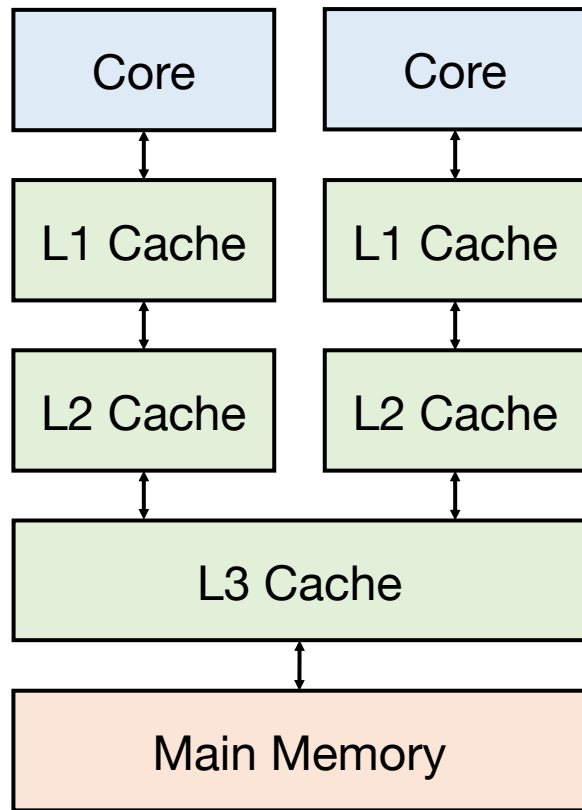


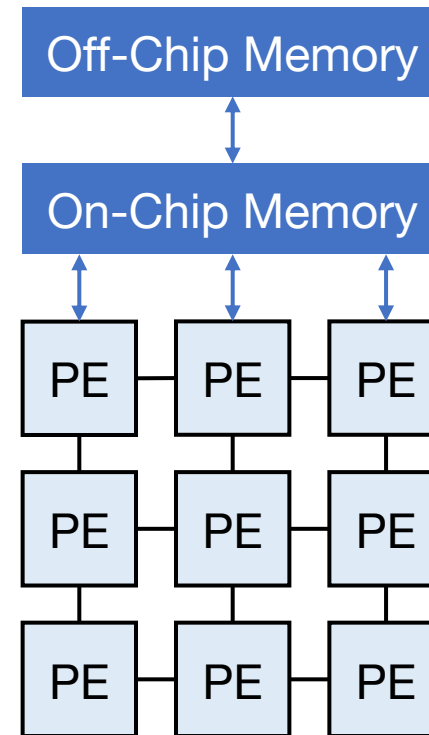
# 高位合成における分離型データ オーケストレーションの自動合成

薄井真之・高前田伸也（東大）

# 背景：アクセラレータ



CPU



※PE = processing element ≡ 演算器

Accelerator

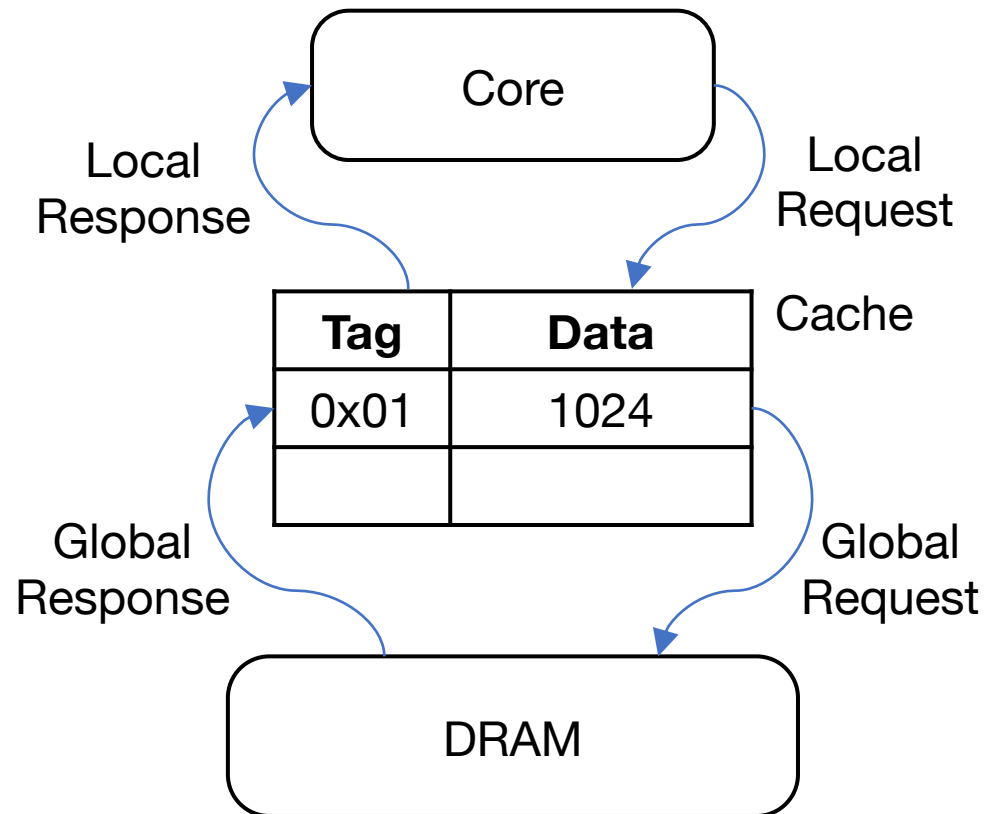
# 背景：高位合成

- アクセラレータの設計には多大な労力を要する
- 高位合成は抽象度を上げることによって生産性を向上させる

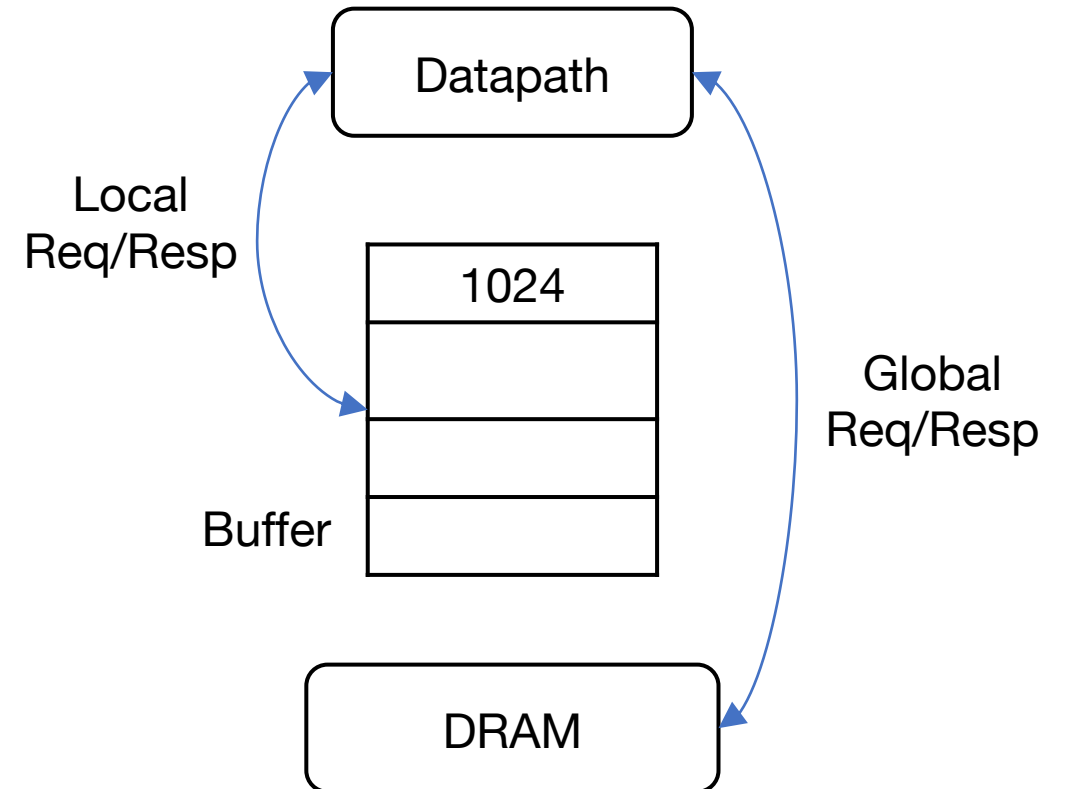


# 背景：アクセラレータのメモリシステム1

## *Implicit* Data Orchestration in CPU

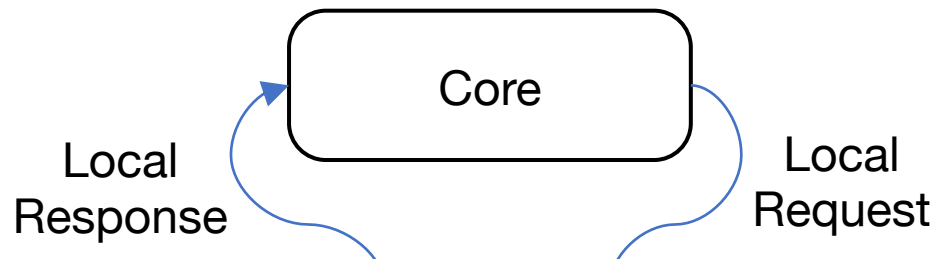


## *Explicit* Data Orchestration in Accelerator

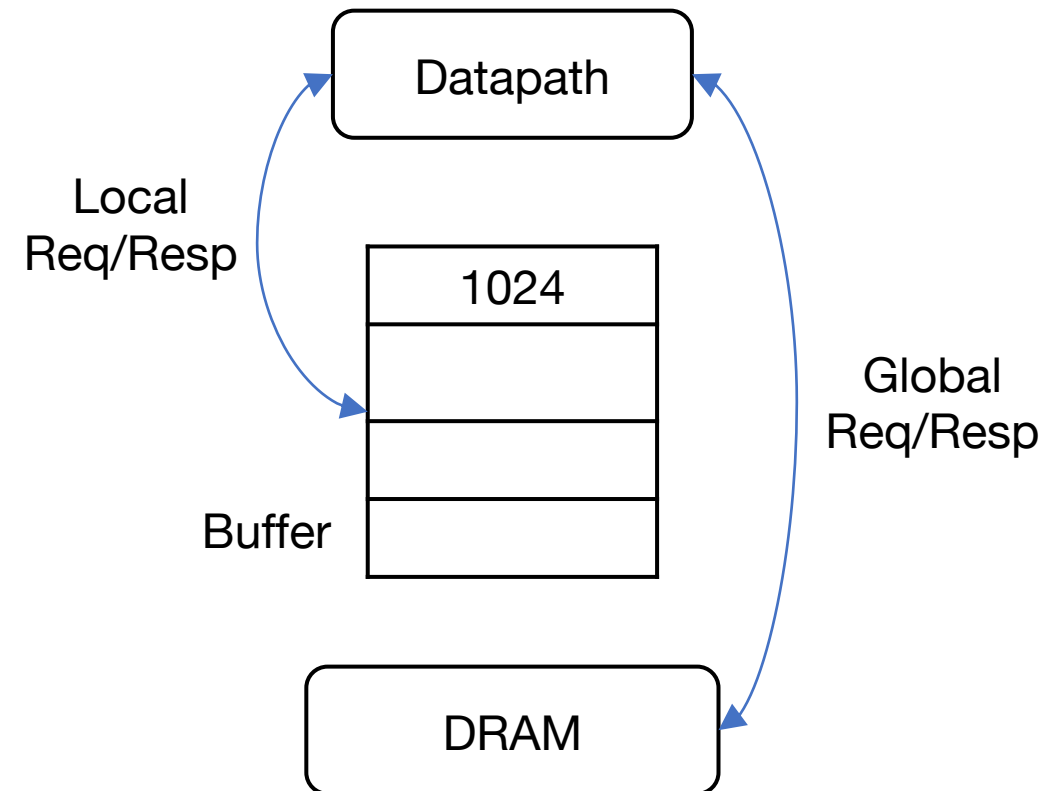


# 背景：アクセラレータのメモリシステム1

## *Implicit* Data Orchestration in CPU

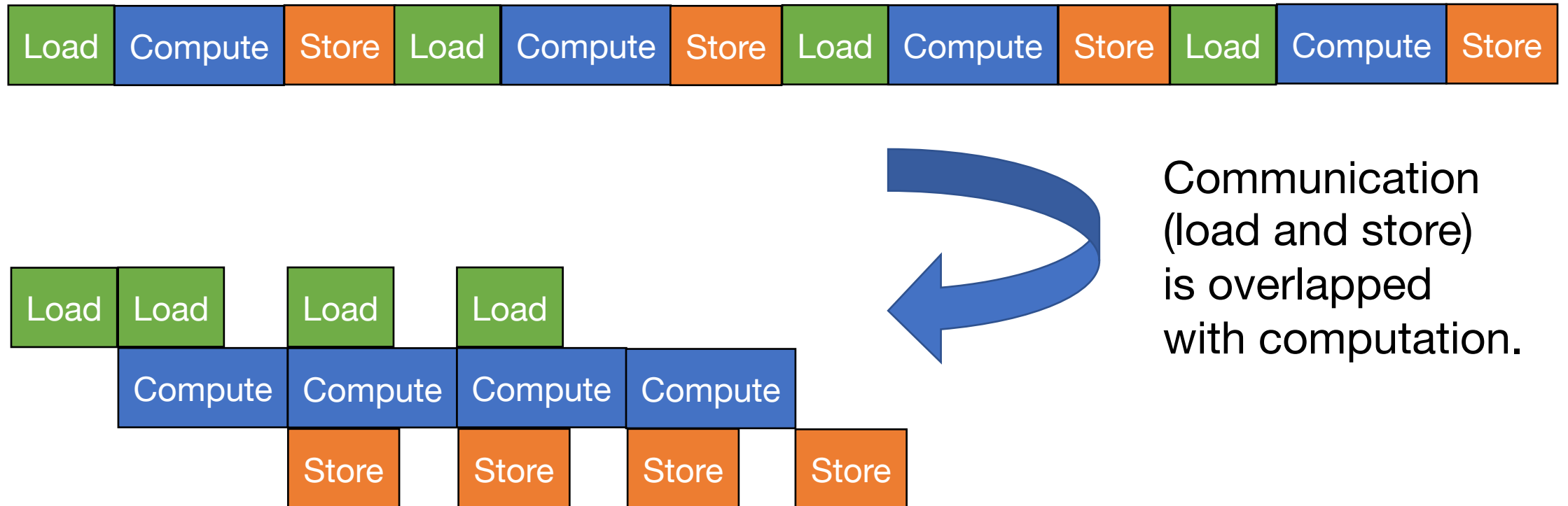


## *Explicit* Data Orchestration in Accelerator



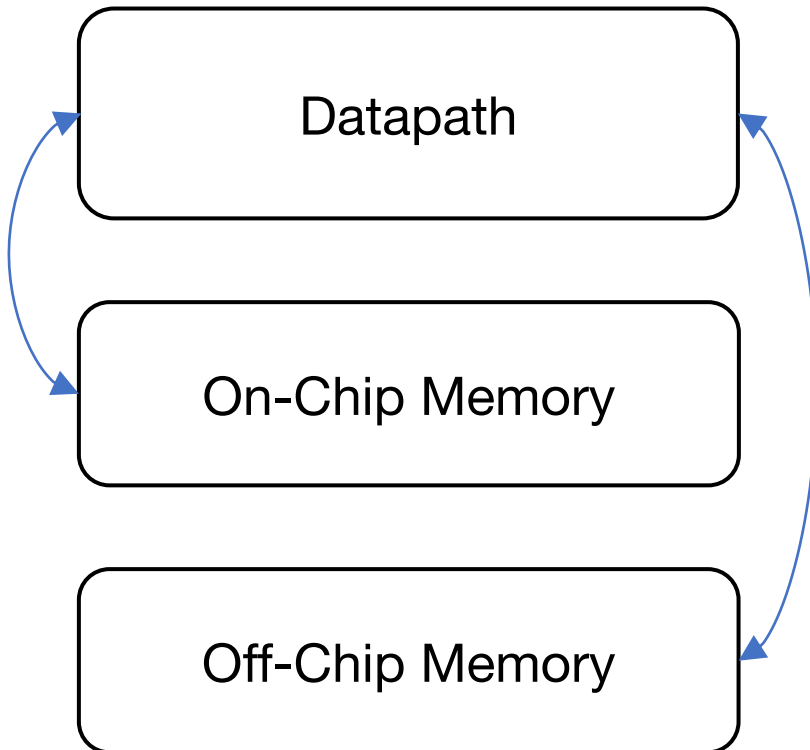
Explicit data orchestrationの利点：  
データ移動を明示的に制御する  
ことで、ドメイン知識を活用し  
性能を向上させることができる

# 背景：計算と通信のオーバーラップ

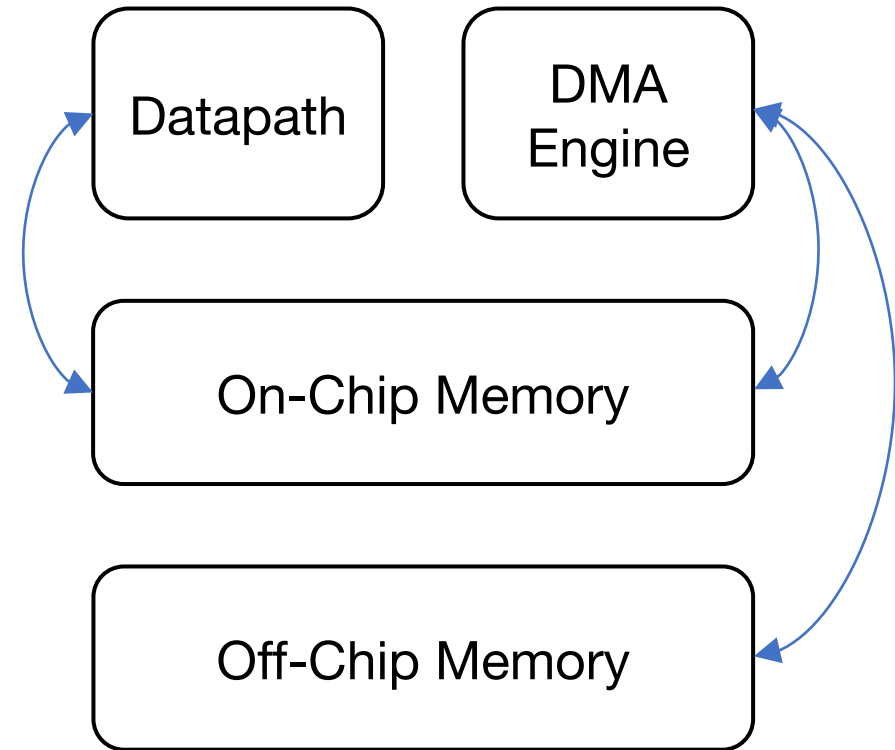


# 背景：アクセラレータのメモリシステム2

Explicit *Coupled*  
Data Orchestration



Explicit *Decoupled*  
Data Orchestration  
(EDDO)



# 設計の課題

- EDDOは計算と通信をオーバーラップさせて性能を向上させるが、計算をするモジュールと通信をするモジュールを分離しなければならず、設計が複雑になる
- 特に、分離したモジュール間の同期が問題になる



# 提案手法

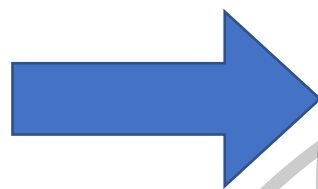
本研究では、高位合成において、データオーケストレーション機構を自動的に分離することにより、設計を容易にする

# 提案手法 (図解)

```

a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
    
```

Our method



Computation

```

for i in range(matrix_size):
    ram_a.wait_not_empty()
    for j in range(matrix_size):
        ram_b.wait_not_empty()
        inner_product(ram_a, ram_b, ram_c,
                      j, matrix_size)
        ram_b.pop()
        ram_c.push()
        ram_a.pop()
        ram_c.wait_not_full()
    
```

Data Movement for Buffer A

```

a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    ram_a.push()
    a_addr += matrix_size * (datawidth // 8)
    ram_a.wait_not_full()
    
```

Data Movement  
for Buffer C

```

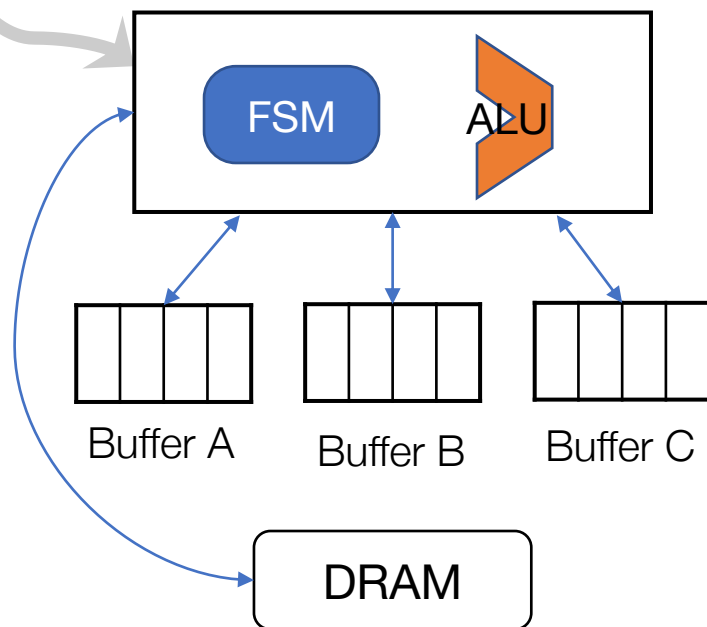
c_addr = c_offset
for i in range(matrix_size):
    ram_c.wait_not_empty()
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    c_addr += matrix_size * (datawidth // 8)
    ram_c.pop()
    
```

```

for i in range(matrix_size):
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        ram_b.push()
        b_addr += matrix_size * (datawidth // 8)
        ram_b.wait_not_full()
    
```

Data Movement for Buffer B

Computation + Communication



Computation



Buffer A

Buffer B

Buffer C

Communication

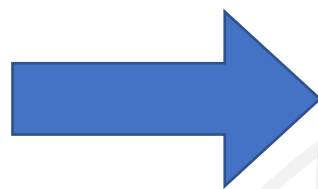
DRAM

# 提案手法 (図解)

```

a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
    
```

Our method



Computation

```

for i in range(matrix_size):
    ram_a.wait_not_empty()
    for j in range(matrix_size):
        ram_b.wait_not_empty()
        inner_product(ram_a, ram_b, ram_c,
                      j, matrix_size)

    ram_b.pop()
    ram_c.push()
    ram_a.pop()
    ram_c.wait_not_full()
    
```

Data Movement for Buffer A

```

a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    ram_a.push()
    a_addr += matrix_size * (datawidth // 8)
    ram_a.wait_not_full()
    
```

Data Movement  
for Buffer C

```

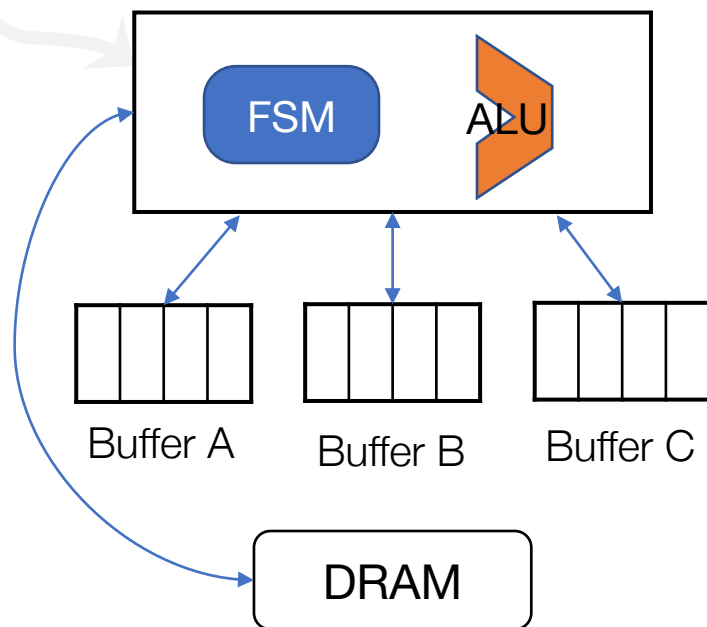
c_addr = c_offset
for i in range(matrix_size):
    ram_c.wait_not_empty()
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    c_addr += matrix_size * (datawidth // 8)
    ram_c.pop()
    
```

```

for i in range(matrix_size):
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        ram_b.push()
        b_addr += matrix_size * (datawidth // 8)
        ram_b.wait_not_full()
    
```

Data Movement for Buffer B

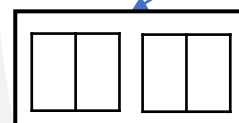
Computation + Communication



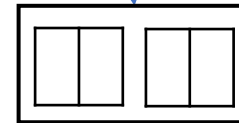
Computation



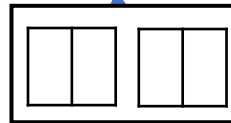
Buffer A



Buffer B



Buffer C



Communication

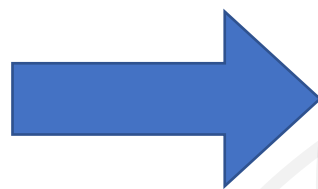


# 提案手法 (図解)

```

a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
    
```

Our method



Computation

```

for i in range(matrix_size):
    ram_a.wait_not_empty()
    for j in range(matrix_size):
        ram_b.wait_not_empty()
        inner_product(ram_a, ram_b, ram_c,
                      j, matrix_size)

    ram_b.pop()
    ram_c.push()
    ram_a.pop()
    ram_c.wait_not_full()
    
```

Data Movement for Buffer A

```

a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    ram_a.push()
    a_addr += matrix_size * (datawidth // 8)
    ram_a.wait_not_full()
    
```

Data Movement  
for Buffer C

```

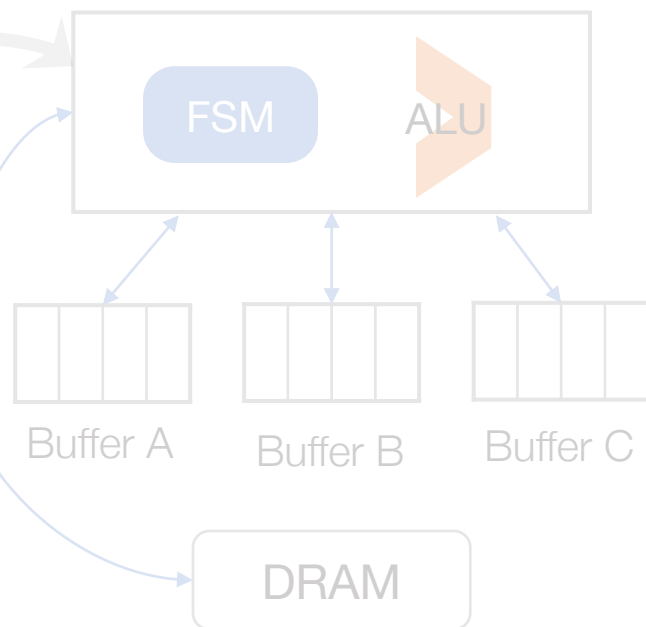
c_addr = c_offset
for i in range(matrix_size):
    ram_c.wait_not_empty()
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    c_addr += matrix_size * (datawidth // 8)
    ram_c.pop()
    
```

```

for i in range(matrix_size):
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        ram_b.push()
        b_addr += matrix_size * (datawidth // 8)
        ram_b.wait_not_full()
    
```

Data Movement for Buffer B

Computation + Communication



Computation



Buffer A



Buffer B



Buffer C



DRAM



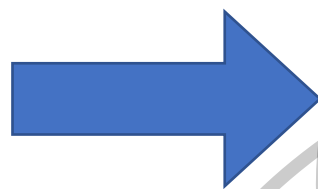
Communication

# 提案手法 (図解)

```

a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
    
```

Our method



Computation

```

for i in range(matrix_size):
    ram_a.wait_not_empty()
    for j in range(matrix_size):
        ram_b.wait_not_empty()
        inner_product(ram_a, ram_b, ram_c,
                      j, matrix_size)
        ram_b.pop()
        ram_c.push()
        ram_a.pop()
        ram_c.wait_not_full()
    
```

Data Movement for Buffer A

```

a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    ram_a.push()
    a_addr += matrix_size * (datawidth // 8)
    ram_a.wait_not_full()
    
```

Data Movement  
for Buffer C

```

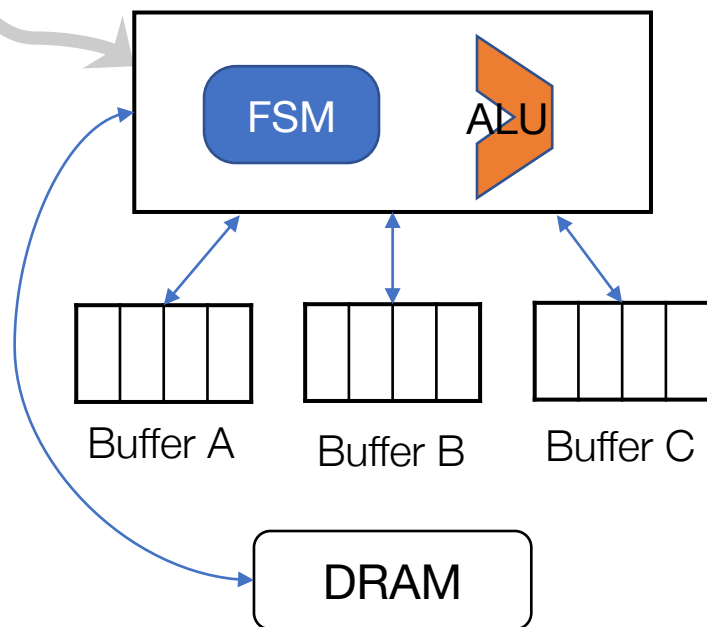
c_addr = c_offset
for i in range(matrix_size):
    ram_c.wait_not_empty()
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    c_addr += matrix_size * (datawidth // 8)
    ram_c.pop()
    
```

```

for i in range(matrix_size):
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        ram_b.push()
        b_addr += matrix_size * (datawidth // 8)
        ram_b.wait_not_full()
    
```

Data Movement for Buffer B

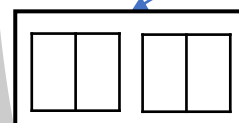
Computation + Communication



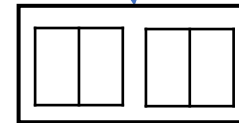
Computation



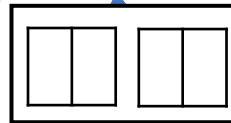
Buffer A



Buffer B



Buffer C



DRAM

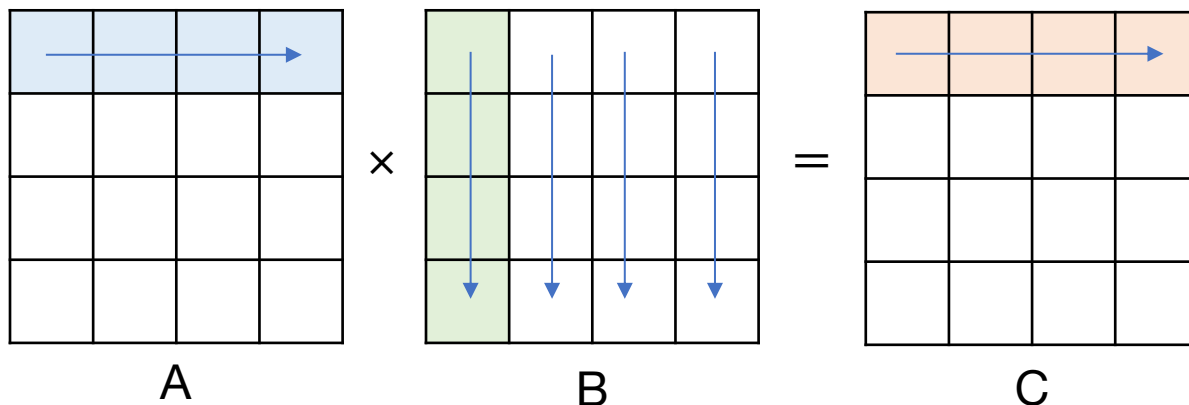


Communication

# コード例：行列積

```
a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size) ← RAM A：行列A用のバッファ
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size) ← RAM B：行列B用のバッファ
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size) ← RAM C：行列C用のバッファ
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
```

行列積： $A = BC$

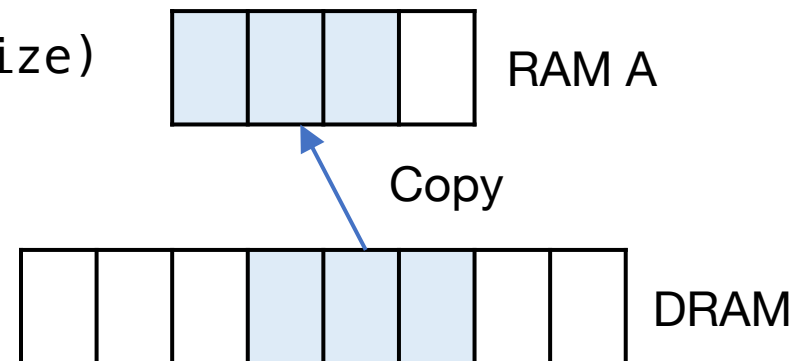


# コード例：分離過程

```
a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
```

RAM AのDMA転送に着目

※DMA転送ではメモリの領域をコピーする



# コード例：分離過程

```
a_addr = a_offset          RAM AのDMAに必要な部分を抽出
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
```



# コード例：分離過程

抽出結果：

```
a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
```

これでは不十分

さらに同期が必要

# 同期のためのAPI



`write()` : データを書き込む

`wait_not_empty()` : バッファが埋まるまで待つ



`push()` : バッファを入れ替える



`write()` : データを書き込む

`read()` : データを読み出す



`push()` : バッファを入れ替える



`wait_not_full()` : バッファが空くまで待つ

`read()` : データを読み出す

# 同期のためのAPI



`wait_not_full()` : バッファが空くまで待つ

`read()` : データを読み出す



`pop()` : バッファを入れ替える



`write()` : データを書き込む

`read()` : データを読み出す

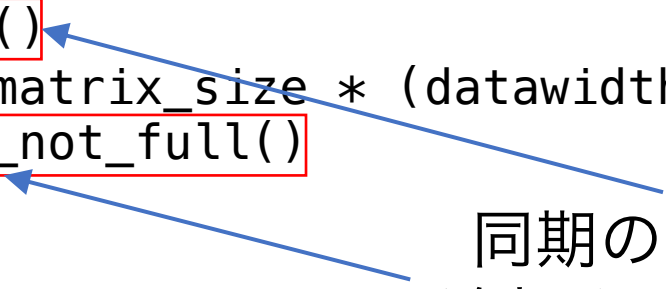
double buffering自体はよく用いられる手法だが、それをデータオーケストレーション機構の自動分離に適用するため、適切な構造と操作を提案している

# コード例：分離過程

最終的に分離されたコード：

```
a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    ram_a.push()
    a_addr += matrix_size * (datawidth // 8)
    ram_a.wait_not_full()
```

同期のために  
追加された部分



# コード例

## Data Movement for Buffer A

```
a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    ram_a.push()
    a_addr += matrix_size * (datawidth // 8)
    ram_a.wait_not_full()
```

## Data Movement for Buffer C

```
c_addr = c_offset
for i in range(matrix_size):
    ram_c.wait_not_empty()
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    c_addr += matrix_size * (datawidth // 8)
    ram_c.pop()
```

※RAM Aと同様にRAM B, C  
についても分離する

## Computation

```
for i in range(matrix_size):
    ram_a.wait_not_empty()
    for j in range(matrix_size):
        ram_b.wait_not_empty()
        inner_product(
            ram_a, ram_b, ram_c,
            j, matrix_size)
    ram_b.pop()
    ram_c.push()
    ram_a.pop()
    ram_c.wait_not_full()
```

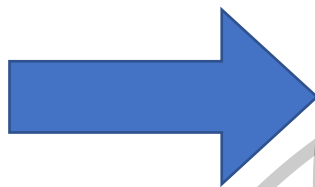
## Data Movement for Buffer B

```
for i in range(matrix_size):
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        ram_b.push()
        b_addr += matrix_size * (datawidth // 8)
    ram_b.wait_not_full()
```

# コード例：最終結果

```
a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
```

Our method



Computation

```
for i in range(matrix_size):
    ram_a.wait_not_empty()
    for j in range(matrix_size):
        ram_b.wait_not_empty()
        inner_product(ram_a, ram_b, ram_c,
                      j, matrix_size)
        ram_b.pop()
        ram_c.push()
        ram_a.pop()
        ram_c.wait_not_full()
```

Data Movement for Buffer A

```
a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    ram_a.push()
    a_addr += matrix_size * (datawidth // 8)
    ram_a.wait_not_full()
```

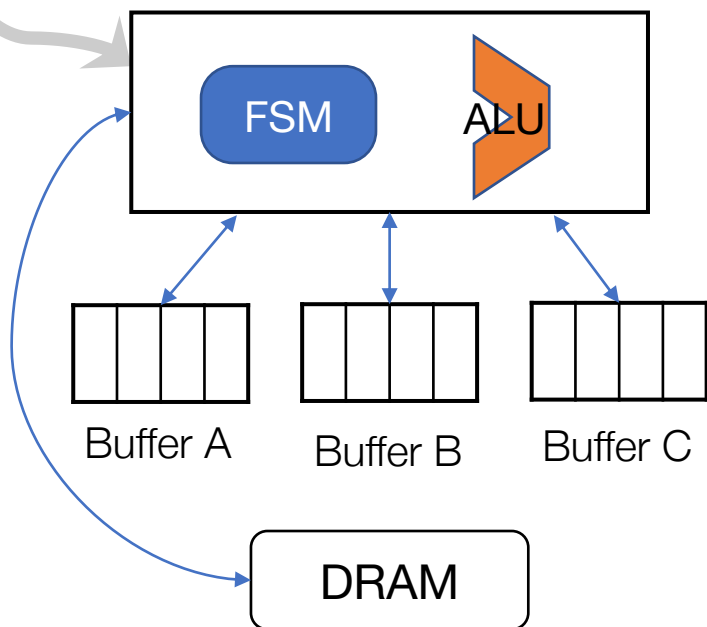
Data Movement  
for Buffer C

```
c_addr = c_offset
for i in range(matrix_size):
    ram_c.wait_not_empty()
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    c_addr += matrix_size * (datawidth // 8)
    ram_c.pop()
```

```
for i in range(matrix_size):
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        ram_b.push()
        b_addr += matrix_size * (datawidth // 8)
        ram_b.wait_not_full()
```

Data Movement for Buffer B

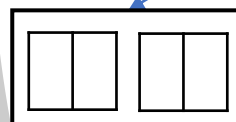
Computation + Communication



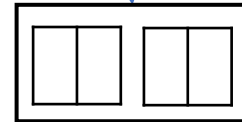
Computation



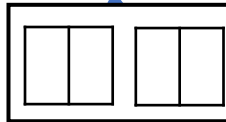
Buffer A



Buffer B



Buffer C



DRAM



Communication

# 説明と実装の差異

- 先ほどの説明ではコードを分離してからAPIを挿入していたが、実際の実装ではAPIを挿入してからコードを分離している
- 分離後のコードではモジュール間の相互作用の情報が失われていて、自動的にAPIを挿入するには情報が足りないことが理由である

先ほどの説明

コード分離



API挿入

実際の実装

API挿入



コード分離

# 実際の流れ

```
a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
```

## APIの自動挿入

```
a_addr = a_offset
c_addr = c_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    b_addr = b_offset
    ram_a.push()
    ram_a.wait_not_empty()
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        ram_b.push()
        ram_b.wait_not_empty()
        inner_product(ram_a, ram_b, ram_c, j, matrix_size)
        b_addr += matrix_size * (datawidth // 8)
        ram_b.pop()
        ram_b.wait_not_full()
    ram_c.push()
    ram_c.wait_not_empty()
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    a_addr += matrix_size * (datawidth // 8)
    c_addr += matrix_size * (datawidth // 8)
    ram_a.pop()
    ram_a.wait_not_full()
    ram_c.pop()
    ram_c.wait_not_full()
```

```
a_addr = a_offset
for i in range(matrix_size):
    ram_a.dma_read(axi_a, 0, a_addr, matrix_size)
    ram_a.push()
    a_addr += matrix_size * (datawidth // 8)
    ram_a.wait_not_full()
```

```
c_addr = c_offset
for i in range(matrix_size):
    ram_c.wait_not_empty()
    ram_c.dma_write(axi_c, 0, c_addr, matrix_size)
    c_addr += matrix_size * (datawidth // 8)
    ram_c.pop()
```

```
for i in range(matrix_size):
    b_addr = b_offset
    for j in range(matrix_size):
        ram_b.dma_read(axi_b, 0, b_addr, matrix_size)
        ram_b.push()
        b_addr += matrix_size * (datawidth // 8)
        ram_b.wait_not_full()
```

## データオーケストレーションの分離

```
for i in range(matrix_size):
    ram_a.wait_not_empty()
    for j in range(matrix_size):
        ram_b.wait_not_empty()
        inner_product(
            ram_a, ram_b, ram_c,
            j, matrix_size)
        ram_b.pop()
    ram_c.push()
    ram_a.pop()
    ram_c.wait_not_full()
```



# APIの自動挿入

Insert push() and  
wait\_not\_empty()

Producer Part

Consumer Part

Insert pop() and  
wait\_not\_full()

- コードをproducer partとconsumer partに分け、その境界に適切なAPIを挿入する
- producer→consumerではpush()とwait\_not\_empty()を、consumer→producerではpop()とwait\_not\_full()を挿入する
- ループ（for文またはwhile文）は端と端が繋がっていると考える

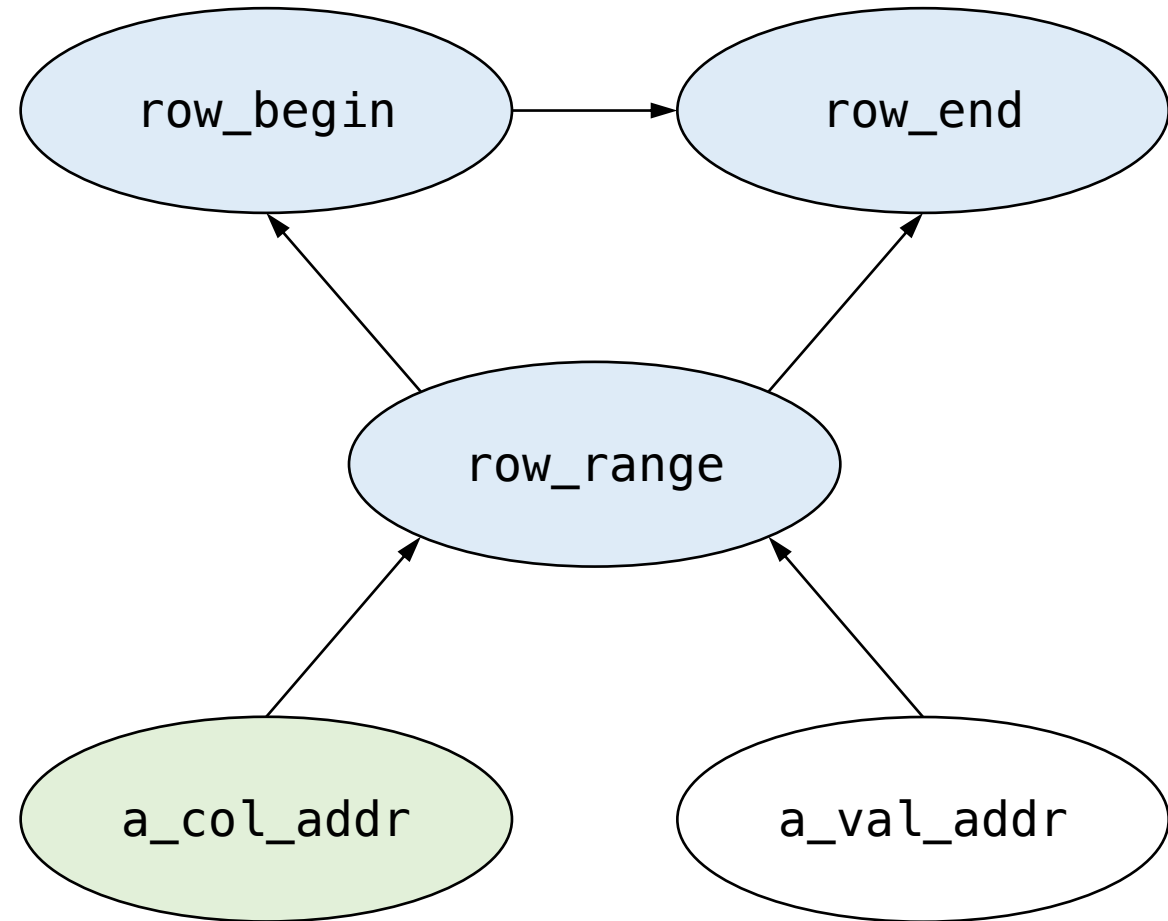
```
1 acc = 0
2 for i in range(block_num):
3     ram.dma_read(
4         axi, 0, i * block_size,
5         block_size)
6     for j in range(block_size):
7         acc += ram.read(j)
```



```
1 acc = 0
2 for i in range(block_num):
3     ram.dma_read(
4         axi, 0, i * block_size,
5         block_size)
6     ram.push()
7     ram.wait_not_empty()
8     for j in range(block_size):
9         acc += ram.read(j)
10    ram.pop()
11    ram.wait_not_full()
```

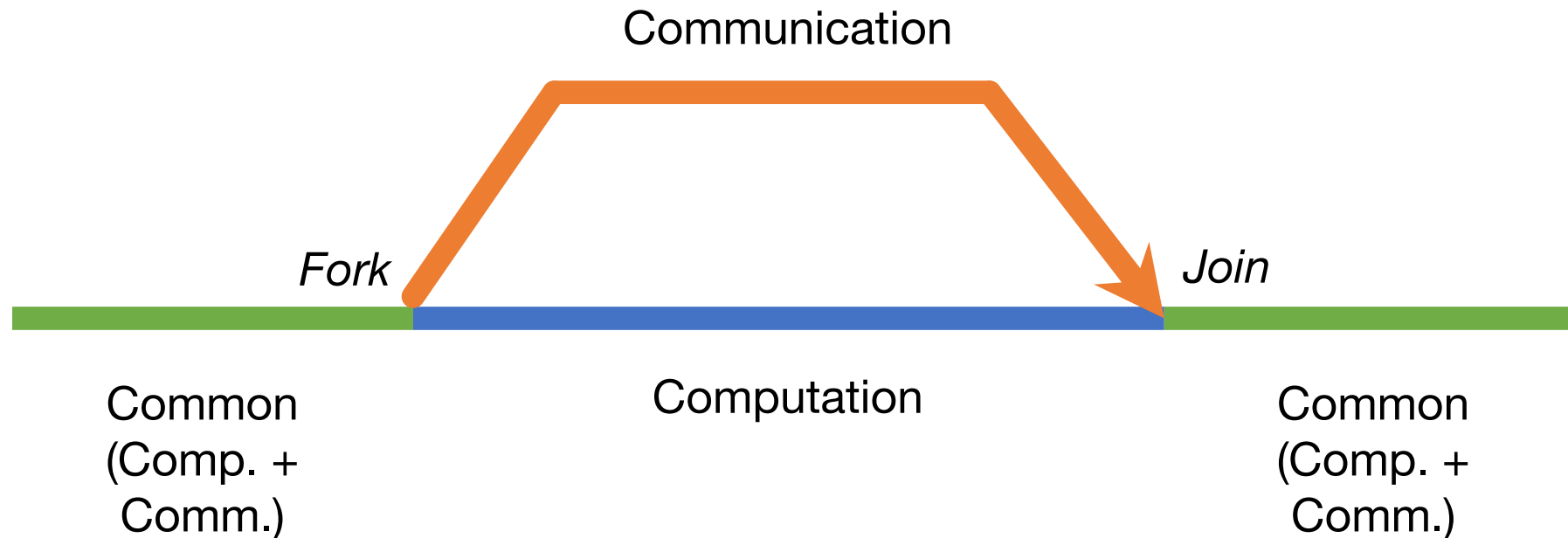
# データオーケストレーションの分離

```
row_begin = ram_a_row.read(0)
a_col_addr = a_col_offset
a_val_addr = a_val_offset
for i in range(1, a_height + 1):
    row_end = ram_a_row.read(i)
    row_range = row_end - row_begin
    row_begin = row_end
    if row_range > 0:
        ram_a_col.dma_read(
            axi_a_col, 0, a_col_addr, row_range)
        ram_a_val.dma_read(
            axi_a_val, 0, a_val_addr, row_range)
        a_col_addr += row_range << log_word
        a_val_addr += row_range << log_word
```



# fork/joinによる部分的な分離

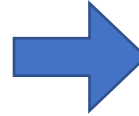
- 依存関係が原因で完全には分離できない場合がある
- 典型的にはsparse workloadの間接参照
- これに対し、途中でforkおよびjoinすることで、部分的に分離する



# コード例2

## APIの自動挿入

```
def main():
    acc = 0
    for i in range(size):
        a_addr = a_offset + i*size*word_size
        ram_a.dma_read(axi_a, 0, a_addr, size)
        for j in range(size):
            b_addr = (b_offset +
                      ram_a.read(j)*size*word_size)
            ram_b.dma_read(axi_b, 0, b_addr, size)
            for k in range(size):
                acc += ram_a.read(k)*ram_b.read(k)
    return acc
```



```
def main():
    acc = 0
    for i in range(size):
        a_addr = a_offset + i*size*word_size
        ram_a.dma_read(axi_a, 0, a_addr, size)
        ram_a.push()
        ram_a.wait_not_empty()
        for j in range(size):
            b_addr = (b_offset +
                      ram_a.read(j)*size*word_size)
            ram_b.dma_read(axi_b, 0, b_addr, size)
            ram_b.push()
            ram_b.wait_not_empty()
            for k in range(size):
                acc += ram_a.read(k)*ram_b.read(k)
            ram_b.pop()
            ram_b.wait_not_full()
        ram_a.pop()
        ram_a.wait_not_full()
    return acc
```

# コード例2

```
def main():  
    acc = 0  
    thd_a = Thread(target=comm_a)  
    thd_a.start()  
    for i in range(size):  
        ram_a.wait_not_empty()  
        thd_b = Thread(target=comm_b)  
        thd_b.start()  
        for j in range(size):  
            ram_b.wait_not_empty()  
            for k in range(size):  
                acc += ram_a.read(k)*ram_b.read(k)  
            ram_b.pop()  
        thd_b.join()  
        ram_a.pop()  
    thd_a.join()  
    return acc
```

RAM A forks

RAM B forks

RAM B joins

RAM A joins

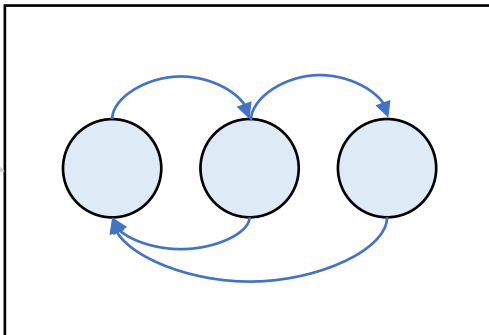
```
def comm_a():  
    for i in range(size):  
        a_addr = a_offset + i*size*word_size  
        ram_a.dma_read(axi_a, 0, a_addr, size)  
        ram_a.push()  
        ram_a.wait_not_full()
```

```
def comm_b():  
    for j in range(size):  
        b_addr = (b_offset +  
                  ram_a.read(j)*size*word_size)  
        ram_b.dma_read(axi_b, 0, b_addr, size)  
        ram_b.push()  
        ram_b.wait_not_full()
```

# コード例2

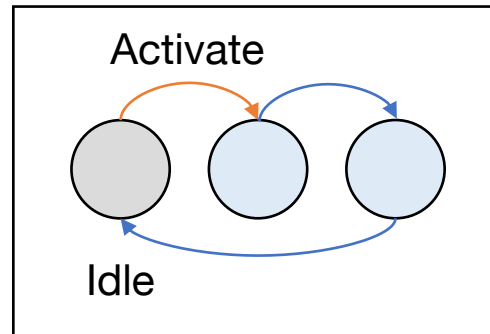
```
def main():  
    acc = 0  
    thd_a = Thread(target=comm_a)  
    thd_a.start()  
    for i in range(size):  
        ram_a.wait_not_empty()  
        thd_b = Thread(target=comm_b)  
        thd_b.start()  
        for j in range(size):  
            ram_b.wait_not_empty()  
            for k in range(size):  
                acc += ram_a.read(k)*ram_b.read(k)  
            ram_b.pop()  
        thd_b.join()  
        ram_a.pop()  
    thd_a.join()  
    return acc
```

FSM



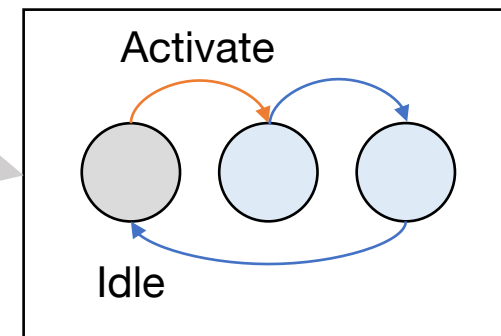
```
def comm_a():  
    for i in range(size):  
        a_addr = a_offset + i*size*word_size  
        ram_a.dma_read(axi_a, 0, a_addr, size)  
        ram_a.push()  
        ram_a.wait_not_full()
```

FSM



```
def comm_b():  
    for j in range(size):  
        b_addr = (b_offset +  
                  ram_a.read(j)*size*word_size)  
        ram_b.dma_read(axi_b, 0, b_addr, size)  
        ram_b.push()  
        ram_b.wait_not_full()
```

FSM



# 部分的な分離の実装：階層的な分離

- fork/joinによる部分的な分離を実際に実装するには、forkやjoinの位置を決定する必要がある
- すべての位置の組合せを列挙すると、組合せの数が二次関数的に増加する
- 候補を限定するため、抽象構文木（AST）を根から辿っていったって、for文やwhile文など特定の種類のノードに遭遇した場合に、その前後でforkとjoinができないかを判定する
- forkとjoinができるか（分離できるか）の判定は、読み出されているSRAMに書き込まれているかで行う

# コード例2

```
def main():  
    acc = 0  
    for i in range(size):  
        a_addr = a_offset + i*size*word_size  
        ram_a.dma_read(axi_a, 0, a_addr, size)  
        for j in range(size):  
            b_addr = (b_offset +  
                      ram_a.read(j)*size*word_size)  
            ram_b.dma_read(axi_b, 0, b_addr, size)  
            for k in range(size):  
                acc += ram_a.read(k)*ram_b.read(k)  
    return acc
```

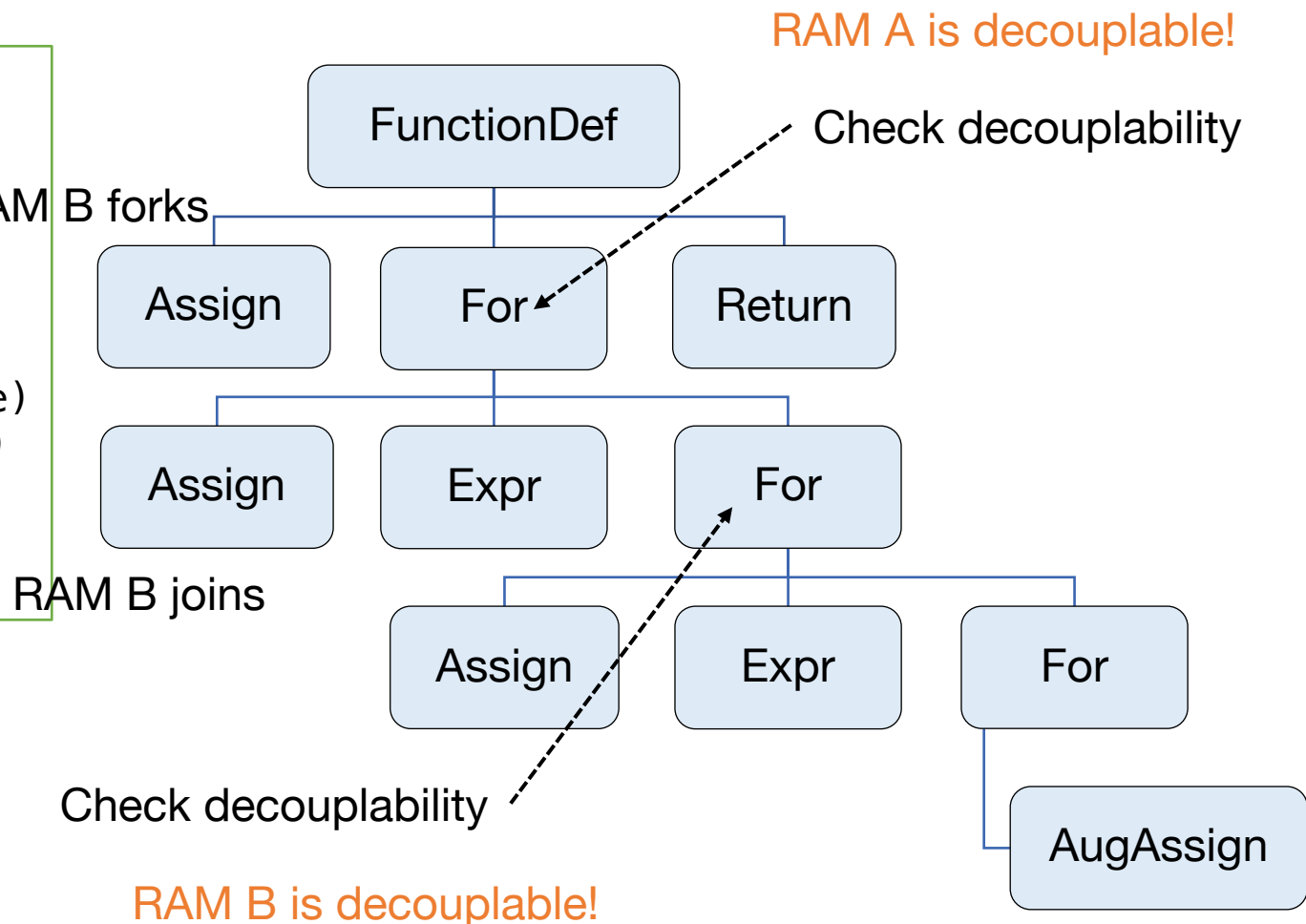
RAM A forks

RAM B forks

RAM A joins

RAM B joins

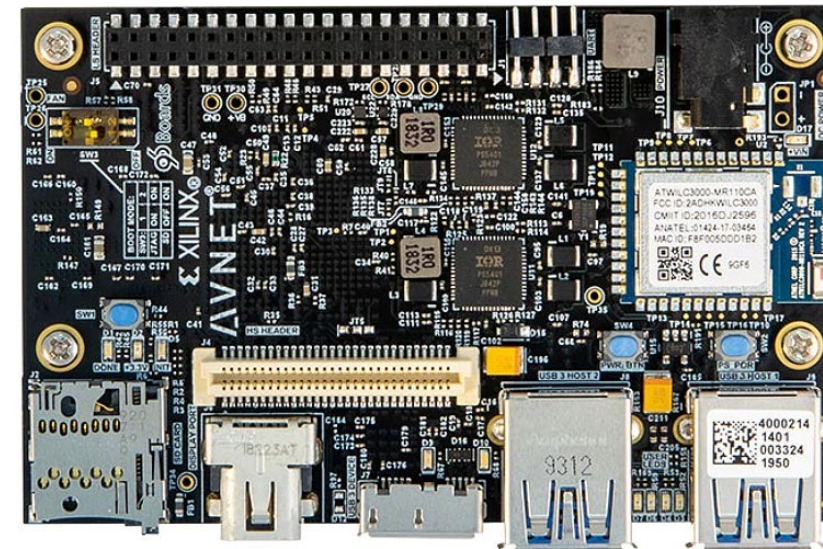
ASTを根から辿っていき、for文やwhile文など特定の種類のノードに遭遇したとき、その前後でforkとjoinができないか試みる





# 評価方法

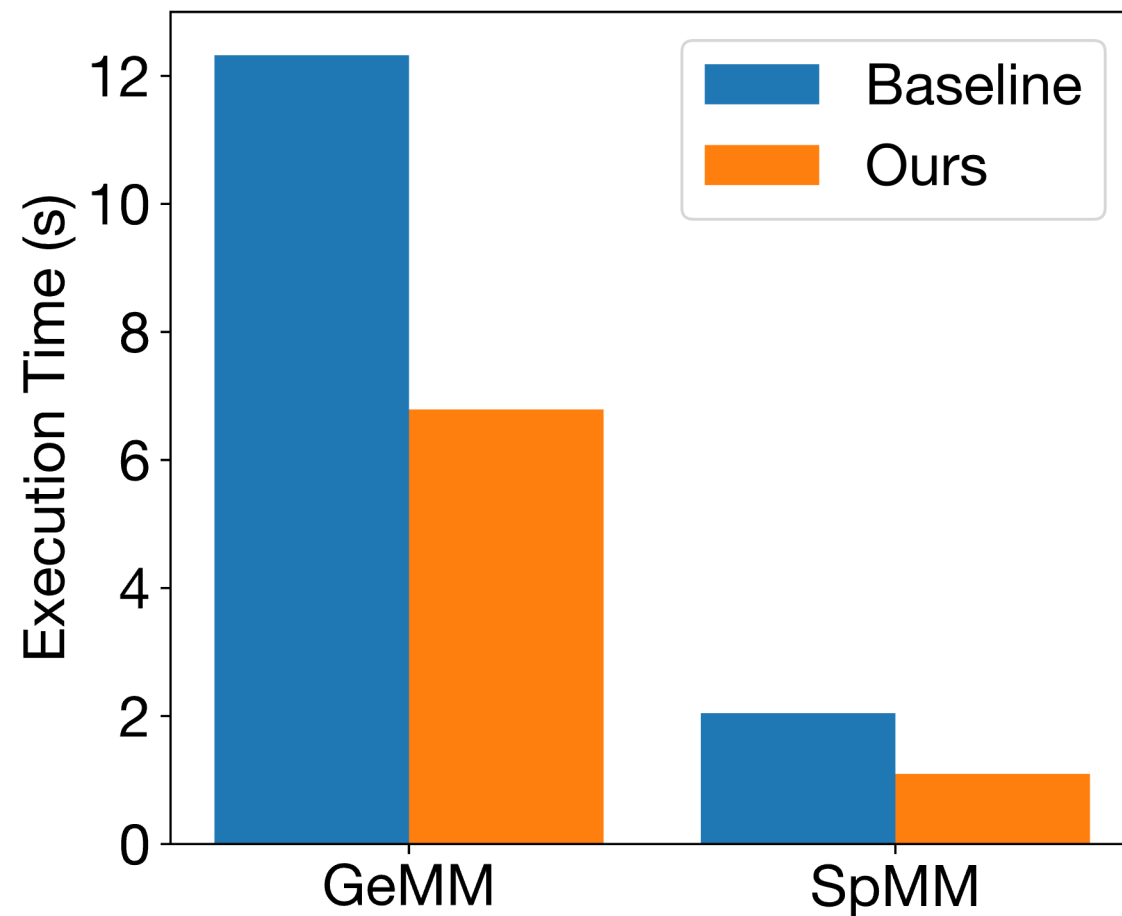
- 提案手法を高位合成ツールのVeriloggen (<https://github.com/PyHDI/veriloggen>) をもとに実装した
- 評価のためにFPGA上にアクセラレータを実装した
- FPGAボード：Ultra96-V2
- EDAツール：Vivado 2022.2
- ワークロード：
  - general matrix multiplication (GeMM)
  - sparse-matrix dense-matrix multiplication (SpMM)



cited from Avnet

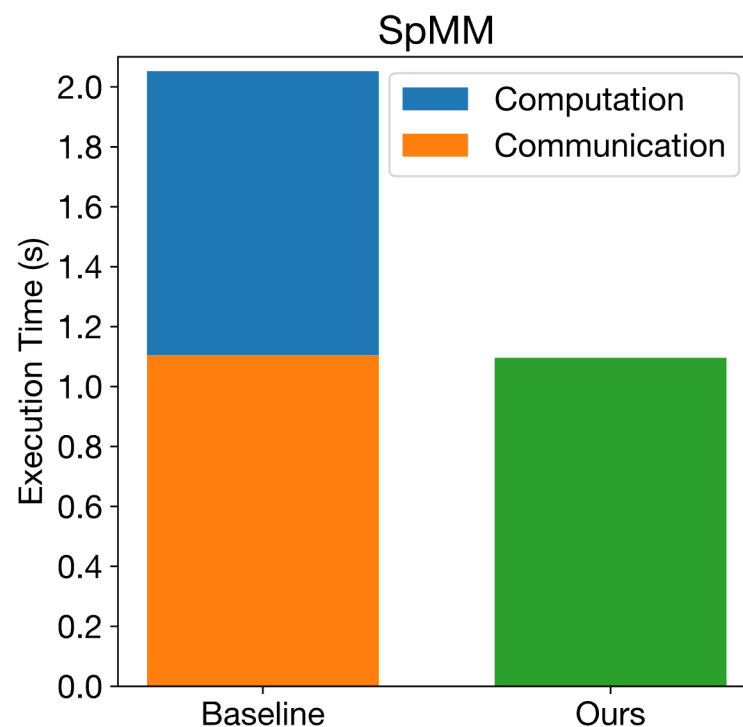
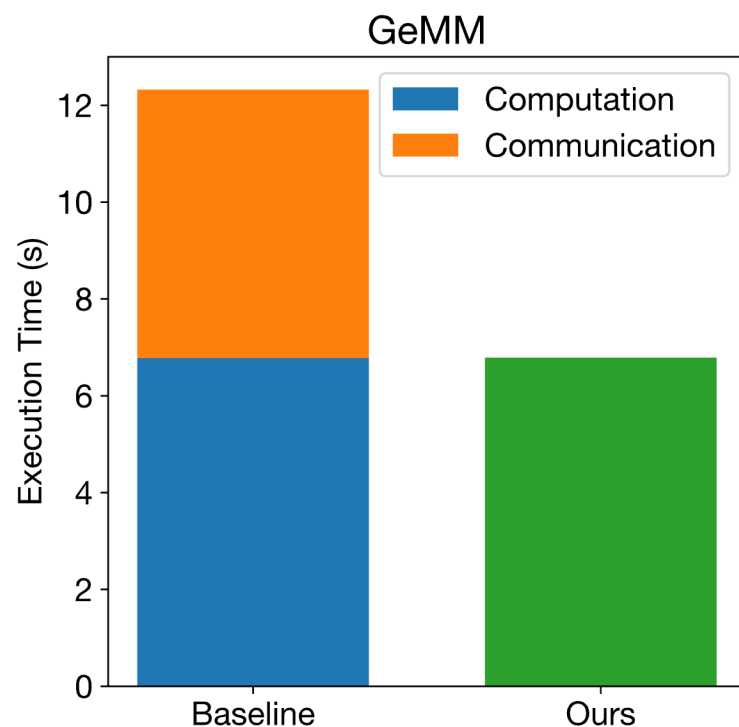
# 評価結果：実行時間

- 実機で実行時間を計測した
- 提案手法は実行時間を半分近く削減した



# 評価結果：実行時間

- 計算と通信に費やされる時間の内訳を含めたグラフを示す
- 提案手法によって計算と通信がオーバーラップされた結果、実行時間は計算と通信に費やされる時間のうち大きい方になっている

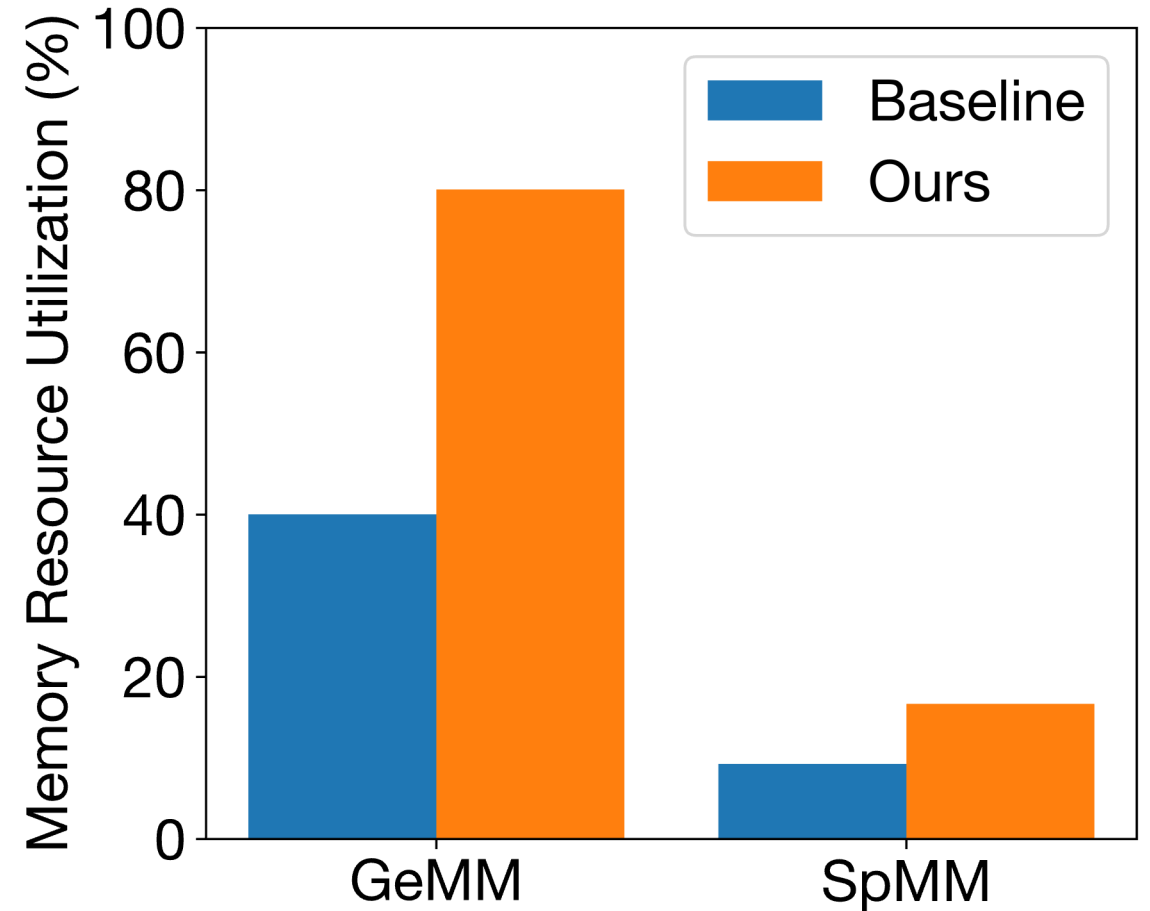


## 注意

- GeMMとSpMMでグラフの積み上げの順番が逆になっている
- GeMMは計算の時間の方が大きく、SpMMは通信の時間の方が大きい

# 評価結果：資源量

- EDAツールによりメモリ資源の使用率（使用したBRAMの割合）を測定した
- 提案手法はdouble bufferingに基づいているため、メモリ資源の利用量は倍になる



# 実行時間の解析

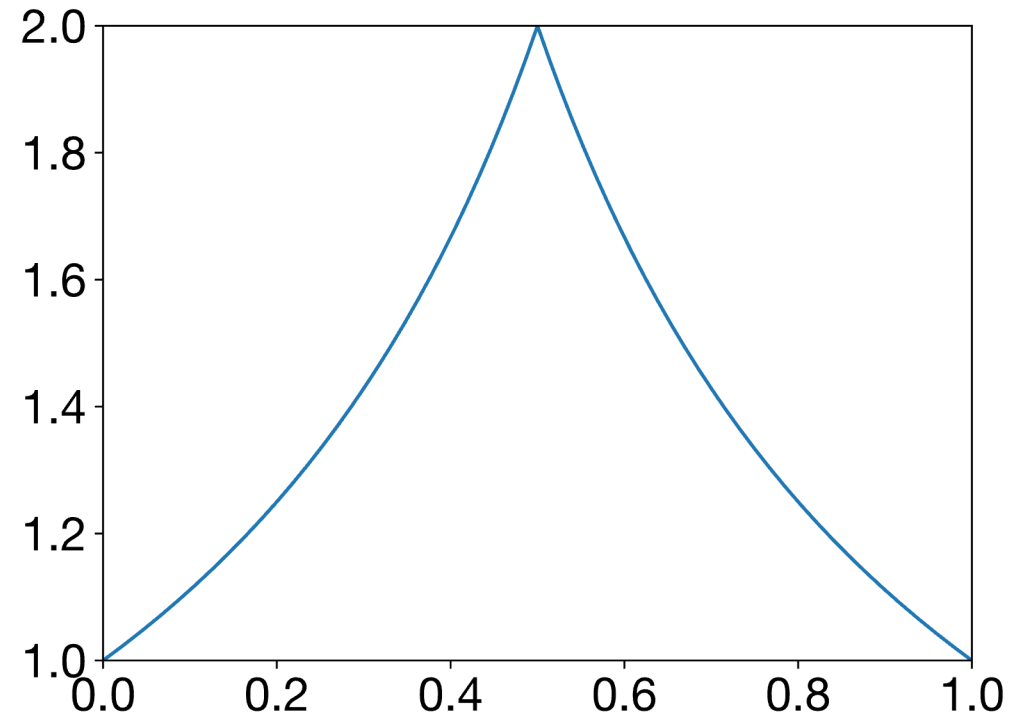
- 計算と通信に費やされる時間をそれぞれ $l_{\text{comp}}$ と $l_{\text{comm}}$ とする
- ベースラインの実行時間は単純に $l_{\text{comp}} + l_{\text{comm}}$ である
- 提案手法により計算と通信がオーバーラップされた場合、実行時間は $\max(l_{\text{comp}}, l_{\text{comm}})$ となる
- 性能向上はこれらの比として以下のように得られる
$$\frac{l_{\text{comp}} + l_{\text{comm}}}{\max(l_{\text{comp}}, l_{\text{comm}})} \leq \frac{\max(l_{\text{comp}}, l_{\text{comm}}) + \max(l_{\text{comp}}, l_{\text{comm}})}{\max(l_{\text{comp}}, l_{\text{comm}})} = 2$$
- ゆえに性能向上の上界は2である

# 実行時間の解析

- $r = l_{\text{comp}} / (l_{\text{comp}} + l_{\text{comm}})$ とおくと、 $r$ は計算に費やされる割合であり、 $1 - r$ は実行時間のうち通信に費やされる割合である
- このとき性能向上は以下のように表される

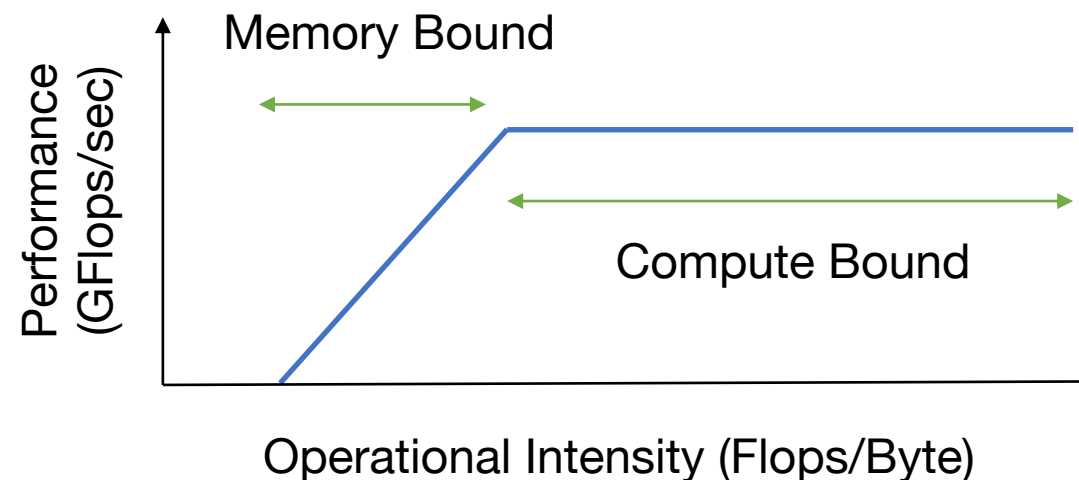
$$\frac{l_{\text{comp}} + l_{\text{comm}}}{\max(l_{\text{comp}}, l_{\text{comm}})} = \frac{1}{\max(r, 1 - r)}$$

- これを $r$ についての関数としてプロットすると図のようになる
- 計算と通信に費やされる時間が同程度 ( $r = 0.5$ 付近) のときに性能向上が大きくなることがわかる



# 実験結果の考察

- 実験では計算と通信に割かれる時間が同程度であったために、理論限界である2に近い性能向上が得られていた
- roofline modelを考慮すると、計算と通信のバランスが取れるようにアーキテクチャを最適化するのが適当だと言えるので、計算と通信に割かれる時間が近いというのはある程度妥当な条件ではある
- しかし、計算または通信にどうしても時間が偏ってしまうようなアプリケーションでは、提案手法の効果が小さくなってしまうと考えられる



# まとめ

- データオーケストレーション機構を分離することにより、計算と通信をオーバーラップさせ、性能を向上させることができる
- 提案手法はそれを自動的に分離することによって設計を容易にする



# 参考文献

- Pellauer, M., Shao, Y., Clemons, J., Crago, N., Hegde, K., Venkatesan, R., Keckler, S., Fletcher, C., & Emer, J. (2019). Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 137–151). Association for Computing Machinery.
- Avnet. Ultra96-V2.  
<https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/>