AMD
Smarter Choice

Encryption

Encryption

Goal: Optimization walkthrough using encryption as the example

AES - Advanced Encryption Standard

AES

Works on 128 bits at a time in a 4x4 state array or 16 *byte* blocks

input bytes

in_0	in_4	in_8	in_{12}
in_1	in_5	in_9	in_{13}
in_2	in_6	in_{10}	in_{14}
in_3	in_7	in_{11}	in_{15}

→

State array

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

→

output bytes

out_0	out_4	out_8	out_{12}
out_1	out_5	out_9	out_{13}
out_2	out_6	out_{10}	out_{14}
out_3	out_7	out_{11}	out_{15}

AES - Cipher Algorithm

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Core Loop

Steps:

1) SubBytes

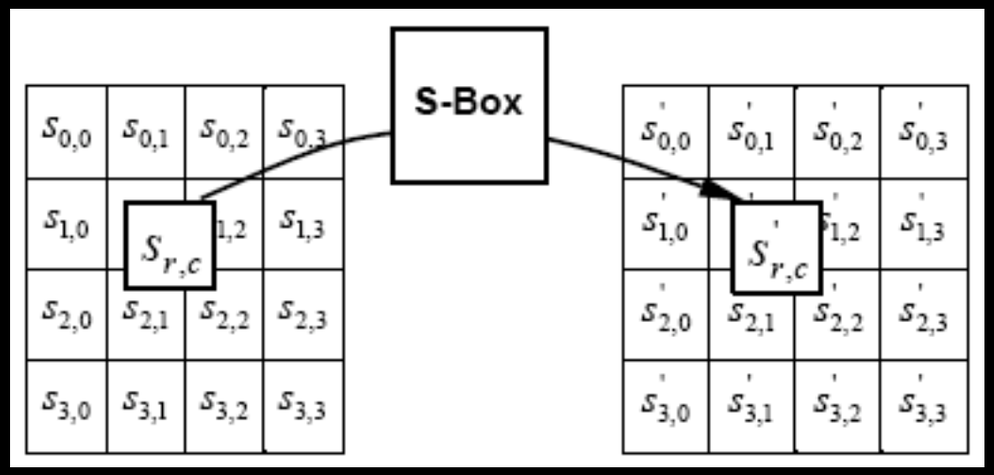
2) ShiftRows

3) MixColumns

4) AddRoundKey

SubBytes

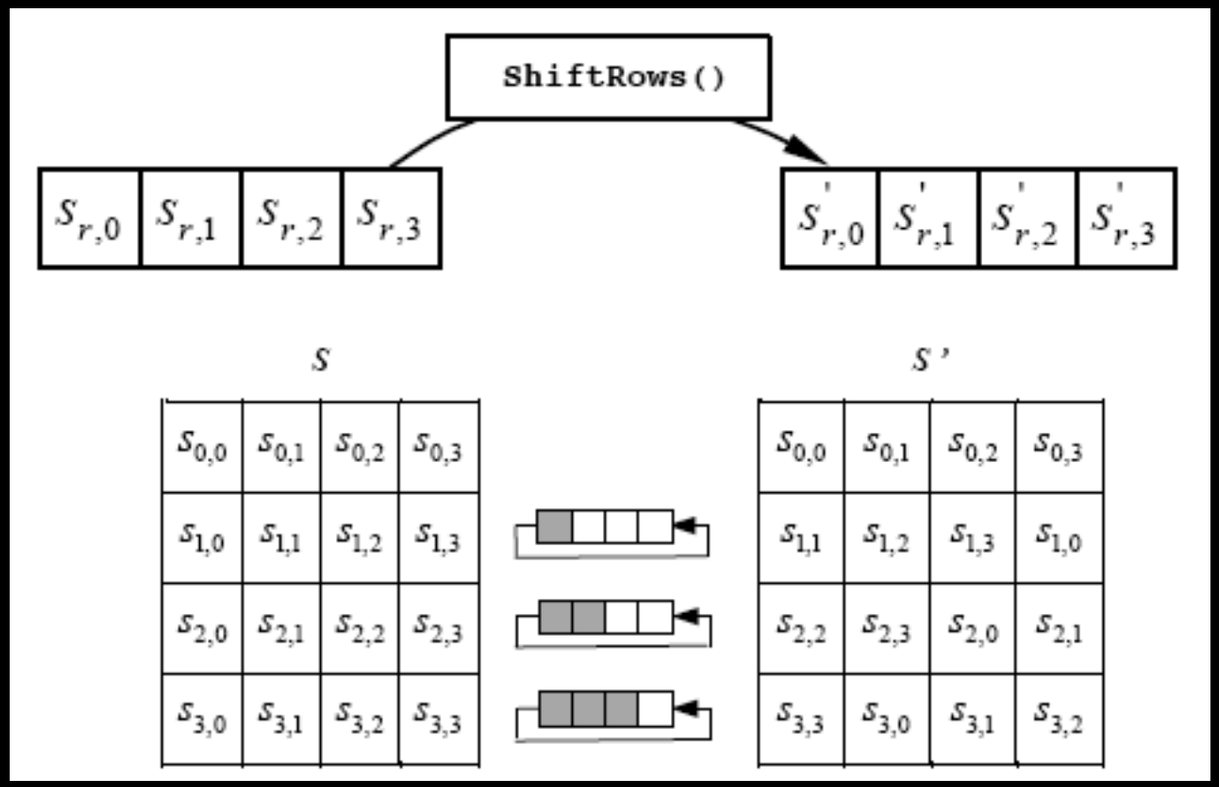
SubBytes is a simple transformation applied to each *byte*



$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

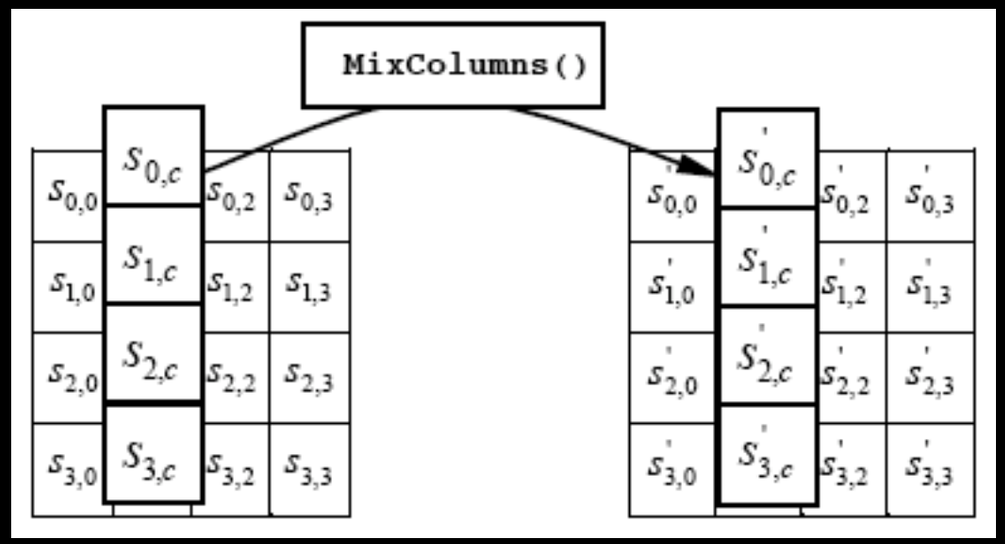
ShiftRows


ShiftRows()
$S_{r,0}$ $S_{r,1}$ $S_{r,2}$ $S_{r,3}$
$S'_{r,0}$ $S'_{r,1}$ $S'_{r,2}$ $S'_{r,3}$
S
S'
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

MixColumns

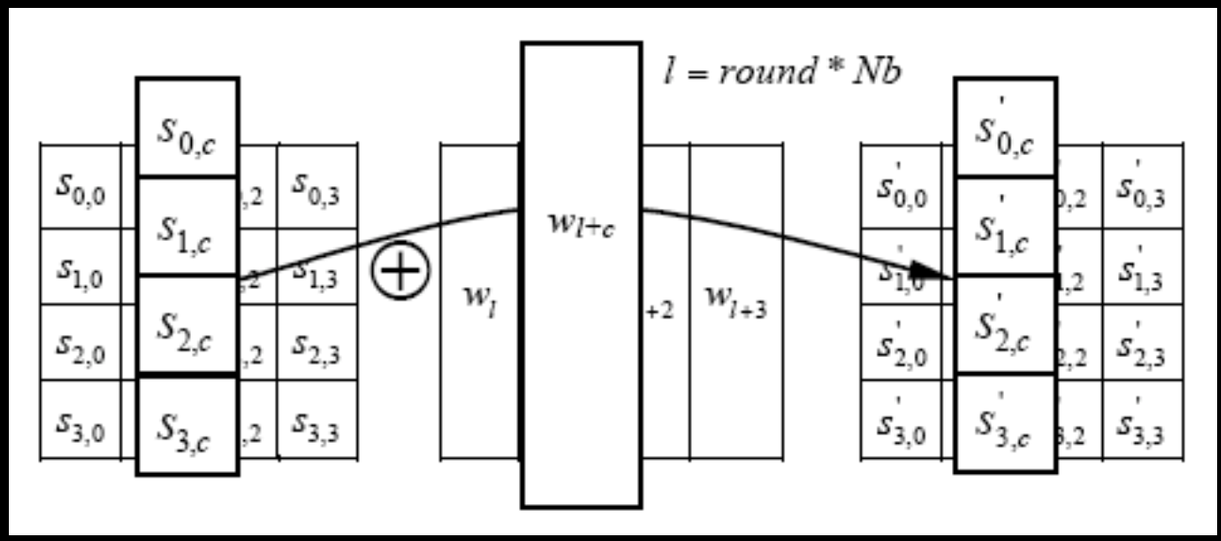
Finite field multiplies (binary polynomials)

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$



AddRoundKey

Add (XOR) the key to the state array



How do we implement this on the GPU?

AMD Smarter Choice

How do we represent the state array?

input bytes

in_0	in_4	in_8	in_{12}
in_1	in_5	in_9	in_{13}
in_2	in_6	in_{10}	in_{14}
in_3	in_7	in_{11}	in_{15}

→

State array

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

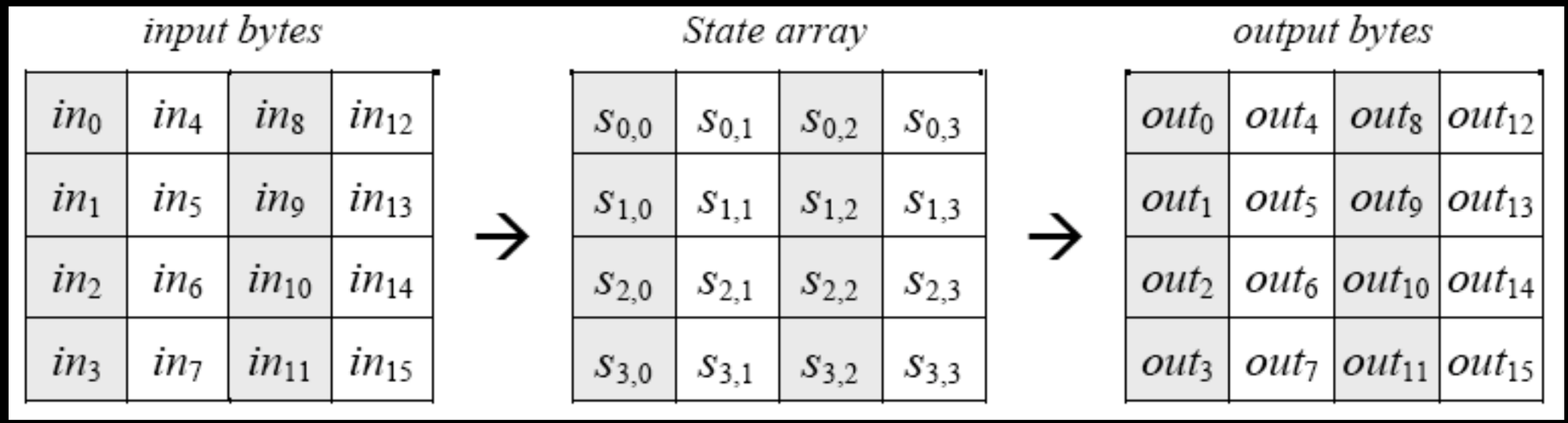
→

output bytes

out_0	out_4	out_8	out_{12}
out_1	out_5	out_9	out_{13}
out_2	out_6	out_{10}	out_{14}
out_3	out_7	out_{11}	out_{15}

How do we implement this on the GPU?

How do we represent the state array?



Four registers - four components each

r0.xyzw
r1.xyzw
r2.xyzw
r3.xyzw

How to implement MixColumns?

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$

How to implement MixColumns?

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$

What about now?

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$

How to implement MixColumns?

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$

What about now?

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$

Use lookup tables

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$

How big a table do we need?

Use lookup tables

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$

How big a table do we need?

Bytes: 256 entries

How many tables do we need?

Use lookup tables

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}
$$

How big a table do we need?

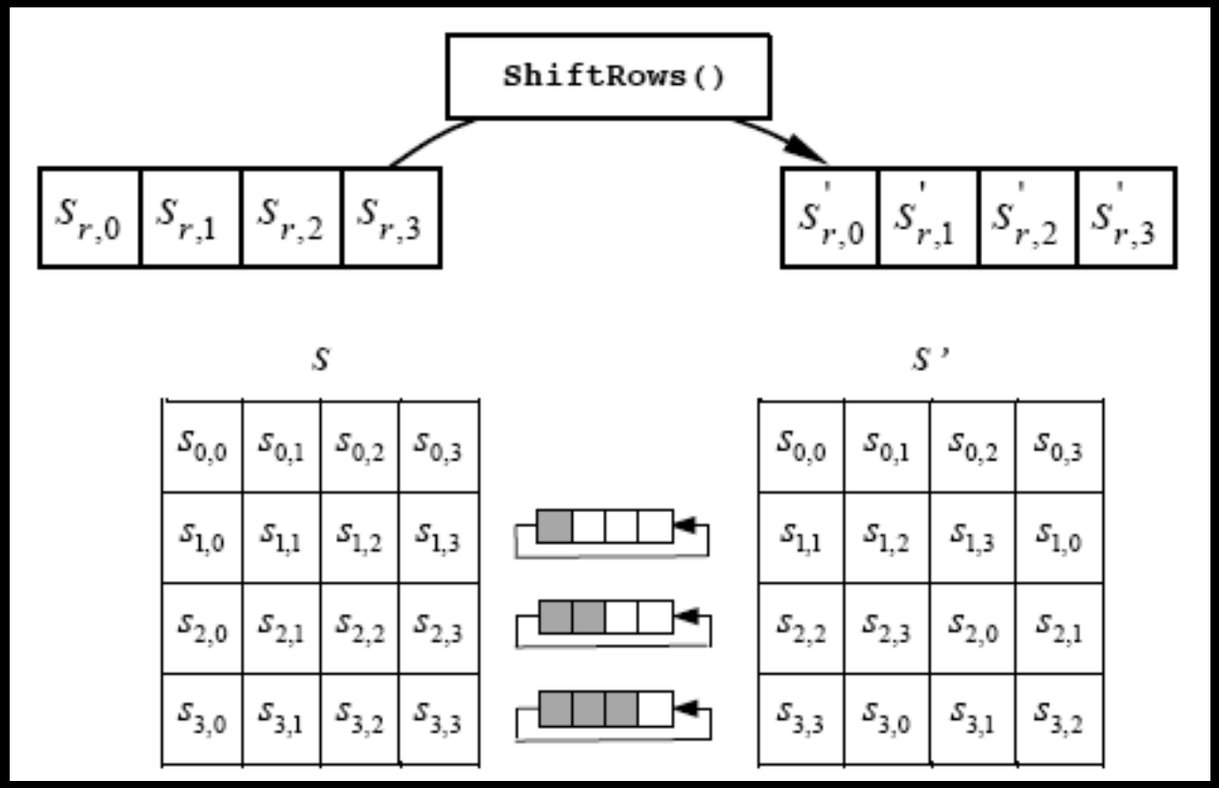
Bytes: 256 entries

How many tables do we need?

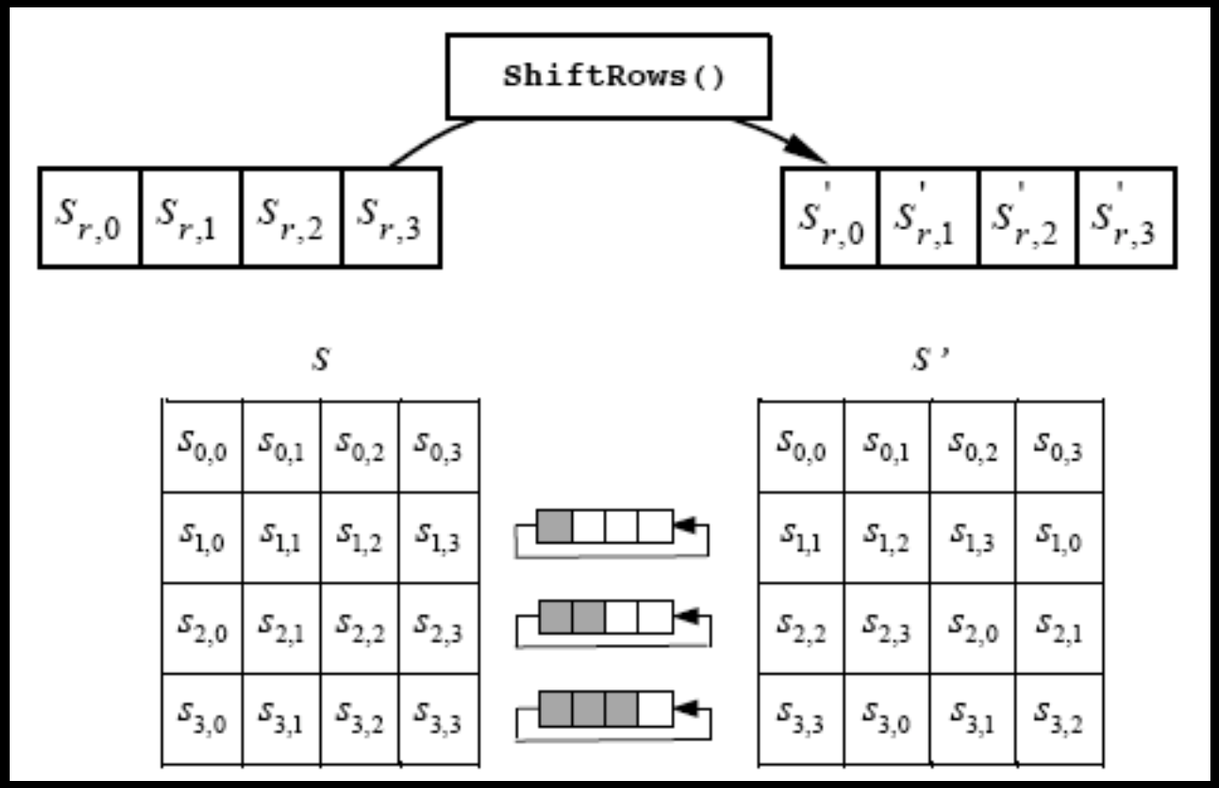
Swizzilng: arbitrary ordering (one table)

Total: One table 256x4bytes

How to implement ShiftRows?



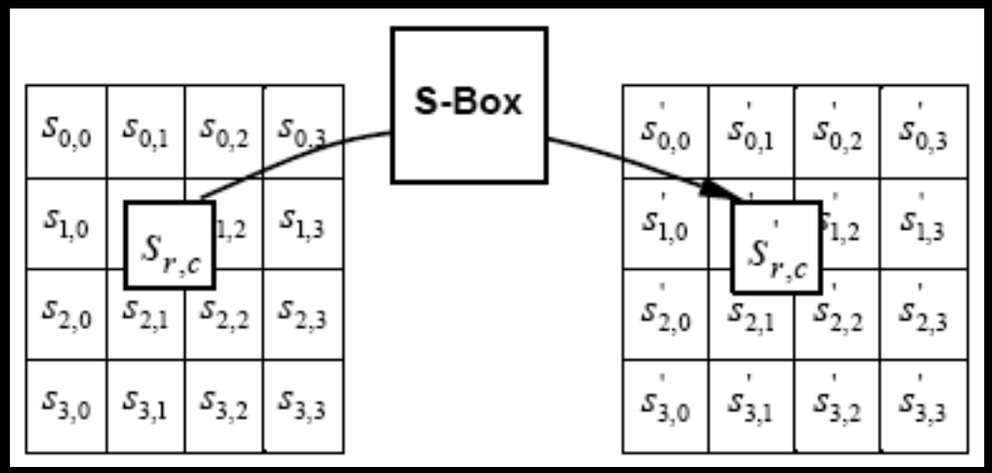
How to implement ShiftRows?



Swizzling is free:

r0′.xyzw = r0.xyzw
r1′.xyzw = r1.wxyz
r2′.xyzw = r2.zwxy
r3′.xyzw = r3.yzwx

How to implement SubBytes?



Lookup table again

How big and how many tables?

SubBytes table?

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

SubBytes table?

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

MixColumns table can be pre-computed with SubBytes transform. No SubBytes table is needed.

Putting it all together

SubBytes + MixColumns

```
float4 c0, r0;

c0 =   txMcol[r0.w].wzyx
     ^ txMcol[r3.z].xwzy
     ^ txMcol[r2.y].yxwz
     ^ txMcol[r1.x].zyxw;

r0 = c0 ^ tKeyadd[round_offset]
```

Shiftrows: component swizzling

Add RoundKey: pre-computed round key lookup

What about the XORs?

R6XX or DX10 hardware supports native integer operations

What about previous generations?

XOR on floating point hardware

How do you do a XOR using only floating point hardware?

XOR on floating point hardware

How do you do a XOR using only floating point hardware?

```
float4 XOR_CALC(float4 a, float4 b)
{
    float4 ret;

    a*=256;
    b*=256;

    float4 pa = frac(a/2.f)*2.f;
    float4 pb = frac(b/2.f)*2.f;
    ret = (pa==pb) ? 0 : 1;
    a -= pa;
    b -= pb;
    pa = frac(a/4.f)*4.f;
    pb = frac(b/4.f)*4.f;
    ret += (pa==pb) ? 0 : 2;
    a -= pa;
    b -= pb;
    pa = frac(a/8.f)*8.f;
    pb = frac(b/8.f)*8.f;
    ret += (pa==pb) ? 0 : 4;
    a -= pa;
    b -= pb;
    a -= pa;
    b -= pb;
    pa = frac(a/32.f)*32.f;
    pb = frac(b/32.f)*32.f;
    ret += (pa==pb) ? 0 : 16;
    a -= pa;
    b -= pb;
    pa = frac(a/64.f)*64.f;
    pb = frac(b/64.f)*64.f;
    ret += (pa==pb) ? 0 : 32;
    a -= pa;
    b -= pb;
    pa = frac(a/128.f)*128.f;
    pb = frac(b/128.f)*128.f;
    ret += (pa==pb) ? 0 : 64;
    a -= pa;
    b -= pb;
    pa = a;
    pb = b;
    ret += (pa==pb) ? 0 : 128;
    return ret/255;
}
```

Using XOR tables

```
float4 c0, r0;

c0 =    txMcol[r0.w].wzyx
     ^ txMcol[r3.z].xwzy
     ^ txMcol[r2.y].yxwz
     ^ txMcol[r1.x].zyxw;
```

```
float4 a0,a1,b0,b1,c0,t0,t1;

a0 = txMcol[r0.w].wzyx;
a1 = txMcol[r3.z].xwzy;
t0 = XOR(a0, a1);

b0 = txMcol[r2.y].yxwz;
b1 = txMcol[r1.x].zyxw;
t1 = XOR(a, b);

c0 = XOR(t0, t1);
```

```
float4 XOR(a,b)
{
  float4 out;
  out.x = Txor[a.x][b.x];
  out.y = Txor[a.y][b.y];
  out.z = Txor[a.z][b.z];
  out.w = Txor[a.w][b.w];
  return out;
}
```

Using XOR tables

```
float4 c0, r0;

c0 =    txMcol[r0.w].wzyx
     ^ txMcol[r3.z].xwzy
     ^ txMcol[r2.y].yxwz
     ^ txMcol[r1.x].zyxw;
```

```
float4 a0,a1,b0,b1,c0,t0,t1;

a0 = txMcol[r0.w].wzyx;
a1 = txMcol[r3.z].xwzy;
t0 = XOR(a0, a1);

b0 = txMcol[r2.y].yxwz;
b1 = txMcol[r1.x].zyxw;
t1 = XOR(a, b);

c0 = XOR(t0, t1);
```

```
float4 XOR(a,b)
{
  float4 out;
  out.x = Txor[a.x][b.x];
  out.y = Txor[a.y][b.y];
  out.z = Txor[a.z][b.z];
  out.w = Txor[a.w][b.w];
  return out;
}
```

```
float4 a, b, c0, r0;

a =   txMcol[r0.w][r3.z];
b =   txMcol[r2.y][r1.x];
c0 = XOR(a, b);
```

Analyzing the performance

Whether using ALU or textures what are the performance implications?

Analyzing the performance

Whether using ALU or textures what are the performance implications?

- ALU:TEX ratio
- # fetch instructions
- Memory access patterns
- Texture sizes

XOR tables achieves rates of ~300 Mbps

Can we go faster?

Latency hiding

Latency hiding

Use ALU instructions to hide memory fetch latency

Solution:

Use both ALU and fetches for XOR calculations

Mixed instructions reach ~990 Mbps

What about with native XOR hardware?

```
…

int4 c0, c1, c2, c3;
for(int i=0; i<9*4; i+=4)
{
    c0 = txMCol.Load(int2(r0.x,0)).xyzw ^ txMCol.Load(int2(r1.y,0)).wxyz ^
         txMCol.Load(int2(r2.z,0)).zwxy ^ txMCol.Load(int2(r3.w,0)).yzwx;
    c1 = txMCol.Load(int2(r1.x,0)).xyzw ^ txMCol.Load(int2(r2.y,0)).wxyz ^
         txMCol.Load(int2(r3.z,0)).zwxy ^ txMCol.Load(int2(r0.w,0)).yzwx;
    c2 = txMCol.Load(int2(r2.x,0)).xyzw ^ txMCol.Load(int2(r3.y,0)).wxyz ^
         txMCol.Load(int2(r0.z,0)).zwxy ^ txMCol.Load(int2(r1.w,0)).yzwx;
    c3 = txMCol.Load(int2(r3.x,0)).xyzw ^ txMCol.Load(int2(r0.y,0)).wxyz ^
         txMCol.Load(int2(r1.z,0)).zwxy ^ txMCol.Load(int2(r2.w,0)).yzwx;

    r0 = c0 ^ keys[4 + i];
    r1 = c1 ^ keys[5 + i];
    r2 = c2 ^ keys[6 + i];
    r3 = c3 ^ keys[7 + i];
}

…
```

Native XOR reaches performance of ~3.5 Gbps

What are the performance issues?

Can we do better?

Can we do better?

Bitslicing - treat the processor as a vector processor with each bit representing an ALU unit (i.e. a 32-bit processor is a virtualized 32 vector processor)

Everything in AES is be computed using AND, OR, NOT, and XOR gates

Using bitslicing, GPU achieves 18.5 Gbps