

암호연구회 최종 발표 진행 사항

-내용 추가 + 가독성이나 흐름 수정 예정-

<https://youtu.be/og79uZSWROE>

개요

- 목적
- 이유

최종 및 세부 목표

- 최종 목표
- 세부 목표

본 과제에서 사용할 NPU

- 사진 및 스펙

오버뷰 / 전체 플로우

- 그림으로 정리

(추가)2022 암호연구회 결과물을 NPU에서 실행한다면?

- 덕영/세영이 돌려봤고 결과 정리해서 넣을 예정

NPU 컴파일러 사용을 위한 고려 사항

Pytorch 사용

- NPU에서 암호 연산을 수행하기 위해서는 기본적으로 Python 환경 사용
- CPU가 아닌 NPU상에서 동작 시키기 위해서는 딥러닝 프레임워크 중 하나인 Pytorch를 사용해야 함
 - Python의 기본 연산자를 사용할 경우 CPU에서 실행
 - Pytorch는 Tensor를 사용하며, 내장 함수들이 존재
 - 기존 딥러닝 모델 구현에 사용
 - 하드웨어 가속 장치에서 실행 가능

Pytorch 사용한 딥러닝 모델 예시

```
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        # 입력에서 은닉층으로 가는 완전 연결층
        self.fc1 = nn.Linear(input_size, hidden_size)
        # 은닉층에서 출력으로 가는 완전 연결층
        self.fc2 = nn.Linear(hidden_size, output_size)
        # 활성화 함수 ReLU
        self.relu = nn.ReLU()

    def forward(self, x):
        # 입력 -> 은닉층 -> 활성화 함수 -> 출력
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

본 과제의 구현물을 NPU에서 실행시키기 위해서는 Pytorch 사용

NPU 자료형 관점에서의 구현 고려 사항

NPU의 지원 자료형

- 각 NPU가 지원하는 자료형으로 암호 연산을 수행해야 함
 - Furiosa Warboy : INT8
 - Rebellions ATOM : Float16, INT8
- 16, 32비트 단위의 암호 연산을 8비트 단위로 변경
- Multi-precision 구현의 필요성
 - 덧셈 연산 시 캐리가 발생하지만, INT8 자료형만 사용 가능
 - 따라서 8비트 중 4비트에 데이터를 할당하고, 상위 4비트는 캐리 저장을 위해 사용해야 함

Multi-precision 구현

- 리벨리온 ATOM은 INT8 과 FP16 자료형을 사용할 수 있지만 아래와 같은 몇가지 문제점이 발생하므로 INT8만 사용
 - Float 타입은 비트 연산자인 bitwise shift 연산 불가
 - Float16/INT8 조합의 Multi-precision 연산을 구현하려면 자료형 혼합 필요 (Float, Int)
 - 그러나, Multi-precision 연산이 사용되는 ADD는 암호 연산에서 수시로 사용
 - 한 번만 사용되는 것이 아니라 라운드 함수 중간마다 다른 연산들과 함께 사용 됨
 - 단순히 Int(float) 방식으로 변환 불가
 - 타입 변환을 중간마다 계속 할 경우 오버헤드 발생
- 따라서, 두 NPU에서 공통적으로 8/4 조합의 Multi-precision 구현 적용

NPU 연산자 관점에서의 구현 고려 사항

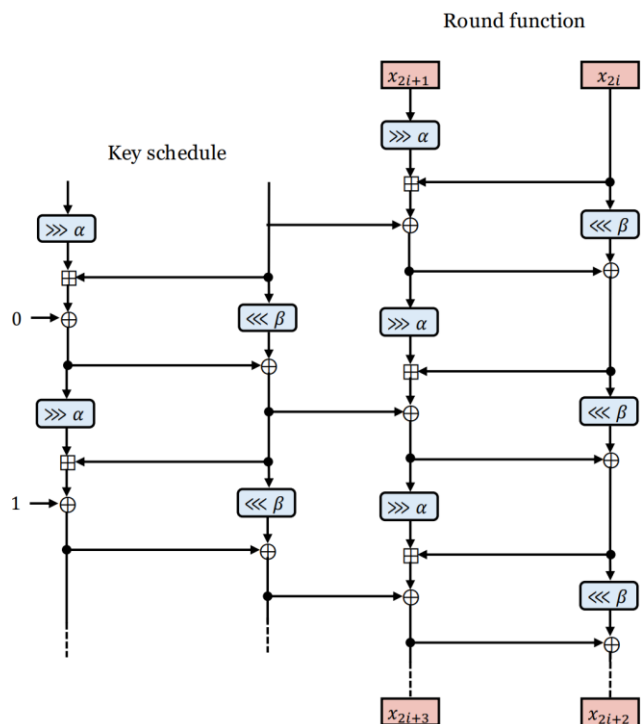
NPU가 지원하는 연산자

- Pytorch를 사용하므로 내장 연산자 활용 가능
- 그러나 모든 연산자가 가속 지원이 되는 것은 아님
 - Add, Multiply 등의 산술 연산은 가속 가능
 - 비트 단위 논리 연산은 가속 불가능

PyTorch OPs	Supported	Accelerated
torch.add	O	O
torch.bitwise_and	O	-
torch.bitwise_not	O	-
torch.bitwise_xor	O	-

NPU가 지원하는 연산자

- 본 과제에서 타겟으로 하는 블록 암호 연산의 경우 NPU에 친화적인 연산을 사용하는가? **NO!**
 - 대부분의 블록 암호는 비트 연산자를 사용함
 - 그러나 비트 연산은 가속 지원되지 않음

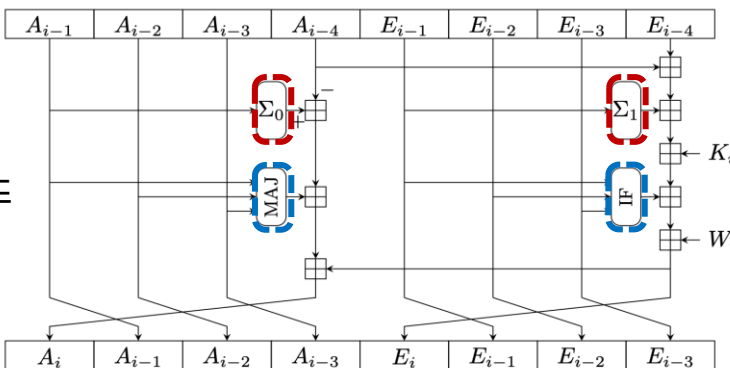


SPECK

메시지
확장

$$W_i = \begin{cases} M_i & 0 \leq i < 16, \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} & 16 \leq i < 80. \end{cases}$$

라운드
함수



SHA512

사용되는 논리 연산

\ggg	Right rotation
\oplus	XOR
\wedge	or
\boxplus	Modulo addition

NPU 상에서의 암호 구현 설계 방식

Hybrid vs. All NPU

Hybrid

CPU에서 비트 연산 + NPU에서 산술 연산

- NPU 산술 연산이 적절하지만 대용량 데이터가 아님
 - NPU는 딥러닝 데이터셋과 같이 대용량에서 효율적
- 비트 연산의 성능은 저하되지 않음
- 스위칭 과정에서 오버헤드 발생 위험

All NPU

NPU에서 모든 연산

- 비트 단위 연산을 산술 연산으로 변경
 - 비트 연산에 비해 계산 복잡도 증가
- 분기/반복이 NPU에서 수행되도록 조작 필요

NPU에서의 암호 구현과 그에 따른 요구사항이나 한계점 분석하는 것이 목적이므로

All NPU 방식 선택

NPU 상에서의 구현 테크닉

NPU 친화적인 구현을 위해 사용한 테크닉

NPU 컴파일러 사용



Pytorch 구조 적용
Tensor 사용

딥러닝 모델 구조 필요

NPU 활용 연산



Multi-precision

NPU의 제한된 자료형 활용

비트연산 to 산술연산

모든 연산에 NPU 활용

연산 효율성 향상



분기/반복 제거

CPU 사용 지점 제거

벡터화

Torch 내장 함수로 효율적 연산

제한적인 환경인 NPU에서 동작 가능하면서 최대한 효율적으로 블록암호를 구현

Pytorch 사용 - 구조

```
class ENC(nn.Module):
    def __init__(self):
        super(ENC, self).__init__()
        self.round_function = ROUND_FUNCTION()

    def forward(self, p, ks):
        x, y = p[0], p[1]
        x, y = self.round_function(torch.stack([x, y]), ks[0])
        x, y = self.round_function(torch.stack([x, y]), ks[1])
        x, y = self.round_function(torch.stack([x, y]), ks[2])
        x, y = self.round_function(torch.stack([x, y]), ks[3])
        x, y = self.round_function(torch.stack([x, y]), ks[4])
        x, y = self.round_function(torch.stack([x, y]), ks[5])
        x, y = self.round_function(torch.stack([x, y]), ks[6])
        x, y = self.round_function(torch.stack([x, y]), ks[7])
        x, y = self.round_function(torch.stack([x, y]), ks[8])
        x, y = self.round_function(torch.stack([x, y]), ks[9])
        x, y = self.round_function(torch.stack([x, y]), ks[10])
        x, y = self.round_function(torch.stack([x, y]), ks[11])
        x, y = self.round_function(torch.stack([x, y]), ks[12])
        x, y = self.round_function(torch.stack([x, y]), ks[13])
        x, y = self.round_function(torch.stack([x, y]), ks[14])
        x, y = self.round_function(torch.stack([x, y]), ks[15])
        x, y = self.round_function(torch.stack([x, y]), ks[16])
        x, y = self.round_function(torch.stack([x, y]), ks[17])
        x, y = self.round_function(torch.stack([x, y]), ks[18])
        x, y = self.round_function(torch.stack([x, y]), ks[19])
        x, y = self.round_function(torch.stack([x, y]), ks[20])
        x, y = self.round_function(torch.stack([x, y]), ks[21])

        return torch.stack([x, y])
```

ENC 함수에 필요한
라운드 함수 클래스 설정

Pytorch 사용 - 구조

```
class ENC(nn.Module):
    def __init__(self):
        super(ENC, self).__init__()
        self.round_function = ROUND_FUNCTION()

    def forward(self, p, ks):
        x, y = p[0], p[1]
        x, y = self.round_function(torch.stack([x, y]), ks[0])
        x, y = self.round_function(torch.stack([x, y]), ks[1])
        x, y = self.round_function(torch.stack([x, y]), ks[2])
        x, y = self.round_function(torch.stack([x, y]), ks[3])
        x, y = self.round_function(torch.stack([x, y]), ks[4])
        x, y = self.round_function(torch.stack([x, y]), ks[5])
        x, y = self.round_function(torch.stack([x, y]), ks[6])
        x, y = self.round_function(torch.stack([x, y]), ks[7])
        x, y = self.round_function(torch.stack([x, y]), ks[8])
        x, y = self.round_function(torch.stack([x, y]), ks[9])
        x, y = self.round_function(torch.stack([x, y]), ks[10])
        x, y = self.round_function(torch.stack([x, y]), ks[11])
        x, y = self.round_function(torch.stack([x, y]), ks[12])
        x, y = self.round_function(torch.stack([x, y]), ks[13])
        x, y = self.round_function(torch.stack([x, y]), ks[14])
        x, y = self.round_function(torch.stack([x, y]), ks[15])
        x, y = self.round_function(torch.stack([x, y]), ks[16])
        x, y = self.round_function(torch.stack([x, y]), ks[17])
        x, y = self.round_function(torch.stack([x, y]), ks[18])
        x, y = self.round_function(torch.stack([x, y]), ks[19])
        x, y = self.round_function(torch.stack([x, y]), ks[20])
        x, y = self.round_function(torch.stack([x, y]), ks[21])

        return torch.stack([x, y])
```

라운드 함수 호출

```
class ROUND_FUNCTION(nn.Module):
    def __init__(self):
        super(ROUND_FUNCTION, self).__init__()
        self.xor_t = XOR_alrt()
        self.ror_7 = ROR_alrt()
        self.rol_2 = ROL_alrt()
        self.add_mp = ADD()

    def forward(self, p, k):
        p[0] = self.ror_7(p[0])
        p[0] = self.add_mp(p[0], p[1])
        p[0] = self.xor_t(p[0], k)
        p[1] = self.rol_2(p[1])
        p[1] = self.xor_t(p[1], p[0])

        return p[0], p[1]
```

라운드 함수에 필요한
XOR, ROT 클래스 설정

라운드 함수의 각 연산 수행

Pytorch 사용 - 구조

```
class ENC(nn.Module):
    def __init__(self):
        super(ENC, self).__init__()
        self.round_function = ROUND_FUNCTION()

    def forward(self, p, ks):
        x, y = p[0], p[1]
        x, y = self.round_function(torch.stack([x, y]), ks[0])
        x, y = self.round_function(torch.stack([x, y]), ks[1])
        x, y = self.round_function(torch.stack([x, y]), ks[2])
        x, y = self.round_function(torch.stack([x, y]), ks[3])
        x, y = self.round_function(torch.stack([x, y]), ks[4])
        x, y = self.round_function(torch.stack([x, y]), ks[5])
        x, y = self.round_function(torch.stack([x, y]), ks[6])
        x, y = self.round_function(torch.stack([x, y]), ks[7])
        x, y = self.round_function(torch.stack([x, y]), ks[8])
        x, y = self.round_function(torch.stack([x, y]), ks[9])
        x, y = self.round_function(torch.stack([x, y]), ks[10])
        x, y = self.round_function(torch.stack([x, y]), ks[11])
        x, y = self.round_function(torch.stack([x, y]), ks[12])
        x, y = self.round_function(torch.stack([x, y]), ks[13])
        x, y = self.round_function(torch.stack([x, y]), ks[14])
        x, y = self.round_function(torch.stack([x, y]), ks[15])
        x, y = self.round_function(torch.stack([x, y]), ks[16])
        x, y = self.round_function(torch.stack([x, y]), ks[17])
        x, y = self.round_function(torch.stack([x, y]), ks[18])
        x, y = self.round_function(torch.stack([x, y]), ks[19])
        x, y = self.round_function(torch.stack([x, y]), ks[20])
        x, y = self.round_function(torch.stack([x, y]), ks[21])

        return torch.stack([x, y])
```

라운드 함수 호출



```
class ROUND_FUNCTION(nn.Module):
    def __init__(self):
        super(ROUND_FUNCTION, self).__init__()
        self.xor_t = XOR_alrt()
        self.ror_7 = ROR_alrt()
        self.rol_2 = ROL_alrt()
        self.add_mp = ADD()

    def forward(self, p, k):
        p[0] = self.ror_7(p[0])
        p[0] = self.add_mp(p[0], p[1])
        p[0] = self.xor_t(p[0], k)
        p[1] = self.rol_2(p[1])
        p[1] = self.xor_t(p[1], p[0])

        return p[0], p[1]
```



```
class ROR_alrt(nn.Module):
```

```
class ADD(nn.Module):
```

```
class XOR_alrt(nn.Module):
```

```
class ROL_alrt(nn.Module):
```

```
class XOR_alrt(nn.Module):
```

라운드 함수에 필요한
XOR, ROT 클래스 설정

라운드 함수의 각 연산 수행

해당 함수들 또한
Pytorch Module로 구현되어 있음

Pytorch 사용 - 텐서 및 내장 함수

- 사용되는 변수들은 모두 텐서로 정의

```
pt = torch.tensor([[ 6, 5, 7, 4], [ 6, 9, 4, 12]], dtype=torch.uint8)
```

```
shift_1 = torch.tensor(8,dtype=torch.uint8)
shift_2 = torch.tensor(2,dtype=torch.uint8)
mask_1 = torch.tensor(0x01,dtype=torch.uint8)
mask_2 = torch.tensor(0x0E,dtype=torch.uint8)
```

- 내장 함수 사용하여 텐서 연산

```
l = torch.tensor(16, dtype = torch.uint8)
s = torch.sub(15, torch.div(r, 4, rounding_mode='trunc').to(torch.uint8))
idx = torch.remainder(torch.add(torch.arange(16), s), l)
idx_plus_1 = torch.remainder(torch.add(idx, 1), l)
```

일반적 산술 연산 (덧셈, 뺄셈, 나머지, 나눗셈)

```
return torch.stack([x, y])
```

stack (묶음)

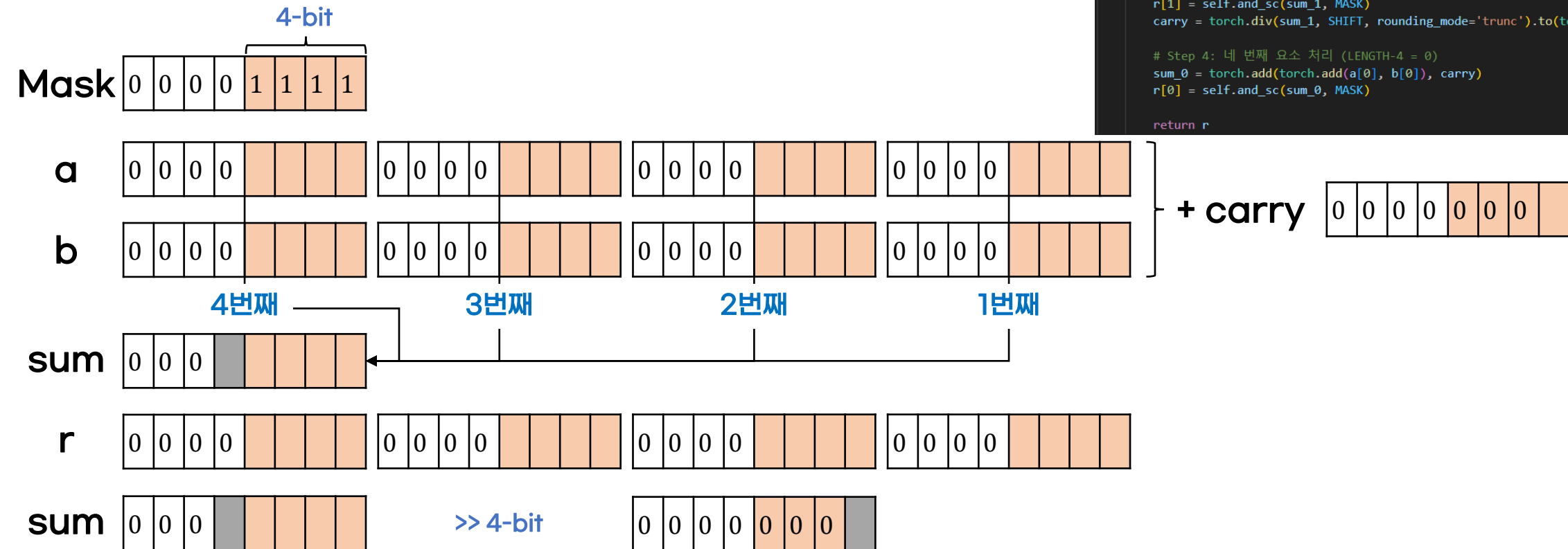
```
a.unsqueeze(1)
b.unsqueeze(1)
```

Unsqueeze (차원 축소)

NPU 활용을 위해 텐서를 사용하며, 산술 연산 및 텐서 조작 연산 가능

Multi-precision addition 구현

- NPU에서 사용 가능한 자료형 ($2n$ -bit)에 따라 조합 결정
 - $2n$ -bit / n -bit (e.g., 8-bit / 4-bit)
- 아래와 같이 8-bit 중 4-bit만 사용하며, 배열 형태로 구성
 - 16-bit의 데이터를 표현하기 위해 8-bit 변수 4개 사용
 - 연산 대상 길이에 따라 배열 길이가 달라짐



```
class ADD(nn.Module): # a : [4-bit, 4-bit, 4-bit, 4-bit], b: [4-bit, 4-bit, 4-bit, 4-bit]
    def __init__(self):
        super(ADD, self).__init__()
        self.and_sc = AND_SC()

    def forward(self, a, b):
        MASK = torch.tensor(0x0F, dtype=torch.uint8) # 4-bit 마스크
        SHIFT = torch.tensor(0x10, dtype=torch.uint8) # 4-bit 시프트 값
        carry = torch.tensor(0x00, dtype=torch.uint8) # 초기 캐리
        r = torch.zeros_like(a, dtype=torch.uint8) # 결과 저장할 텐서

        # Step 1: 마지막 요소 처리 (LENGTH-1 = 3)
        sum_3 = torch.add(torch.add(a[3], b[3]), carry)
        r[3] = self.and_sc(sum_3, MASK)
        carry = torch.div(sum_3, SHIFT, rounding_mode='trunc').to(torch.uint8)

        # Step 2: 두 번째 요소 처리 (LENGTH-2 = 2)
        sum_2 = torch.add(torch.add(a[2], b[2]), carry)
        r[2] = self.and_sc(sum_2, MASK)
        carry = torch.div(sum_2, SHIFT, rounding_mode='trunc').to(torch.uint8)

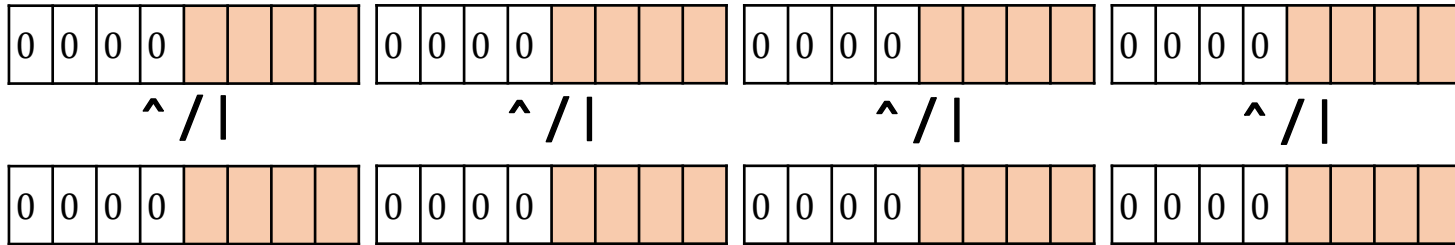
        # Step 3: 세 번째 요소 처리 (LENGTH-3 = 1)
        sum_1 = torch.add(torch.add(a[1], b[1]), carry)
        r[1] = self.and_sc(sum_1, MASK)
        carry = torch.div(sum_1, SHIFT, rounding_mode='trunc').to(torch.uint8)

        # Step 4: 네 번째 요소 처리 (LENGTH-4 = 0)
        sum_0 = torch.add(torch.add(a[0], b[0]), carry)
        r[0] = self.and_sc(sum_0, MASK)

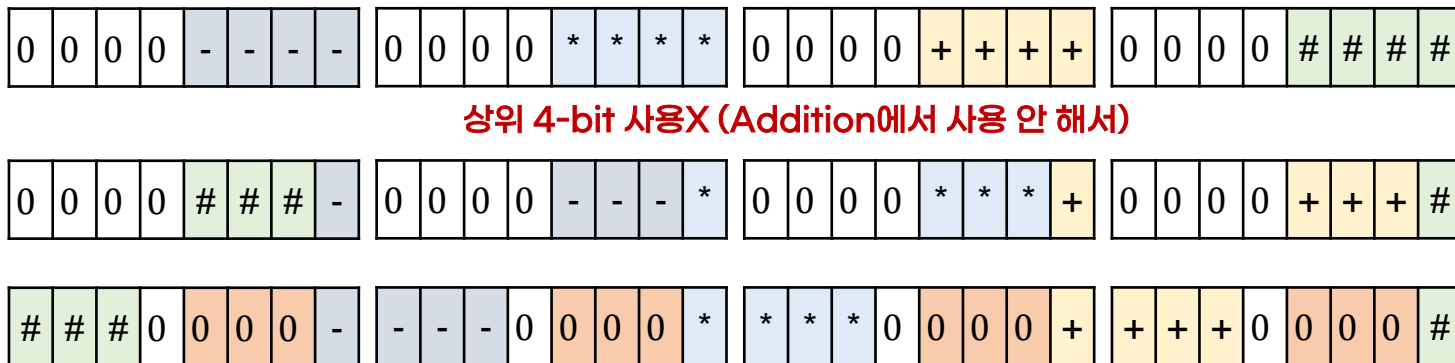
        return r
```

Multi-precision을 사용한 비트 연산 로직

- Addition 연산에서 사용하는 형태로 연산하기 위해 다른 연산들 또한 아래와 같이 구성
 - 16-bitwise XOR / OR



- 16-bitwise Rotation (Left, Right)
 - ex: >>>3



상위 4-bit 사용X (Addition에서 사용 안 해서)



X !!!

NPU 사용 위해 비트 연산을 산술 연산으로 변경

- NPU는 비트 연산 지원 X
 - 컴파일 불가

XOR 연산

→ 각 비트에 대해 아래와 같은 연산 수행 후,
가중치 곱셈하여 비트 별 결과 합산

$$(a + b) - 2 \cdot (a \cdot b)$$

NOT 연산

→ 각 비트를 1에서 뺀 후, 가중치 곱셈

AND 연산

→ 각 비트에 대한 곱셈 후, 가중치를 곱함

```
class XOR_alrt(nn.Module):
    def forward(self, a, b):
        # 각 비트에 해당하는 가중치 (2^i)를 미리 계산
        #bit_weights = torch.tensor([torch.pow(2, i) for i in range(8)], dtype=torch.uint8) # [1, 2, 4, 8, 16, 32, 64, 128]
        bit_weights = torch.pow(torch.tensor(2, dtype=torch.uint8), torch.arange(8, dtype=torch.uint8))

        bit_a = torch remainder(torch.div(a.unsqueeze(1), bit_weights, rounding_mode='trunc'), 2) # [n, 8] 형태
        bit_b = torch remainder(torch.div(b.unsqueeze(1), bit_weights, rounding_mode='trunc'), 2) # [n, 8] 형태

        # XOR 연산: (bit_a + bit_b) - 2 * (bit_a * bit_b)
        xor_bits = torch.sub(torch.add(bit_a, bit_b), torch.mul(2, torch.mul(bit_a, bit_b)))

        # 각 비트 가중치를 곱해서 최종 결과 계산 (비트별 결과를 합산)
        result = torch.sum(torch.multiply(xor_bits, bit_weights), dim=1).to(torch.uint8)

        return result
```

```
class NOT_t(nn.Module):
    def forward(self, a):
        bit_weights = torch.pow(torch.tensor(2, dtype=torch.uint8), torch.arange(8, dtype=torch.uint8)) # [1, 2, 4, 8, 16, 32, 64, 128]
        bit_a = torch remainder(torch.div(a.unsqueeze(1), bit_weights, rounding_mode='trunc'), 2) # [n, 8] 형태

        not_bit = torch.sub(1, bit_a)
        result = torch.sum(torch.multiply(not_bit, bit_weights), dim=-1).to(torch.uint8)

        return result
```

```
class AND_SC(nn.Module):
    def forward(self, a, b):

        # 각 비트에 해당하는 가중치를 미리 계산
        bit_weights = torch.pow(torch.tensor(2, dtype=torch.uint8), torch.arange(8, dtype=torch.uint8))
        # 8비트씩 비트 연산을 하기 위해 각 비트 추출 (브로드캐스트 적용)
        bit_a = torch remainder(torch.div(a, bit_weights, rounding_mode='trunc'), 2) # 비트 추출
        bit_b = torch remainder(torch.div(b, bit_weights, rounding_mode='trunc'), 2)

        # AND 연산 수행 (각 비트에 대해 동시에)
        and_bits = torch.multiply(bit_a, bit_b)

        # 각 비트 가중치(2^i)를 곱해서 최종 결과 계산
        result = torch.sum(torch.multiply(and_bits, bit_weights)).to(torch.uint8) # 각 비트에 대해 AND 결과를 가중합

        return result
```

NPU 사용 위해 비트 연산을 산술 연산으로 변경

- NPU는 비트 연산 지원 X
 - 컴파일 불가

Rotation 연산

→ Bitwise shift 연산이 불가하므로 대체

Left shift : Multiply

Right shift : Division

```
class ROR_NOT_ZERO(nn.Module):
    def __init__(self):
        super(ROR_NOT_ZERO, self).__init__()
        self.or_s = OR_s()
        self.and_s = AND_s()

    def forward(self, x, r, mask_1, mask_2, sft_1, sft_2):
        l = torch.tensor(16, dtype = torch.uint8)
        s = torch.sub(15, torch.div(r, 4, rounding_mode='trunc').to(torch.uint8)) # s = 15 - (r // 4)
        idx = torch.remainder(torch.add(torch.arange(16), s), 1)
        idx_plus_1 = torch.remainder(torch.add(idx, 1), 1)

        t = torch.zeros(16, dtype=torch.int8)

        t[0] = self.or_s(self.and_s(torch.multiply(x[idx[0]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[0]], sft_2), mask_2))
        t[1] = self.or_s(self.and_s(torch.multiply(x[idx[1]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[1]], sft_2), mask_2))
        t[2] = self.or_s(self.and_s(torch.multiply(x[idx[2]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[2]], sft_2), mask_2))
        t[3] = self.or_s(self.and_s(torch.multiply(x[idx[3]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[3]], sft_2), mask_2))
        t[4] = self.or_s(self.and_s(torch.multiply(x[idx[4]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[4]], sft_2), mask_2))
        t[5] = self.or_s(self.and_s(torch.multiply(x[idx[5]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[5]], sft_2), mask_2))
        t[6] = self.or_s(self.and_s(torch.multiply(x[idx[6]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[6]], sft_2), mask_2))
        t[7] = self.or_s(self.and_s(torch.multiply(x[idx[7]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[7]], sft_2), mask_2))
        t[8] = self.or_s(self.and_s(torch.multiply(x[idx[8]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[8]], sft_2), mask_2))
        t[9] = self.or_s(self.and_s(torch.multiply(x[idx[9]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[9]], sft_2), mask_2))
        t[10] = self.or_s(self.and_s(torch.multiply(x[idx[10]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[10]], sft_2), mask_2))
        t[11] = self.or_s(self.and_s(torch.multiply(x[idx[11]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[11]], sft_2), mask_2))
        t[12] = self.or_s(self.and_s(torch.multiply(x[idx[12]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[12]], sft_2), mask_2))
        t[13] = self.or_s(self.and_s(torch.multiply(x[idx[13]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[13]], sft_2), mask_2))
        t[14] = self.or_s(self.and_s(torch.multiply(x[idx[14]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[14]], sft_2), mask_2))
        t[15] = self.or_s(self.and_s(torch.multiply(x[idx[15]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[15]], sft_2), mask_2))

        return t
```

CPU 사용 피하기 위해 분기/반복 제거

- SHA512의 경우 라운드에 따라 라운드 함수가 다름
 - 따라서, 조건문+반복문이 사용됨
- 라운드 함수를 두 종류로 나누고, Unloop 시킴

```
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[0], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[1], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[2], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[3], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[4], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[5], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[6], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[7], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[8], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[9], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[10], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[11], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[12], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[13], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[14], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16 rnd[15], a, b, c, d, e, f, g, h, kk, W)
```

```
k = torch.zeros(16, dtype=torch.uint8)
k = kk[round]
w = torch.zeros(16, dtype=torch.uint8)
w = W[round]
t1 = self.t1(e, f, g, h, k, w)
t2 = self.t2(a, b, c)
h = g
g = f
f = e
e = self.add_(d, t1)
d = c
c = b
b = a
a = self.add_(t1, t2)
```

```
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[16], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[17], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[18], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[19], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[20], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[21], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[22], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[23], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[24], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[25], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[26], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[27], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[28], a, b, c, d, e, f, g, h, kk, W)
a, b, c, d, e, f, g, h = self.roundfunc16a rnd[29], a, b, c, d, e, f, g, h, kk, W)
```

```
k = torch.zeros(16, dtype=torch.uint8)
k = kk[round]
w = torch.zeros(16, dtype=torch.uint8)
w = self.next_16a(W)
t1 = self.t1(e, f, g, h, k, w)
t2 = self.t2(a, b, c)
h = g
g = f
f = e
e = self.add_(d, t1)
d = c
c = b
b = a
a = self.add_(t1, t2)
```

CPU 사용 피하기 위해 분기/반복 제거

- SHA512의 Rotation 함수 구현 시 조건문 필요
 - 14,18,41 비트 회전 등 여러 경우가 존재 (SPECK은 간단)
 - 이를 4개의 케이스로 나누어 따로 구현
 - Shift 하지 않는 경우 0으로 나누어야 함 → 제로 디비전 에러 발생
 - 따라서, rotation 0인 경우와 아닌 경우를 나누어 구현

0 비트 shift 필요 (div 연산 미사용)

```
def forward(self, x, r):
    mask_1 = torch.tensor(0x0F, dtype=torch.uint8)
    mask_2 = torch.tensor(0x0F, dtype=torch.uint8)
    sft_1 = torch.tensor(16, dtype=torch.uint8) # left : 2^sft_1 = 16
    l = torch.tensor(16, dtype=torch.uint8)
    s = torch.sub(15, torch.div(r, 4, rounding_mode='trunc').to(torch.uint8)) # s = 15 - (r // 4)
    idx = torch.remainder(torch.add(torch.arange(16), s), 1)
    idx_plus_1 = torch.remainder(torch.add(idx, 1), 1)
    # idx = (torch.arange(16) + s) % 1
    # idx_plus_1 = (idx + 1) % 1
    t = torch.zeros(16, dtype=torch.uint8)

    t[0] = self.or_s(self.and_s(torch.mul(x[idx[0]], sft_1), mask_1), self.and_s((x[idx_plus_1[0]]), mask_2))
    t[1] = self.or_s(self.and_s(torch.mul(x[idx[1]], sft_1), mask_1), self.and_s((x[idx_plus_1[1]]), mask_2))
    t[2] = self.or_s(self.and_s(torch.mul(x[idx[2]], sft_1), mask_1), self.and_s((x[idx_plus_1[2]]), mask_2))
    t[3] = self.or_s(self.and_s(torch.mul(x[idx[3]], sft_1), mask_1), self.and_s((x[idx_plus_1[3]]), mask_2))
    t[4] = self.or_s(self.and_s(torch.mul(x[idx[4]], sft_1), mask_1), self.and_s((x[idx_plus_1[4]]), mask_2))
    t[5] = self.or_s(self.and_s(torch.mul(x[idx[5]], sft_1), mask_1), self.and_s((x[idx_plus_1[5]]), mask_2))
    t[6] = self.or_s(self.and_s(torch.mul(x[idx[6]], sft_1), mask_1), self.and_s((x[idx_plus_1[6]]), mask_2))
    t[7] = self.or_s(self.and_s(torch.mul(x[idx[7]], sft_1), mask_1), self.and_s((x[idx_plus_1[7]]), mask_2))
    t[8] = self.or_s(self.and_s(torch.mul(x[idx[8]], sft_1), mask_1), self.and_s((x[idx_plus_1[8]]), mask_2))
    t[9] = self.or_s(self.and_s(torch.mul(x[idx[9]], sft_1), mask_1), self.and_s((x[idx_plus_1[9]]), mask_2))
    t[10] = self.or_s(self.and_s(torch.mul(x[idx[10]], sft_1), mask_1), self.and_s((x[idx_plus_1[10]]), mask_2))
    t[11] = self.or_s(self.and_s(torch.mul(x[idx[11]], sft_1), mask_1), self.and_s((x[idx_plus_1[11]]), mask_2))
    t[12] = self.or_s(self.and_s(torch.mul(x[idx[12]], sft_1), mask_1), self.and_s((x[idx_plus_1[12]]), mask_2))
    t[13] = self.or_s(self.and_s(torch.mul(x[idx[13]], sft_1), mask_1), self.and_s((x[idx_plus_1[13]]), mask_2))
    t[14] = self.or_s(self.and_s(torch.mul(x[idx[14]], sft_1), mask_1), self.and_s((x[idx_plus_1[14]]), mask_2))
    t[15] = self.or_s(self.and_s(torch.mul(x[idx[15]], sft_1), mask_1), self.and_s((x[idx_plus_1[15]]), mask_2))
```

0 비트 shift 필요 X (div로 right shift)

```
def forward(self, x, r, mask_1, mask_2, sft_1, sft_2):
    l = torch.tensor(16, dtype=torch.uint8)
    s = torch.sub(15, torch.div(r, 4, rounding_mode='trunc').to(torch.uint8)) # s = 15 - (r // 4)
    idx = torch.remainder(torch.add(torch.arange(16), s), 1)
    idx_plus_1 = torch.remainder(torch.add(idx, 1), 1)

    t = torch.zeros(16, dtype=torch.int8)

    t[0] = self.or_s(self.and_s(torch.multiply(x[idx[0]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[0]], sft_2), mask_2))
    t[1] = self.or_s(self.and_s(torch.multiply(x[idx[1]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[1]], sft_2), mask_2))
    t[2] = self.or_s(self.and_s(torch.multiply(x[idx[2]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[2]], sft_2), mask_2))
    t[3] = self.or_s(self.and_s(torch.multiply(x[idx[3]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[3]], sft_2), mask_2))
    t[4] = self.or_s(self.and_s(torch.multiply(x[idx[4]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[4]], sft_2), mask_2))
    t[5] = self.or_s(self.and_s(torch.multiply(x[idx[5]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[5]], sft_2), mask_2))
    t[6] = self.or_s(self.and_s(torch.multiply(x[idx[6]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[6]], sft_2), mask_2))
    t[7] = self.or_s(self.and_s(torch.multiply(x[idx[7]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[7]], sft_2), mask_2))
    t[8] = self.or_s(self.and_s(torch.multiply(x[idx[8]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[8]], sft_2), mask_2))
    t[9] = self.or_s(self.and_s(torch.multiply(x[idx[9]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[9]], sft_2), mask_2))
    t[10] = self.or_s(self.and_s(torch.multiply(x[idx[10]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[10]], sft_2), mask_2))
    t[11] = self.or_s(self.and_s(torch.multiply(x[idx[11]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[11]], sft_2), mask_2))
    t[12] = self.or_s(self.and_s(torch.multiply(x[idx[12]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[12]], sft_2), mask_2))
    t[13] = self.or_s(self.and_s(torch.multiply(x[idx[13]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[13]], sft_2), mask_2))
    t[14] = self.or_s(self.and_s(torch.multiply(x[idx[14]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[14]], sft_2), mask_2))
    t[15] = self.or_s(self.and_s(torch.multiply(x[idx[15]], sft_1), mask_1), self.and_s(torch.div(x[idx_plus_1[15]], sft_2), mask_2))
```

CPU 사용 피하기 위해 분기/반복 제거

- SHA512의 Rotation 함수 구현 시 조건문 필요
 - 14,18,41 비트 회전 등 여러 경우가 존재
 - 이를 4개의 케이스로 나누어 따로 구현
 - Shift 하지 않는 경우 0으로 나누어야 함 → 제로 디비전 에러 발생
 - 따라서, rotation 0인 경우와 아닌 경우를 나누어 구현

```
class SIGMA0(nn.Module):
    def __init__(self):
        super(SIGMA0, self).__init__()
        self.xor = XOR t()
        self.ror_28 = ROR_0()
        self.ror_34 = ROR_2()
        self.ror_39 = ROR_3()

    def forward(self, x):
        return self.xor(self.xor(self.ror_28(x,28), self.ror_34(x,34)), self.ror_39(x,39))
```

Rotation 함수 호출 시 4가지 케이스 중 선택
8/4 조합의 Multi-precision을 사용하므로
 $ROR_n \rightarrow n = \text{회전 비트 수} // 4$

```
class ROR_1(nn.Module): # ror = 1
    def __init__(self):
        super(ROR_1, self).__init__()
        self.r_not_zero = ROR_NOT_ZERO()

    def forward(self, x, r):
        mask_1 = torch.tensor(0x08, dtype=torch.uint8)
        mask_2 = torch.tensor(0x07, dtype=torch.uint8)
        sft_1 = torch.tensor(8, dtype=torch.uint8)
        sft_2 = torch.tensor(2, dtype=torch.uint8)
        t = self.r_not_zero(x, r, mask_1, mask_2, sft_1, sft_2)
        return t
```

```
class ROR_2(nn.Module): # ror = 2
    def __init__(self):
        super(ROR_2, self).__init__()
        self.r_not_zero = ROR_NOT_ZERO()

    def forward(self, x, r):
        mask_1 = torch.tensor(0x0C, dtype=torch.uint8)
        mask_2 = torch.tensor(0x03, dtype=torch.uint8)
        sft = torch.tensor(4, dtype=torch.uint8)
        t = self.r_not_zero(x, r, mask_1, mask_2, sft, sft)

        return t
```

```
class ROR_3(nn.Module): # ror = 3
    def __init__(self):
        super(ROR_3, self).__init__()
        self.r_not_zero = ROR_NOT_ZERO()

    def forward(self, x, r):
        mask_1 = torch.tensor(0x0E, dtype=torch.uint8)
        mask_2 = torch.tensor(0x01, dtype=torch.uint8)
        sft_1 = torch.tensor(2, dtype=torch.uint8)
        sft_2 = torch.tensor(8, dtype=torch.uint8)

        t = self.r_not_zero(x, r, mask_1, mask_2, sft_1, sft_2)
        return t
```

NPU 상에서의 연산 벡터화

- NPU에서 효율적으로 연산하기 위해 벡터화 시킨 후 텐서 연산
- 특히, 각 요소에 접근하는 반복문이 언루프 된 경우 벡터화를 함께 적용해야 효과적

Case 1: 비트 단위 연산 벡터화

- 원래의 bitwise 연산은 아래 과정이 필요 없음
- 그러나, 산술 연산으로 변경했으므로 비트 연산을 위한 텐서 조작 필요
- 연산 대상을 8비트로 분해하고, 분해된 8비트를 벡터화 하여 한번에 연산 수행
 - a와 b에 대한 비트 추출 시, unsqueeze 통해 비트 가중치와 형태 맞춤 → (n,8) 형태

```
bit_a = torch remainder(torch.div a.unsqueeze(1), bit_weights, rounding_mode='trunc'), 2)  
bit_b = torch remainder(torch.div b.unsqueeze(1), bit_weights, rounding_mode='trunc'), 2)
```

NPU 상에서의 연산 벡터화

- NPU에서 효율적으로 연산하기 위해 벡터화 시킨 후 텐서 연산
- 특히, 각 요소에 접근하는 반복문이 언루프 된 경우 벡터화를 함께 적용해야 효과적

Case 2: index 16개를 한번에 계산

- Rotation은 Shift, or 을 혼용하여 구현 → index 조작 필요
- For문 사용하지 않고 torch.arange(16)을 통해 0~15까지 한번에 생성하고 모든 index 계산

```
l = torch.tensor(16, dtype = torch.uint8)
s = torch.sub(15, torch.div(r, 4, rounding_mode='trunc').to(torch.uint8))
idx = torch.remainder(torch.add(torch.arange(16), s), l)
idx_plus_1 = torch.remainder(torch.add(idx, 1), l)
```

반복문 방식:

```
python
result = []
for i in range(len(a)):
    result.append(a[i] + b[i])
```

벡터화된 방식:

```
python
result = a + b # 벡터화된 연산
```


NPU 상에서의 연산 벡터화

- NPU에서 효율적으로 연산하기 위해 벡터화 시킨 후 텐서 연산
- 특히, 각 요소에 접근하는 반복문이 언루프 된 경우 벡터화를 함께 적용해야 효과적

Case 3: 비트별 가중치를 배열로 생성

- `torch.arange(8) → [0,1, ..., 7]`
- `torch.pow(2, torch.arange(8)) → [1,2, ..., 128]`
- 8비트 형태로 추출되었으므로, 비트 가중치 (`bit_weights`)와 한번에 브로드캐스트 연산
- 비트별 결과 합산

```
#bit weights = torch.tensor([torch.pow(2, i) for i in range(8)], dtype=torch.uint8) # [1, 2, 4, 8, 16, 32, 64, 128]
bit_weights = torch.pow(torch.tensor(2, dtype=torch.uint8), torch.arange(8, dtype=torch.uint8))

bit_a = torch remainder(torch.div(a.unsqueeze(1), bit_weights, rounding_mode='trunc'), 2) # [n, 8] 형태
bit_b = torch remainder(torch.div(b.unsqueeze(1), bit_weights, rounding_mode='trunc'), 2) # [n, 8] 형태

# XOR 연산: (bit_a + bit_b) - 2 * (bit_a * bit_b)
xor_bits = torch.sub(torch.add(bit_a, bit_b), torch.mul(2, torch.mul(bit_a, bit_b)))

# 각 비트 가중치를 곱해서 최종 결과 계산 (비트별 결과를 합산)
result = torch.sum(torch.multiply(xor_bits, bit_weights), dim=1).to(torch.uint8)
```


NPU 상에서의 연산 벡터화

- 하지만, Multi-precision 덧셈은 순차적 연산 필요
 - 4비트 데이터 연산마다 **순차적으로 캐리**가 발생
 - 발생한 캐리는 **다음 단위 연산에 사용**
- 따라서 반복되는 로직임에도, 순차적 연산 수행

index
최하위 블록 [3] { 캐리 덧셈
캐리 발생

그 상위 블록 [2] { 캐리 덧셈
캐리 발생

그 상위 블록 [1] { 캐리 덧셈
캐리 발생

최상위 블록 [0] { 캐리 덧셈

```
class ADD(nn.Module): # a : [4-bit, 4-bit, 4-bit, 4-bit], b: [4-bit, 4-bit, 4-bit, 4-bit]

    def __init__(self):
        super(ADD, self).__init__()
        self.and_sc = AND_SC()

    def forward(self, a, b):
        MASK = torch.tensor(0x0F, dtype=torch.uint8) # 4-bit 마스크
        SHIFT = torch.tensor(0x10, dtype=torch.uint8) # 4-bit 시프트 값
        carry = torch.tensor(0x00, dtype=torch.uint8) # 초기 캐리
        r = torch.zeros_like(a, dtype=torch.uint8) # 결과 저장할 텐서

        # Step 1: 마지막 요소 처리 (LENGTH-1 = 3)
        sum_3 = torch.add(torch.add(a[3], b[3]), carry)
        r[3] = self.and_sc(sum_3, MASK)
        carry = torch.div(sum_3, SHIFT, rounding_mode='trunc').to(torch.uint8)

        # Step 2: 두 번째 요소 처리 (LENGTH-2 = 2)
        sum_2 = torch.add(torch.add(a[2], b[2]), carry)
        r[2] = self.and_sc(sum_2, MASK)
        carry = torch.div(sum_2, SHIFT, rounding_mode='trunc').to(torch.uint8)

        # Step 3: 세 번째 요소 처리 (LENGTH-3 = 1)
        sum_1 = torch.add(torch.add(a[1], b[1]), carry)
        r[1] = self.and_sc(sum_1, MASK)
        carry = torch.div(sum_1, SHIFT, rounding_mode='trunc').to(torch.uint8)

        # Step 4: 네 번째 요소 처리 (LENGTH-4 = 0)
        sum_0 = torch.add(torch.add(a[0], b[0]), carry)
        r[0] = self.and_sc(sum_0, MASK)

        return r
```

NPU 상에서의 실행

TorchScript 및 ONNX 모델로 변환

- Torch.jit.script로 변경 후, ONNX로 변환

- **Torch.jit.script**

- PyTorch 모델의 모든 연산을 분석하고, TorchScript라는 중간 표현으로 변환
- PyTorch는 기본적으로 동적 그래프 방식으로 동작하지만, TorchScript는 이를 **정적 그래프 방식**으로 변환
- 변환된 모델은 ONNX로 변환하기 위한 중간 단계로 사용
 - Pytorch 모델을 바로 ONNX로 변환하면 아래와 같은 에러 발생하므로 중간 단계 필요

torch.onnx.symbolic_registry.UnsupportedOperatorError:
Exporting the operator ::resolve_conj to ONNX opset version 11 is not supported

- **ONNX** (Open Neural Network Exchange)

- 딥러닝 프레임워크와 하드웨어 호환성 보장을 위한 표준 포맷
- 그래프 형태 → Node, Tensor, Operator로 구성
- 장점
 - 여러 하드웨어에서의 최적화 지원 (GPU, NPU, FPGA 등)
- 단점
 - 연산자 미지원 및 불일치 발생 가능
 - 복잡한 모델 변환 시 변환 실패나 성능 저하 존재

```
scripted_model = torch.jit.script(enc)
torch.onnx.export(scripted_model, (pt, ks), "speck_enc_op.onnx", opset_version=14)
```



```
BS BELDC2BELpytorchSUBACK1.12.0:W
D
NAKOnnx::ScatterND_20725DC2NAKOnnx::ScatterND_21968SUB
Identity_0"BS Identity
=
DC1Onnx::Equal_20721DC2DC2Onnx::Concat_21967SUB
Identity_1"BS Identity
D
NAKOnnx::ScatterND_20725DC2NAKOnnx::ScatterND_21966SUB
Identity_2"BS Identity
=
DC1Onnx::Equal_20721DC2DC2Onnx::Concat_21965SUB
Identity_3"BS Identity
D
NAKOnnx::ScatterND_20731DC2NAKOnnx::ScatterND_21964SUB
Identity_4"BS Identity
=
DC1Onnx::Equal_20721DC2DC2Onnx::Concat_21963SUB
Identity_5"BS Identity
8
SIOnnx::Div_20751DC2SIOnnx::Div_21962SUB
Identity_6"BS Identity
D
NAKOnnx::ScatterND_20728DC2NAKOnnx::ScatterND_21961SUB
Identity_7"BS Identity
```

NPU에서 실행하기 위한 컴파일

- 리벨리온 ATOM

```
input_info=[("p.1", [2, 4], torch.uint8), ("ks.1", [22, 4], torch.uint8)],  
compiled_model = rebel.compile_from_torch(scripted_model, input_info, npu="RBLN-CA02")  
compiled_model.save("speck_npu.rbln")  
module = rebel.Runtime("speck_npu.rbln", tensor_type="pt")
```

- 퓨리오사 Warboy (컴파일 방식은 맞음, 하지만 내부 컴파일 에러)

```
model_path = "speck_enc_op.onnx"  
# Furiosa NPU에서 사용할 세션 생성  
speck_enc = session.create(model_path)
```

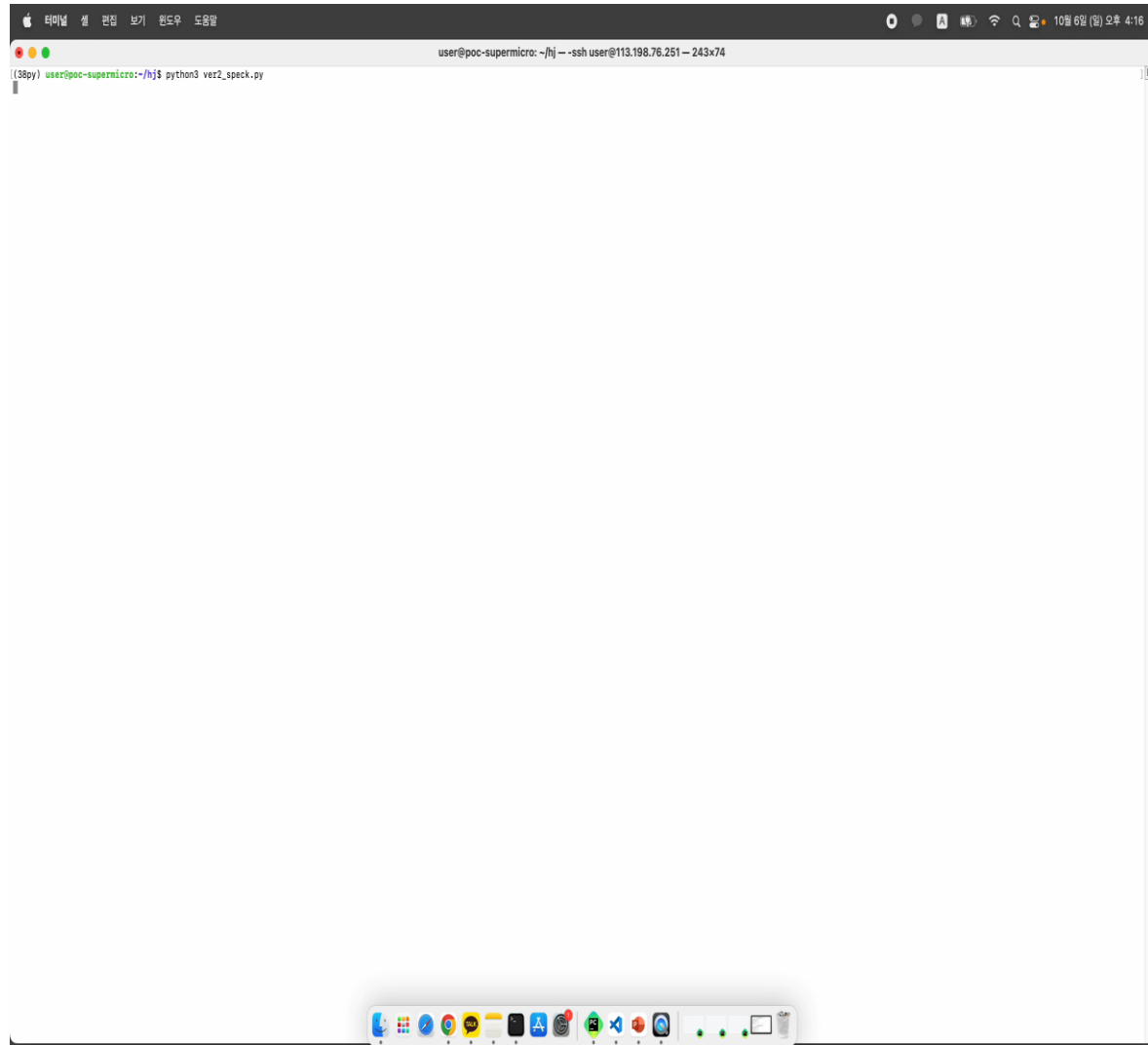
```
# 입력 텐서 준비  
inputs = {"p.1": pt, "ks.1": ks}  
#inputs = {"p.1", "ks.1"}
```

```
# NPU에서 모델 실행  
outputs = speck_enc.run(inputs)
```

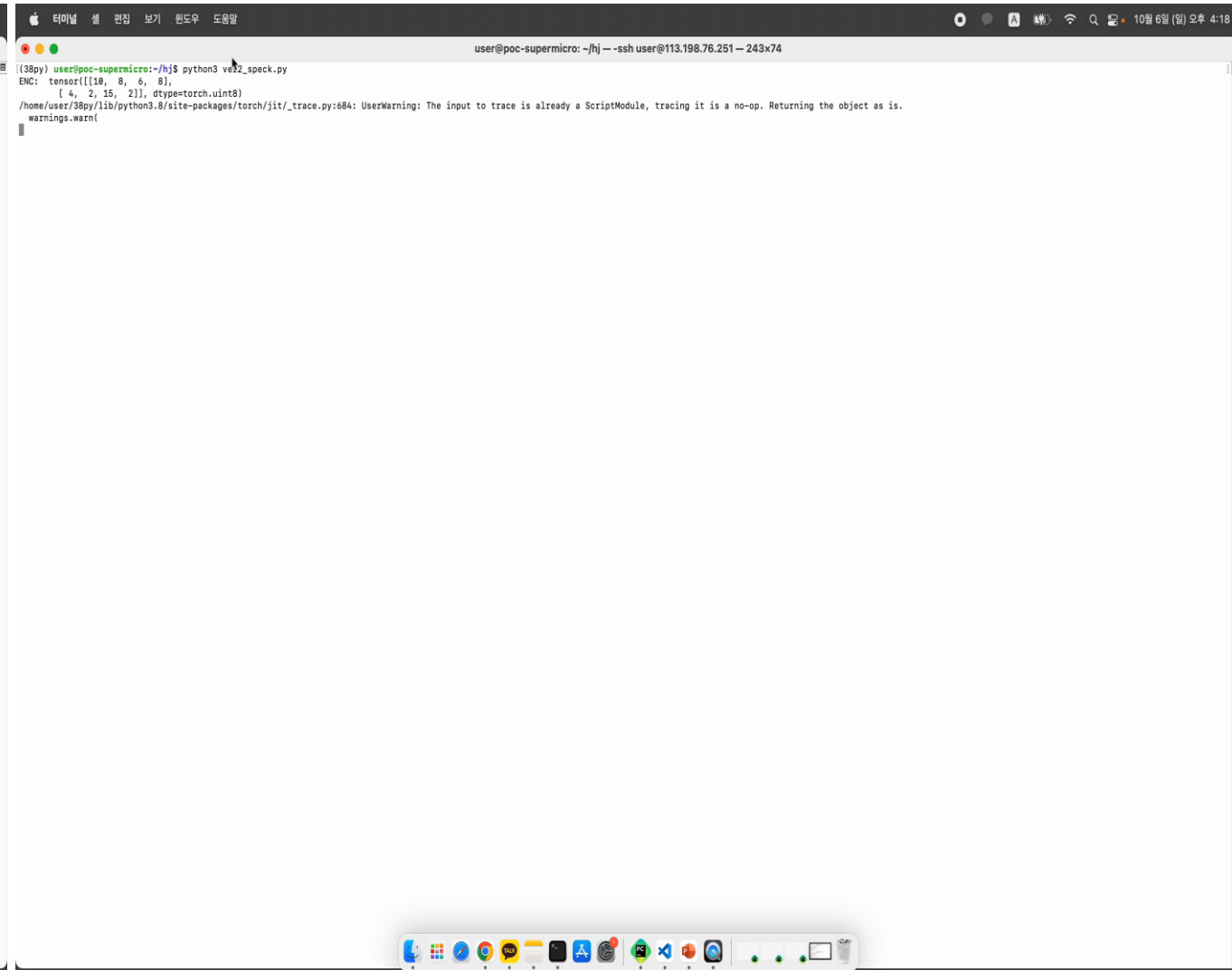
Rebellions - NPU에서 실행 (SPECK)

- 리벨리온 ATOM

SHA 512는 에러 해결 되면 넣을 예정



A terminal window titled "user@poc-supermicro: ~/hj - ssh user@113.198.76.251 - 243x74". The prompt is "(38py) user@poc-supermicro:~/hj\$". The command "python3 ver2_speck.py" has been entered and the cursor is at the end of the line.



A terminal window titled "user@poc-supermicro: ~/hj - ssh user@113.198.76.251 - 243x74". The prompt is "(38py) user@poc-supermicro:~/hj\$". The command "python3 ver2_speck.py" has been executed, resulting in the following output:

```
ENC: tensor([[[[10,  8,  6,  8],  
               [ 4,  2, 15,  2]], dtype=torch.uint8])  
/home/user/38py/11b/python3.8/site-packages/torch/jit/_trace.py:684: UserWarning: The input to trace is already a ScriptModule, tracing it is a no-op. Returning the object as is.  
warnings.warn(  
|
```

Rebellions - NPU에서 실행 (SPECK)

- 리벨리온 ATOM에서 SPECK 실행

CPU 결과

```
(38py) user@poc-supermicro:~/hj$ python3 ver2_speck.py
ENC: tensor([[[[10, 8, 6, 8],
               [ 4, 2, 15, 2]]], dtype=torch.uint8)
/home/user/.38py/lib/python3.8/site-packages/torch/jit/_trace.py:684: UserWarning:
  warnings.warn(
RBLN SDK compiler version: 0.5.9
-- Target NPU: RBLN-CA02
-- Tensor parallel size: 1
+-----+
|Compile(#0), mod_name=default, input_info_index=0|
+-----+
Computation graph generation ██████████ 100% 02:32
Serializing compiled model to speck_npu.rbln ...
Compiled model serialized. Elapsed time: 0:00:00
Load model completed. Elapsed time: 0:00:00
```

컴파일 성공

그러나 NPU 실행 시 테스트벡터 불일치

Compiled Model Description			
I/O Type	Key	Shape	Dtype
Input	p.1	(2, 4)	torch.uint8
Input	ks.1	(22, 4)	torch.uint8
Output	0	(2, 4)	torch.uint8
Compiler version		0.5.9	
Memory		0.0B	

NPU 결과

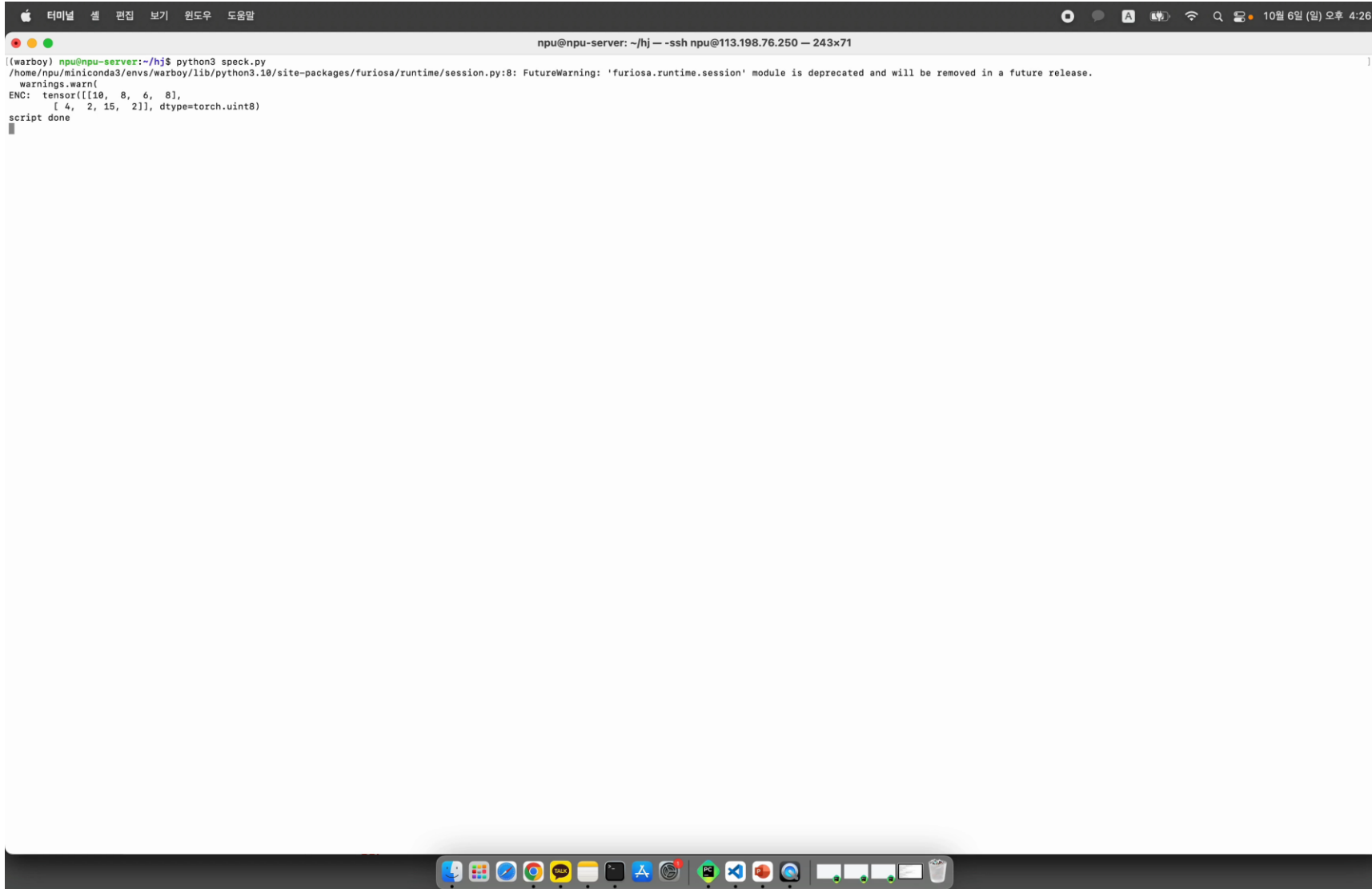
```
tensor([[0, 1, 1, 1],
        [0, 1, 1, 1]], dtype=torch.uint8)
```

컴파일러 내부 과정에서 뭔가 달라지는 듯 (문의 예정)

→ 지난 번에 문의 했을 때는 컴파일러 내부 동작이나 이런 건 보안 상 못 알려준다고 하긴 했습니다..

Furiosa - NPU에서 실행 (SPECK)

- 퓨리오사 Warboy (에러 해결되면 다시 첨부)



The screenshot shows a terminal window titled "npu@npu-server: ~/hj - ssh npu@113.198.76.250 - 243x71". The terminal output is as follows:

```
((warboy) npu@npu-server:~/hj$ python3 speck.py
/home/npu/miniconda3/envs/warboy/lib/python3.10/site-packages/furiosa/runtime/session.py:8: FutureWarning: 'furiosa.runtime.session' module is deprecated and will be removed in a future release.
warnings.warn(
ENC: tensor([[10,  8,  6,  8],
              [ 4,  2, 15, 21]], dtype=torch.uint8)
script done
```

The terminal window is part of a macOS desktop environment, with the dock visible at the bottom showing various application icons.

Furiosa - NPU에서 실행 (SPECK)

- 컴파일 에러 난 상태 (문의 후 수정 예정)
 - SPECK은 리벨리온에서는 되는 데 퓨리오사에서는 안됨
 - Sha512가 speck 보다 훨씬 복잡하긴 해도 설계 원칙이나 기본 연산들은 다르지 않은데 리벨리온 컴파일 에러 발생
 - 퓨리오사는 둘 다 안됨 (컴파일 에러 => 문의 필요할 듯)

퓨리오사 ONNX 변환 후 모델 로드까지는 가능

```
((38py) user@poc-supermicro:~/hj$ python3 ver2_sha512.py
tensor([[[13, 13, 10, 15, 3, 5, 10, 1, 9, 3, 6, 1, 7, 10, 11, 10],
         [12, 12, 4, 1, 7, 3, 4, 9, 10, 14, 2, 0, 4, 1, 3, 1],
         [ 1, 2, 14, 6, 15, 10, 4, 14, 8, 9, 10, 9, 7, 14, 10, 2],
         [ 0, 10, 9, 14, 14, 14, 14, 6, 4, 11, 5, 5, 13, 3, 9, 10],
         [ 2, 1, 9, 2, 9, 9, 2, 10, 2, 7, 4, 15, 12, 1, 10, 8],
         [ 3, 6, 11, 10, 3, 12, 2, 3, 10, 3, 15, 14, 14, 11, 11, 13],
         [ 4, 5, 4, 13, 4, 4, 2, 3, 6, 4, 3, 12, 14, 8, 0, 14],
         [ 2, 10, 9, 10, 12, 9, 4, 15, 10, 5, 4, 12, 10, 4, 9, 15]],
        dtype=torch.uint8)
Traceback (most recent call last):
  File "ver2_sha512.py", line 775, in <module>
    torch.onnx.export(scripted_model, mm, "sha512_enc_op.onnx", opset_version=14)
  File "/home/user/38py/lib/python3.8/site-packages/torch/onnx/utils.py", line 551, in export
    _export(
  File "/home/user/38py/lib/python3.8/site-packages/torch/onnx/utils.py", line 1648, in _export
    graph, params_dict, torch_out = _model_to_graph(
  File "/home/user/38py/lib/python3.8/site-packages/torch/onnx/utils.py", line 1174, in _model_to_graph
    graph = _optimize_graph(
  File "/home/user/38py/lib/python3.8/site-packages/torch/onnx/utils.py", line 714, in _optimize_graph
    graph = _C._jit_pass_onnx(graph, operator_export_type)
  File "/home/user/38py/lib/python3.8/site-packages/torch/onnx/utils.py", line 1997, in _run_symbolic_function
    return symbolic_fn(graph_context, *inputs, **attrs)
  File "/home/user/38py/lib/python3.8/site-packages/torch/onnx/symbolic_opset9.py", line 6952, in prim_constant
    raise errors.SymbolicValueError(
torch.onnx.errors.SymbolicValueError: Unsupported prim::Constant kind: 'ival'. Please send a bug report at https://github.com/pytorch/pytorch/issues.
Inputs:
  Empty
Outputs:
  #0: 32 defined in (%32 : int[][]) = prim::Constant[value=[[6, 10, 0, 9, 14, 6, 6, 7, 15, 3, 11, 12, 12, 9, 0, 8], [11, 11, 6, 7, 10, 14, 8, 5, 8, 4, 12, 10, 10, 7, 3, 11], [3, 12, 6, 14, 15, 3, 7, 2, 15, 14, 9, 4, 15, 8, 2, 11], [10, 5, 11, 3, 14, 6, 12, 1, 15], [1, 15, 8, 3, 13, 9, 10, 11, 15, 11, 4, 1, 11, 13, 6, 11], [5, 11, 14, 0, 12, 13, 1, 9, 1, 3, 7, 14, 2, 1, 7]]] (type 'List[List[int]]') in the TorchScript graph. The containing node has kind 'prim::Constant'.
```

상수 관련 에러 발생하며,
해당 에러는 고친 적 있어서
다시 확인 예정

퓨리오사 런타임 에러

```
(warboy) npu@npu-server:~/hj$ python3 speck.py
/home/npu/miniconda3/envs/warboy/lib/python3.10/site-packages/furiosa/runtime/session.py:8: FutureWarning:
ENC: tensor([[[10, 8, 6, 8],
              [ 4, 2, 15, 2]], dtype=torch.uint8)
script done
export done
load done

2024-10-03T06:56:04.664881Z INFO furiosa_rt_core::driver::event_driven::coord: FuriosaRT (v0.10.3)
2024-10-03T06:56:04.674347Z INFO furiosa_rt_core::driver::event_driven::coord: Found furiosa-compiler
2024-10-03T06:56:04.674403Z INFO furiosa_rt_core::driver::event_driven::coord: Found libhal (type)
2024-10-03T06:56:04.674424Z INFO furiosa_rt_core::driver::event_driven::coord: [Runtime-0] detected
2024-10-03T06:56:04.697647Z INFO furiosa_rt_core::driver::event_driven::coord: - [0] npu:0:0-1 (FuriosaRT)
2024-10-03T06:56:04.698088Z INFO furiosa_rt_core::driver::event_driven::coord: [Runtime-0] start
2024-10-03T06:56:04.699535Z INFO furiosa::runtime: Saving the compilation log into /home/npu/.local/state/furiosa/logs/compiler-20241003065604-f5v2
2024-10-03T06:56:04.704594Z INFO furiosa_rt_core::driver::event_driven::coord: [Runtime-0] created
2024-10-03T06:56:04.727975Z INFO furiosa_rt_core::driver::event_driven::coord: [Sess-35e66b48] compilation failed
[1/6] Q Compiling from onnx to dfp
ERROR: the static shape of tensor /round_function/round_7/Reshape_1_output_0 has to be provided
2024-10-03T06:56:05.151372Z INFO furiosa_rt_core::driver::event_driven::coord: compilation failed
=====
Compilation Failure Report
=====
- furiosa-runtime version: 0.10.3 (rev: 394c19392 built at 2023-11-22T08:53:04Z)
- furiosa-compiler version: 0.10.1 (rev: 8b00177dc built at 2023-11-23T01:50:08Z)
- libhal version: 0.12.0 (rev: 56530c0 built at 2023-11-16T12:34:03Z)

The compiler log can be found at /home/npu/.local/state/furiosa/logs/compiler-20241003065604-f5v2

If you have a problem, please follow the bug reporting guide at
https://furiosa-ai.github.io/docs/latest/en/customer-support/bugs.html
=====
2024-10-03T06:56:05.166765Z INFO furiosa_rt_core::npu::raw: NPU (npu:0:0-1) has been closed
2024-10-03T06:56:05.174986Z INFO furiosa_rt_core::driver::event_driven::coord: [Runtime-0] stopped
Traceback (most recent call last):
  File "/home/npu/hj/speck.py", line 281, in <module>
    speck_enc = session.create(model_path)
  File "<string>", line 70, in wrapped
runtime.FuriosaRuntimeError: runtime error: Compilation error: Other error
```


NPU 상에서 실행 시의 실질적인 어려움

NPU 구현 시 제한 사항

- 퓨리오사 Warboy의 경우, 텐서 연산 과정에서 크기를 명확하게 정의하라는 에러가 발생
 - 리벨리온 ATOM의 경우, 길이나 형태 관련 내장 함수 사용 시 컴파일러 내부 기능과 충돌하여 컴파일 불가
- } 두 NPU에서
상반된 에러 발생
- 비트 연산자는 ONNX로 변경 시 모두 미지원
 - 산술 연산으로 대체 필요
 - 기본 Python 연산에서는 쉬운 부분들이 제한적임
 - Torch.jit.script나 NPU에서는 모든 인자가 명시적으로 Tensor 타입만 가능
 - Int 변수 사용 불가 (Int 값을 Tensor로 변환)
 - 하지만, 리스트 인덱싱 등에서는 정수형 스칼라 값이 필요 → 이를 위한 함수를 사용하면 에러 발생
 - 텐서가 아니면 에러가 발생
 - 그래서 텐서를 정의하고 텐서 연산을 사용한 후, 결과 값은 텐서라고 명시하지 않았더니 해결
 - 하지만 해당 결과 값을 다시 연산하면 에러 없이 연산 가능 (즉, 결과 값도 텐서라는 의미)
→ 정확한 이유는 알 수 없으나 해결!
 - 개발 용이성을 위해 개인 pc 사용한 경우에는 torch 라이브러리의 모델 export가 가능했으나, 동일 코드도 NPU에선 불가능
 - NPU에서는 동일 라이브러리의 동일 함수도 안되는 경우 발생
 - NPU에서는 ONNX로 바로 변환되지 않으며, Torch.jit.Script로 변환 필수

NPU의 미구현 에러

- `NotImplementedError: The following operators are not implemented: ['aten::item', 'prim::dtype']`
 - `item()`, `dtype` 연산들은 NPU에서 구현되지 않았으므로 사용 권장하지 않으며 대체 연산 필요
 - 본 구현에서는 `rotation`시에 텐서로부터 스칼라 값을 뽑아서 `rotation` 인덱스로 사용할 때 필요
 - `Item()`은 텐서를 Python 기본 자료형으로 바꾸므로 NPU에서 사용 불가 → `dtype` 에러 야기
 - 사용 불가하여 인덱스 텐서 배열을 인덱스로 사용하도록 `텐서[인덱스 배열[인덱스]]`와 같이 사용
 - 해당 방식이 텐서 형태를 계속 유지할 수 있으므로 권장
- `NotImplementedError: The following operators are not implemented: ['prim::Print']`
 - `Print` 사용 시 컴파일 불가
 - NPU에서 중간 결과 값 디버깅이 어려움
- (현재까지) 블록 암호 구현에서 발견한 미구현 에러들을 피해가려면?
 - Python 기본 연산 사용하지 않고 텐서 연산을 유지
 - 타입/사이즈/프린트 함수 사용 지양

하드웨어 별 차이

- 동일 구현, 동일 onnx 파일임에도 불구하고 하드웨어 별로 다른 컴파일 에러 발생
 - 각 하드웨어에 맞게 코드를 변경하였으나 그에 따른 다른 에러가 발생
 - 현재 원인 파악 불가 (기업 기밀이라고 함..)
 - 하지만 다시 문의할 예정 ...
- 다른 요소가 뭐가 있을지 여쭙보고 싶습니다

논의 분석 뭐 그런 거

일단 결과 정리

- 비트 연산들은 NPU에 올리기 위한 ONNX 모델에서 미지원
 - 수학적 연산으로 대체 (딥러닝 추론 칩이므로 산술 연산 지원 및 가속 가능)
- 암호 연산은 분기/반복이 필요하고, 딥러닝처럼 대용량 데이터가 아니어서 NPU만 사용하기엔 불리함
 - 분기문 같은 경우는 케이스를 쪼개고 어느정도 하드코딩을 하는 게 나을 것으로 생각됨
- 현재의 NPU에서는 모든 연산을 수행하는 것은 비효율일 수 있음
- NPU별로 컴파일러를 제공하므로 구현 차이점이 존재
- 더 큰 자료형 필요
 - Pytorch의 tensor 연산이 32, 64 비트 데이터 + 대용량 데이터에 최적임
 - 지금처럼 8비트 자료형에 단일 블록 연산은 그리 효율적이지 않음..
- NPU의 핵심 자료형인 Float는 활용하기 어려움 (일단 지금까지의 판단으로는 그럼)
 - Int() 이런 식의 형변환이 어려움
- 추가...

NPU 상에서의 고속 구현의 한계

- 아직 시간을 재보진 못했음 (NPU 컴파일 시 발생하는 에러 해결이 너무 많았음..) → 측정 되면 추가
- NPU 적용 시 기대되는 효과로 **가속화**가 있었음
 - 그러나, 자료형 및 연산자 등의 제한으로 인해 구현에 제한이 많음
 - 특히, 비트 연산을 산술 연산으로 대체 + 8/4 조합 Multi-precision 사용으로 인해 효율성 저하
 - Pytorch의 텐서 연산은 16, 32 비트에 처리에 최적화 되어 있다고 하지만, NPU 사용 시 8비트 사용하게 됨
- 제한적 환경에서 최대한 효율적 구현을 해보았으나, 구조적으로 비효율적

고속 구현에는 아직 제한 사항이 많음

암호 구현을 위한 개선 요구 사항 (추가예정)

- 더 많은 자료형 및 연산자를 지원
 - INT 8만 사용하는 것은 딥러닝 추론에는 효율적이지만 암호 구현에서는 비효율적
 - 비트 연산 제공 필요
- 두 NPU 모두 더 넓은 범위의 SDK 지원이 필요
 - 너무 딥러닝 작업에만 적합한 기능들만 지원 중 → 좀 더 범용적인 개발이 쉽게 가능했으면..
 - 사이즈나 타입 연산을 좀 더 자유롭게 하고 싶음...
- 현재까지 메모리 관련 개선 사항은 없음
 - 두 NPU 모두 온 칩 메모리가 32MB
 - 단일 평문에 대한 구현만 진행하였으므로 현재까지 메모리 문제 없음

GPU 최적화 기술과 NPU 최적화 기술 비교

GPU 최적화 기법

데이터 블록 크기 조절

처리할 데이터를 효율적으로
블록으로 나누어 처리

NPU 구현은 제한된 자료형만

사용 가능하여 적용 어려움

분기/반복문은 GPU 친화적 X

분기문 줄이기
Un-loop하여 병렬 구현 권장

NPU 구현도 동일한 최적화 포인트

NPU 최적화 기법

삼항 연산자 뭐 이런거 있었던거 같은데 아무튼 추가할 거 있으면 추가

NPU 친화적인 암호는 무엇인가? 수정예정

- 비트 연산을 최소화하고 산술 연산을 극대화 하는 암호가 적합
- 따라서 격자 기반 암호 등이 적합
 - 산술 연산을 주로 사용함
 - 다항식 곱셈, 벡터 연산 등
 - 대규모 병렬 연산을 요구하는 구조
 - 큰 수나 행렬 연산을 사용

결론

결론.. 추가 예정

- NPU와 블록 암호 구현의 갭
- 그렇다면 NPU에 잘 어울리는 암호 구현이 있는가?
- 있다면 이유랑 뭐 그런 거..

감사합니다.