Another Automatic Kernel Generator on Huawei NPU with Polyhedral Compilation

Shenghu Jiang, Mingli Sun, Xiaohua Shi
姜圣虎，孙明利，史晓华

Beihang University

Outline

- Polyhedral Compilation
- Front End
- Back End
- Recursive Approach for Schedule Generation
- Auto-tuning
- Auto-fusion
- Future Work

Polyhedral Compilation

- Integer Set Library (ISL)

```
A[i, j] : 0 <= i < 16 and 0 <= j < 256
B[i, j] : 0 <= i < 16 and 0 <= j < 256
```

Input

```
C[i] : 0 <= i < 16
```

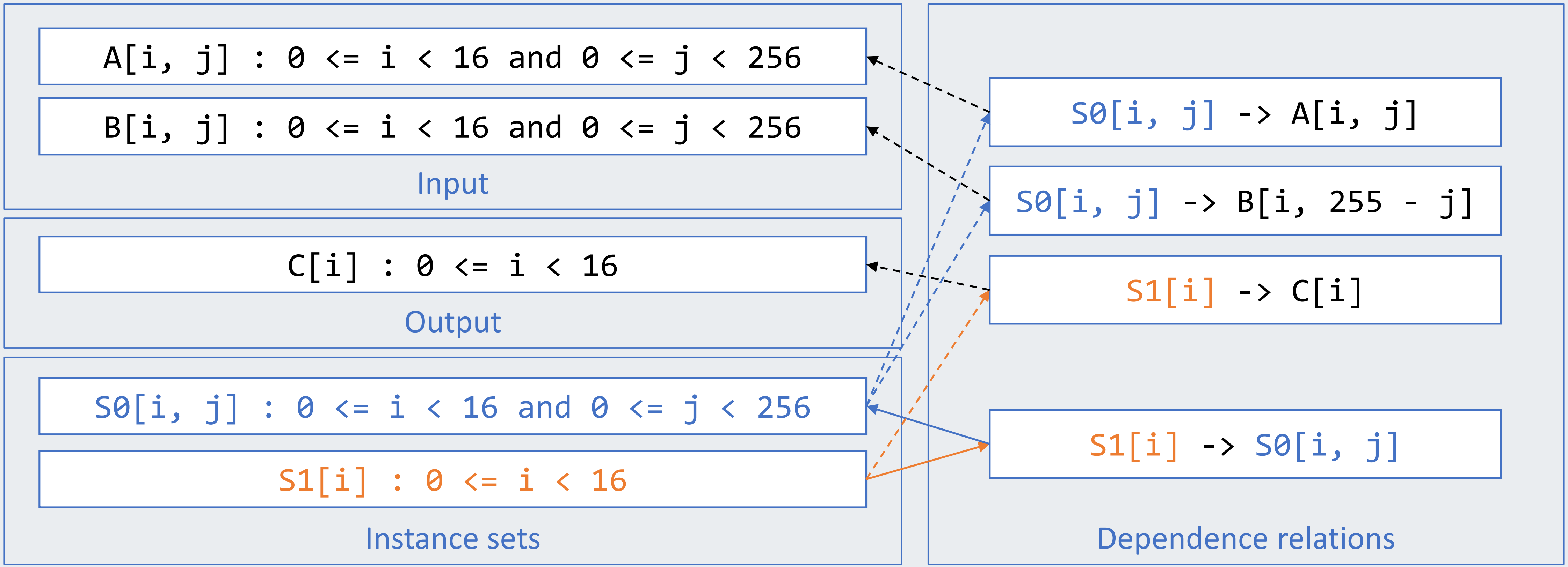
Output

```
S0[i, j] : 0 <= i < 16 and 0 <= j < 256
S1[i] : 0 <= i < 16
```

Instance sets

Polyhedral Compilation

- Integer Set Library (ISL)



Polyhedral Compilation

- Integer Set Library (ISL)

```
A[i, j] : 0 <= i < 16 and 0 <= j < 256
B[i, j] : 0 <= i < 16 and 0 <= j < 256
```

Input

```
C[i] : 0 <= i < 16
```

Output

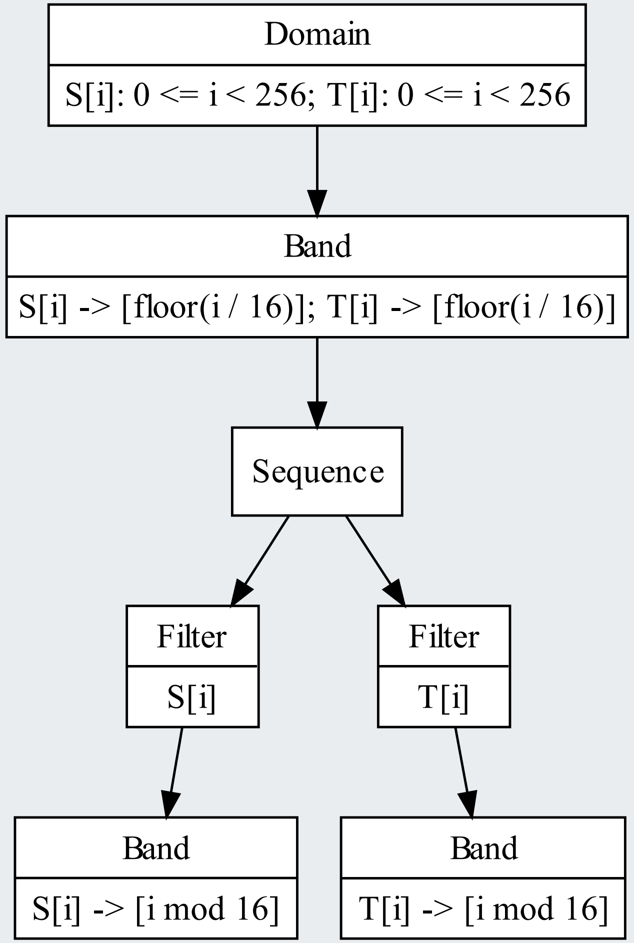
```
S0[i, j] : 0 <= i < 16 and 0 <= j < 256
S1[i] : 0 <= i < 16
```

Instance sets

```
S0[0, 0]
S0[0, 1]
...
S0[0, 255]
S1[0]
S0[1, 0]
...
S0[15, 255]
S1[15]
```

Schedule

Schedule Trees




```
for (c0 = 0; c0 < 16; c0 += 1) {
    for (c1 = 0; c1 < 16; c1 += 1) {
        S(c0 * 16 + c1);
    }
    for (c2 = 0; c2 < 16; c2 += 1) {
        T(c0 * 16 + c2);
    }
}
```

Front End – IR Code

```
opdef Conv2D<
    N: int, C0: int, C1: int, H: int, W: int, OutC0: int, OutC1: int,
    Kernel: int, Stride: int, Group: int
>(
    data[N, C0, H, W, C1],
    weight[OutC0, floordiv(C0, Group), Kernel, Kernel, C1, OutC1]
) -> (
    res[N, OutC0, add(floordiv(sub(H, Kernel), Stride), 1),
        add(floordiv(sub(W, Kernel), Stride), 1), OutC1]
) {
    mult[n, u, h, w, x, i, j, y, v] = mul(
        data[n, x, add(h, i), add(w, j), y],
        weight[u, sub(x, mul(floordiv(x, floordiv(C0, Group)), floordiv(C0, Group))), i, j, y, v]
    )
    res[n, u, h, w, v] = reduce<add, 0.0>({
        mult[n_, u_, h_, w_, x, i, j, y, v_]: and(
            eq(n, n_), eq(u, u_), eq(v, v_), eq(mul(h, Stride), h_), eq(mul(w, Stride), w_),
            le(0, i), lt(i, Kernel), le(0, j), lt(j, Kernel), le(0, x), lt(x, C0), le(0, y), lt(y, C1)
        )
    })
}
```

Front End – the Polyhedral Model

IR Code

Eliminate parameters &

Preprocess

Instance Sets

Dependence Relations

```
opdef Dot<N: int, M: int>(A[N, M], B[N, M]) -> (C[N]) {
    T[i, j] = mul(A[i, j], B[i, j])
    C[i] = reduce<add, 0.0>({
        T[u, v]: and(eq(u, i), le(0, v), lt(v, M))
    )
}
```

Front End – the Polyhedral Model

IR Code

Eliminate parameters &

Preprocess

Instance Sets

Dependence Relations

```
opdef Dot<>(A[16, 256], B[16, 256]) -> (C[16]) {
    T[i, j] = mul(A[i, j], B[i, j])
    C[i] = reduce<add, 0.0>({
        T[u, v]: and(eq(u, i), le(0, v), lt(v, 256))
    )
}
```

Front End – the Polyhedral Model

IR Code

Eliminate parameters &

Preprocess

Instance Sets

Dependence Relations

```
T[i, j] : 0 <= i < 16 and 0 <= j < 256
C[i] : 0 <= i < 16
```

Instance sets

```
C[i] -> _C_gm[i]
...
C[i] -> T[i, j]
```

Dependence relations

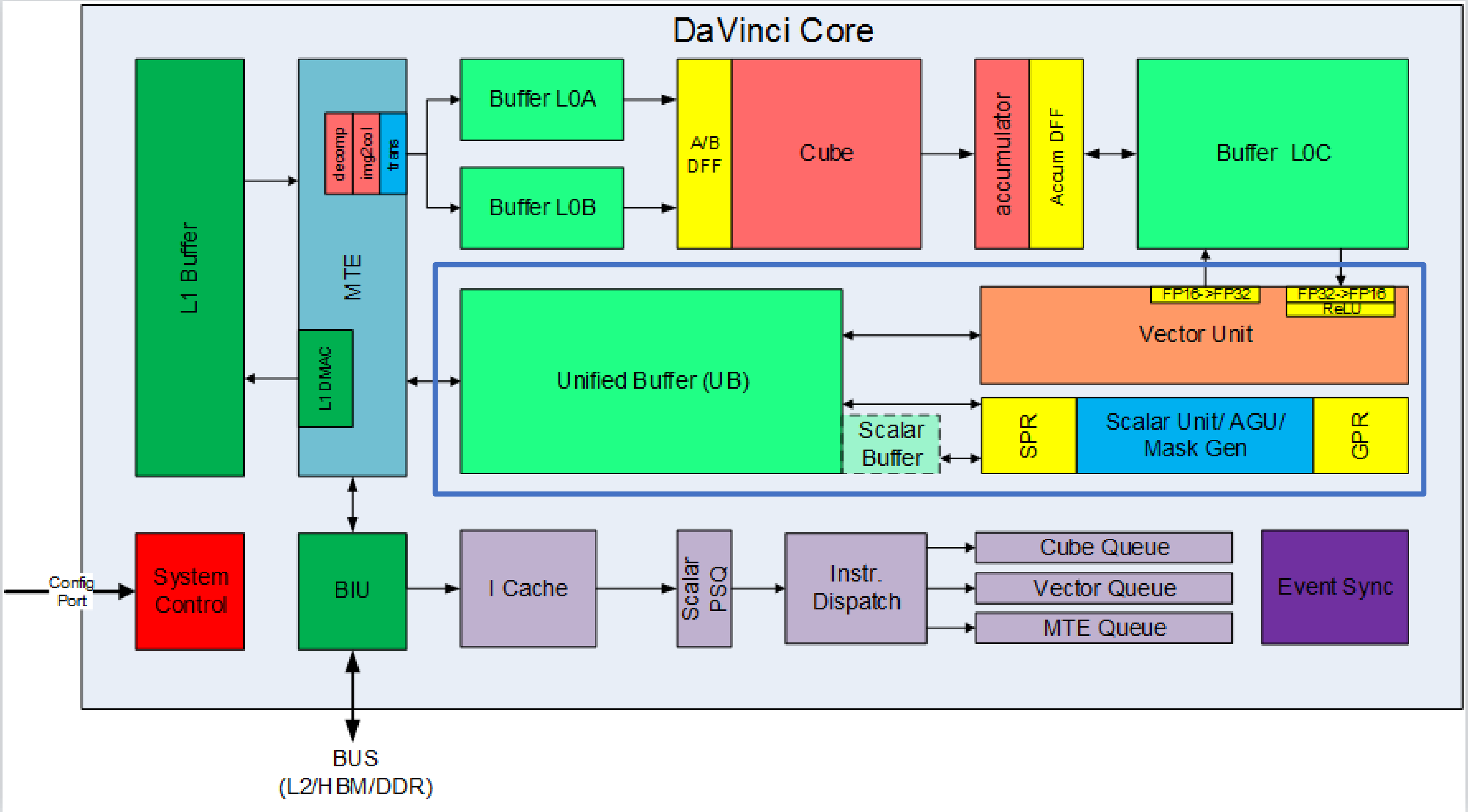
Front End – Fusion

```
opdef Dot<>(A[16, 256], B[16, 256]) -> (C[16]) {
    T[i, j] = mul(A[i, j], B[i, j])
    C[i] = reduce<add, 0.0>({
        T[u, v]: and(eq(u, i), le(0, v), lt(v, 256))
    )
}
```

```
opdef ReLU<>(A[16]) -> (B[16]) {
    B[i] = relu(A[i])
}
```

```
opdef Dot_ReLU<>(A_0[16, 256], B_0[16, 256]) -> (B_1[16]) {
    T_0[i, j] = mul(A_0[i, j], B_0[i, j])
    C_0[i] = reduce<add, 0.0>({
        T_0[u, v]: and(eq(u, i), le(0, v), lt(v, 256))
    )
    B_1[i] = relu(C_0[i])
}
```

Da Vinci Architecture



Da Vinci Architecture on Huawei Ascend 310 NPU

Back End: TIK

- TIK: a wrapper of the TVM IR builder
- Low-level interfaces:
  - Tensor management, control flows, emitting instructions, ...

```
from tbe import tik
tik_instance = tik.Tik()
data_A = tik_instance.Tensor('float16', (128,), name='data_A', scope=tik.scope_gm)
data_B = tik_instance.Tensor('float16', (128,), name='data_B', scope=tik.scope_gm)
data_C = tik_instance.Tensor('float16', (128,), name='data_C', scope=tik.scope_gm)
data_A_ub = tik_instance.Tensor('float16', (128,), name='data_A_ub', scope=tik.scope_ubuf)
data_B_ub = tik_instance.Tensor('float16', (128,), name='data_B_ub', scope=tik.scope_ubuf)
data_C_ub = tik_instance.Tensor('float16', (128,), name='data_C_ub', scope=tik.scope_ubuf)
tik_instance.data_move(data_A_ub, data_A, 0, 1, 128 //16, 0, 0)
tik_instance.data_move(data_B_ub, data_B, 0, 1, 128 //16, 0, 0)
tik_instance.vec_add(128, data_C_ub[0], data_A_ub[0], data_B_ub[0], 1, 8, 8, 8)
tik_instance.data_move(data_C, data_C_ub, 0, 1, 128 //16, 0, 0)
tik_instance.BuildCCE(kernel_name='simple_add', inputs=[data_A, data_B], outputs=[data_C])
```

Recursive Approach for Schedule Generation

Input

- Instances
  - Computation (IR)
  - Prefix schedule
- Dependence relations
- Access relations
- Tunable parameters
- …

?

Output

- Schedule tree (subtree)
- TIK API arguments and calls

Recursive Approach for Schedule Generation

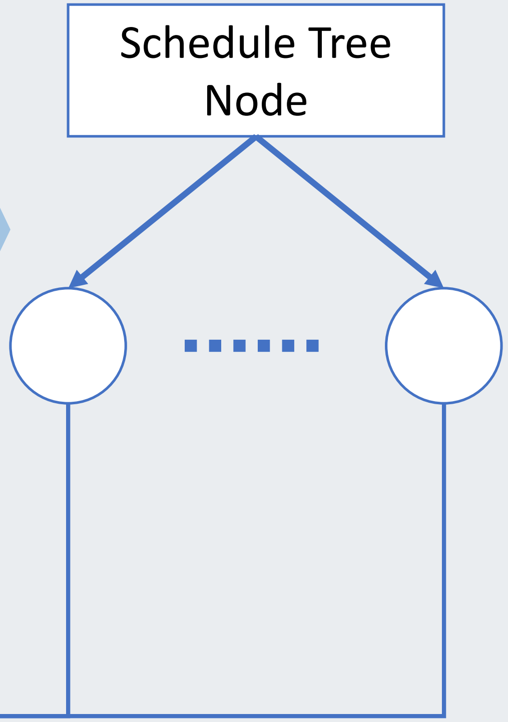
Input

- Instances
  - Computation (IR)
  - Prefix schedule
- Dependence relations
- Access relations
- Tunable parameters
- …

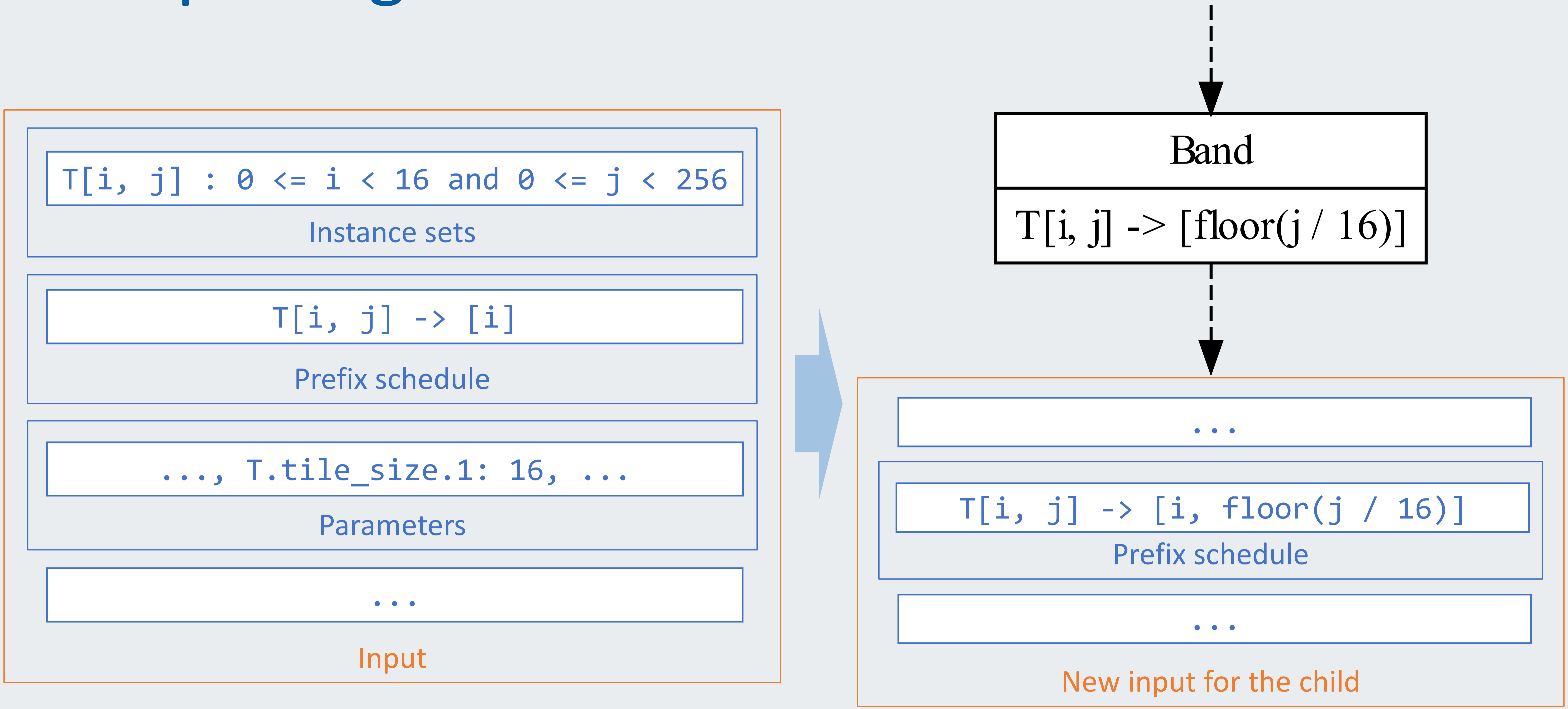
Output (each step)

- Root of the subtree
  - Maybe a chain
- API calls for the root

- Inputs of the recursion for the children



Loop Tiling



T[i, j] : 0 <= i < 16 and 0 <= j < 256
Instance sets
T[i, j] -> [i]
Prefix schedule
..., T.tile_size.1: 16, ...
Parameters
...
Input
Band
T[i, j] -> [floor(j / 16)]
...
T[i, j] -> [i, floor(j / 16)]
Prefix schedule
...
New input for the child

Vectorization

```
T[i, j] : 0 <= i < 16 and 0 <= j < 256
```

Instance sets

```
T[i, j] -> [i, floor(j / 16)]
```

Prefix schedule

```
T[i, j] -> T_ub[k] : k = j mod 16
```

Access

```
...
```

Input

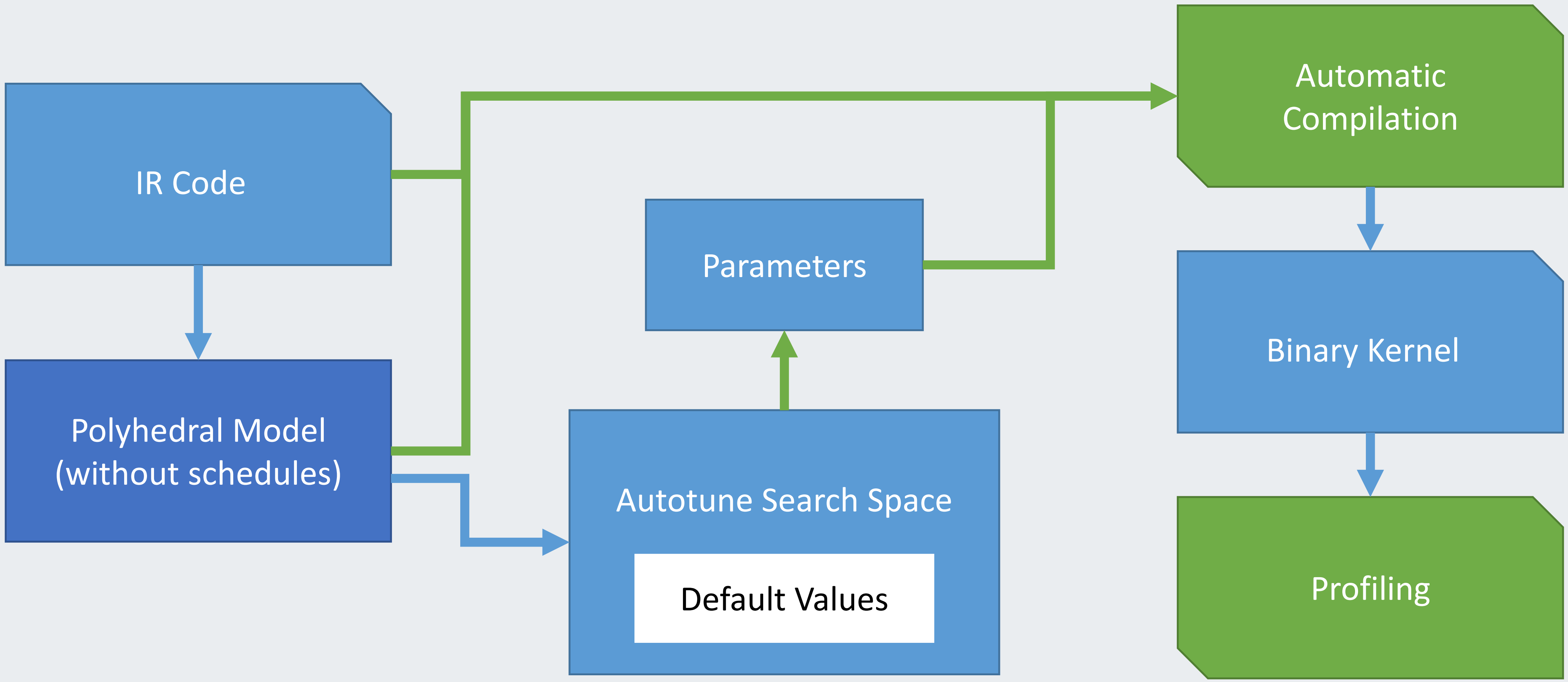
Expansion

T[i, j] -> T_impl[i, floor(j / 16)]

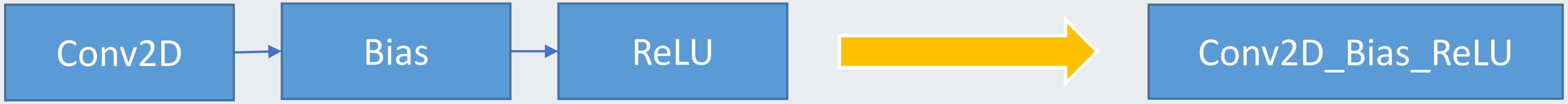
```
tensor = ctx.tensor_table.get('T_ub')
ctx.tik_instance.vec_mul(tensor, ...)
```

T_impl

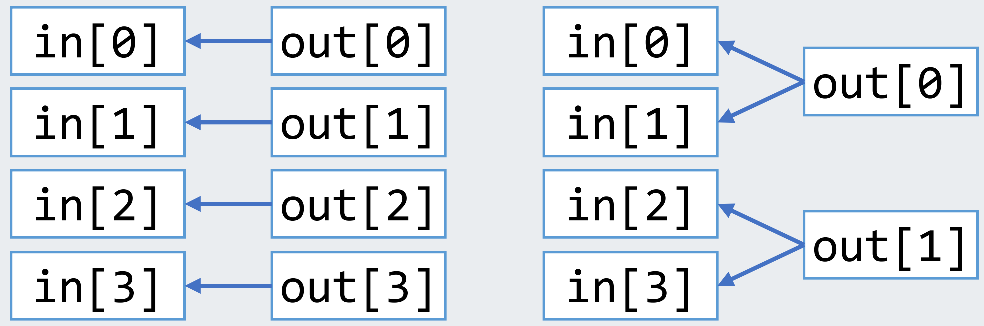
Auto-tuning


IR Code
Polyhedral Model
(without schedules)
Parameters
Autotune Search Space
Default Values
Automatic
Compilation
Binary Kernel
Profiling

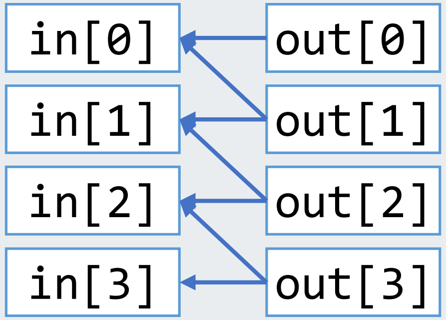
Auto-fusion



- Dependence relations of outputs on inputs
  - Injective → "fusable" (in most cases)
  - Typically, elementwise operators will be detected as fusable kernels



Injective



Not injective

Future Work

- Utilize “cube” instructions and other hardware accelerations
- Computations → low-level structures and statements
  - Complex
  - Lots of unnecessary details in the implementation
  - Solution: More levels of IR? More stages?
- Recursive compilation? Beam Search!
  - Redesign of auto-tuning
- ...

Acknowledgments

HUAWEI

Thank you for listening

Shenghu Jiang
ChieloNewctle@Yandex.com

Mingli Sun
2239625661@qq.com

Xiaohua Shi
xhshi@buaa.edu.cn