazioni, con conseguente ripartizione dei blocchi rispetto ai work-item:

***global_size* = *blocchi*:** ciascun work-item si occupa di applicare AES a un blocco dell'input;

***global_size* > *blocchi*:** *blocchi* work-item si occupano di applicare AES a un blocco dell'input, *globalsize* − *blocchi* work-item non vengono schedulati;

***global_size* < *blocchi*:** ciascun work-item applica AES a $\frac{global\_size}{blocchi}$ blocchi; se il *resto* della divisione è diverso da zero, *resto* work-item dovranno occuparsi di un ulteriore blocco dell'input.

In poche parole PAES cerca di ripartire uniformemente i blocchi da cifrare tra i work-item disponibili senza porre vincoli artificiosi a queste due variabili. In questo modo è possibile dimensionare il numero di work-item a seconda delle peculiarità dell'hardware a disposizione.

In termini strettamente riguardanti OpenCL, quanto scritto sopra equivale a definire uno spazio degli indici monodimensionale la cui grandezza non è necessariamente fissata a priori; anche la suddivisione dei work-item in work-group non è cablata nello schema di parallelismo ed è adattabile rispetto al dispositivo di calcolo usato di volta in volta. Il modello di suddivisione dei dati corrisponde al SPMD (vedi sezione 2.4.1) in quanto ciascun work-item è istanza del medesimo kernel ma il flusso di esecuzione può seguire percorsi differenti.

4.2 Strumenti utilizzati

La scelta di OpenCL impone l'uso del suo dialetto C99 (vedi sezione 2.5) per la creazione dei kernel che saranno eseguiti nei device.

Per quanto concerne il programma lato host esistono API OpenCL per numerosi linguaggi di programmazione, ma quelle fornite dai produttori sono utilizzabili direttamente soltanto da programmi C o C++. Poiché lo stato delle diverse implementazioni di OpenCL è in continua evoluzione, per evitare problemi dovuti alla sincronizzazione delle modifiche tra le API originali e quelle dei vari binding, PAES sfrutta direttamente le API C/C++ fornite dai produttori.

Il programma lato host è stato scritto con il linguaggio C99, ovvero la più recente evoluzione standardizzata dall'International Standard Organization (ISO) del C, le cui peculiarità utilizzate in PAES sono:

1. possibilità di commentare con la sequenza "//";
2. possibilità di dichiarare variabili in qualsiasi punto del programma;
3. introduzione di un tipo di dato booleano, ovvero un intero che può assumere soltanto i valori 0 e 1; qualora un altro intero venga convertito in booleano, se il suo valore è 0 rimane 0, se è diverso da 0 diventa 1. È così possibile mantenere la compatibilità con la rappresentazione dei valori booleani del C classico usando un tipo di dati più espressivo dei generici interi.

Il compilatore di riferimento è il GCC, versione 4; sia i test che lo sviluppo sono stati eseguiti su sistemi Linux, sia a 32 bit che a 64 bit. Le implementazioni di OpenCL utilizzate sono rispettivamente quella della NVIDIA Corporation e quella di AMD, nelle ultime versioni disponibili al momento della stesura di questo elaborato, rispettivamente la 2.3 e la 2.0.

Per quanto riguarda il programma in esecuzione nell'host sono stati ampiamente usati Data Display Debugger (DDD)[1] e Valgrind[2]: il primo è un'interfaccia grafica per molti debugger, tra quello usato durante lo sviluppo di PAES, il GNU Debugger (GDB); Valgrind invece consente di eseguire un programma in un ambiente emulato per rilevare errori nell'allocazione e nell'uso della memoria, estremamente comuni nei programmi sviluppati in C.

Il codice è ampiamente documentato mediante l'uso di Doxygen[3]: si tratta di uno strumento che consente di inserire la documentazione dei vari metodi, costanti, tipi di dato, etc., direttamente all'interno del codice, esattamente come JavaDoc.

I programmi di test descritti nella sezione 6.1 sono basati su Python 2.6 e richiedono una shell compatibile con la Bourne Shell (es. la diffusissima BASH).

[1] http://www.gnu.org/software/ddd/
[2] http://valgrind.org/
[3] http://www.stack.nl/~dimitri/doxygen/

4.3 Struttura dei file sorgenti

PAES è formato da 7 file sorgenti, ciascuno con un ruolo ben preciso:

***paes_functions.c*:** contiene la funzione *apply_aes* che consente al programma lato host di eseguire la procedura AES sul device OpenCL;

***paes_functions.h*:** contiene i prototipi delle funzioni che sono esportate per l'uso esterno rispetto a *paes_functions.c*.

***paes.cl*:** contiene il kernel che esegue un round di AES specificato dal programmatore sui dati passatigli in ingresso e le funzioni accessorie per assolvere questo compito;

***paes_constants_and_datatypes.h*:** contiene le costanti e i tipi di dato non primitivi condivisi tra il programma lato host e il kernel OpenCL;

***paes_size.h*:** contiene dei valori usati per determinare local e global size (vedi sottosezione 5.1.2);

***sha256.c*:** contiene un'implementazione della funzione di hashing SHA-256, utilizzata da PAES per trattare le password inserite dall'utente;

***sha256.h*:** contiene i prototipi delle funzioni esportate da *sha256.c*;

***paes.c*:** contiene la funzione *main* e tutte le funzioni strettamente necessarie all'interfaccia a riga di comando del programma.

Insieme ai sorgenti, PAES comprende anche due file accessori:

***doxygen.cfg*:** contiene la configurazione di Doxygen personalizzata per PAES;

***Makefile*:** contiene le regole per la compilazione di PAES e altri compiti secondari, come l'indentazione automatica del codice sorgente secondo lo standard K&R.

4.4 Interfaccia utente

Poiché PAES è un progetto con finalità principalmente di ricerca, la complessità dell'interfaccia utente è minima rispetto a quella del nucleo del programma. In effetti PAES è un semplice programma a riga di comando (vedi figura 4.1) il cui comportamento è definito dalle opzioni specificate dall'utente descritte nella tabella 4.1.

```
$ ./paes -i in -o out -m encrypt -k 192 -p 'pass' -d gpu


-------- PAES --------


PARAMETERS:
  Input file: in
  Output file: out
  AES mode: encrypt
  Key size: 192
  Device: gpu
  File size: 134217728 bytes

Loading OpenCL source code...
clCreateContextFromType...
```

Figura 4.1: PAES in azione.

La gestione delle opzioni della riga di comando è stata implementata mediante la libreria *getopt* abitualmente usata nei programmi Unix. In particolare, chiamate successive alla funzione *getopt* restituiranno di volta in volta il carattere delle varie opzioni della riga di comando specificate dall'utente o -1 se non ve ne sono più; nella variabile *optarg* è contenuto un puntatore all'eventuale valore dell'argomento associato all'opzione corrente. Oltre alle variabili *argc* e *argv*, che contengono gli argomenti separati e il loro numero, *getopt* richiede un terzo argomento che consiste nella lista di lettere rappresentanti le varie opzioni, eventualmente seguite dai due punti qualora l'opzione richieda

Opzione	Significato
-i	file di input
-o	file di output
-m	modalità di uso di AES, può essere `encrypt` o `decrypt`
-k	dimensione della chiave (`128`, `192` o `256`)
-p	password usata per derivare la chiave per cifrare
-d	dispositivo da usare per il calcolo (`cpu` o `gpu`)

Tabella 4.1: Opzioni della riga di comando di PAES.

```
do {
    c = getopt(argc, argv, "hi:o:m:k:p:d:g:l:");
    switch (c) {
        case 'i':
            *input_file_name = (char *) malloc(sizeof(char) *
                strlen(optarg) + 1);
            strcpy(*input_file_name, optarg);
            break;
        // ... similmente per le restanti opzioni
    }
} while (c != -1);
```

Codice 4.1: *Lettura delle opzioni della riga di comando in PAES.*

un argomento aggiuntivo. Il codice 4.1 è un estratto del codice di PAES che mostra il semplicissimo utilizzo di *getopt*.

5 Implementazione

One of my most productive days was throwing away 1000 lines of code.

K. Thompson

5.1 Programma host

La componente di PAES in esecuzione sull'host si occupa di controllare le richieste dell'utente e preparare il device per l'esecuzione del compito desiderato. La lettura delle richieste lato utente avviene mediante la libreria *getopt*, come descritto nella sezione 4.4. Una volta letti i dati in input PAES controlla che questi abbiano senso: ad esempio il dispositivo da usare deve essere `cpu` o `gpu`, nessun'altro valore è ammissibile.

La password inserita dall'utente invece viene trattata in modo particolare; l'utente infatti può inserire parole chiave di lunghezza arbitraria, mentre AES necessita di chiavi dalla lunghezza prefissata (128, 192 o 256 bit). Per questo motivo PAES calcola l'hash SHA-256 della password in questione: poiché questa funzione produce un hash di 256 bit è sempre possibile ricavare da quest'ultimo una chiave AES di qualsiasi lunghezza ammissibile.

La funzione SHA-256 utilizzata è stata ripresa dal codice di GNU Privacy Guard (GnuPG)[1] e leggermente adattata per farla funzionare al di fuori del suo programma originario. Qualora l'utente non specifici la password mediante l'argomento della riga di comando `-p` gli verrà richiesta interattivamente mediante la funzione *getpass*: l'inserimento della password seguirà lo standard de facto in Unix, quindi a schermo non verrà visualizzato ciò che l'utente sta digitando ma alla pressione del tasto INVIO la password verrà memorizzata correttamente.

Successivamente PAES leggerà dal disco il contenuto del file da cifrare e richiamerà la funzione *apply_aes*, che si occuperà di preparare ed eseguire il programma OpenCL sul dispositivo specificato. Il risultato di quest'ultima operazione verrà infine scritto nel file di output.

5.1.1 La funzione apply_aes

Questa funzione, contenuta nel file *paes_functions.c* (vedi sezione 4.3), si occupa di preparare ed eseguire il programma OpenCL sul dispositivo specificato dall'utente. Il processo è composto da una serie di passaggi piuttosto semplici, anche se tediosi, basati principalmente su chiamate API di OpenCL. Per ogni chiamata viene controllato se è stata eseguita con successo: in caso contrario l'esecuzione della funzione termina, rilasciando comunque tutti gli oggetti allocati fino a quel momento. Se tutte le chiamate vengono eseguite con successo, la funzione termina normalmente e la memoria allocata viene comunque liberata. Negli esempi di codice riportati di seguito le istruzioni per il controllo degli errori sono omesse per brevità e chiarezza.

Il primo passo consiste nel determinare l'implementazione di OpenCL (definita con il sostantivo "piattaforma") che PAES dovrà utilizzare. Verrà sempre usata la prima implementazione trovata, in quanto in un'installazione realistica non vi saranno mai più piattaforme OpenCL contemporaneamente onde evitare conflitti e potenziali problemi di compatibilità. Il codice 5.1 mostra le

[1] http://www.gnupg.org/

```
clGetPlatformIDs(0, NULL, &num_platforms);
platforms = (cl_platform_id *) malloc(sizeof(cl_platform_id) *
    num_platforms);
clGetPlatformIDs(num_platforms, platforms, NULL);
cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,
    (cl_context_properties)platforms[0], 0};
cl_context_properties *cprops;
if (NULL == platforms[0]) cprops = NULL;
else cprops = cps;
context = clCreateContextFromType(cprops, device_type[device],
    NULL, NULL, &error);
```

Codice 5.1: *Scelta dell'implementazione di OpenCL da utilizzare.*

chiamate necessarie per determinare la piattaforma OpenCL presente sulla macchina. Dalla piattaforma verrà poi ricavato il contesto nel quale eseguire i kernel OpenCL (vedi sezione 2.2.1). Si noti la necessità di effettuare una chiamata OpenCL per controllare quante piattaforme sono effettivamente disponibili prima di allocare un'array per memorizzarle: questo modo di procedere è molto comune nell'API OpenCL ove sia necessario reperire informazioni di grandezza variabile da memorizzare nell'heap.

Grazie al contesto è ora possibile selezionare il device sul quale eseguire effettivamente il calcolo. Sulla base dell'input dell'utente tale device può essere una GPU o una CPU; anche in questo caso viene selezionato il primo device della tipologia desiderata in quanto generalmente l'unico del genere presente sulla macchina (si noti che una CPU multi-core è comunque vista come un'unica CPU contenente più CU). Questo processo è visibile nel codice di Figura 5.2. Anche in questo caso viene prima trovato il numero di dispositivi disponibili, poi vengono memorizzati i riferimenti agli stessi in un'array di lunghezza adeguata. Poiché verrà sempre usato il primo device trovato, d'ora in poi verrà sempre usato il primo elemento dell'array *devices*, ovvero *devices[0]*.

Una volta trovato il dispositivo si può creare una coda dei comandi asso-

```
size_t context_information_size;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL,
    &context_information_size);
devices = (cl_device_id *) malloc(context_information_size);
clGetContextInfo(context, CL_CONTEXT_DEVICES,
    context_information_size, devices, NULL);
```

Codice 5.2: *Selezione del device da utilizzare.*

```
clCreateCommandQueue(context, devices[0],
    CL_QUEUE_PROFILING_ENABLE, &error);
```

Codice 5.3: *Creazione di una coda di comandi.*

ciata allo stesso come descritto nel codice 5.3. È fondamentale specificare il parametro *CL_QUEUE_PROFILING_ENABLE* che consente di abilitare la misura dei tempi di esecuzione dei comandi eseguiti nella coda appena creata.

Poiché AES richiede che la chiave ricavata dalla password dell'utente venga espansa per generare le diverse sottochiavi, è necessario procedere con questa operazione (vedi codice 5.4). Poiché la creazione delle sottochiavi ha un costo molto piccolo e indipendente dalla dimensione dei dati in ingresso questo calcolo viene svolto direttamente dal programma host per evitare inutili complicazioni. Il comportamento delle funzioni *get_round_key_size* e *key_expansion* ricalca perfettamente quanto riportato nello standard AES ed è stato già descritto nella sezione 3.5.

A questo punto è necessario scrivere nella memoria del dispositivo di calcolo i dati su cui dovrà essere applicato AES. Prima di tutto si crea un buffer OpenCL mediante la chiamata *clCreateBuffer*; il buffer viene creato in modalità lettura e scrittura poiché servirà anche per contenere il risultato

```
cl_uint round_key_size = get_round_key_size(key_size_bits);
cl_uchar *round_key = key_expansion(key, key_size_bits);
```

Codice 5.4: *Creazione delle sottochiavi AES.*

```
cl_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE,
    sizeof(cl_uchar) * size, NULL, &error);
clEnqueueWriteBuffer(command_queue, cl_buffer, CL_TRUE, 0,
    sizeof(cl_uchar) * size, (void *) buffer, 0, NULL,
    &event_write);
```

Codice 5.5: *Creazione e scrittura del buffer nella memoria del dispositivo.*

```
program = clCreateProgramWithSource(context, 1,
    (const char **) &source, &source_size, &error);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG,
     300, build_log, NULL);
printf("\nBuild log:\n%s\n", build_log);
```

Codice 5.6: *Compilazione del programma da eseguire sul dispositivo.*

della computazione. Quindi, tramite la funzione *clEnqueueWriteBuffer*, viene messa nella coda dei comandi un'istruzione per la scrittura del buffer nella memoria del device; all'operazione di scrittura viene associato un evento che servirà in seguito per misurare le prestazioni. Il terzo parametro di questa chiamata indica che l'operazione è bloccante, quindi il flusso del programma host rimarrà fermo fino al suo completamento. Le operazioni illustrate in questo paragrafo sono implementate dal codice 5.5.

Il programma OpenCL deve essere compilato per il device in cui verrà eseguito. Purtroppo al momento della stesura l'implementazione OpenCL di AMD non supporta il caricamento di file binari compilati in precedenza, quindi è necessario compilare il sorgente OpenCL sul momento. Il codice 5.6 mostra le istruzioni necessarie per la compilazione del sorgente: il codice del programma OpenCL è contenuto nella variabile *source*, di dimensione *source_size*, che è stata caricata in precedenza con il contenuto del file sorgente *paes.cl* (vedi sezione 4.3).

Avendo a disposizione il programma compilato, grazie alla funzione *clCreateKernel* è possibile selezionare il codice binario del kernel che verrà eseguito

nel device. Il codice 5.6 mostra come creare un oggetto kernel che verrà usato per eseguire la computazione nonché il modo per impostare gli argomenti che rimarranno fissi durante la stessa grazie alla funzione *clSetKernelArg*. Alcune note importati su quest'ultima operazione:

1. Gli argomenti per il kernel destinati a essere memorizzati nella Global Memory o nella Constant Memory necessitano di un apposito buffer OpenCL e sarà quest'ultimo a essere usato come argomento; poiché per le sottochiavi non era ancora stato creato un buffer è necessario farlo ora. Questo buffer è impostato in modalità sola lettura in quanto durante la computazione non è mai necessario scrivervi.

2. Gli argomenti che verranno salvati nella Private Memory possono invece essere passati senza ricorrere a buffer intermedi.

3. *clSetKernelArg* richiede che venga specificato l'indice dell'argomento da passare; tale indice parte da 0 e ricalca fedelmente l'ordine degli argomenti della funzione kernel.

La creazione del kernel e l'impostazione degli argomenti fissi è implementata dal codice 5.7; il nome del kernel ovviamente deve essere lo stesso di quello dichiarato nel sorgente OpenCL (vedi sezione 5.2).

Il codice 5.8 mostra infine l'esecuzione del kernel. Il kernel in questione esegue soltanto un round AES specificato in un parametro impostato nel programma host; tale parametro deve quindi essere cambiato di volta in volta, a differenza di quelli impostati precedentemente che rimangono fissi. Il kernel viene eseguito sullo spazio degli indici definiti da *global_size* e *local_size*; il metodo per determinare tali valori è descritto nella sezione 5.1.2. Mettere in coda l'esecuzione del kernel non è un'operazione bloccante quindi è necessario ricorrere alla chiamata *clFinish* per sincronizzare il programma host con quello eseguito sul device. I tempi di esecuzione dei vari round vengono sommati per ottenere il tempo totale impiegato per la cifratura AES.

```
clCreateKernel(program, "kernel_aes", &error);

clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *) &cl_buffer);
clSetKernelArg(kernel, 1, sizeof(cl_ulong), (void *) &blocks);
clSetKernelArg(kernel, 2, sizeof(cl_uint), (void *) &mode);

cl_rkey = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
   sizeof(cl_uchar) * round_key_size, round_key, &error2);
clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *) &cl_rkey);

clSetKernelArg(kernel, 4, sizeof(cl_uint), (void *) &rounds);
```

Codice 5.7: *Creazione del kernel e impostazione degli argomenti fissi.*

```
double execution_time = 0;
for (cl_uint round = 0; round <= rounds; ++round) {
    clSetKernelArg(kernel, 5, sizeof(cl_uint), (void *) &round);

    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
        &global_size, &local_size, 0, NULL, &event_execute);
    clFinish(command_queue);

    execution_time += execution_time_msecs(event_execute);
}
```

Codice 5.8: *Esecuzione del kernel.*

```
clEnqueueReadBuffer(command_queue, cl_buffer, CL_TRUE, 0, sizeof
    (cl_uchar) * size, buffer, 0, NULL, &event_read);
```

Codice 5.9: *Lettura del risultato dell'esecuzione del kernel.*

Per ottenere il risultato della computazione è necessaria ancora un'altra istruzione per la coda dei comandi del device, descritta nel codice di Figura 5.9. L'istruzione è anch'essa bloccante e viene associata a un evento per misurarne il tempo d'esecuzione.

È infine fondamentale rilasciare scrupolosamente tutti gli oggetti allocati nel corso di questa lunga serie di operazioni per evitare che la macchina e i device utilizzati vengano saturati nel corso di esecuzioni successive del programma; a tale scopo, oltre alla consueta istruzione *free*, OpenCL mette a disposizione una serie di chiamate API per deallocare le varie tipologie di oggetti specifiche di OpenCL stesso. Nel caso di PAES sono necessarie rispettivamente *clReleaseEvent*, *clReleaseMemObject*, *clReleaseKernel*, *clReleaseProgram*, *clReleaseCommandQueue* e *clReleaseContext*.

5.1.2 Global e local size

PAES consente di gestire global e local size in due modi:

1. staticamente;
2. dinamicamente.

L'uso di un file header dedicato (in questo caso *paes_size.h*) rende possibile la scrittura di politiche dinamiche per determinare global e local size in base alla dimensione dell'input. Nel file possono essere contenuti i seguenti valori:

PAES_DYNAMIC_SIZE: se definito, abilita la politica dinamica;

PAES_MAX_LOCAL_SIZE: il massimo valore da assegnare alla local size;

```
static double execution_time_msecs(cl_event event)
{
    cl_ulong start, end;
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
        sizeof(cl_ulong), &start, NULL);
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
        sizeof(cl_ulong), &end, NULL);
    return (end - start) * 1.0E-6;
}
```

Codice 5.10: *Misura delle prestazioni di un comando tramite eventi.*

PAES_GLOBAL_LOCAL_RATIO: il rapporto tra global e local size.

PAES cerca di impostare una global size pari al numero di blocchi AES del file da cifrare mentre la local size è ottenuta come rapporto tra global e local size. Qualora il numero di workgroup, ovvero $\frac{global\_size}{local\_size}$, superi i limiti fisici del device, la global size viene ridimensionata.

Se invece si desidera usare valori fissi per local e global size bisogna impostare le seguenti costanti simboliche nel file header:

PAES_STATIC_SIZE: se definito, abilita la politica statica;

PAES_LOCAL_SIZE: il valore da assegnare alla local size;

PAES_GLOBAL_SIZE: il valora da essegnare alla global size.

5.1.3 Misura delle prestazioni

Le prestazioni del programma eseguito sul dispositivo OpenCL vengono misurate mediante gli eventi messi a disposizione da OpenCL stesso. Dato un evento associato a un comando della coda di esecuzione terminato correttamente, è possibile risalire al tempo di esecuzione del comando stesso mediante la semplice funzione riportata nel codice di Figura 5.10.

```
__kernel __attribute__ ((vec_type_hint(uchar)))
void kernel_aes(
    __global uchar16 *buffer,
    const ulong blocks,
    const uint mode,
    __constant const uchar16 *round_key,
    const uint rounds,
    const uint round)
```

Codice 5.11: *Prototipo del kernel di PAES.*

L'esecuzione viene misurata con una risoluzione nell'ordine teorico dei nanosecondi, ma per gli scopi di PAES il valore viene riportato in millisecondi per fornire una misura più chiara e comunque assolutamente significativa.

5.2 Kernel

Come visto nella sezione 4.3 il kernel OpenCL di PAES e le funzioni da esso utilizzate sono contenute nel file *paes.cl*. La funzione kernel, richiamata dal programma host, ha il prototipo riportato nel codice di Figura 5.11.

Come da standard, il prototipo deve iniziare con il qualificatore `__kernel`. Il qualificatore `__attribute__` e il suo argomento sono definiti dallo standard in questo modo:

> *__attribute__((vec_type_hint(<type>))) è un suggerimento opzionale per il compilatore; serve a rappresentare la larghezza computazionale della funzione kernel e dovrebbe servire come base per il calcolo dell'uso della larghezza di banda del processore quando il compilatore sta cercando di autovettorizzare il codice.*[3]

I parametri del kernel hanno questo significato:

***buffer*:** il vettore contenente i blocchi da cifrare, che verranno sovrascritti coi blocchi cifrati, allocati nella Global Memory;

***blocks*:** il numero di blocchi AES contenuti nel buffer;

***mode*:** indica se si vuole cifrare o decifrare i dati;

***round_key*:** le sottochiavi, allocate nella Constant Memory;

***rounds*:** il numero totale di round previsto per la lunghezza di chiave selezionata dall'utente;

***round*:** il round che il kernel deve eseguire.

Il buffer, passato dal programma host come array di singoli *char* privi di segno viene qui rappresentato come array di *uchar16*: si tratta di un tipo vettoriale di OpenCL contenente 16 byte, ovvero proprio la dimensione di un blocco AES. Il cambio di rappresentazione non inficia la corretta lettura dei valori nel buffer in quanto in layout dei dati in memoria rimane identico; d'altro canto l'uso di *uchar16* consente di semplificare la scrittura del codice AES rendendolo più comprensibile, sia per la mappatura diretta dei blocchi sia per alcune peculiarità dei tipi vettoriali OpenCL che saranno descritte in seguito.

Prima di tutto ciascuna istanza del kernel calcola l'intervallo di blocchi del buffer di input che dovrà cifrare. Il calcolo, descritto nel codice di Figura 5.12 implementa la logica descritta nella sezione 4.1 che, ricordo, ripartisce uniformemente i blocchi da cifrare tra i work-item disponibili.

A questo punto il work-item ha tutte le informazioni necessarie per svolgere la sua parte di computazione, il cui schema è riportato nel codice di Figura 5.13. Innanzitutto ciascuna istanza del kernel controlla se deve effettivamente cifrare dei blocchi (qualora $global\_worksize < blocks$, alcuni work-item non devono fare nulla); se questo è il caso, AES è applicato a tutti i blocchi di propria competenza.

Il codice del kernel per cifrare un blocco è riportato nel codice di Figura 5.14, quello per decifrare è molto simile. Le istruzioni condizionali del preprocessore vengono usate nella compilazione in fase di test (vedi sezione 6.1) allo scopo di

```
size_t global_work_size = get_global_size(0);
size_t global_id = get_global_id(0);

size_t blocks_per_work_item;
if (global_work_size < blocks)
    blocks_per_work_item = blocks / global_work_size;
else
    blocks_per_work_item = 1;

size_t reminder;
if (global_work_size < blocks)
    reminder = blocks % global_work_size;
else
    reminder = 0;

size_t from_block = global_id * blocks_per_work_item;
if (global_id < reminder)
    from_block += global_id;
else
    from_block += reminder;
```

Codice 5.12: *Selezione dei blocchi per ciascun work-item.*

```
if (from_block < blocks) {
    size_t to_block = from_block + blocks_per_work_item;
    if (global_id < reminder)
        to_block += 1;

    if (mode == AES_MODE_ENCRYPT)
        for (size_t b = from_block; b < to_block; ++b)
            // Cifra il blocco buffer[b]
    else
        for (size_t b = from_block; b < to_block; ++b)
            // Decifra il blocco buffer[b]
}
```

Codice 5.13: *Codice principale del kernel.*

```
#ifdef SHIFT_ROWS
    shift_rows(b, buffer);
#elif defined(MIX_COLUMNS)
    mix_columns(b, buffer);
#elif defined(ADD_ROUND_KEY)
    buffer[b] ^= round_key[rounds];
#elif defined(SUB_BYTES)
    sub_bytes(b, buffer, sbox_encrypt);
#else
    if (round == 0) {
        buffer[b] ^= round_key[0];
    } else if (round == rounds) {
        sub_bytes(b, buffer, sbox_encrypt);
        shift_rows(b, buffer);
        buffer[b] ^= round_key[rounds];
    } else {
        sub_bytes(b, buffer, sbox_encrypt);
        shift_rows(b, buffer);
        mix_columns(b, buffer);
        buffer[b] ^= round_key[round];
    }
#endif
```

Codice 5.14: *Codice kernel per la cifratura di un blocco.*

```
__constant const uchar sbox_encrypt[256] = AES_SBOX_ENCRYPT;
__constant const uchar sbox_decrypt[256] = AES_SBOX_DECRYPT;

void sub_bytes(size_t block, __global uchar16 *buffer,
    __constant const uchar *sbox)
{
     buffer[block] = (uchar16)(sbox[buffer[block][0]],
                               sbox[buffer[block][1]],
                               sbox[buffer[block][2]],
                               sbox[buffer[block][3]],
                               // ...
                               sbox[buffer[block][13]],
                               sbox[buffer[block][14]],
                               sbox[buffer[block][15]]);
}
```

Codice 5.15: *Funzione SubBytes definita nel kernel.*

consentire controlli sulla conformità allo standard delle singole funzioni AES; normalmente verrà sempre compilato il codice contenuto nel ramo *#else*, che in effetti implementa un singolo round AES con gli opportuni accorgimenti per il primo e l'ultimo round.

Le primitive AES sono implementate come funzioni all'interno del medesimo sorgente OpenCL, con l'esclusione della *AddRoundKey* che è rappresentabile da una singola istruzione; questo perché l'uso dei tipi di dato vettoriali OpenCL consente l'applicazione di operazioni a tutti gli elementi del vettore contemporaneamente (in questo caso, l'*or* esclusivo).

Come esempio delle restanti funzioni AES, il codice 5.15 riporta la S-box. Il parametro *block*, ovvero il blocco su cui lavorare, e *buffer*, corrispondente al *buffer* di input/output della funzione kernel, sono comuni a tutte le primitive AES; la tabella usata per contenere i valori di tutte le sostituzioni possibili, una peculiarità della funzione *sub_bytes*, è invece un puntatore a un'area della Constant Memory allocata con una costante globale nel codice sorgente OpenCL.

Anche in questo caso viene sfruttata una sintassi peculiare dei tipi vettoriali OpenCL, il costruttore anonimo; i singoli elementi del risultato sono ottenuti con la sostituzione mediante la tabella che rappresenta la S-box. Poiché tale tabella è un parametro, la funzione *sub_bytes* può essere usata sia per cifrare che per decifrare semplicemente cambiando quest'ultimo.

6 Metodologia

Though this be madness, yet there is method in't.

W. Shakespeare

Poiché lo scopo principale di PAES era quello di esplorare l'uso di OpenCL è stata posta particolare attenzione alla metodologia usata e alle sue differenze e somiglianze rispetto a quella canonica per lo sviluppo di programmi C.

OpenCL è uno standard ben definito, tuttavia le sue implementazioni sono ancora piuttosto instabili. Per questo motivo è stato opportuno isolare le chiamate OpenCL all'interno di una procedura (in questo caso si tratta di *apply_aes*, descritta nella sezione 5.1.1); in questo modo si limita l'impatto dei problemi delle implementazioni OpenCL a una parte di codice molto ben definita.

Il resto del programma host è quasi completamente svincolato dalle idiosincrasie di OpenCL salvo che per un punto fondamentale: i tipi di dato. Infatti la dimensione dei tipi di dato standard del linguaggio C (es. *int*) può variare tra l'host e il particolare device usato; a questo scopo OpenCL definisce dei tipi di dato da utilizzare al posto di quelli standard la cui dimensione è garantita essere identica sia nell'host che nei device. Ad esempio in PAES l'input è

rappresentato come un vettore di tipo *cl_uchar*, il corrispettivo OpenCL del tipo *unsigned char*.

Dato che OpenCL consente di eseguire programmi su device assolutamente eterogenei, PAES è stato strutturato in modo da avere la maggior libertà possibile nella scelta di local e global size (vedi sezione 4.1); l'uso di un file header dedicato per questi valori (vedi sottosezione 5.1.2) consente di averne più copie pronte per l'uso, ciascuna ottimizzata per un particolare dispositivo.

Sebbene il programma per il device sia scritto in un linguaggio con alcune limitazioni rispetto al C standard (vedi sezione 2.5) è stato opportuno evitare la duplicazione di definizioni di costanti e tipi di dato tra il programma host e quello per il device: a questo scopo è stato creato il file *paes_constants_and_datatypes.h*. Poiché il dialetto C OpenCL non supporta l'uso delle enumerazioni è stato necessario usare la direttiva del preprocessore *#define* per la definizione di valori costanti; in questo caso è stato scelto un compromesso tra minor duplicazione del codice e maggior pericolo di inconsistenze dei tipi di dato.

PAES è stato strutturato in maniera da poter essere usato in modo pressoché indolore da altri programmi. In effetti per riutilizzare il codice di PAES è sufficiente richiamare la funzione *apply_aes*, per il cui funzionamento sono necessari i file *paes.cl*, *paes_constants_and_datatypes.h*, *paes_size.h*, *paes_functions.h* e *paes_functions.c*. Sempre nell'ottica di favorire il riuso del codice tutte le funzioni, i tipi di dato e le costanti sono state ampiamente documentate.

6.1 Automazione dei test

Durante lo sviluppo di PAES è stato fondamentale valutarne sia la correttezza che le prestazioni.

Lo sviluppo dei test per la conformità rispetto allo standard AES e per la misura delle prestazioni è iniziato parallelamente rispetto a quello del nucleo

di PAES; lungi dall'essere un peso, infatti, questo modo di procedere agevola il lavoro complessivo da svolgere rendendo più semplice mantenere sotto controllo la complessità del codice che, inevitabilmente, cresce col passare del tempo [10].

Come riferimento per i test, sia di correttezza che di prestazioni, è stata selezionata un'implementazione seriale di AES corretta il cui codice è di facile comprensione, a patto che si conosca bene il funzionamento di AES.

I test sono implementati mediante script Python; oltre a essere portabili come e più di PAES, ciò li rende anche facilmente modificabili. In questo modo si è evitato di rendere pesante l'aggiornamento dei test parallelamente con la crescita di PAES, cosa che avrebbe potuto disincentivarlo completamente.

Tutti i test di PAES sono implementati mediante discendenti della classe *BaseTest*. Qui sono definiti una serie di metodi di utilità generale (vedi tabella 6.1) che verranno poi usati dalle sottoclassi. In particolare i discendenti dovranno implementare gli specifici test da eseguire nel metodo *test* e richiamare il metodo *run* nel codice principale dello script.

Grazie a *BaseTest* tutti i test vengono eseguiti in directory temporanee: se il test ha successo la sua directory temporanea viene cancellata, altrimenti rimane sul disco, a disposizione per ulteriori indagini. Inoltre *BaseTest* fornisce alle sottoclassi i metodi per la scrittura di un file di log dal formato predefinito con tutte le informazioni necessarie per contestualizzare i risultati di un test (nome del test, nome host, device utilizzato, durata...).

I test che sono stati effettivamente implementati sfruttando questa infrastruttura sono:

TestBijectivity: controlla che l'implementazione di PAES sia biettiva, ovvero l'applicazione della procedura di cifratura prima e di decifratura poi a un input qualsiasi dovrebbe produrre un risultato identico all'input stesso; ovviamente non si tratta di un test di correttezza ma di un metodo semplice e veloce per rilevare errori grossolani.

Metodo	**Descrizione**
compile_paes(opt)	compila PAES e prepara l'ambiente per eseguirlo
compile_aes(opt)	compila l'AES di riferimento
cleanup	elimina la directory temporanea del test
create_dummy(size)	crea un file casuale di dimensione *size*
paes(opt)	esegue PAES con le opzioni specificate
aes(opt)	esegue l'AES di riferimento con le opzioni specificate
diff(f1, f2)	controlla se i due file *f1* e *f2* sono uguali
echo(msg)	scrive su un file di log il messaggio specificato

Tabella 6.1: Principali metodi della classe BaseTest.

TestConformance: controlla che PAES produca gli stessi risultati dell'implementazione AES di riferimento, sia globalmente sia a livello delle singole operazioni AES (*SubBytes*, *ShiftRows*, *MixColumns*, *AddRoundKey*); il tutto sia per la cifratura che per la decifratura.

TestPerformance: rileva le prestazioni di PAES per una serie di file di dimensione crescente; le rilevazioni sono fatte sia per la cifratura che per la decifratura nonché per le operazioni di lettura e scrittura verso il device.

7 Valutazione delle prestazioni

Software is getting slower more rapidly than hardware becomes faster.

N. Wirth

Le analisi delle prestazioni di PAES sono state svolte, sia per quanto riguarda la versione OpenCL su CPU che per quella seriale, su un processore Intel Dual Core, avente le caratteristiche seguenti:

```
Name: Intel(R) Core(TM)2 CPU 6300 @ 1.86GHz (64 bit)
Vendor: GenuineIntel
Max compute units: 2
Max work item sizes: [1024, 1024, 1024]
Max work group size: 1024
Max clock frequency: 1.86 GHz
Max memory allocation size: 1 Gb
Global memory size: 3 Gb
Max constant buffer size: 64 Kb
Max constant arguments: 8
Local memory size: 32 Kb
```

I test su GPU invece si sono avvalsi di una scheda grafica GeForce GTX 275, le cui caratteristiche principali sono:

```
Name: GeForce GTX 275 (32 bit)
Vendor: NVIDIA Corporation
Max compute units: 30
Max work item sizes: [512, 512, 64]
Max work group size: 512
Max clock frequency: 1.40 GHz
Max memory allocation size: 223 Mb
Global memory size: 895 Mb
Max constant buffer size: 64 Kb
Max constant arguments: 9
Local memory size: 16 Kb
```

Queste caratteristiche sono state rilevate mediante un semplice programma diagnostico che sfrutta una serie di chiamate API OpenCL per interrogare i device; i dati riportati sopra sono il risultato della sua esecuzione sui due device.

Si noti come la CPU abbia soltanto 2 core operanti in modalità a 64 bit, rispetto alle 30 CU a 32 bit della GPU; tuttavia quest'ultime sono piuttosto specializzate per programmi che sfruttano il paradigma SIMD o SPMD mentre le unità di calcolo della CPU sono state concepite per un uso più generale. La frequenza dei core della CPU è leggermente superiore a quella delle CU della GPU.

Il parametro `Global memory size` indica la quantità di Global Memory disponibile per un singolo kernel: la dotazione della CPU è decisamente migliore rispetto a quella della GPU. Un altro parametro interessante è la dimensione massima per l'allocazione di una singola variabile: la GPU è limitata a 223 Mb, perciò i test riportati di seguito si limiteranno al raggiungimento di questa soglia.

Anche in ambito software la scelta di questi due device ha determinato una notevole eterogeneità: i test su CPU si sono necessariamente avvalsi dell'implementazione OpenCL di AMD mentre per la GPU è stata usata quella della NVIDIA Corporation (vedi sezione 2.6).

7.1 Input/output

I test di banda riguardano le operazioni di lettura e scrittura nella Global Memory, sia della CPU che della GPU; ovviamente non hanno alcun senso per il programma seriale, basato su un modello di programmazione differente.

La dimensione della chiave AES non influisce sulle operazioni di scrittura dei dati da cifrare nella Global Memory del device né sulla successiva lettura del risultato.

Le tabelle 7.1 e 7.2 riportano i risultati dei test rispettivamente in lettura e scrittura. È evidente che il costo sia della lettura che della scrittura è pressoché esponenziale rispetto alla grandezza dei dati da cifrare: si tratta di un overhead che non è possibile eliminare mediante artifici a livello di scrittura del codice sorgente OpenCL. Nei grafici 7.1 e 7.3 sono rappresentati i risultati dei test su CPU mentre quelli su GPU sono nei grafici 7.2 e 7.4.

7.2 AES

I test prestazionali più importanti sono sicuramente quelli che coinvolgono il codice che implementa AES. I rilevamenti sono stati effettuati per la versione seriale e quella OpenCL (sia su CPU che su GPU), per ciascuna lunghezza di chiave, sia per la cifratura che per la decifratura. I dati riportati nella tabella 7.3 e rappresentati nel grafico 7.5 sono una sintesi dei risultati dei test basata su questi due criteri:

1. sia la cifratura che la decifratura hanno lo stesso andamento in quanto le operazioni da svolgere sono del tutto simili;

2. le tre lunghezze di chiave AES aumentano il numero di round allo stesso modo per tutte le implementazioni; pertanto, fissata la scelta tra AES seriale o PAES su CPU o GPU la lunghezza della chiave non influisce sull'andamento dei risultati.

Per il primo motivo sono riportati soltanto i risultati della cifratura, mentre per il secondo sono stati aggregati tramite media aritmetica i rilevamenti di ciascuna implementazione per tutte e tre le lunghezze di chiave.

Le tradizionali misure basate sullo speed-up non sono applicabili, in generale, per due motivi:

1. OpenCL non consente di controllare direttamente il numero di unità di calcolo che il programma può usare;

2. il programma è stato eseguito su dispositivi eterogenei.

Per dare un'idea dei rapporti tra le prestazioni delle varie implementazioni, nella tabella 7.4 sono riportate tre misure:

1. il rapporto dei tempi d'esecuzione tra le versioni OpenCL su CPU e GPU;

2. il rapporto dei tempi d'esecuzione tra la versione seriale e quella OpenCL su GPU;

3. il rapporto dei tempi d'esecuzione tra la versione seriale e quella OpenCL su CPU;

Sulla base di questi indici è possibile fare alcune osservazioni; ad esempio:

1. PAES su CPU è sempre più veloce del programma seriale;

2. PAES su GPU è più veloce del programma seriale a partire da input di dimensione 512 Kb;

3. PAES su GPU è più veloce di PAES su CPU a partire da input di 8 Mb.

In generale, all'aumentare della dimensione dei dati da cifrare, PAES su GPU è la versione più performante; per input di piccole dimensioni invece l'overhead dovuto alle operazioni di lettura e scrittura sulla memoria della GPU supera il guadagno nei tempi di calcolo.

Dimensione (byte)	**CPU (ms)**	**GPU (ms)**
128	0.000	0.003
256	0.001	0.003
512	0.001	0.003
1024	0.001	0.004
2048	0.002	0.004
4096	0.002	0.005
8192	0.007	0.007
16384	0.015	0.010
32768	0.032	0.017
65536	0.060	0.030
131072	0.122	0.108
262144	0.237	0.202
524288	0.466	0.387
1048576	1.147	0.774
2097152	2.433	1.172
4194304	4.716	2.001
8388608	9.434	3.593
16777216	18.621	6.892
33554432	37.023	13.436
67108864	73.977	26.475
134217728	162.418	52.453
234700800	267.740	91.343

Tabella 7.1: Tempi di lettura dalla Global Memory.

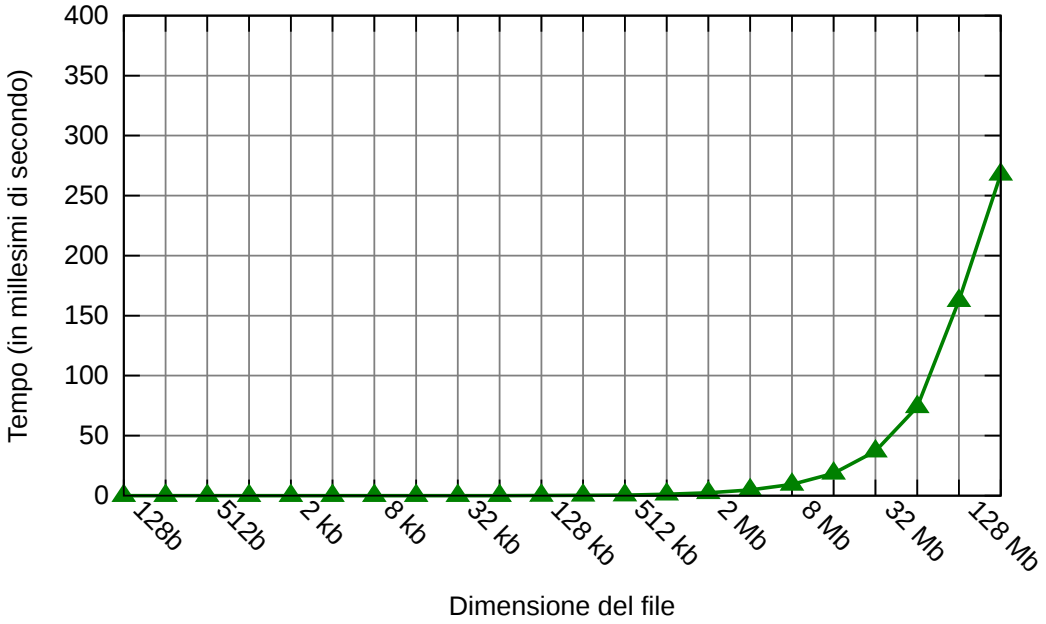

Figura 7.1: Tempi di lettura dalla Global Memory della CPU.

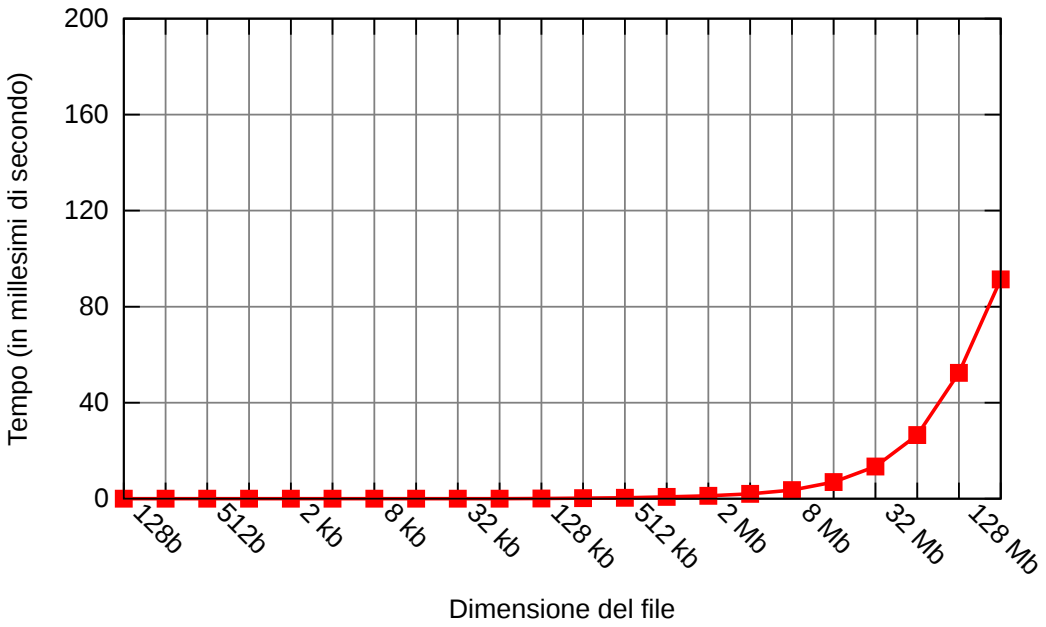

Figura 7.2: Tempi di lettura dalla Global Memory della GPU.

Dimensione (byte)	**CPU (ms)**	**GPU (ms)**
128	0.003	0.018
256	0.004	0.020
512	0.004	0.021
1024	0.004	0.021
2048	0.004	0.021
4096	0.007	0.022
8192	0.010	0.026
16384	0.018	0.034
32768	0.032	0.046
65536	0.051	0.076
131072	0.096	0.130
262144	0.182	0.246
524288	0.370	0.483
1048576	0.759	0.952
2097152	1.875	1.560
4194304	3.881	2.764
8388608	7.219	5.121
16777216	13.750	9.861
33554432	26.883	19.291
67108864	53.088	38.644
134217728	107.619	76.040
234700800	184.678	132.751

Tabella 7.2: Tempi di scrittura nella Global Memory.

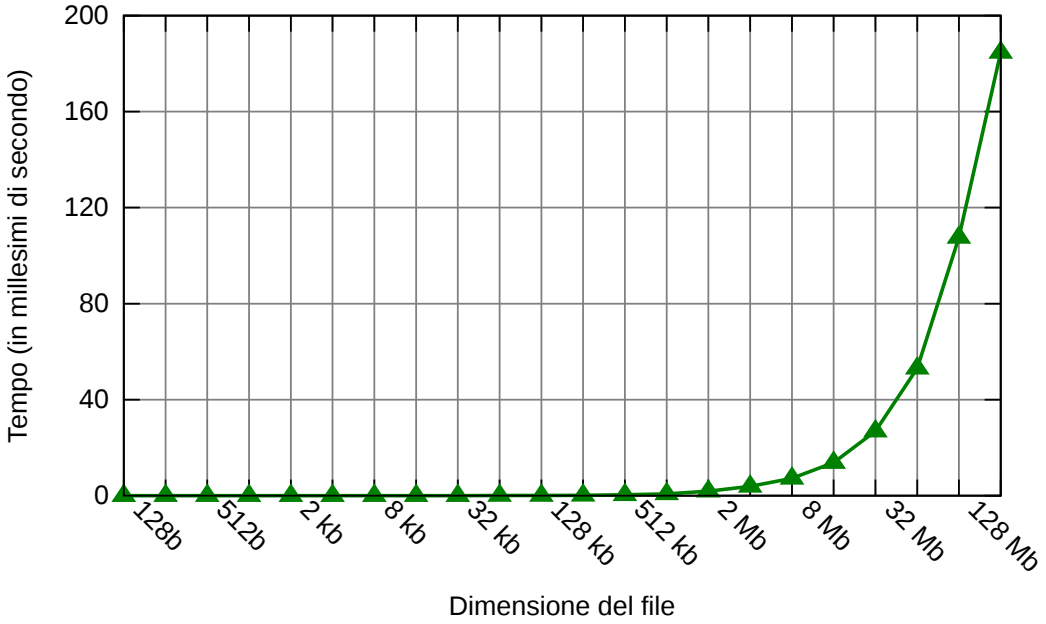

Figura 7.3: Tempi di scrittura nella Global Memory della CPU.

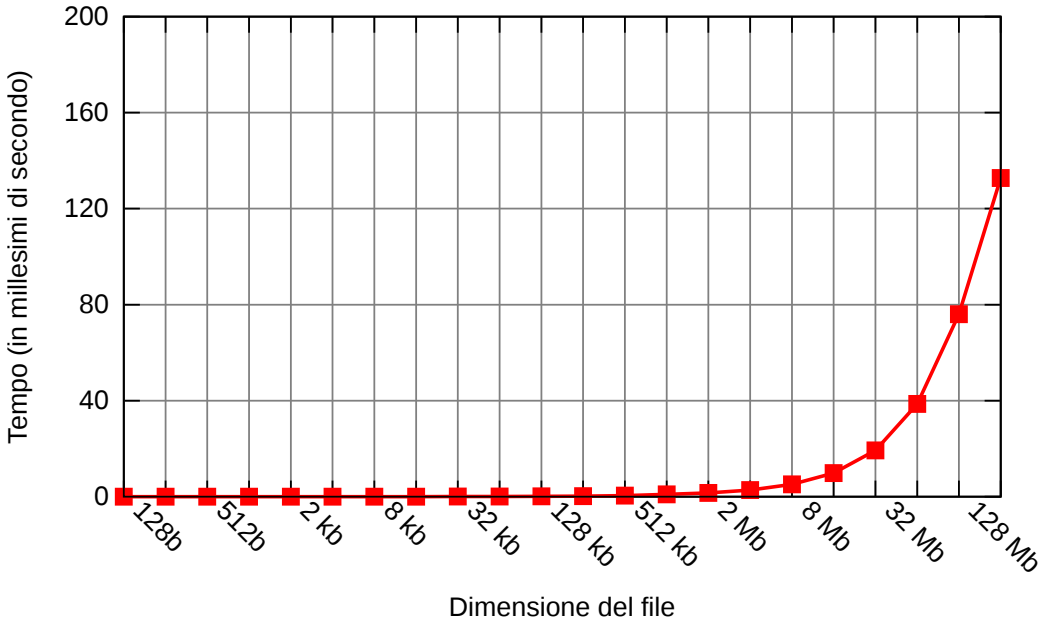

Figura 7.4: Tempi di scrittura nella Global Memory della GPU.

Dimensione (byte)	**CPU (ms)**	**GPU (ms)**	**Seriale**
128	0.075	0.705	0.217
256	0.104	0.722	0.426
512	0.160	0.751	0.846
1024	0.274	0.801	1.752
2048	0.286	0.905	3.415
4096	0.524	0.911	6.754
8192	0.976	0.905	13.610
16384	1.883	0.948	27.128
32768	3.678	1.022	54.313
65536	7.292	1.931	113.525
131072	14.492	2.936	217.586
262144	30.559	4.955	436.502
524288	57.661	8.893	869.043
1048576	115.667	16.840	1741.804
2097152	230.564	33.398	3484.572
4194304	462.034	66.055	6995.495
8388608	925.355	131.624	13995.803
16777216	1850.096	262.747	27884.374
33554432	3700.182	524.854	55445.609
67108864	7697.433	1049.197	111911.418
134217728	15285.159	2806.445	222053.770
234700800	26178.837	6687.997	386443.750

Tabella 7.3: Tempi di cifratura.

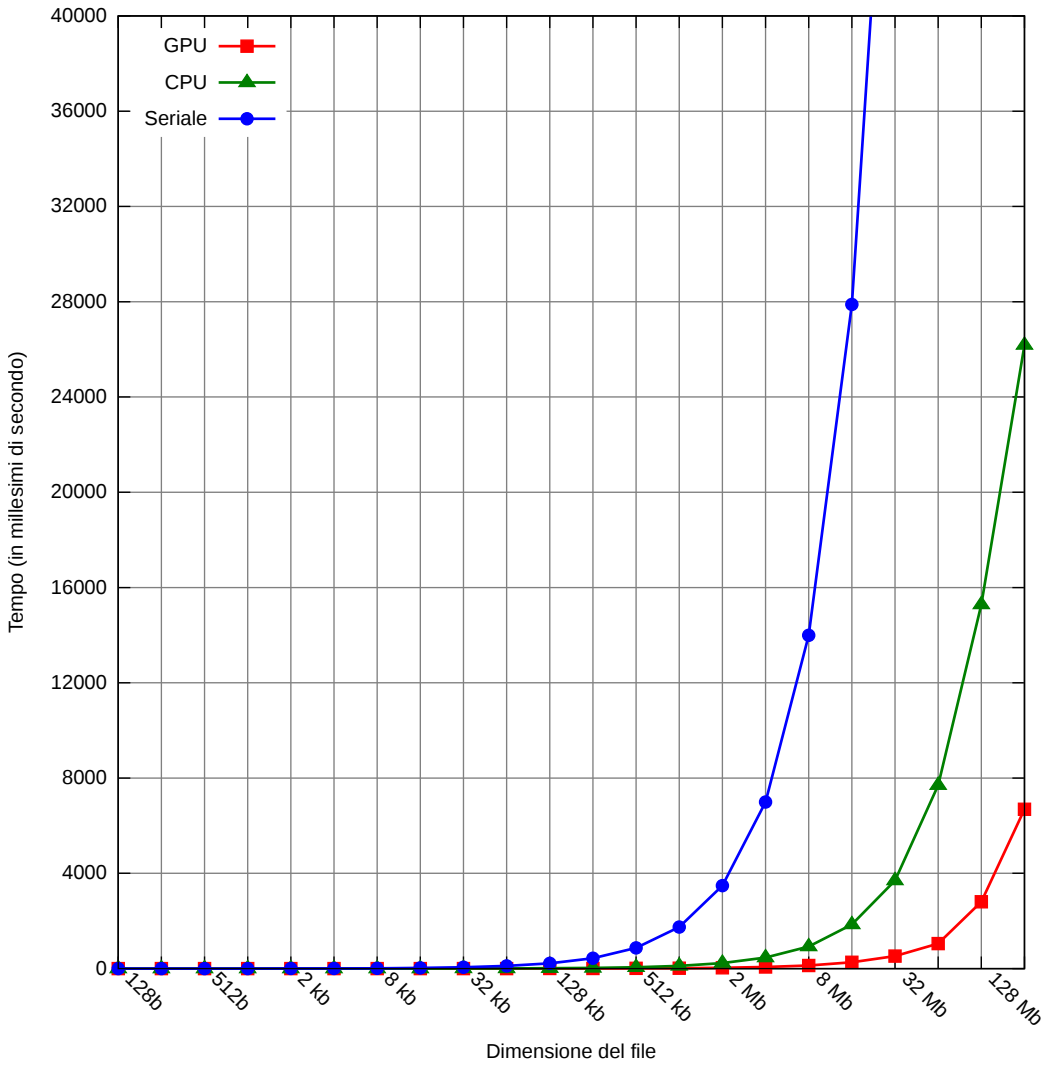

Figura 7.5: Tempi di cifratura.

Dimensione (byte)	**GPU/CPU**	**GPU/Seriale**	**CPU/Seriale**
128	0.106	0.308	2.893
256	0.144	0.590	4.096
512	0.213	1.126	5.287
1024	0.342	2.187	6.394
2048	0.316	3.773	11.941
4096	0.575	7.414	12.889
8192	1.078	15.039	13.945
16384	1.986	28.616	14.407
32768	3.599	53.144	14.767
65536	3.776	58.791	15.568
131072	4.936	74.110	15.014
262144	6.167	88.093	14.284
524288	6.484	97.722	15.072
1048576	6.869	103.433	15.059
2097152	6.904	104.335	15.113
4194304	6.995	105.904	15.141
8388608	7.030	106.332	15.125
16777216	7.041	106.126	15.072
33554432	7.050	105.640	14.985
67108864	7.336	106.664	14.539
134217728	5.446	79.123	14.527
234700800	3.914	57.782	14.762

Tabella 7.4: Rapporti vari tra i tempi di cifratura.

Conclusioni

Lo sviluppo di PAES è stata un'ottima occasione per esplorare l'uso delle architetture multi/many-core nell'ambito di un problema reale e, in particolare, l'uso di OpenCL. L'esperienza accumulata, oltre ad avallare previsioni ottimistiche sulla crescita di queste tecnologie, può essere in buona parte riutilizzata in progetti futuri, soprattutto nell'ambito dell'esperimento MaCGO.

Lo sviluppo di programmi paralleli con OpenCL ha una serie di complicazioni: quelle intrinseche nella programmazione parallela e quelle derivanti dalla diversa cura che i vari produttori ripongono nelle loro implementazioni. Per queste ragioni è stato fondamentale sviluppare PAES usando implementazioni OpenCL differenti contemporaneamente e corredarlo di una serie di test la cui semplicità d'uso ha reso possibile eseguirli molto spesso.

Nonostante l'uso di OpenCL consenta di scrivere programmi in grado di utilizzare in modo trasparente diverse tipologie di dispositivi di calcolo, è stato necessario porre particolare attenzione per sfruttare al meglio le peculiarità degli stessi. Il tipo di adattamento descritto nella sottosezione 5.1.2 è un modello da seguire anche per altri progetti basati su OpenCL. Sempre al fine di rendere un programma OpenCL in grado sfruttare al meglio dispositivi eterogenei, è fondamentale strutturarne il parallelismo in modo flessibile e privo di vincoli non strettamente necessari, ove possibile.

Grazie alla natura modulare di PAES, il suo codice potrà essere facilmente integrato e riutilizzato in altri progetti.

Bibliografia

[1] *Federal Information Processing Standards Publication 197.* National Institute of Standards and Technology, 2001.

[2] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 1965.

[3] Aaftab Munshi. *The OpenCL Specification 1.0.* Khronos OpenCL Working Group, 2009.

[4] *NVIDIA OpenCL Best Practices Guide.* NVIDIA Corporation, 2009.

[5] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. *Report on the Development of the Advanced Encryption Standard (AES).* National Institute of Standards and Technology, 2000.

[6] Douglas R. Stinson. *Cryptography Theory and Practice.* Chapman & Hall/CRC, 2006.

[7] Niels Ferguson and Bruce Schneier. *Practical Cryptography.* John Wiley & Sons, 2003.

[8] Alex Biryukov and Dmitry Khovratovich. Related-key cryptanalysis of the full aes-192 and aes-256. 2009.

[9] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key recovery attacks of practical complexity on aes variants with up to 10 rounds. 2009.

[10] Erdogmus, Hakan, Morisio, and Torchiano. On the effectiveness of test-first approach to programming. In *Proceedings of the IEEE Transactions on Software Engineering, 31*, 2005.

Elenco delle tabelle

Elenco dei codici

Elenco delle figure