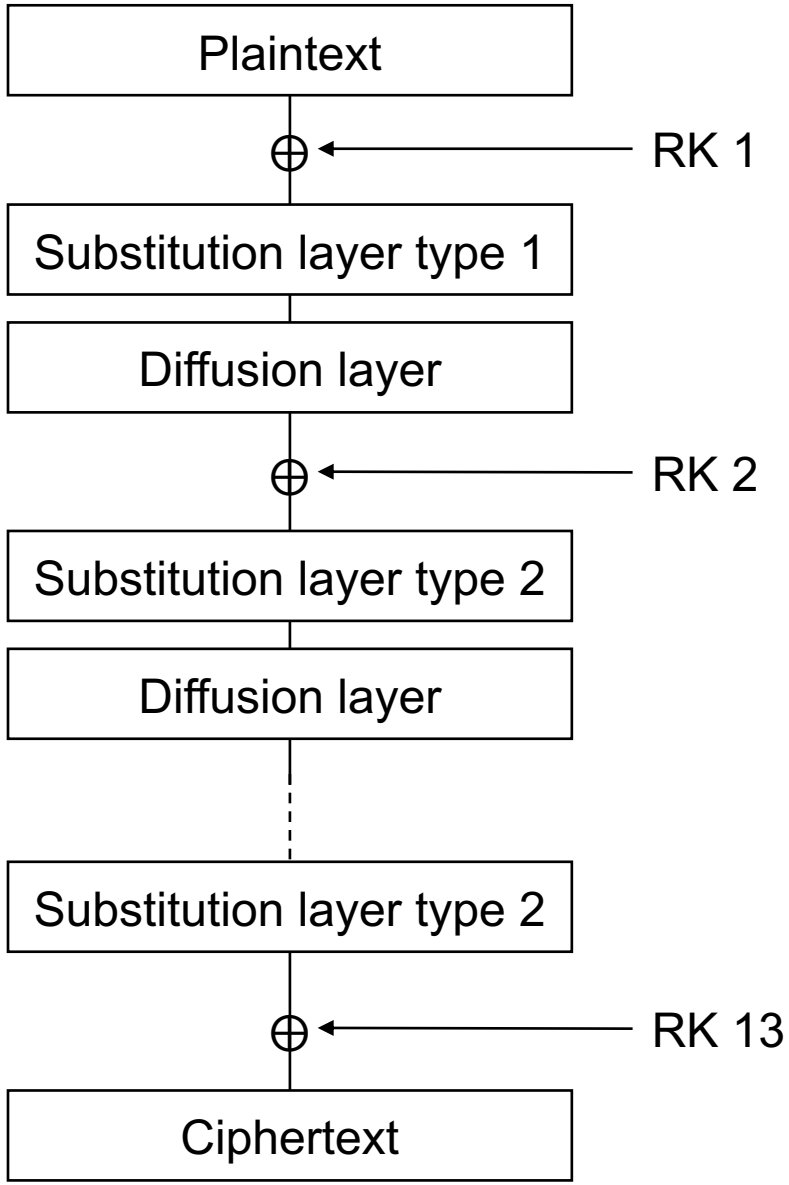
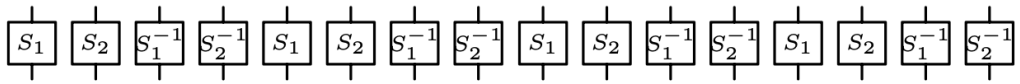
GPU ARIA 구현

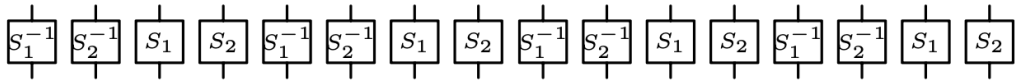
https://youtu.be/pvq_4GD-fJE

ARIA block cihper 소개





(a) S-box layer type 1

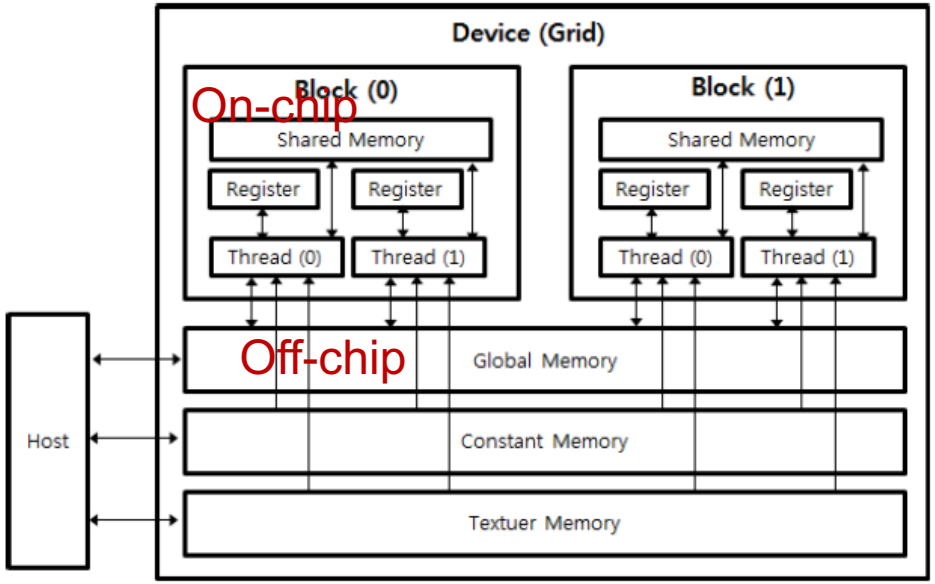


(b) S-box layer type 2

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \end{pmatrix} = \begin{pmatrix} 0&0&0&1&1&0&1&0&1&1&0&0&0&1&1&0 \\ 0&0&1&0&0&1&0&1&1&1&0&0&1&0&0&1 \\ 0&1&0&0&1&0&1&0&0&0&1&1&1&0&0&1 \\ 1&0&0&0&0&1&0&1&0&0&1&1&0&1&1&0 \\ 1&0&1&0&0&1&0&0&1&0&0&1&0&0&1&1 \\ 0&1&0&1&1&0&0&0&0&1&1&0&0&0&1&1 \\ 1&0&1&0&0&0&0&1&0&1&1&0&1&1&0&0 \\ 0&1&0&1&0&0&1&0&1&0&0&1&1&1&0&0 \\ 1&1&0&0&1&0&0&1&0&0&1&0&0&1&0&1 \\ 1&1&0&0&0&1&1&0&0&0&0&1&1&0&1&0 \\ 0&0&1&1&0&1&1&0&1&0&0&0&0&1&0&1 \\ 0&0&1&1&1&0&0&1&0&1&0&0&1&0&1&0 \\ 0&1&1&0&0&0&1&1&0&1&0&1&1&0&0&0 \\ 1&0&0&1&0&0&1&1&1&0&1&0&0&1&0&0 \\ 1&0&0&1&1&1&0&0&0&1&0&1&0&0&1&0 \\ 0&1&1&0&1&1&0&0&1&0&1&0&0&0&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{pmatrix}$$

GPU 메모리 구조

- 메모리 접근 속도
  - 글로벌 메모리 〈 로컬 메모리 〈 공유 메모리
- 메모리 용량
  - 공유 메모리 〈 로컬 메모리 〈 글로벌 메모리



GPU Warp and Bank

- Block의 thread는 Warp라는 그룹으로 실행
  - 워프에는 32개의 연속된 thread로 이루어짐
  - 32개의 thread는 SIMD와 유사한 방식으로 동작
  - 공유메모리의 액세스도 Warp 단위로 실행.

- 공유 메모리는 높은 bandwidth를 위해 Bank라는 32개의 동일한 사이즈의 메모리 모듈로 나뉨.

GPU 뱅크 충돌

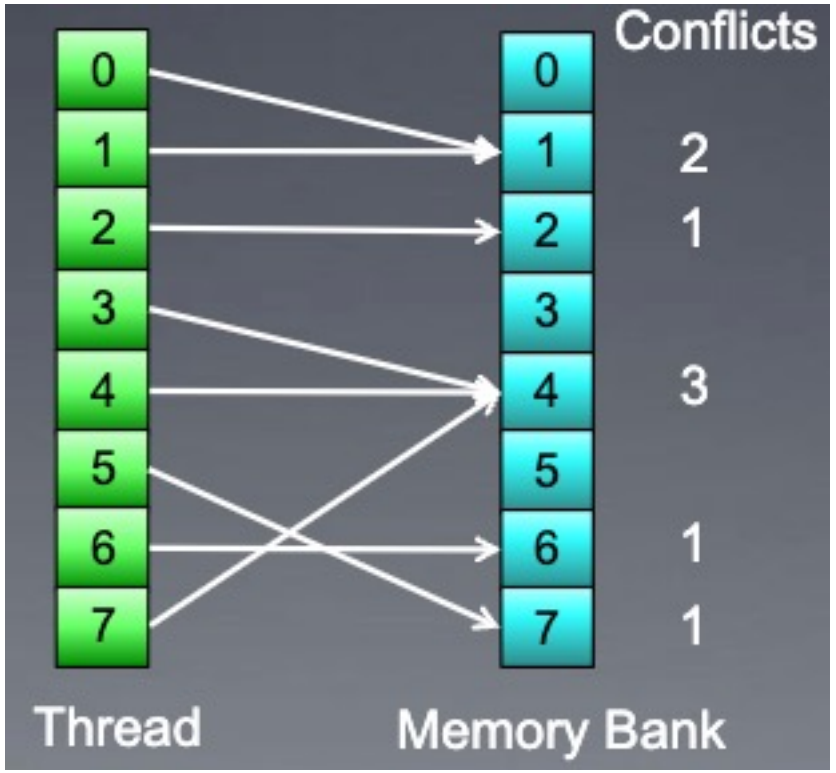
Bank 0	0x00	0x01	0x02	0x03
Bank 1	0x04	0x05	0x06	0x07
Bank 2	0x08	0x09	0x10	0x11

Bank 0	0x00	0x03	0x06	0x09
Bank 1	0x01	0x04	0x07	0x10
Bank 2	0x02	0x05	0x08	0x11

GPU 뱅크충돌

- 뱅크 충돌(bank conflict)
  - 서로 다른 Thread가 하나의 Bank에 접근하게 되면서, 순차적으로 처리하게 되면서 생기는 문제(서로 다른 thread가 순차적으로 연산을 하는 것은 병렬 연산에 의도하지 않은 행동)

- 뱅크 충돌을 피하기 위해서 서로 다른 뱅크를 사용하도록 프로그래밍이 필요함.



GPU Aria global memory

- Sbox 테이블을 글로벌 메모리에 저장하여 사용할 때의 구현

```
__global__ void testCuda_Func(u8* input, u8* output, u8* rkad, u32* S1, u32* S2, u32* X1, u32* X2) {

}

__host__ void testCuda(u8* in. u8*out){
    cudaMalloc((void**)&S1D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&S2D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&X1D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&X2D, TABLE_SIZE * sizeof(u32));

    cudaMemcpy(S1D, S1, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(S2D, S2, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(X1D, X1, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(X2D, X2, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);

    testCuda_Func <<<block_num, thread_num >>> (dev_in, dev_out, dev_rk, S1D, S2D, X1D, X2D);
}
```

GPU Aria Shared memory

- Sbox 테이블을 공유 메모리에 저장하여 사용할 때의 구현

```
__global__ void testCuda_Func(u8* input, u8* output, u8* rkad, u32* S1_G, u32* S2_G, u32* X1_G, u32* X2_G) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ u32 S1[TABLE_SIZE];
    __shared__ u32 S2[TABLE_SIZE];
    __shared__ u32 X1[TABLE_SIZE];
    __shared__ u32 X2[TABLE_SIZE];

    if (threadIdx.x < TABLE_SIZE) {
        S1[threadIdx.x] = S1_G[threadIdx.x];
        S2[threadIdx.x] = S2_G[threadIdx.x];
        X1[threadIdx.x] = X1_G[threadIdx.x];
        X2[threadIdx.x] = X2_G[threadIdx.x];
    }
    __syncthreads();
}

__host__ void testCuda(u8* in. u8*out){
    cudaMalloc((void**)&S1D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&S2D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&X1D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&X2D, TABLE_SIZE * sizeof(u32));

    cudaMemcpy(S1D, S1, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(S2D, S2, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(X1D, X1, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(X2D, X2, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);

    testCuda_Func <<<block_num, thread_num >>> (dev_in, dev_out, dev_rk, S1D, S2D, X1D, X2D);
}
```

GPU Aria Shared memory – bank conflict 해결

```cpp
__global__ void testCuda_Func(u8* input, u8* output, u8* rkad, u32* S1_G, u32* S2_G, u32* X1_G, u32* X2_G) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int warpIndex = threadIdx.x & 9;

    __shared__ u32 S1[TABLE_SIZE][10];
    __shared__ u32 S2[TABLE_SIZE][10];
    __shared__ u32 X1[TABLE_SIZE][10];
    __shared__ u32 X2[TABLE_SIZE][10];

    if (threadIdx.x < TABLE_SIZE) {
        S1[threadIdx.x] = S1_G[threadIdx.x];
        S2[threadIdx.x] = S2_G[threadIdx.x];
        X1[threadIdx.x] = X1_G[threadIdx.x];
        X2[threadIdx.x] = X2_G[threadIdx.x];
    }
    __syncthreads();
}

__host__ void testCuda(u8* in. u8*out){
    cudaMalloc((void**)&S1D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&S2D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&X1D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&X2D, TABLE_SIZE * sizeof(u32));

    cudaMemcpy(S1D, S1, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(S2D, S2, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(X1D, X1, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(X2D, X2, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);

    testCuda_Func <<<block_num, thread_num >>> (dev_in, dev_out, dev_rk, S1D, S2D, X1D, X2D);
}
```

```cpp
__global__ void testCuda_Func(u8* input, u8* output, u8* rkad, u32* S1_G, u32* S2_G, u32* X1_G, u32* X2_G) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int warpIndex = threadIdx.x & 9;

    __shared__ u32 S1[10][TABLE_SIZE];
    __shared__ u32 S2[10][TABLE_SIZE];
    __shared__ u32 X1[10][TABLE_SIZE];
    __shared__ u32 X2[10][TABLE_SIZE];

    if (threadIdx.x < TABLE_SIZE) {
        S1[threadIdx.x] = S1_G[threadIdx.x];
        S2[threadIdx.x] = S2_G[threadIdx.x];
        X1[threadIdx.x] = X1_G[threadIdx.x];
        X2[threadIdx.x] = X2_G[threadIdx.x];
    }
    __syncthreads();
}

__host__ void testCuda(u8* in. u8*out){
    cudaMalloc((void**)&S1D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&S2D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&X1D, TABLE_SIZE * sizeof(u32));
    cudaMalloc((void**)&X2D, TABLE_SIZE * sizeof(u32));

    cudaMemcpy(S1D, S1, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(S2D, S2, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(X1D, X1, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(X2D, X2, TABLE_SIZE * sizeof(u32), cudaMemcpyHostToDevice);

    testCuda_Func <<<block_num, thread_num >>> (dev_in, dev_out, dev_rk, S1D, S2D, X1D, X2D);
}
```

GPU Aria Shared memory – bank conflict 해결

		r1	r2	r3	
	memcpy	**Key_xor**	**odd_round**	**even_round**	**r1+r2+r1+r3**
__shared__ [256][10]	136	40	75	75	약 230 230
__shared__ [10][256]	106	16	60	60	약 170 152
__shared__ [12][256]	93	15	60	60	약 160 150
__shared__ [256]	9	12	59	58	약 130 141
no shared	x	14	80	79	약 170 187

GPU Aria Shared memory – bank conflict 해결

```
__shared__ u32 S1[32][TABLE_SIZE];
__shared__ u32 S2[32][TABLE_SIZE];
__shared__ u32 X1[32][TABLE_SIZE];
__shared__ u32 X2[32][TABLE_SIZE];
```

```
__shared__ u32 S1[TABLE_SIZE][32];
__shared__ u32 S2[TABLE_SIZE][32];
__shared__ u32 X1[TABLE_SIZE][32];
__shared__ u32 X2[TABLE_SIZE][32];
```

B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16	B17	B18	B19B	B20	B21	B22	B23	B24	B25	B26	B27	B28	B29	B30	B31

감 사 합 니 다