POLITECNICO
MILANO 1863

089020 - Progetto di Ingegneria Informatica

Post Quantum Cryptography

Relazione Finale

Autore: Stefano Formicola
Docente supervisore: Alessandro Barenghi
Docente responsabile: Stefano Zanero

Anno Accademico 2018/2019

Indice

1 Introduzione

Il progetto Post Quantum Cryptography inizia a Ottobre 2018, col professor **Alessandro Barenghi** del DEIB (Dipartimento di Elettronica, Informazione e Bioingegneria) come docente supervisore.

1.1 LEDAcrypt

Il progetto è basato sull'implementazione pubblica di **LEDAcrypt**, un algoritmo crittografico a chiave pubblica la cui robustezza e confidenzialità sono garantite dalla difficoltà della decodifica, che appartiene alla classe dei problemi *NP-hard*. Questo tipo di problemi risulta difficile da risolvere anche con computer quantistici, macchine che sfruttano i fenomeni della meccanica quantistica per risolvere problemi matematici difficili o intrattabili per i computer tradizionali, come la fattorizzazione degli interi.

Quest'ultimo, ad esempio, è alla base dell'infrastruttura crittografica a chiave pubblica moderna e, in futuro, computer quantistici sufficientemente potenti saranno capaci di comprometterne il funzionamento, minando la confidenzialità e l'integrità delle comunicazioni digitali.

1.2 Scopo del progetto

L'obiettivo didattico del progetto consiste nell'ottimizzazione della funzione di decoding, scritta in linguaggio C, per architetture di processori x86-64 che supportano il set di istruzioni SIMD Advanced Vector Extensions 2 (AVX2), sfruttando le *compiler intrinsics* di Intel.

Inoltre, il progetto mi ha consentito di migliorare ed approfondire la capacità di comprendere ed analizzare del codice prodotto da altri autori nonché ragionare su come migliorarlo ed ottimizzarlo, sfruttando la possibilità di eseguire determinati calcoli in parallelo su registri a 256 bit del processore.

2 L'algoritmo

L'algoritmo da ottimizzare rappresenta una procedura di decoding conosciuta col nome di *Bit Flipping (BF)*. Gli algoritmi di Bit Flipping eseguono delle iterazioni su una matrice H, detta di parity-check, per dedurre quali bit di una parola affetta da errori devono essere invertiti in modo che la decodifica abbia successo.

La procedura di riferimento di questo progetto consiste in un algoritmo QDecode che lavora su un vettore s detto sindrome, la matrice sparsa H e la matrice di trasformazione Q. Le operazioni compiute su questi parametri in ingresso sono finalizzate a ricostruire il vettore d'errore e della sindrome s.

Il collo di bottiglia dell'implementazione iniziale consisteva nelle prime istruzioni dell'algoritmo, ovvero quelle in cui QDecode calcola il numero di unsatisfied parity checks presenti nella sindrome s e di seguito rappresentate.

Input: s: sindrome, vettore di dimensione p
Htr: rappresentazione in forma sparsa della matrice trasposta H

```
1  upc ← [0|...|0] ; // upc di dimensione N_0 * p
2  currSyndrome ← s;
3  for i = 0 to n_0 − 1 do
4  |  for valueIdx = 0 to p − 1 do
5  |  |  for HtrIdx = 0 to d_v − 1 do
6  |  |  |  if get_coeff(currSyndrome, (Htr[i][HtrIdx]+valueIdx mod p) = 1 then
7  |  |  |  |  upc[i * p + valueIdx] = upc[i * p + valueIdx] + 1;
8  |  |  |  end
9  |  |  end
10 |  end
11 end
```

Algorithm 1: Calcolo degli UPC della sindrome s

L'idea alla base dell'ottimizzazione consiste nello sfruttare il fatto che gli *upc* sono rappresentati in 8 bit, e quindi è possibile eseguire un caricamento vettoriale di 8 *unsatisfied parity checks* nei moderni processori con parole di memoria da 64 bit. Per implementarla, la matrice *upc* passa da una dimensione $n_0 * p$ a $n_0 * p'$ con p' uguale a p approssimato al più vicino multiplo del numero di byte della parola di memoria del processore sfruttata per vettorizzare il calcolo (8 B per i processori a 64 bit). Nel caso dei processori AVX2, è stato possibile sfruttare registri del processore della dimensione di 256 bit (32 Byte). Inoltre, la funzione *get_coeff* restituirà una rappresentazione "impacchettata" dei bit corrispondenti ai coefficienti

nel range dall'indice passato per parametro ai successivi 64 (o, nell'implementazione AVX2, 256).

Una prima versione ottimizzata dell'algoritmo è rappresentata dallo pseudocodice seguente.

```
1  upc ← [0|...|0] ; // upc di dimensione N_0 * p'
2  currSyndrome[1] ← s;
3  currSyndrome[0] ← s[p − 1];
4  for i = 0 to n_0 − 1 do
5  |  for valueIdx = 0 to p − 1 do
6  |  |  for HtrIdx = 0 to d_v − 1 do
7  |  |  |  basePos = Htr[i][HtrIdx]+valueIdx mod p;
8  |  |  |  packedSynBits ←get_64_coeff(currSyndrome, basePos)
   |  |  |   for upcMatRow = 0 to 7 do
9  |  |  |  |  upcMat[upcMatRow] ←packedSynBits &VEC_COMB_MASK;
10 |  |  |  |  packedSynBits = packedSynBits >> 1 ;
11 |  |  |  end
12 |  |  |  upcMat ← transpose(upcMat);
13 |  |  |  for upcMatRow = 0 to 7 do
14 |  |  |  |  upc[i][valueIdx+8*upcMatCol] ← upcMat[upcMatRow];
15 |  |  |  end
16 |  |  end
17 |  |  valueIdx ← valueIdx + VECTORIZED_READ_BITS;
18 |  end
19 end
20 for i = 0 to n_0 − 1 do
21 |  for j = 0 to 31 do
22 |  |  upc[i][p+j] = upc[i][j];
23 |  end
24 end
```

Algorithm 2: Algoritmo 1 vettorizzato a 64 bit

A partire dall'Algoritmo 2 è possibile ottenere un'ottimizzazione 4 volte superiore se, invece di calcolare 64 bit di indici, se ne calcolano 256 bit, sfruttando le operazioni vettoriali offerte dalle Intrinsics di Intel per i processori della famiglia AVX2.

Così, è possibile caricare in un registro __m256 quattro parole della sindrome, su cui poi è possibile svolgere operazioni logiche ed aritmetiche. Un esempio è l'operazione aritmetica di modulo, la quale nelle versioni non ottimizzate è implementata come

una serie di divisioni con resto. Una versione ottimizzata e che sfrutta gli operatori logici, invece, è la seguente:

```
unsigned int logicalModulo(uint32_t a, uint32_t b) {
    return a - ( ( ( (signed int)(b) - (signed int)(a) ) >> 31) & b ) ;
}
```

Il corrispettivo vettorizzato, invece, è mostrato nel codice seguente e calcola il modulo b di 4 interi a 32 bit contenuti nel registro a:

```
__m256i avx2_epi32_Modulo(__m256i a, uint32_t b) {
   __m256i base = _mm256_set1_epi32(b);

   __m256i t0   = _mm256_sub_epi32(base, a);
   __m256i t1   = _mm256_srai_epi32(t0, 0x1F);
   __m256i t2   = _mm256_and_si256 (t1, base);

   return _mm256_sub_epi32(a, t2);
}
```

Complessivamente, le istruzioni vettoriali usate per l'implementazione del progetto sono istruzioni di INSERT ed EXTRACT nei registri a 256 bit, AND e OR vettoriale, SHIFT intra-word dei registri, LOAD e STORE vettoriale e operazioni aritmetiche come la ADD e la SUB.

3 Trasposta di una matrice AVX2

Ad ogni calcolo degli UPCs, questi andranno salvati nella matrice di UPC nel giusto ordine. Essenzialmente, la matrice di UPCs deve essere trasposta e poi ogni riga sarà salvata nell'opportuna posizione del vettore *unsatisfiedParityChecks*.

Per ottenere ciò ho utilizzato le intrinsics di permutazione dei registri. La situazione iniziale si presenta come in figura 1, dove ad ogni alternanza di colore corrisponde un registro, letto da sinistra verso destra dall'alto in basso.

1	2	3	4	5	6	7	8	
9	10	11	12	13	14	15	16	row0
17	18	19	20	21	22	23	24	
25	26	27	28	29	30	31	32	
33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	row1
49	50	51	52	53	54	55	56	
57	58	59	60	61	62	63	64	
65	66	67	68	69	70	71	72	
73	74	75	76	77	78	79	80	row2
81	82	83	84	85	86	87	88	
89	90	91	92	93	94	95	96	
97	98	99	100	101	102	103	104	
105	106	107	108	109	110	111	112	row3
113	114	115	116	117	118	119	120	
121	122	123	124	125	126	127	128	
129	130	131	132	133	134	135	136	
137	138	139	140	141	142	143	144	row4
145	146	147	148	149	150	151	152	
153	154	155	156	157	158	159	160	
161	162	163	164	165	166	167	168	
169	170	171	172	173	174	175	176	row5
177	178	179	180	181	182	183	184	
185	186	187	188	189	190	191	192	
193	194	195	196	197	198	199	200	
201	202	203	204	205	206	207	208	row6
209	210	211	212	213	214	215	216	
217	218	219	220	221	222	223	224	
225	226	227	228	229	230	231	232	
233	234	235	236	237	238	239	240	row7
241	242	243	244	245	246	247	248	
249	250	251	252	253	254	255	256	

Figura 1: Situazione iniziale della matrice di UPC, rappresentata nell'implementazione come 8 vettori da 256 bit, ovvero 8x32 uint8_t

La prima operazione per ottenere la matrice trasposta consiste nell'istruzione:

```
__m256  __t0  = _mm256_unpacklo_ps(row0, row1);
__m256  __t1  = _mm256_unpackhi_ps(row0, row1);
```

Ovvero l'interleaving della metà inferiore (e, nella seconda istruzione, superiore) di ciascun registro di 128 bit dei due operandi, il cui risultato è mostrato in figura 2.

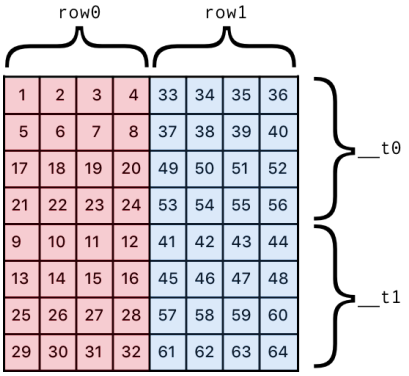

Figura 2: Interleaving di row0 e row1

La stessa operazione è stata applicata alle coppie di registri (row2, row3), (row4, row5), (row6, row7). In figura 3 è mostrato il contenuto di __t2 e __t3.

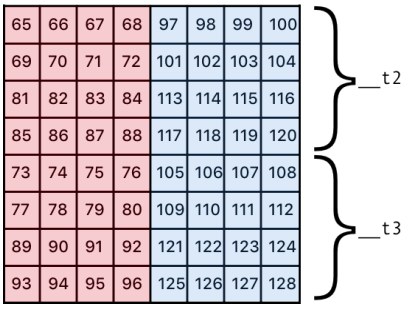

Figura 3: Interleaving di row2 e row3

Prendendo come riferimento i registri __t0, __t1, __t2, __t3, al termine dell'interleaving i valori sono riordinati con la intrinsic *_mm256_shuffle_ps*, la quale li tratta come elementi da 32 bit.

```
__m256  __tt0 = _mm256_shuffle_ps(__t0,__t2, 0x44);
__m256  __tt1 = _mm256_shuffle_ps(__t0,__t2, 0xEE);
__m256  __tt2 = _mm256_shuffle_ps(__t1,__t3, 0x44);
__m256  __tt3 = _mm256_shuffle_ps(__t1,__t3, 0xEE);
```

La prima shuffle produce un risultato intermedio che alterna 64 bit inferiori (partendo dai primi 128 bit) del primo operando all'altro. Il risultato della seconda istruzione, invece, contiene i 64 bit superiori (partendo dei primi 128 bit) del primo operando alternati a quelli del secondo.

I risultati intermedi delle prime 6 shuffle sono mostrati nelle figure 4, 5, 6.

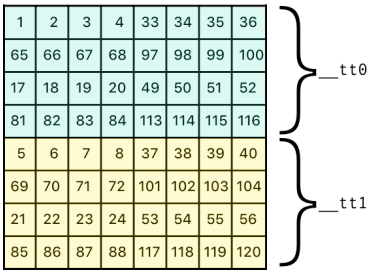

Figura 4: _mm256_shuffle_ps(__t0,__t2) con maschere 0x44 e 0xEE

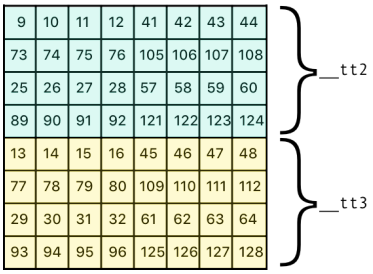

Figura 5: _mm256_shuffle_ps(__t1,__t3) con maschere 0x44 e 0xEE

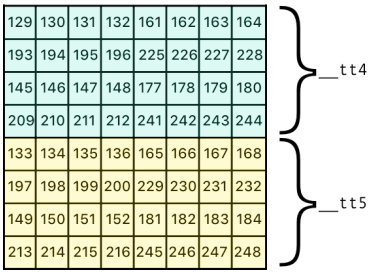

Figura 6: _mm256_shuffle_ps(__t4,__t6) con maschere 0x44 e 0xEE

Il passo finale per ottenere la trasposta della matrice di __m256i è composto da altre due funzioni sui registri intermedi ottenuti finora.

```
row0 = _mm256_permute2f128_ps(__tt0, __tt4, 0x20);
row1 = _mm256_permute2f128_ps(__tt1, __tt5, 0x20);
row2 = _mm256_permute2f128_ps(__tt2, __tt6, 0x20);
row3 = _mm256_permute2f128_ps(__tt3, __tt7, 0x20);
row4 = _mm256_permute2f128_ps(__tt0, __tt4, 0x31);
row5 = _mm256_permute2f128_ps(__tt1, __tt5, 0x31);
row6 = _mm256_permute2f128_ps(__tt2, __tt6, 0x31);
row7 = _mm256_permute2f128_ps(__tt3, __tt7, 0x31);
```

Le istruzioni di permutazione con maschera 0x20 compongono un registro da 256 bit con i 128 bit di parte alta del secondo operando e 128 bit di parte bassa del primo.

La maschera 0x31, invece, permette di comporre un registro il cui risultato contiene i 128 bit di parte alta del primo operando come parte bassa e 128 bit di parte alta del secondo operando come parte alta.

Il risultato, per il registro row0, è illustrato in figura 7.

1	2	3	4	33	34	35	36
65	66	67	68	97	98	99	100
129	130	131	132	161	162	163	164
193	194	195	196	225	226	227	228

Figura 7: _mm256_permute2f128_ps(__tt0,__tt4) con maschere 0x20

La trasposta della matrice, trattata come una 8x8 float, è finita. Ora occorre solo permutare gli elementi interni ad ogni registro per ottenere la vera matrice trasposta, una 8x32 unsigned char. La funzione che permette di ottenere questo risultato è la seguente, il cui risultato è mostrato in figura 8.

```
static inline __m256i permute_row(__m256i row){
   row = _mm256_shuffle_epi8(row, _mm256_set_epi8(15, 11, 7, 3,
                                                  14, 10, 6, 2,
                                                  13,  9, 5, 1,
                                                  12,  8, 4, 0,
                                                  15, 11, 7, 3,
                                                  14, 10, 6, 2,
                                                  13,  9, 5, 1,
                                                  12,  8, 4, 0));
   row = _mm256_permutevar8x32_epi32(row, _mm256_set_epi32(7,3,6,2,5,1,4,0));
   return row;
}
```

1	2	3	4	33	34	35	36
65	66	67	68	97	98	99	100
129	130	131	132	161	162	163	164
193	194	195	196	225	226	227	228

Figura 8: permute_row(row0)

4 Risultati

I risultati del progetto, in termini di performance, sono piuttosto notevoli e mostrano un decremento dei tempi fino a 20 volte rispetto all'implementazione originale.

Nella tabella di seguito sono rappresentati, per ogni categoria di sicurezza e valori di N0, i tempi in millisecondi di un benchmark di 100 test, compresa la varianza.

Alla colonna **Bench 0** corrisponde l'implementazione originale dell'algoritmo, senza ottimizzazione, mentre **Bench 1** rappresenta i risultati del benchmark eseguito sulla prima implementazione vettorizzata dell'algoritmo, a 64 bit.

Category	N0	Bench 0	Bench 1
1	2	3.617 (+,- 0.491)	0.833 (+,- 0.141)
1	3	4.438 (+,- 0.383)	1.020 (+,- 0.111)
1	4	4.629 (+,- 0.492)	1.087 (+,- 0.161)
3	2	9.694 (+,- 0.934)	2.072 (+,- 0.271)
3	3	16.098 (+,- 1.023)	2.106 (+,- 0.252)
3	4	18.327 (+,- 1.029)	2.320 (+,- 0.180)
5	2	13.487 (+,- 0.708)	2.703 (+,- 0.222)
5	3	19.215 (+,- 1.612)	3.344 (+,- 0.344)
5	4	20.009 (+,- 1.468)	3.732 (+,- 0.397)

Per quanto riguarda l'implementazione finale, ottimizzata per le architetture AVX2, sono state prese in considerazione due implementazioni dell'algoritmo.
La prima, **Bench 2**, fa uso del modulo aritmetico per calcolare singolarmente gli indici rimanenti della sindrome, non multipli del fattore di coalescenza 256.
La seconda implementazione, **Bench 3**, calcola il modulo di questi indici inserendoli in un unico registro __m256i e poi ne esegue le operazioni, effettuando così più operazioni di insert.

Category	N0	Bench 2	Bench 3
1	2	0.183 (+,- 0.021)	0.176 (+,- 0.032)
1	3	0.290 (+,- 0.047)	0.433 (+,- 0.054)
1	4	0.532 (+,- 0.065)	1.509 (+,- 0.082)
3	2	0.468 (+,- 0.078)	0.456 (+,- 0.066)
3	3	0.490 (+,- 0.050)	0.429 (+,- 0.037)
3	4	1.286 (+,- 0.106)	1.305 (+,- 0.098)
5	2	0.762 (+,- 0.109)	0.674 (+,- 0.054)
5	3	0.721 (+,- 0.059)	1.343 (+,- 0.103)
5	4	1.055 (+,- 0.100)	2.084 (+,- 0.132)

5 Competenze acquisite

I risultati didattici del progetto sono stati numerosi e mi hanno permesso di approfondire meglio diverse fasi dello sviluppo di un progetto in C.

Ovviamente, grande importanza è attribuita alla capacità di "vettorizzare" un algoritmo e a comprendere le diverse istruzioni offerte dalle *compiler intrinsics* di Intel, riuscendo anche a combinare istruzioni appartenenti a famiglie di processori più vecchie, come le SSE. Altri tipi di ottimizzazioni, che ho imparato durante l'implementazione AVX2 dell'algoritmo, sono legate al tipo e al numero di bit scelti per rappresentare i dati in memoria, riuscendo ad aggregare fino a 32 iterazioni di un for-loop su dati a 8 bit in un'unica istruzione operante su registri a 256 bit.

Altre competenze che ho acquisito sin dall'inizio del progetto riguardano l'utilizzo delle direttive al preprocessore come la define di una macro, le direttive condizionali o il token-pasting operator, ma anche i benefici legati all'utilizzo dell'inlining delle funzioni. Un altro strumento molto utile e che prima dell'inizio del progetto non avevo mai usato è *GNU make*, col quale ho avuto modo di modificare diversi parametri legati alla fase di compilazione, sia in fase di debug che di rilascio del codice.

Una fase molto importante in fase di sviluppo dell'implementazione è stata quella di debug: ho avuto modo di approfondire l'utilizzo di diversi suite per il debug come *Valgrind* ed il suo strumento per il profiling *Callgrind*, il cui output è stato visualizzato con Qcachegrind per macOS. Avendo lavorato al progetto interamente da macOS, inizialmente ho riscontrato difficoltà nell'installare Valgrind sull'ultima versione dell'OS, Mojave, a causa dell'incompatibilità con alcune librerie a 32-bit, il cui supporto è cessato. Questo mi ha portato a studiare ed imparare a configurare un ambiente di sviluppo Linux basato su Docker, utile per le prime fasi di compilazione e debug da linea di comando, riducendo i tempi necessari per l'avvio di una VM da VirtualBox. Successivamente, ho avuto modo di interagire con la community di Homebrew su GitHub e di risolvere una issue per l'installazione di Valgrind su macOS, riuscendo a fare debugging senza l'utilizzo di un Docker container.

Uno strumento molto utile e degno di nota, che ho avuto modo di apprezzare imparando ad usarlo, è il *debugger LLDB*, attraverso il quale ho potuto confrontare i valori in bit degli unsatisfied parity checks ed altri parametri del decoder tra la versione di riferimento e quella in fase di sviluppo, così come ho potuto analizzare il codice una riga alla volta attraverso l'uso semplificato dei breakpoint, condizionali e non.