LEA Competition 2014
High Performance Implementations of LEA

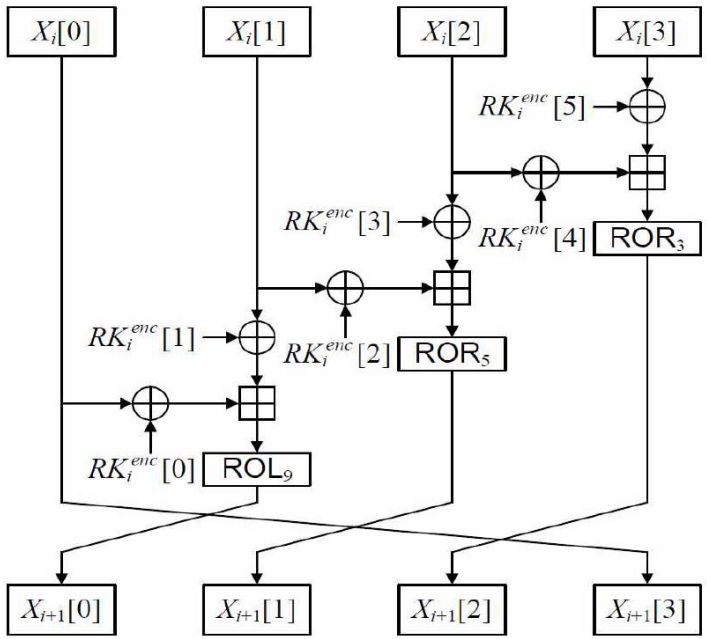
Hwajeong Seo Jihyun Kim Hanju Seo Howon Kim

Pusan National University

2014/06/12

Target Block Cipher

- **LEA**
  - ARX-based block cipher(Addition,Rotation,eXclusive-or)
  - 32-bit wise word size
  - Single architecture for 128-, 192-, 256-bit



Target Devices

- **8-bit Embedded Platform AVR**
  - 8-bit processor, 7.3728 MHz
  - 128KB EEPROM, 4KB RAM, 32 registers

Mnemonics	Operands	Description	Operation	#Clock
ADD	Rd, Rr	Add without Carry	Rd ← Rd+Rr	1
ADC	Rd, Rr	Add with Carry	Rd ← Rd+Rr+C	1
EOR	Rd, Rr	Exclusive OR	Rd ← Rd⊕Rr	1
LSL	Rd	Logical Shift Left	C|Rd ← Rd<<1	1
LSR	Rd	Logical Shift Right	Rd|C ← 1>>Rd	1
ROL	Rd	Rotate Left Through Carry	C|Rd ← Rd<<1||C	1
ROR	Rd	Rotate Right Through Carry	Rd|C ← C||1>>Rd	1
LD	Rd, X	Load Indirect	Rd ← (X)	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	2

Target Devices

- **16-bit Embedded Platform MSP**
  - 16-bit processor, 8 MHz
  - 32-48KB flash memory, 10KB RAM, 12 registers

Mnemonics	Operands	Description	Operation	#Clock
ADD	Rr, Rd	Add without Carry	Rd ← Rd+Rr	1
ADDC	Rr, Rd	Add with Carry	Rd ← Rd+Rr+C	1
AND	Rr, Rd	Logical AND	Rd ← Rd&Rr	1
XOR	Rr, Rd	Exclusive OR	Rd ← Rd⊕Rr	1
RLA	Rd	Logical Shift Left	C|Rd ← Rd<<1	1
RRA	Rd	Logical Shift Right	Rd|C ← 1>>Rd	1
RLC	Rd	Rotate Left Through Carry	C|Rd ← Rd<<1||C	1
RRC	Rd	Rotate Right Through Carry	Rd|C ← C||1>>Rd	1

Target Devices

- **32-bit Embedded Platform ARM**
  - 32-bit processor, $1 \sim 2$ GHz
  - 16 registers, most instructions in a single cycle

Mnemonics	Operands	Description	Operation	#Clock
ADD	Rd, Rr	Add without Carry	Rd ← Rd+Rr	1
EOR	Rd, Rr	Exclusive OR	Rd ← Rd⊕Rr	1
ROL	Rd	Rotate Left Through Carry	C|Rd ← Rd<<1||C	1
ROR	Rd	Rotate Right Through Carry	Rd|C ← C||1>>Rd	1

Target Devices

- **SIMD Embedded Platform NEON**
  - SIMD extensions on ARM
  - sixteen 128-bit registers
  - packed 8, 16, 32, 64-bit operations

Mnemonics	Operands	Description	Cycles	Source	Result	Writeback
VADD	Qd,Qn,Qm	Vector Addition	1	-,2,2	3	6
VEOR	Qd,Qn,Qm	Vector Exclusive-or	1	-,2,2	3	6
VSHL	Qd,Qm,#imm	Vector Left Shift	1	-,2,-	3	6
VSRI	Qd,Qm,#imm	Vector Right Shift with Insert	2	2,2,-	4	7

Target Devices

- **Cloud Platform**
  - Include mobile and personal computers: no limit on devices
  - Operated over web browsers: Chrome, Internet Explore, Firefox
  - Javascript, ASM.js

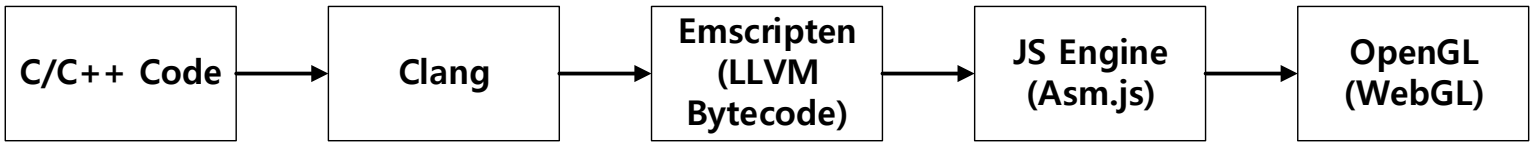
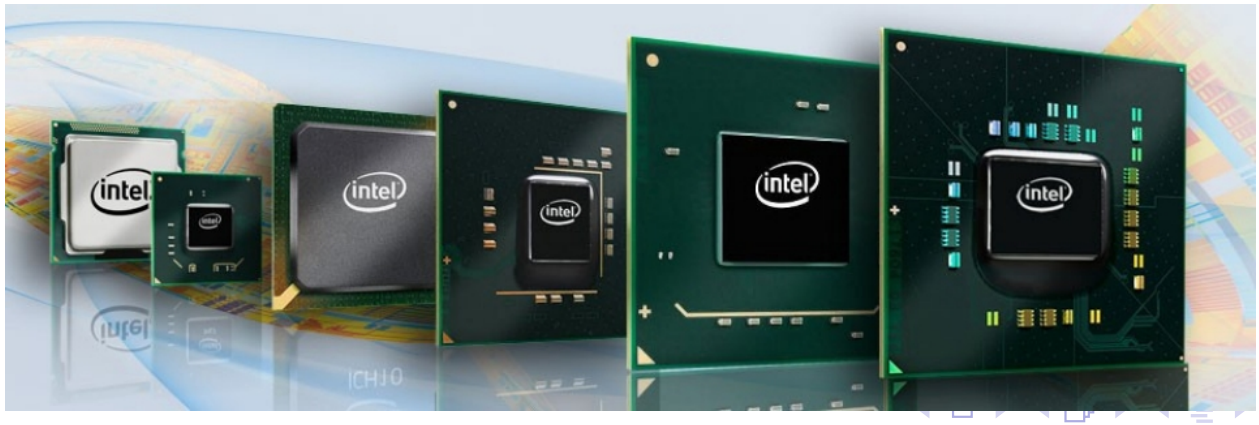


Figure: Hierarchy structure of asm.js

Target Devices

- **Intel CPU**
  - 32-, 64-bit word size, high frequency with multi-core
  - Various parallel computing capabilities
    - SIMD: SSE2, SSE3, SSSE3, SSE4, AVX, AVX2
    - SIMT: OpenMP

Mnemonics	Operands	Description	Operation	L
_mm_add_epi32	Rd, Rr	Addition of Packed 32-bit	Rd[3:0]←Rd[3:0]+Rr[3:0]	2
_mm_xor_si128	Rd, Rr	Bitwise OR of 128-bit	Rd←Rd⊕Rr	2
_mm_slli_epi32	Rd, #imm	Left Shift Packed 32-bit by imm	Rd[3:0]←Rd[3:0]<<#imm	2
_mm_srli_epi32	Rd, #imm	Right Shift Packed 32-bit by imm	Rd[3:0]←Rd[3:0]>>#imm	2



Target Devices

- **NVIDIA GPU**
  - Advanced general purpose computation environment (CUDA)
  - Massively threaded computation with thousand of threads

Mnemonics	Operands	Description	Operation
ADD	Rd, Rr1, Rr2	Add without Carry	Rd ← Rd+Rr
XOR	Rd, Rr1, Rr2	Exclusive OR	Rd ← Rd⊕Rr
SHL	Rd, Rr, #imm	Logical Shift Left by imm	Rd ← Rd<<#imm
SHR	Rd, Rr, #imm	Logical Shift Right by imm	Rd ← #imm>>Rd

8-bit AVR

Addition	Exclusive-or	Right Rotation
`ADD R12, R16`	`EOR R12, R16`	`CLR R20`
`ADC R13, R17`	`EOR R13, R17`	`LSR R15`
`ADC R14, R18`	`EOR R14, R18`	`ROR R14`
`ADC R15, R19`	`EOR R15, R19`	`ROR R13`
		`ROR R12`
		`ROR R20`
		`EOR R15, R20`

Efficient Shift Offset and Direction in AVR

Input: direction d, offset o.
Output: direction d, offset o.
1. $o = o \bmod 8$
2. if o>4
3. $o = 8 - o$
4. $d = !d$
5. return d, o

	X3				X2				X1				X0			
Round1	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
Round2	R3	R2	R1	R0	R15	R14	R13	R12	R8	R11	R10	R9	R6	R5	R4	R7
Round3	R6	R5	R4	R7	R3	R2	R1	R0	R12	R15	R14	R13	R11	R10	R9	R8
Round4	R11	R10	R9	R8	R6	R5	R4	R7	R0	R3	R2	R1	R15	R14	R13	R12

Figure: Register status for LEA encryption in AVR

16-bit MSP

Addition	Exclusive-or	Right Rotation
`ADD R8, R10`	`XOR R8,R10`	`CLR R12`
`ADDC R9, R11`	`XOR R9,R11`	`RRA R11`
		`RRC R10`
		`RRC R12`
		`AND #0X1FFF, R11`
		`ADD R12, R11`

Efficient Shift Offset and Direction in MSP

Input: direction d, offset o.
Output: direction d, offset o.

1. $o = o \bmod 16$
2. if o>8
3. $o = 16 - o$
4. $d = !d$
5. return d, o

	X3		X2		X1		X0	
Round1	R11	R10	R9	R8	R7	R6	R5	R4
Round2	R5	R4	R11	R10	R9	R8	R6	R7
Round3	R6	R7	R5	R4	R11	R10	R8	R9
Round4	R8	R9	R6	R7	R5	R4	R10	R11
Round5	R10	R11	R8	R9	R6	R7	R4	R5
Round6	R4	R5	R10	R11	R8	R9	R7	R6
Round7	R7	R6	R4	R5	R10	R11	R9	R8
Round8	R9	R8	R7	R6	R4	R5	R11	R10

Figure: Register status for LEA encryption in MSP

32-bit ARM

Addition	Exclusive-or	Right Rotation by 31
`ADD R8, R8, R5`	`EOR R8, R8, R5`	`ROR R8, #31`

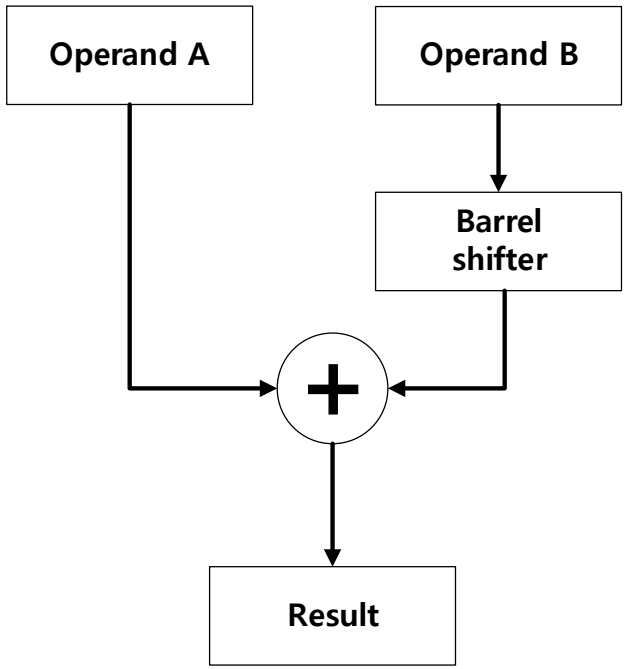


Figure: Bitwise exclusive-or with inline barrel shifter for LEA

SIMD NEON

Addition	Exclusive-or	Right Rotation by 9
`vadd.i32 q1, q1, q0`	`veor q1, q1, q0`	`vshl.i32 q1, q0, #9` `vsri.32 q1, q0, #23`

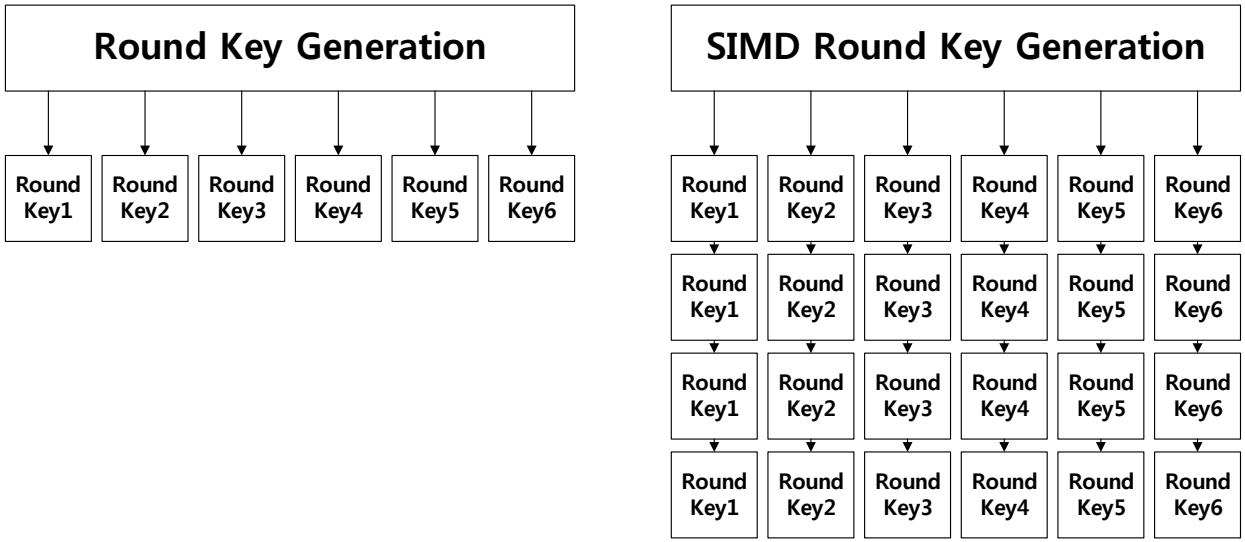


Figure: Key schedule in NEON

Cloud Platform

- Javascript does not support `unsigned int`.
- We can get `unsigned int` with `unsigned int` right shift.
- Emscripten allows us to use C/C++ compatible Javascript.

```
function ROL(input, offset){
 input = ( (input<<offset)>>>0) | (input>>>(32-offset) );
 return input
}//Left rotation
```

```
function ROR(input, offset){
 input = ( (input<<32-offset)>>>0) | (input>>>(offset) );
 return input
}//Right rotation
```

Intel CPU (1/4)

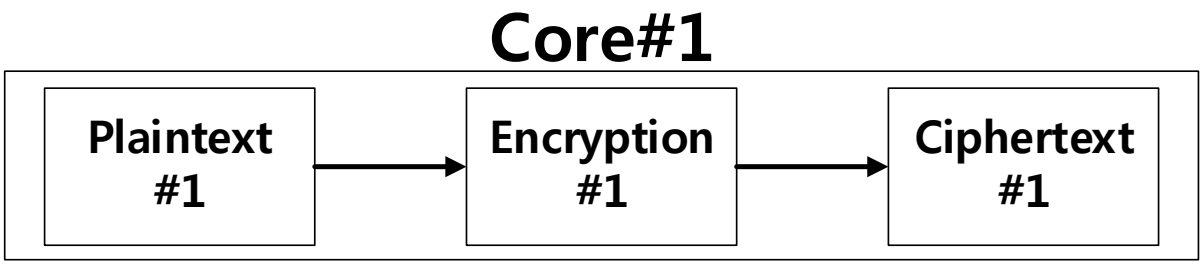


Figure: SISD based LEA implementation

Intel CPU (2/4)

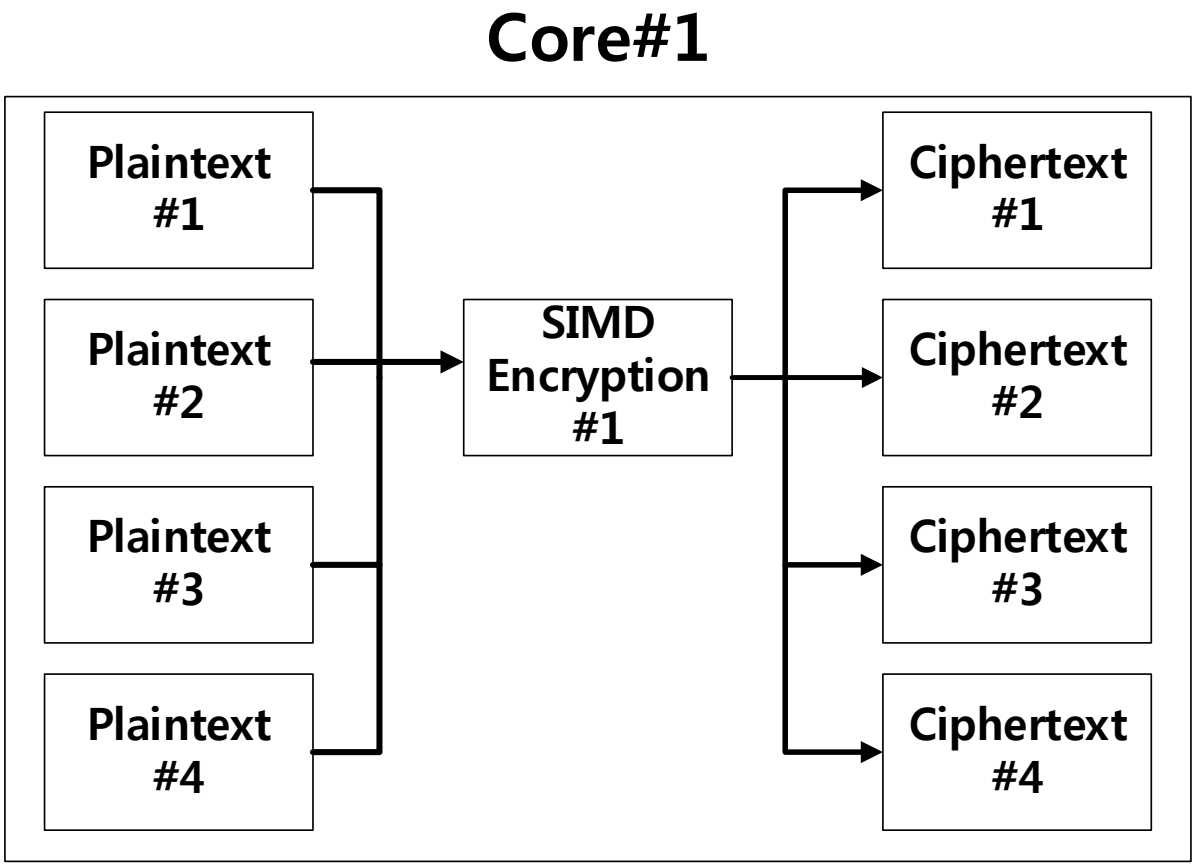


Figure: SIMD based LEA implementation

Intel CPU (3/4)

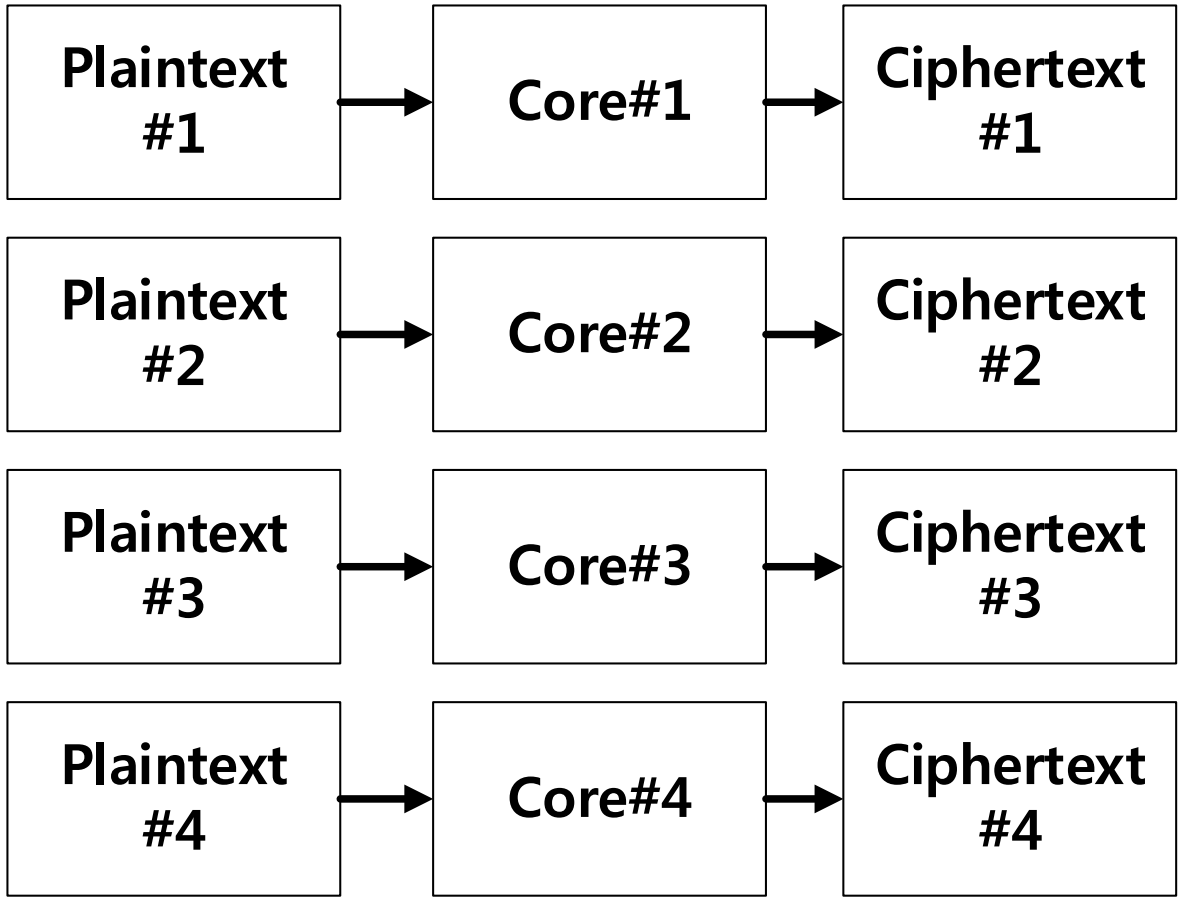


Figure: SISD+SIMT based LEA implementation

Intel CPU (4/4)

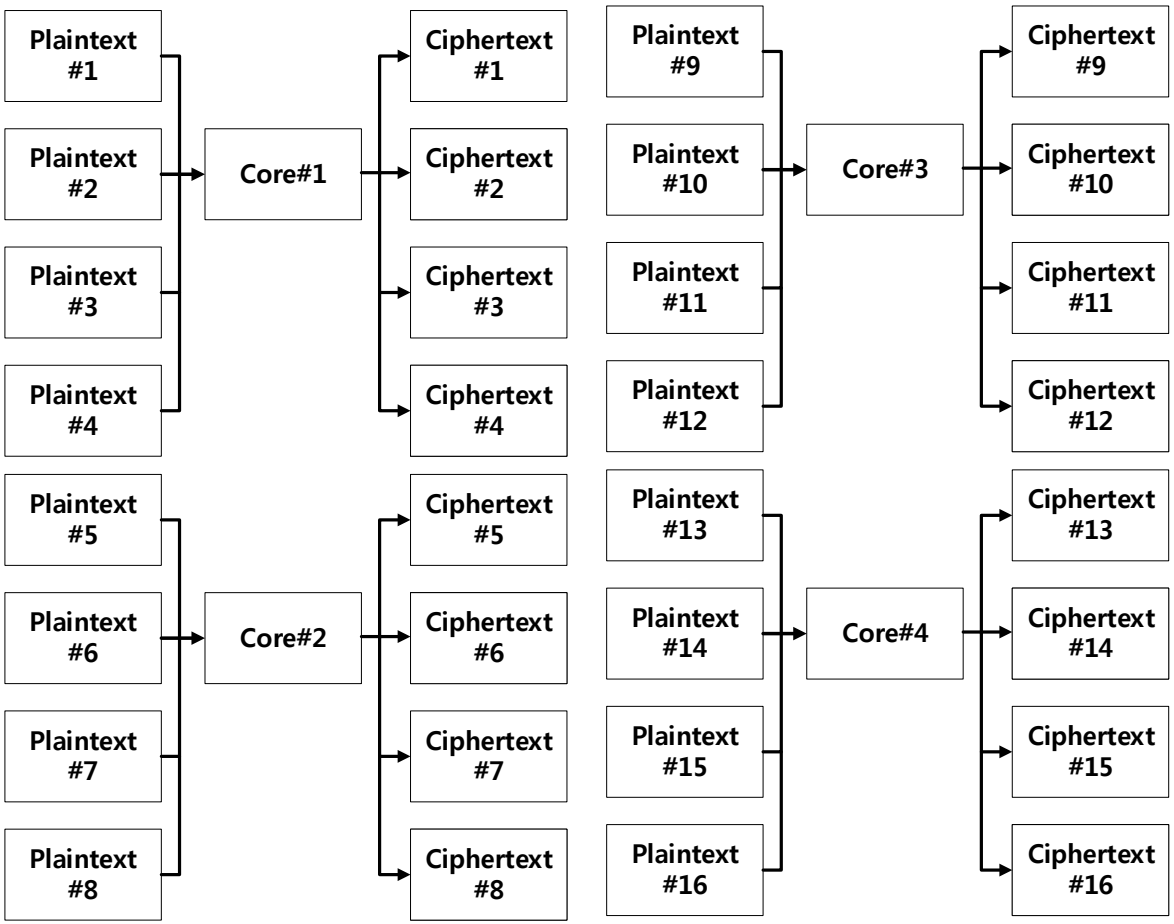


Figure: SIMD+SIMT based LEA implementation

NVIDIA GPU

- By referring ICISC'13, we assigned several encryptions on single thread.

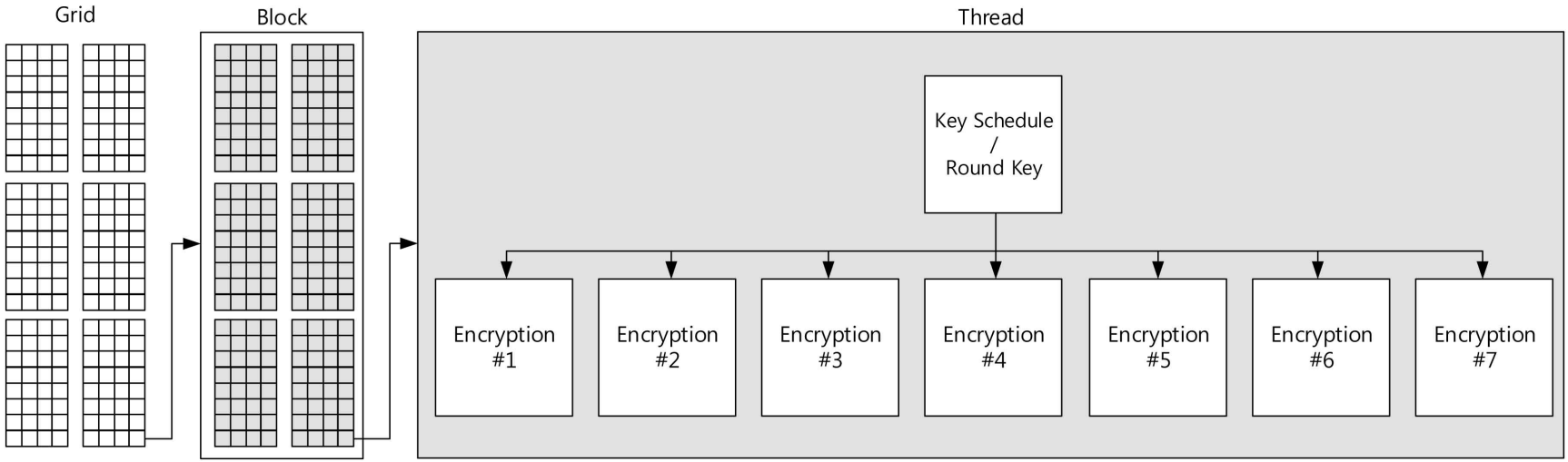


Figure: Multiple encryption in a thread approach

Third Party

- C#: using `System.UInt23` for `unsigned int`.

unsigned char	unsigned int
`using u8 = System.Byte`	`using u32 = System.UInt32`

- Java: using `unsigned int shift` for `unsigned int`.

```
function ROL(input, offset){
 input = ( (input<<offset)>>>0) | (input>>>(32-offset) );
 return input
}//Left rotation
```

```
function ROR(input, offset){
 input = ( (input<<32-offset)>>>0) | (input>>>(offset) );
 return input
}//Right rotation
```

- Python: using mask with `0xffffffff` for `unsigned int`.

```
def ROL(a,size):
size = size % 32
c = (a<<size) & 0xffffffff)
d = (a>>(32-size)) & 0xffffffff
return c | d
```

```
def ROR(a,size):
size = size % 32
c = (a<<(32-size)) & 0xffffffff
d = (a>>size) & 0xffffffff
return c | d
```

8-bit AVR

- Compared with previous 128-bit LEA implementation over AVR, performance is enhanced by 11.7 %.

Table: Comparison results on AVR, Key and Enc are measured in cycles/byte and code size in bytes, †: key scheduling is written in C language, ‡: reference code.

Method	Key	Enc	Code size
Proposed			
LEA 128-bit	229.1	167.6	11690
LEA 192-bit	3600.1†	222.9	7296
LEA 256-bit	3209†	254.1	8558
LEA 128-bit‡	2683.3	1349.2	1556
LEA 192-bit‡	3600.1	1572.4	2036
LEA 256-bit‡	3209	1795.7	2470
LEA 128-bit [23]	n/a	190	n/a
AES 128-bit [22]	46.7	124.5	1912

16-bit MSP

- Compared with 128-bit LEA reference implementation over AVR, performance is enhanced by 84.8 %.

Table: Comparison results on MSP, Key and Enc are measured in cycles/byte and code size in bytes, †: key scheduling is written in C language, ‡: reference code, ⋆: On-the-fly method.

Method	Key	Enc	Code size
Proposed			
LEA 128-bit	206.4	157.6	10150
LEA 192-bit	2958.6†	176.4	5570
LEA 256-bit	2546†	201.1	6590
LEA 128-bit‡	2367.3	1033.9	772
LEA 192-bit‡	2958.6	1229.9	996
LEA 256-bit‡	2546	1405.1	1328
AES 128-bit⋆ [30]	n/a	339.5	2536
AES 256-bit⋆ [30]	n/a	472	2830
AES 128-bit⋆ [12]	n/a	180	5860

32-bit ARM & SIMD NEON

- Compared with previous 128-bit LEA and AES implementation over ARM, performance is enhanced by 13.9 % and 49.1 %.

Table: Comparison results on ARM, Key and Enc are measured in cycles/byte and code size in bytes, †: key scheduling is written in C language, ‡: reference code.

Method	Key	Enc	Code size
Proposed			
ARM-LEA 128-bit	11.4	17.3	10872
ARM-LEA 192-bit	108.3†	21.7	11984
ARM-LEA 256-bit	104.4†	24.7	11648
NEON-LEA 128-bit	30.3†	12.3	9868
NEON-LEA 192-bit	34†	14	10476
NEON-LEA 256-bit	30.2†	15.5	10424
LEA 128-bit‡	94.3	46.5	7252
LEA 192-bit‡	108.3	58.7	7668
LEA 256-bit‡	104.4	72.2	7256
ARM-LEA 128-bit [23]	n/a	20.1	n/a
NEON-LEA 128-bit [25]	n/a	10.1	n/a
AES 128-bit [22]	n/a	34	n/a

Cloud Platform (1/3)

Table: Comparison results of Javascript on desktop, Key and Enc are measured in cycles/byte and code size in bytes, †: Javascript, ‡: asm.js, *C*: Chrome, *IE*: Internet Explorer, *FF*: Firefox, [*L*]: Looped, [*U*]: Unrolled.

Method	Key	Enc	Code size
128-bit^{†,C}[*L*]	114.8	65.9	2309
128-bit^{†,IE}[*L*]	998.8	1232.5	2309
128-bit^{†,FF}[*L*]	252.9	246.5	2309
128-bit^{†,C}[*U*]	114.8	240.1	5510
128-bit^{†,IE}[*U*]	998.8	1997.5	5510
128-bit^{†,FF}[*U*]	252.9	276.3	5510
192-bit^{†,C}[*L*]	155.8	72.3	2750
192-bit^{†,IE}[*L*]	1374.2	1466.3	2750
192-bit^{†,FF}[*L*]	311.7	278.4	2750
192-bit^{†,C}[*U*]	155.8	297.5	6839
192-bit^{†,IE}[*U*]	1374.2	2358.8	6839
192-bit^{†,FF}[*U*]	311.7	331.5	6839
256-bit^{†,C}[*L*]	138.1	89.3	3055
256-bit^{†,IE}[*L*]	1317.5	1870	3055
256-bit^{†,FF}[*L*]	278.4	312.4	3055
256-bit^{†,C}[*U*]	138.1	369.8	7769
256-bit^{†,IE}[*U*]	1317.5	3102.5	7769
256-bit^{†,FF}[*U*]	278.4	401.6	7769

Method	Key	Enc	Code size
AES 128-bit^{†,C} [27]	n/a	1176	6057
AES 128-bit^{†,IE} [27]	n/a	1073.7	6057
AES 128-bit^{†,FF} [27]	n/a	818.1	6057

Chrome is fastest browser for Javascript. LEA-128 over Chrome is 94.4 % faster than AES-128 over Chrome.

Cloud Platform (2/3)

Table: Comparison results of Javascript on desktop, Key and Enc are measured in cycles/byte and code size in bytes, †: Javascript, ‡: asm.js, *C*: Chrome, *IE*: Internet Explorer, *FF*: Firefox, [*L*]: Looped, [*U*]: Unrolled.

Method	Key	Enc	Code size
128-bit^{‡,C}[*L*]	63.8	36.1	224642
128-bit^{‡,IE}[*L*]	688.5	716.1	224642
128-bit^{‡,FF}[*L*]	44.6	23.4	224642
128-bit^{‡,C}[*U*]	63.8	42.5	228614
128-bit^{‡,IE}[*U*]	688.5	637.5	228614
128-bit^{‡,FF}[*U*]	44.6	25.5	228614
192-bit^{‡,C}[*L*]	79.3	46.8	224927
192-bit^{‡,IE}[*L*]	804.7	992.4	224927
192-bit^{‡,FF}[*L*]	73.7	34	224927
192-bit^{‡,C}[*U*]	79.3	51	229644
192-bit^{‡,IE}[*U*]	804.7	771.4	229644
192-bit^{‡,FF}[*U*]	73.7	34	229644
256-bit^{‡,C}[*L*]	74.4	48.9	225414
256-bit^{‡,IE}[*L*]	708.7	1162.4	225414
256-bit^{‡,FF}[*L*]	69.1	36.1	225414
256-bit^{‡,C}[*U*]	74.4	59.5	230846
256-bit^{‡,IE}[*U*]	708.7	1000.9	230846
256-bit^{‡,FF}[*U*]	69.1	38.3	230846

Method	Key	Enc	Code size
AES 128-bit^{†,C} [27]	n/a	1176	6057
AES 128-bit^{†,IE} [27]	n/a	1073.7	6057
AES 128-bit^{†,FF} [27]	n/a	818.1	6057

Firefox is fastest browser for asm.js. LEA-128 over Firefox is 97.1 % faster than AES-128 in Javascript.

Cloud Platform (3/3)

Table: Comparison results of Javascript on mobile platform, Key and Enc are measured in cycles/byte and code size in bytes, †: Javascript, C: Chrome, W: Webkit, FF: Firefox, $[L]$: Looped, $[U]$: Unrolled.

Method	Key	Enc	Code size
128-bit$^{\dagger,C}[L]$	1032	350	2309
128-bit$^{\dagger,W}[L]$	1466	934	2309
128-bit$^{\dagger,FF}[L]$	2345	1102	2309
128-bit$^{\dagger,C}[U]$	1032	1697	5510
128-bit$^{\dagger,W}[U]$	1466	2614	5510
128-bit$^{\dagger,FF}[U]$	2345	3132	5510
192-bit$^{\dagger,C}[L]$	793	367	2750
192-bit$^{\dagger,W}[L]$	1509	882	2750
192-bit$^{\dagger,FF}[L]$	2065	1155	2750
192-bit$^{\dagger,C}[U]$	793	2205	6839
192-bit$^{\dagger,W}[U]$	1509	3199	6839
192-bit$^{\dagger,FF}[U]$	2065	5267	6839
256-bit$^{\dagger,C}[L]$	778	455	3055
256-bit$^{\dagger,W}[L]$	1165	1018	3055
256-bit$^{\dagger,FF}[L]$	1732	1522	3055
256-bit$^{\dagger,C}[U]$	778	2572	7769
256-bit$^{\dagger,W}[U]$	1165	3787	7769
256-bit$^{\dagger,FF}[U]$	1732	4987	7769

Method	Key	Enc	Code size
AES 128-bit^{†,C} [27]	n/a	3524	6057
AES 128-bit^{†,W} [27]	n/a	4581	6057
AES 128-bit^{†,FF} [27]	n/a	4012	6057

Chrome is fastest browser on mobile. LEA-128 is 90 % faster than AES-128 in Javascript.

Intel CPU (1/2)

Table: Comparison results on Intel CPU, Key and Enc are measured in cycles/byte and code size in bytes, †: OpenMP ‡: reference code, 1: Intel Core-i5 2500, 2: Intel Core i7-860, 3: Intel Core i7-960

Method	Key	Enc	Code size
LEA 128-bit	22.1	14.1	3072
LEA 192-bit	28.2	16.4	3584
LEA 256-bit	33.1	18.5	4096
SIMD-LEA 128-bit	42.3	4	5632
SIMD-LEA 192-bit	40.4	6	6656
SIMD-LEA 256-bit	47.6	6.7	7680
LEA 128-bit†	n/a	5	3072
LEA 192-bit†	n/a	4	3584
LEA 256-bit†	n/a	4.5	4096
SIMD-LEA 128-bit†	n/a	1.1	5632
SIMD-LEA 192-bit†	n/a	1.6	6656
SIMD-LEA 256-bit†	n/a	1.5	7680
LEA 128-bit‡	22.1	17.2	2048
LEA 192-bit‡	28.2	20	2048
LEA 256-bit‡	33.1	22.7	2048
LEA 128-bit[1] [23]	n/a	9.29	n/a
SIMD-LEA 128-bit[2] [23]	n/a	4.19	n/a
AES 128-bit[1] [8]	n/a	11.35	n/a
SIMD-AES 128-bit[2] [16]	n/a	6.92	n/a
AES 128-bit†,[3] [21]	n/a	1.8	n/a

Intel CPU (2/2)

- Thanks to hyper threading, we can exploit eight threads at once under four physical cores.

Table: Detailed results on OpenMP, Enc are measured in cycles/byte and code size in bytes, †: Looped ‡: Unrolled

Method	#1	#2	#3	#4	#8	#16	#32	#64
Proposed								
LEA 128-bit†	14.3	7.5	5.4	5	7.4	10.4	11.1	28.2
LEA 192-bit†	14.1	7.5	6.2	6	4	13.5	11.1	27
LEA 256-bit†	16.3	8.5	7.4	6.8	4.5	9.6	10.9	17.6
LEA 128-bit‡	14	7.5	5.9	5.7	5.9	24.9	10.4	21.6
LEA 192-bit‡	16.3	8.5	7	6.7	4.5	13.2	22.8	20.9
LEA 256-bit‡	18.6	9.9	8	7.7	4.8	6.7	11.7	20.2
SIMD-LEA 128-bit†	4.6	2.4	2.1	2	1.2	4.6	9.3	5.5
SIMD-LEA 192-bit†	7.2	3.7	2.9	2.6	1.6	4.5	3.6	6.2
SIMD-LEA 256-bit†	7.8	4	3.2	2.9	1.7	6	9.1	5.9
SIMD-LEA 128-bit‡	3.9	2	1.8	1.8	1.1	2.7	2.9	4.6
SIMD-LEA 192-bit‡	6	3.1	2.5	2.3	1.6	3	4.9	5.3
SIMD-LEA 256-bit‡	6.7	3.4	2.7	2.6	1.5	7.3	3.9	6.3
AES 128-bit [21]	4.8	n/a	n/a	n/a	1.8	n/a	n/a	n/a

NVIDIA GPU (1/2)

Table: Comparison results on Nvidia GPU, Key and Enc are measured in cycles/byte, Gigabyte/second and code size in bytes, †: without memory transfer, ‡: with memory transfer, ⋆: non-coalescing code, 1: GTX 680 2: GTX 285

Method	Enc(cycles/byte)	Enc(Gigabyte/second)	Code size
Proposed			
LEA 128-bit^{†,1}	0.06	17.21	71168
LEA 192-bit^{†,1}	0.07	14.12	76800
LEA 256-bit^{†,1}	0.08	12.57	81920
LEA 128-bit^{‡,1}	0.42	2.39	71168
LEA 192-bit^{‡,1}	0.44	2.31	76800
LEA 256-bit^{‡,1}	0.45	2.21	81920
LEA 128-bit^{†,*,1}	0.09	10.77	17408
LEA 192-bit^{†,*,1}	0.13	7.71	17408
LEA 256-bit^{†,*,1}	0.14	7.28	17408
LEA 128-bit^{‡,*,1}	0.72	1.40	17408
LEA 192-bit^{‡,*,1}	0.75	1.34	17408
LEA 256-bit^{‡,*,1}	0.75	1.34	17408
LEA 128-bit^{†,1} [25]	-	17.4	-
LEA 128-bit^{‡,1} [25]	-	2.5	-
AES 128-bit^{†,2} [14]	-	9.3	-
AES 128-bit^{‡,2} [14]	-	2.8	-

NVIDIA GPU (2/2)

- Performance is peak where ratio of encryption/thread is 3.

Table: Detailed results of non-coalescing program on GPU, Enc are measured in cycles/byte / GBps and code size in bytes, †: without memory transfer ‡: with memory transfer

Method	#1	#2	#3	#4
Proposed				
LEA 128-bit†	0.234/4.285	0.142/7.084	0.093/10.773	0.107/9.362
LEA 192-bit†	0.321/3.133	0.224/4.481	0.130/7.710	0.125/8.065
LEA 256-bit†	0.370/2.717	0.213/4.723	0.138/7.281	0.130/7.71
LEA 128-bit‡	0.795/1.264	0.791/1.273	0.717/1.402	0.923/1.090
LEA 192-bit‡	0.871/1.154	0.836/1.199	0.753/1.335	0.927/1.085
LEA 256-bit‡	0.895/1.124	0.854/1.178	0.749/1.342	0.915/1.099

3rd Party Implementations

- The order of speed is C#, Java and Python.
- Java is operated over virtual machine.
- Python is interpretor language and ranked at the slowest language.

Table: Comparison results on Intel CPU, Key and Enc are measured in cycles/byte and code size in bytes, †: C# ‡: Java, ⋆: Python

Method	Key	Enc
Proposed		
LEA 128-bit†	82.9	44.6
LEA 192-bit†	133.2	51
LEA 256-bit†	111.6	59.5
LEA 128-bit‡	233.8	913.8
LEA 192-bit‡	212.5	956.3
LEA 256-bit‡	191.3	1083.8
LEA 128-bit⋆	3168.4	1625.6
LEA 192-bit⋆	3570	1912.3
LEA 256-bit⋆	3527.5	2103.8

Future technologies at work

NEON: 256-bit wise word instruction
AVX512: 512-bit wise word instruction
CUDA capability 3.5: rotation instruction
CUDA 6.0: combined memory architecture

Contributions

Novel optimization methods on various platforms
Advanced implementation results on processors

Thank you for your attention.