AKG: Automatic Kernel Generation for Neural Processing Units using Polyhedral Transformations

Jie Zhao[1] Bojie Li[2] Wang Nie[2] Zhen Geng[2]
Renwei Zhang[2] Xiong Gao[2] Bin Cheng[2] Chen Wu[2]
Yun Cheng[2] Zheng Li[2] Peng Di[2†] Kun Zhang[2‡] Xuefeng Jin[2]

[1]State Key Laboratory of Mathematical Engineering and Advanced Computing, China
[2]Huawei Technologies Co. Ltd., China
[†]Now with Ant Group, China [‡]Now with Tencent Penglai Lab, China

2021.06.23, Virtual, Canada

42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)

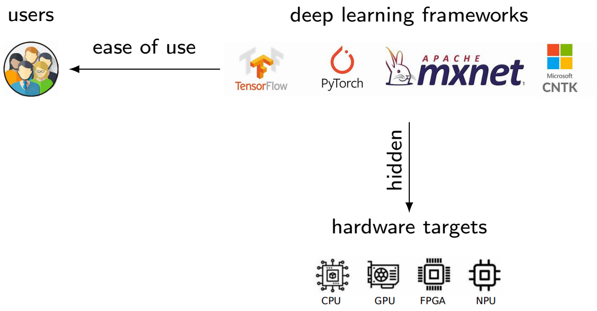
Outline

Why DL Compilers for NPUs

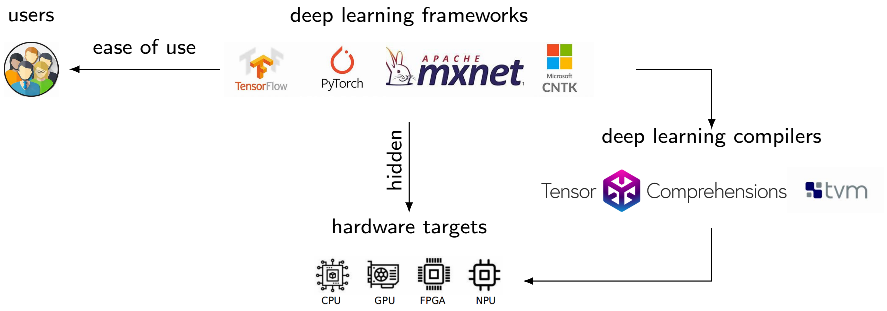
deep learning frameworks

TensorFlow PyTorch APACHE mxnet Microsoft CNTK

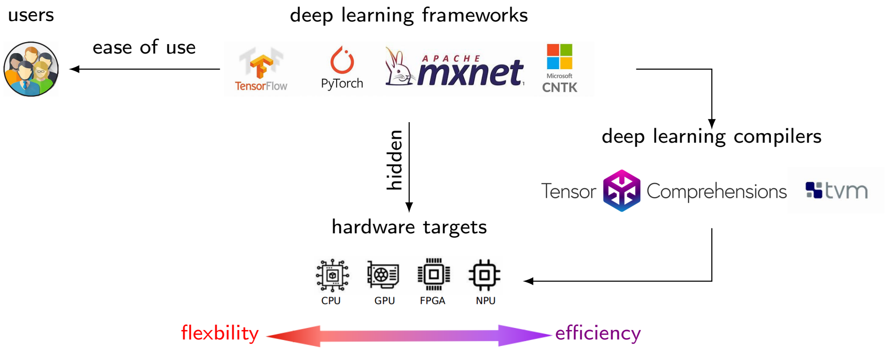
Why DL Compilers for NPUs



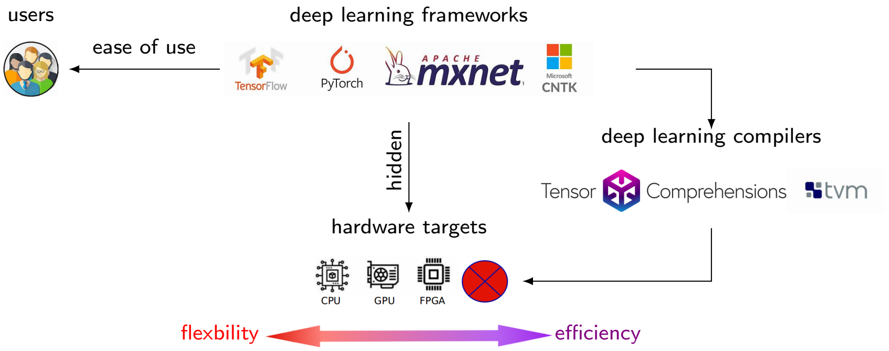
Why DL Compilers for NPUs



Why DL Compilers for NPUs

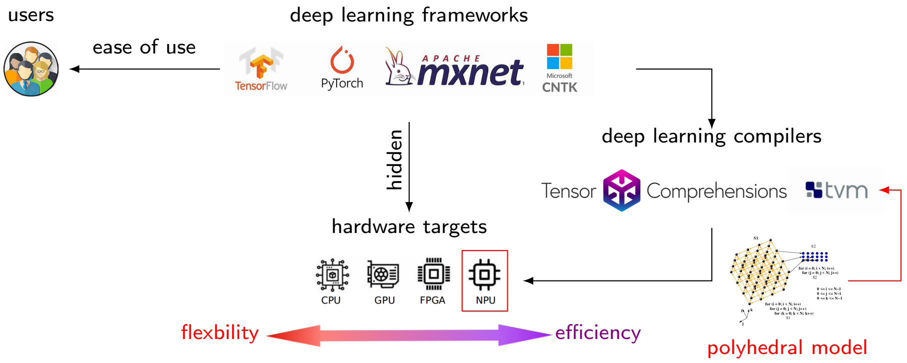


Why DL Compilers for NPUs



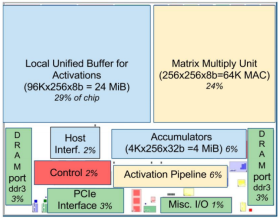
- Prior DL compilers [2, 10] do not support code generation for NPUs.

Why DL Compilers for NPUs



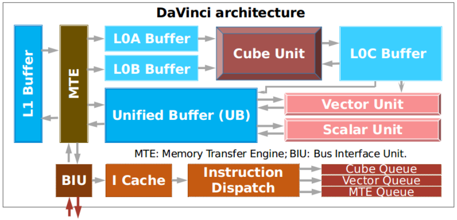
- Prior DL compilers [2, 10] do not support code generation for NPUs.
- We present AKG in this paper to implement AUTOMATIC KERNEL GENERATION for NPUs using Polyhedral Transformations.

Challenges faced by DL Compilers for NPUs



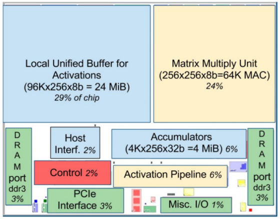
Local Unified Buffer for Activations (96Kx256x8b = 24 MiB) 29% of chip
Matrix Multiply Unit (256x256x8b=64K MAC) 24%
DRAM port ddr3 3%
Host Interf. 2%
Accumulators (4Kx256x32b =4 MiB) 6%
DRAM port ddr3 3%
Control 2%
Activation Pipeline 6%
PCIe Interface 3%
Misc. I/O 1%

Google TPU [6]



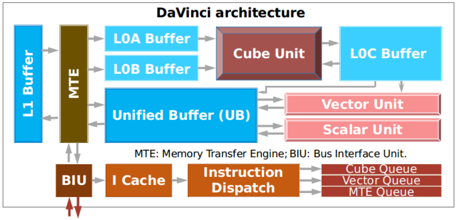
Huawei Ascend [8]

Challenges faced by DL Compilers for NPUs



Local Unified Buffer for Activations (96Kx256x8b = 24 MiB) 29% of chip
Matrix Multiply Unit (256x256x8b=64K MAC) 24%
DRAM port ddr3 3%
Host Interf. 2%
Accumulators (4Kx256x32b =4 MiB) 6%
DRAM port ddr3 3%
Control 2%
Activation Pipeline 6%
PCIe Interface 3%
Misc. I/O 1%

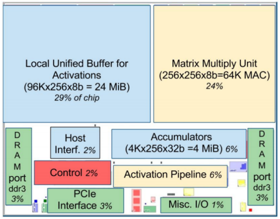
Google TPU [6]



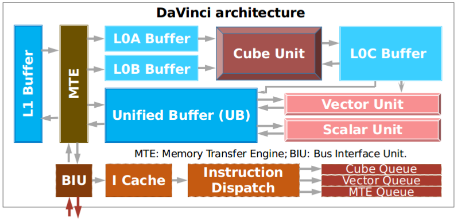
Huawei Ascend [8]

- Effective scheduling for the conflicting demands of parallelism and locality.

Challenges faced by DL Compilers for NPUs


Local Unified Buffer for Activations (96Kx256x8b = 24 MiB) 29% of chip
Matrix Multiply Unit (256x256x8b=64K MAC) 24%
DRAM port ddr3 3%
Host Interf. 2%
Accumulators (4Kx256x32b =4 MiB) 6%
DRAM port ddr3 3%
Control 2%
Activation Pipeline 6%
PCIe Interface 3%
Misc. I/O 1%

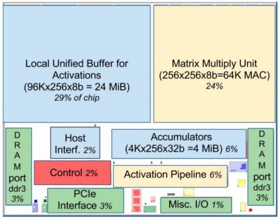
Google TPU [6]



Huawei Ascend [8]

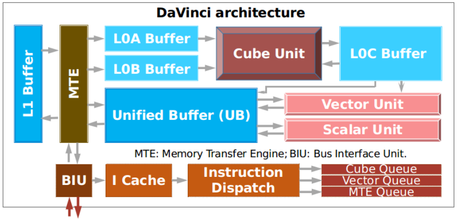
- Effective scheduling for the conflicting demands of parallelism and locality.
- Software-controlled storage management between multi-level, multi-directional memory hierarchy.

Challenges faced by DL Compilers for NPUs



Local Unified Buffer for Activations (96Kx256x8b = 24 MiB) 29% of chip
Matrix Multiply Unit (256x256x8b=64K MAC) 24%
DRAM port ddr3 3%
Host Interf. 2%
Accumulators (4Kx256x32b =4 MiB) 6%
DRAM port ddr3 3%
Control 2%
Activation Pipeline 6%
PCIe Interface 3%
Misc. I/O 1%

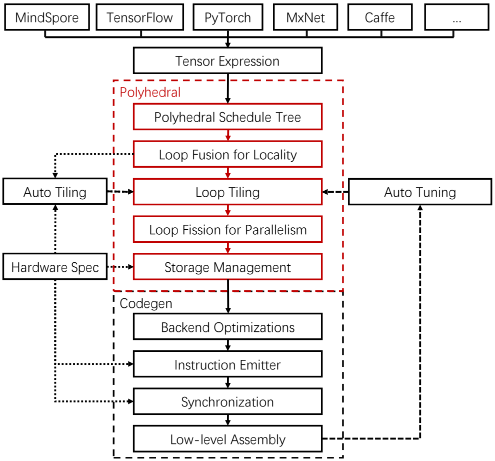
Google TPU [6]



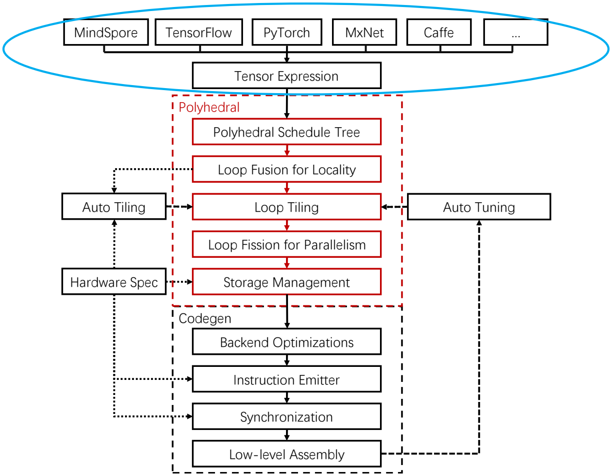
Huawei Ascend [8]

- Effective scheduling for the conflicting demands of parallelism and locality.
- Software-controlled storage management between multi-level, multi-directional memory hierarchy.
- Automatic implementation of domain-specific transformations for convolution.

Overview of Our Approach

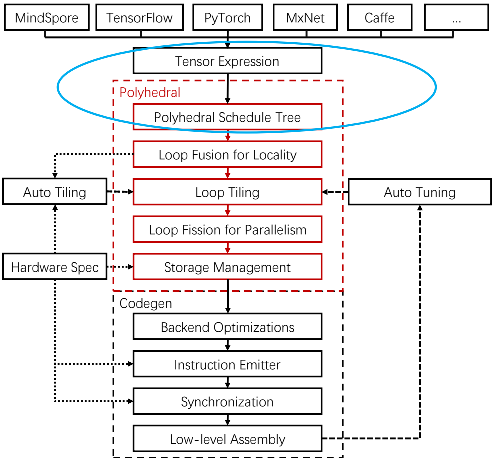


Overview of Our Approach



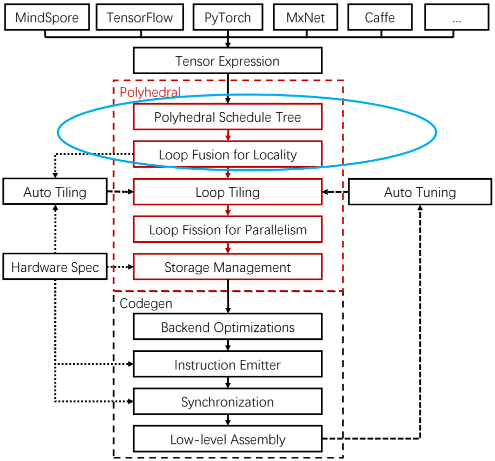
AKG inherits the graph engine and DSL of TVM [2] for expressing tensor computations.

Overview of Our Approach



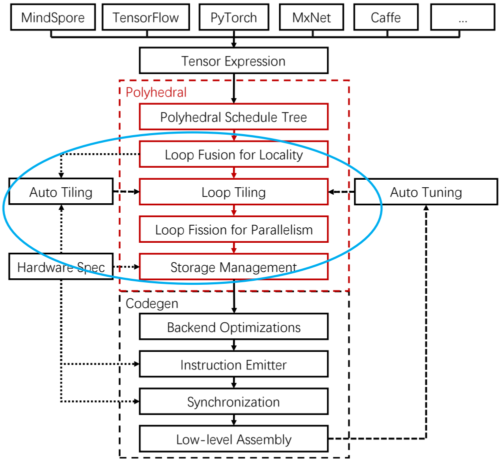
AKG branches from TVM by lowering HalideIR [9] generated by the DSL to schedule trees.

Overview of Our Approach



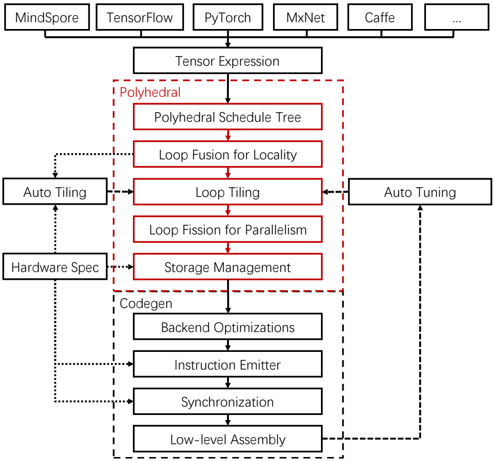
AKG leverages versatile polyhedral scheduling algorithms, exploiting parallelism and locality of programs simultaneously.

Overview of Our Approach



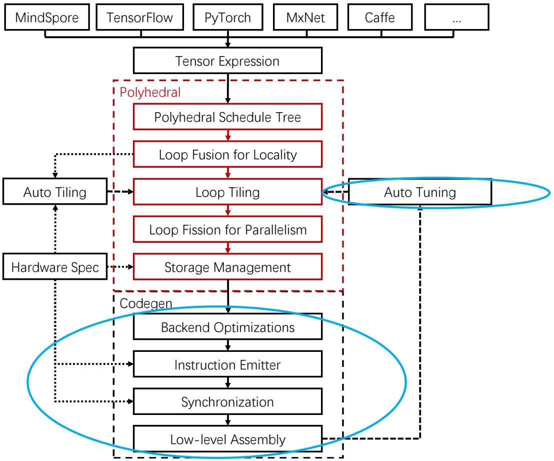
AKG models the interplay between loop fusion and tiling, achiveing automatic decoupled data orchestration between memory hierarchy.

Overview of Our Approach



AKG takes as input an external schedule tree to implement the *img2col* transformations [5] for convolutions.

Overview of Our Approach



AKG also implements vectorization, low-level synchronization, auto-tuning, improving the performance of its generated code.

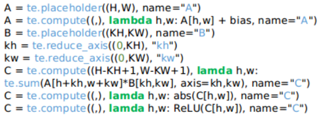
Abstraction Lowering

The polyhedral model [1, 3, 12] is a mathematical abstraction use to analyze and optimize program.

Abstraction Lowering

One can lower a tensor program written by TVM's DSL to a so-called *schedule tree* representation [4] of the polyhedral model.

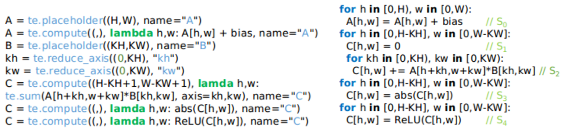
Abstraction Lowering



```
A = te.placeholder((H,W), name="A")
A = te.compute((,), lambda h,w: A[h,w] + bias, name="A")
B = te.placeholder((KH,KW), name="B")
kh = te.reduce_axis((0,KH), "kh")
kw = te.reduce_axis((0,KW), "kw")
C = te.compute((H-KH+1,W-KW+1), lamda h,w:
te.sum(A[h+kh,w+kw]*B[kh,kw], axis=kh,kw), name="C")
C = te.compute((,), lamda h,w: abs(C[h,w]), name="C")
C = te.compute((,), lamda h,w: ReLU(C[h,w]), name="C")
```

One can lower a tensor program written by TVM's DSL to a so-called *schedule tree* representation [4] of the polyhedral model.

Abstraction Lowering

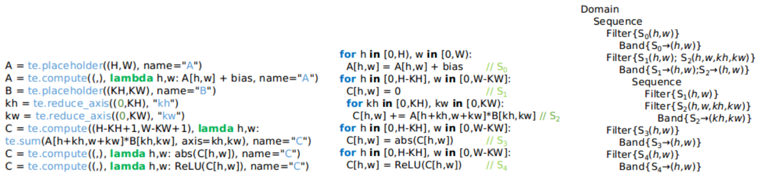


```
A = te.placeholder((H,W), name="A")
A = te.compute((,), lambda h,w: A[h,w] + bias, name="A")
B = te.placeholder((KH,KW), name="B")
kh = te.reduce_axis((0,KH), "kh")
kw = te.reduce_axis((0,KW), "kw")
C = te.compute((H-KH+1,W-KW+1), lamda h,w:
te.sum(A[h+kh,w+kw]*B[kh,kw], axis=kh,kw), name="C")
C = te.compute((,), lamda h,w: abs(C[h,w]), name="C")
C = te.compute((,), lamda h,w: ReLU(C[h,w]), name="C")
```

```
for h in [0,H), w in [0,W):
 A[h,w] = A[h,w] + bias      // S0
for h in [0,H-KH], w in [0,W-KW]:
 C[h,w] = 0                  // S1
 for kh in [0,KH), kw in [0,KW):
  C[h,w] += A[h+kh,w+kw]*B[kh,kw] // S2
for h in [0,H-KH], w in [0,W-KW]:
 C[h,w] = abs(C[h,w])        // S3
for h in [0,H-KH], w in [0,W-KW]:
 C[h,w] = ReLU(C[h,w])       // S4
```

One can lower a tensor program written by TVM's DSL to a so-called *schedule tree* representation [4] of the polyhedral model.

Abstraction Lowering



```
A = te.placeholder((H,W), name="A")
A = te.compute((,), lambda h,w: A[h,w] + bias, name="A")
B = te.placeholder((KH,KW), name="B")
kh = te.reduce_axis((0,KH), "kh")
kw = te.reduce_axis((0,KW), "kw")
C = te.compute((H-KH+1,W-KW+1), lamda h,w:
te.sum(A[h+kh,w+kw]*B[kh,kw], axis=kh,kw), name="C")
C = te.compute((,), lamda h,w: abs(C[h,w]), name="C")
C = te.compute((,), lamda h,w: ReLU(C[h,w]), name="C")
```

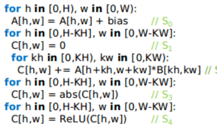
```
for h in [0,H), w in [0,W):
 A[h,w] = A[h,w] + bias        // S_0
for h in [0,H-KH], w in [0,W-KW]:
 C[h,w] = 0                    // S_1
 for kh in [0,KH), kw in [0,KW):
  C[h,w] += A[h+kh,w+kw]*B[kh,kw] // S_2
for h in [0,H-KH], w in [0,W-KW]:
 C[h,w] = abs(C[h,w])          // S_3
for h in [0,H-KH], w in [0,W-KW]:
 C[h,w] = ReLU(C[h,w])         // S_4
```

```
Domain
  Sequence
    Filter{S_0(h,w)}
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw)}
      Band{S_1→(h,w);S_2→(h,w)}
        Sequence
          Filter{S_1(h,w)}
          Filter{S_2(h,w,kh,kw)}
            Band{S_2→(kh,kw)}
    Filter{S_3(h,w)}
      Band{S_3→(h,w)}
    Filter{S_4(h,w)}
      Band{S_4→(h,w)}
```

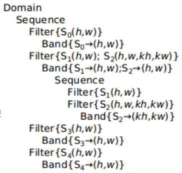
The schedule tree is functional due to its rich set of node types:

- a domain node, filter nodes
- band nodes, sequence nodes and set nodes
- extension nodes
- mark nodes
- and more ...

Versatile Polyhedral Scheduling

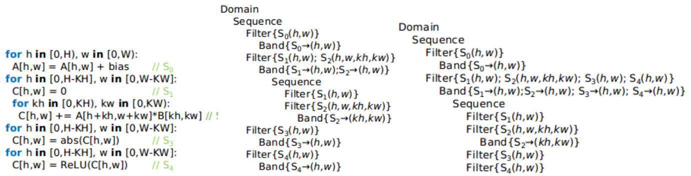


```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias        // S0
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = 0                    // S1
  for kh in [0,KH), kw in [0,KW):
    C[h,w] += A[h+kh,w+kw]*B[kh,kw] // S
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = abs(C[h,w])          // S3
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = ReLU(C[h,w])         // S4
```



```
Domain
  Sequence
    Filter{S0(h,w)}
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw)}
      Band{S1→(h,w);S2→(h,w)}
        Sequence
          Filter{S1(h,w)}
          Filter{S2(h,w,kh,kw)}
            Band{S2→(kh,kw)}
    Filter{S3(h,w)}
      Band{S3→(h,w)}
    Filter{S4(h,w)}
      Band{S4→(h,w)}
```

Versatile Polyhedral Scheduling



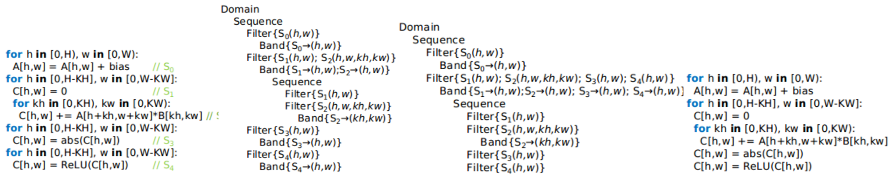
```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias        // S_0
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = 0                    // S_1
  for kh in [0,KH), kw in [0,KW):
   C[h,w] += A[h+kh,w+kw]*B[kh,kw] //
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = abs(C[h,w])          // S_3
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = ReLU(C[h,w])         // S_4
```

```
Domain
  Sequence
    Filter{S_0(h,w)}
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw)}
      Band{S_1→(h,w);S_2→(h,w)}
        Sequence
          Filter{S_1(h,w)}
          Filter{S_2(h,w,kh,kw)}
            Band{S_2→(kh,kw)}
    Filter{S_3(h,w)}
      Band{S_3→(h,w)}
    Filter{S_4(h,w)}
      Band{S_4→(h,w)}
```

```
Domain
  Sequence
    Filter{S_0(h,w)}
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)}
        Sequence
          Filter{S_1(h,w)}
          Filter{S_2(h,w,kh,kw)}
            Band{S_2→(kh,kw)}
          Filter{S_3(h,w)}
          Filter{S_4(h,w)}
```

- We leverage the ILP-based *isl* scheduler [11, 13] to compute new schedules that exploit parallelism and temporal locality simultaneously.

Versatile Polyhedral Scheduling



```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias        // S_0
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = 0                    // S_1
  for kh in [0,KH), kw in [0,KW):
   C[h,w] += A[h+kh,w+kw]*B[kh,kw] //
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = abs(C[h,w])          // S_3
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = ReLU(C[h,w])         // S_4
```

```
Domain
  Sequence
    Filter{S_0(h,w)}
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw)}
      Band{S_1→(h,w);S_2→(h,w)}
        Sequence
          Filter{S_1(h,w)}
          Filter{S_2(h,w,kh,kw)}
            Band{S_2→(kh,kw)}
    Filter{S_3(h,w)}
      Band{S_3→(h,w)}
    Filter{S_4(h,w)}
      Band{S_4→(h,w)}
```

```
Domain
  Sequence
    Filter{S_0(h,w)}
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)]
        Sequence
          Filter{S_1(h,w)}
          Filter{S_2(h,w,kh,kw)}
            Band{S_2→(kh,kw)}
          Filter{S_3(h,w)}
          Filter{S_4(h,w)}
```

```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = 0
  for kh in [0,KH), kw in [0,KW):
   C[h,w] += A[h+kh,w+kw]*B[kh,kw]
  C[h,w] = abs(C[h,w])
  C[h,w] = ReLU(C[h,w])
```

- We leverage the ILP-based *isl* scheduler [11, 13] to compute new schedules that exploit parallelism and temporal locality simultaneously.

Versatile Polyhedral Scheduling

```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias       // S_0
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = 0                   // S_1
  for kh in [0,KH), kw in [0,KW):
    C[h,w] += A[h+kh,w+kw]*B[kh,kw] // S
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = abs(C[h,w])         // S_3
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = ReLU(C[h,w])        // S_4
```

```
Domain
  Sequence
    Filter{S_0(h,w)}
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw)}
      Band{S_1→(h,w);S_2→(h,w)}
        Sequence
          Filter{S_1(h,w)}
          Filter{S_2(h,w,kh,kw)}
            Band{S_2→(kh,kw)}
    Filter{S_3(h,w)}
      Band{S_3→(h,w)}
    Filter{S_4(h,w)}
      Band{S_4→(h,w)}
```

```
Domain
  Sequence
    Filter{S_0(h,w)}
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)}
        Sequence
          Filter{S_1(h,w)}
          Filter{S_2(h,w,kh,kw)}
            Band{S_2→(kh,kw)}
          Filter{S_3(h,w)}
          Filter{S_4(h,w)}
```

```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = 0
  for kh in [0,KH), kw in [0,KW):
    C[h,w] += A[h+kh,w+kw]*B[kh,kw]
  C[h,w] = abs(C[h,w])
  C[h,w] = ReLU(C[h,w])
```

- We leverage the ILP-based *isl* scheduler [11, 13] to compute new schedules that exploit parallelism and temporal locality simultaneously.
- The polyhedral scheduler exposes a wider set of affine transformations than TVM, enabling auxiliary loop transformations like skewing, shifting, scaling.

Versatile Polyhedral Scheduling

```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias        // S0
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = 0                    // S1
  for kh in [0,KH), kw in [0,KW):
    C[h,w] += A[h+kh,w+kw]*B[kh,kw] //
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = abs(C[h,w])          // S3
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = ReLU(C[h,w])         // S4
```

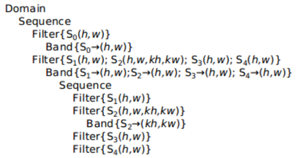
```
Domain
  Sequence
    Filter{S0(h,w)}
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw)}
      Band{S1→(h,w);S2→(h,w)}
        Sequence
          Filter{S1(h,w)}
          Filter{S2(h,w,kh,kw)}
            Band{S2→(kh,kw)}
    Filter{S3(h,w)}
      Band{S3→(h,w)}
    Filter{S4(h,w)}
      Band{S4→(h,w)}
```

```
Domain
  Sequence
    Filter{S0(h,w)}
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}
      Band{S1→(h,w);S2→(h,w); S3→(h,w); S4→(h,w)]
        Sequence
          Filter{S1(h,w)}
          Filter{S2(h,w,kh,kw)}
            Band{S2→(kh,kw)}
          Filter{S3(h,w)}
          Filter{S4(h,w)}
```

```
for h in [0,H), w in [0,W):
  A[h,w] = A[h,w] + bias
for h in [0,H-KH], w in [0,W-KW]:
  C[h,w] = 0
  for kh in [0,KH), kw in [0,KW):
    C[h,w] += A[h+kh,w+kw]*B[kh,kw]
  C[h,w] = abs(C[h,w])
  C[h,w] = ReLU(C[h,w])
```

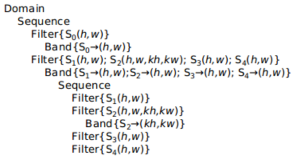
- We leverage the ILP-based *isl* scheduler [11, 13] to compute new schedules that exploit parallelism and temporal locality simultaneously.
- The polyhedral scheduler exposes a wider set of affine transformations than TVM, enabling auxiliary loop transformations like skewing, shifting, scaling.
- The polyhedral model first computes a loop fusion configuration, based on which loop tiling is performed automatically.

Constructing Tile Shapes



```
Domain
   Sequence
      Filter{S_0(h,w)}
         Band{S_0→(h,w)}
      Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
         Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)}
            Sequence
               Filter{S_1(h,w)}
               Filter{S_2(h,w,kh,kw)}
                  Band{S_2→(kh,kw)}
               Filter{S_3(h,w)}
               Filter{S_4(h,w)}
```

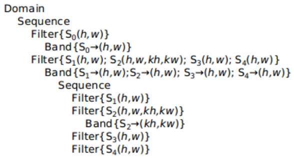
Constructing Tile Shapes



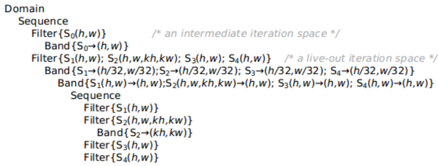
```
Domain
  Sequence
    Filter{$S_0(h,w)$}
      Band{$S_0 \rightarrow (h,w)$}
    Filter{$S_1(h,w)$; $S_2(h,w,kh,kw)$; $S_3(h,w)$; $S_4(h,w)$}
      Band{$S_1 \rightarrow (h,w)$;$S_2 \rightarrow (h,w)$; $S_3 \rightarrow (h,w)$; $S_4 \rightarrow (h,w)$}
        Sequence
          Filter{$S_1(h,w)$}
          Filter{$S_2(h,w,kh,kw)$}
            Band{$S_2 \rightarrow (kh,kw)$}
          Filter{$S_3(h,w)$}
          Filter{$S_4(h,w)$}
```

- The classical polyhedral compilation workflow generates two kernels.

Constructing Tile Shapes



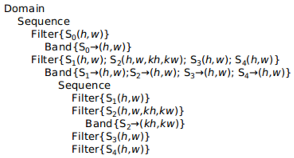
```
Domain
  Sequence
    Filter{S_0(h,w)}
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)}
        Sequence
          Filter{S_1(h,w)}
          Filter{S_2(h,w,kh,kw)}
            Band{S_2→(kh,kw)}
          Filter{S_3(h,w)}
          Filter{S_4(h,w)}
```



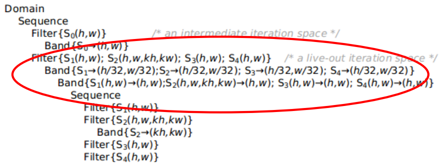
```
Domain
  Sequence
    Filter{S_0(h,w)}          /* an intermediate iteration space */
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}   /* a live-out iteration space */
      Band{S_1→(h/32,w/32);S_2→(h/32,w/32); S_3→(h/32,w/32); S_4→(h/32,w/32)}
        Band{S_1(h,w)→(h,w);S_2(h,w,kh,kw)→(h,w); S_3(h,w)→(h,w); S_4(h,w)→(h,w)}
          Sequence
            Filter{S_1(h,w)}
            Filter{S_2(h,w,kh,kw)}
              Band{S_2→(kh,kw)}
            Filter{S_3(h,w)}
            Filter{S_4(h,w)}
```

- The classical polyhedral compilation workflow generates two kernels.
- We use the reverse strategy proposed in our earlier work [15] to enable the generation of a single kernel.

Constructing Tile Shapes



```
Domain
   Sequence
      Filter{S0(h,w)}
         Band{S0→(h,w)}
      Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}
         Band{S1→(h,w);S2→(h,w); S3→(h,w); S4→(h,w)}
            Sequence
               Filter{S1(h,w)}
               Filter{S2(h,w,kh,kw)}
                  Band{S2→(kh,kw)}
               Filter{S3(h,w)}
               Filter{S4(h,w)}
```



```
Domain
   Sequence
      Filter{S0(h,w)}          /* an intermediate iteration space */
         Band{S0→(h,w)}
      Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}     /* a live-out iteration space */
         Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
            Band{S1(h,w)→(h,w);S2(h,w,kh,kw)→(h,w); S3(h,w)→(h,w); S4(h,w)→(h,w)}
               Sequence
                  Filter{S1(h,w)}
                  Filter{S2(h,w,kh,kw)}
                     Band{S2→(kh,kw)}
                  Filter{S3(h,w)}
                  Filter{S4(h,w)}
```

- The classical polyhedral compilation workflow generates two kernels.
- We use the reverse strategy proposed in our earlier work [15] to enable the generation of a single kernel.
- The reverse strategy first tiles a live-out iteration space,

Constructing Tile Shapes

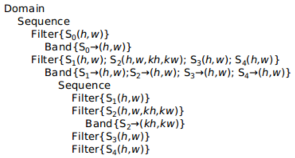
```
Domain
  Sequence
    Filter{S0(h,w)}
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}
      Band{S1→(h,w);S2→(h,w); S3→(h,w); S4→(h,w)}
        Sequence
          Filter{S1(h,w)}
          Filter{S2(h,w,kh,kw)}
            Band{S2→(kh,kw)}
          Filter{S3(h,w)}
          Filter{S4(h,w)}
```

```
Domain
  Sequence
    Filter{S0(h,w)}                /* an intermediate iteration space */
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}    /* a live-out iteration space */
      Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
        Band{S1(h,w)→(h,w);S2(h,w,kh,kw)→(h,w); S3(h,w)→(h,w); S4(h,w)→(h,w)}
          Sequence
            Filter{S1(h,w)}
            Filter{S2(h,w,kh,kw)}
              Band{S2→(kh,kw)}
            Filter{S3(h,w)}
            Filter{S4(h,w)}
```

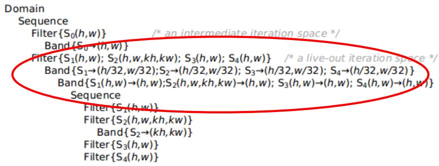
$\{(o_0, o_1) \to A(h', w') : 0 \le o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \le o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \le h' < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \le w' < T_3 \cdot o_1 + KW + T_3 - 1\}$

- The classical polyhedral compilation workflow generates two kernels.
- We use the reverse strategy proposed in our earlier work [15] to enable the generation of a single kernel.
- The reverse strategy first tiles a live-out iteration space,

Constructing Tile Shapes



```
Domain
  Sequence
    Filter{S0(h,w)}
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}
      Band{S1→(h,w);S2→(h,w); S3→(h,w); S4→(h,w)}
        Sequence
          Filter{S1(h,w)}
          Filter{S2(h,w,kh,kw)}
            Band{S2→(kh,kw)}
          Filter{S3(h,w)}
          Filter{S4(h,w)}
```



```
Domain
  Sequence
    Filter{S0(h,w)}              /* an intermediate iteration space */
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}    /* a live-out iteration space */
      Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
        Band{S1(h,w)→(h,w);S2(h,w,kh,kw)→(h,w); S3(h,w)→(h,w); S4(h,w)→(h,w)}
          Sequence
            Filter{S1(h,w)}
            Filter{S2(h,w,kh,kw)}
              Band{S2→(kh,kw)}
            Filter{S3(h,w)}
            Filter{S4(h,w)}
```

$\{(o_0, o_1) \to A(h', w') : 0 \le o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \le o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \le h' < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \le w' < T_3 \cdot o_1 + KW + T_3 - 1\}$

$\{(o_0, o_1) \to S_0(h, w) : 0 \le o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \le o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \le h < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \le w < T_3 \cdot o_1 + KW + T_3 - 1\}$

- The classical polyhedral compilation workflow generates two kernels.
- We use the reverse strategy proposed in our earlier work [15] to enable the generation of a single kernel.
- The reverse strategy first tiles a live-out iteration space, and uses the data tiles to construct tile shapes for intermediate iteration spaces.

Specifying Tile Sizes

- Prior tensor compilers use default tile sizes in compilers.
- We propose a tile-size specification language.

```
stmt_id :: "S_" integer
tile_size :: integer
tile_spec :: tile_size @ buffer
tile_specs :: tile_spec | tile_specs , tile_spec
stmt_spec :: stmt_id : tile_specs
tiling_policy :: stmt_spec | tiling_policy stmt_spec
```

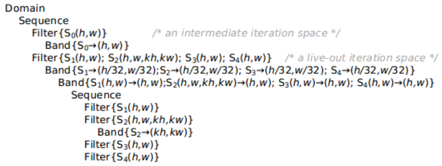
Specifying Tile Sizes

- Prior tensor compilers use default tile sizes in compilers.
- We propose a tile-size specification language.

```
stmt_id :: "S_" integer
tile_size :: integer
tile_spec :: tile_size @ buffer
tile_specs :: tile_spec | tile_specs, tile_spec
stmt_spec :: stmt_id : tile_specs
tiling_policy :: stmt_spec | tiling_policy stmt_spec
```

- This language simplifies the tile size selection issue, which has been automated by compiler.

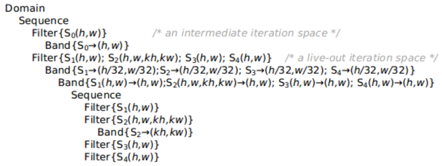
Fusion When Offloading Data



```
Domain
  Sequence
    Filter{S0(h,w)}          /* an intermediate iteration space */
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}    /* a live-out iteration space */
      Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
        Band{S1(h,w)→(h,w);S2(h,w,kh,kw)→(h,w); S3(h,w)→(h,w); S4(h,w)→(h,w)}
          Sequence
            Filter{S1(h,w)}
            Filter{S2(h,w,kh,kw)}
              Band{S2→(kh,kw)}
            Filter{S3(h,w)}
            Filter{S4(h,w)}
```

$$\{(o_0, o_1) \to S_0(h, w) : 0 \le o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \le o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \le h < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \le w < T_3 \cdot o_1 + KW + T_3 - 1\}$$

Fusion When Offloading Data

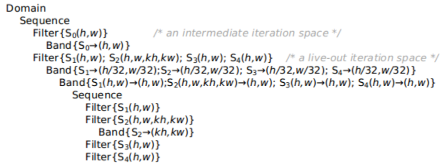


```
Domain
   Sequence
      Filter{S0(h,w)}            /* an intermediate iteration space */
         Band{S0→(h,w)}
      Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}    /* a live-out iteration space */
         Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
            Band{S1(h,w)→(h,w);S2(h,w,kh,kw)→(h,w); S3(h,w)→(h,w); S4(h,w)→(h,w)}
               Sequence
                  Filter{S1(h,w)}
                  Filter{S2(h,w,kh,kw)}
                     Band{S2→(kh,kw)}
                  Filter{S3(h,w)}
                  Filter{S4(h,w)}
```

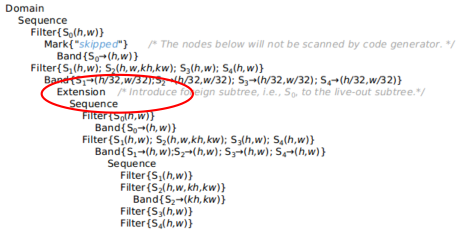
$$\{(o_0, o_1) \to S_0(h, w) : 0 \le o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \le o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \le h < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \le w < T_3 \cdot o_1 + KW + T_3 - 1\}$$

- This relation implies the *overlapped* tile shape [14] of the intermediate iteration space, but it has to be used together with loop fusion.

Fusion When Offloading Data



```
Domain
  Sequence
    Filter{S_0(h,w)}          /* an intermediate iteration space */
      Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}    /* a live-out iteration space */
      Band{S_1→(h/32,w/32);S_2→(h/32,w/32); S_3→(h/32,w/32); S_4→(h/32,w/32)}
        Band{S_1(h,w)→(h,w);S_2(h,w,kh,kw)→(h,w); S_3(h,w)→(h,w); S_4(h,w)→(h,w)}
          Sequence
            Filter{S_1(h,w)}
            Filter{S_2(h,w,kh,kw)}
              Band{S_2→(kh,kw)}
            Filter{S_3(h,w)}
            Filter{S_4(h,w)}
```

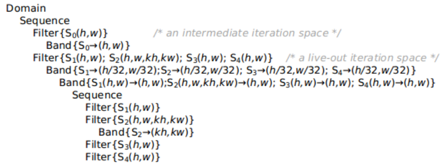


```
Domain
  Sequence
    Filter{S_0(h,w)}
      Mark{"skipped"}      /* The nodes below will not be scanned by code generator. */
        Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h/32,w/32);S_2→(h/32,w/32); S_3→(h/32,w/32); S_4→(h/32,w/32)}
        Extension    /* Introduce foreign subtree, i.e., S_0, to the live-out subtree.*/
          Sequence
            Filter{S_0(h,w)}
              Band{S_0→(h,w)}
            Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
              Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)}
                Sequence
                  Filter{S_1(h,w)}
                  Filter{S_2(h,w,kh,kw)}
                    Band{S_2→(kh,kw)}
                  Filter{S_3(h,w)}
                  Filter{S_4(h,w)}
```

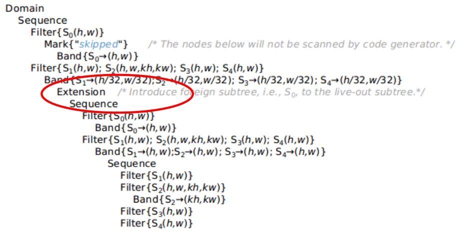
$$\{(o_0, o_1) \rightarrow S_0(h, w) : 0 \leq o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \leq o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \leq h < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \leq w < T_3 \cdot o_1 + KW + T_3 - 1\}$$

- This relation implies the *overlapped* tile shape [14] of the intermediate iteration space, but it has to be used together with loop fusion.

Fusion When Offloading Data



```
Domain
  Sequence
    Filter{S0(h,w)}          /* an intermediate iteration space */
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}    /* a live-out iteration space */
      Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
        Band{S1(h,w)→(h,w);S2(h,w,kh,kw)→(h,w); S3(h,w)→(h,w); S4(h,w)→(h,w)}
          Sequence
            Filter{S1(h,w)}
            Filter{S2(h,w,kh,kw)}
              Band{S2→(kh,kw)}
            Filter{S3(h,w)}
            Filter{S4(h,w)}
```

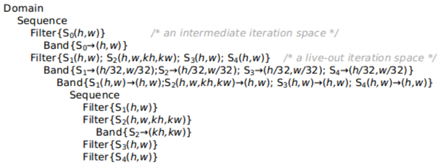


```
Domain
  Sequence
    Filter{S0(h,w)}
      Mark{"skipped"}      /* The nodes below will not be scanned by code generator. */
        Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}
      Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
        Extension    /* Introduce foreign subtree, i.e., S0, to the live-out subtree.*/
          Sequence
            Filter{S0(h,w)}
              Band{S0→(h,w)}
            Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}
              Band{S1→(h,w);S2→(h,w); S3→(h,w); S4→(h,w)}
                Sequence
                  Filter{S1(h,w)}
                  Filter{S2(h,w,kh,kw)}
                    Band{S2→(kh,kw)}
                  Filter{S3(h,w)}
                  Filter{S4(h,w)}
```

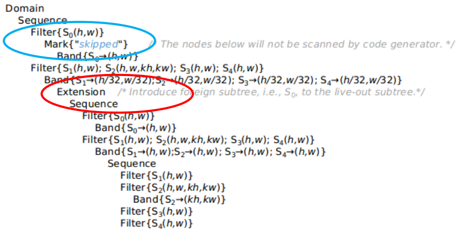
$$\{(o_0, o_1) \to S_0(h, w) : 0 \le o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \le o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \le h < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \le w < T_3 \cdot o_1 + KW + T_3 - 1\}$$

- This relation implies the *overlapped* tile shape [14] of the intermediate iteration space, but it has to be used together with loop fusion.
- The post-tiling fusion strategy models a novel composition of loop transformations.

Fusion When Offloading Data



```
Domain
  Sequence
    Filter{S0(h,w)}           /* an intermediate iteration space */
      Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}    /* a live-out iteration space */
      Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
        Band{S1(h,w)→(h,w);S2(h,w,kh,kw)→(h,w); S3(h,w)→(h,w); S4(h,w)→(h,w)}
          Sequence
            Filter{S1(h,w)}
            Filter{S2(h,w,kh,kw)}
              Band{S2→(kh,kw)}
            Filter{S3(h,w)}
            Filter{S4(h,w)}
```

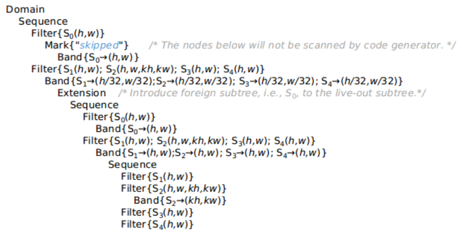


```
Domain
  Sequence
    Filter{S0(h,w)}
      Mark{"skipped"}        /* The nodes below will not be scanned by code generator. */
        Band{S0→(h,w)}
    Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}
      Band{S1→(h/32,w/32);S2→(h/32,w/32); S3→(h/32,w/32); S4→(h/32,w/32)}
        Extension    /* Introduce foreign subtree, i.e., S0, to the live-out subtree.*/
          Sequence
            Filter{S0(h,w)}
              Band{S0→(h,w)}
            Filter{S1(h,w); S2(h,w,kh,kw); S3(h,w); S4(h,w)}
              Band{S1→(h,w);S2→(h,w); S3→(h,w); S4→(h,w)}
                Sequence
                  Filter{S1(h,w)}
                  Filter{S2(h,w,kh,kw)}
                    Band{S2→(kh,kw)}
                  Filter{S3(h,w)}
                  Filter{S4(h,w)}
```

$$\{(o_0, o_1) \rightarrow S_0(h, w) : 0 \leq o_0 < \lceil (H - KH + 1)/T_2 \rceil \wedge 0 \leq o_1 < \lceil (W - KW + 1)/T_3 \rceil \wedge T_2 \cdot o_0 \leq h < T_2 \cdot o_0 + KH + T_2 - 1 \wedge T_3 \cdot o_1 \leq w < T_3 \cdot o_1 + KW + T_3 - 1\}$$

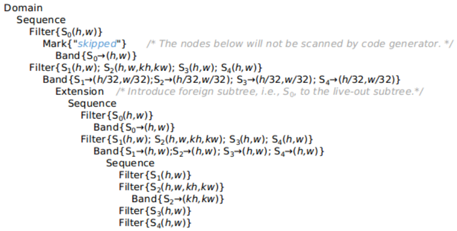
- This relation implies the *overlapped* tile shape [14] of the intermediate iteration space, but it has to be used together with loop fusion.
- The post-tiling fusion strategy models a novel composition of loop transformations.
- The original subtree should be skipped.

Fusion When Forking Data and Intra-Tile Rescheduling



```
Domain
  Sequence
    Filter{S_0(h,w)}
      Mark{"skipped"}     /* The nodes below will not be scanned by code generator. */
        Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h/32,w/32);S_2→(h/32,w/32); S_3→(h/32,w/32); S_4→(h/32,w/32)}
        Extension   /* Introduce foreign subtree, i.e., S_0, to the live-out subtree.*/
          Sequence
            Filter{S_0(h,w)}
              Band{S_0→(h,w)}
            Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
              Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)}
                Sequence
                  Filter{S_1(h,w)}
                  Filter{S_2(h,w,kh,kw)}
                    Band{S_2→(kh,kw)}
                  Filter{S_3(h,w)}
                  Filter{S_4(h,w)}
```

Fusion When Forking Data and Intra-Tile Rescheduling

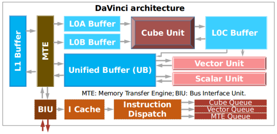


```
Domain
  Sequence
    Filter{S_0(h,w)}
      Mark{"skipped"}     /* The nodes below will not be scanned by code generator. */
        Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h/32,w/32);S_2→(h/32,w/32); S_3→(h/32,w/32); S_4→(h/32,w/32)}
        Extension   /* Introduce foreign subtree, i.e., S_0, to the live-out subtree.*/
          Sequence
            Filter{S_0(h,w)}
              Band{S_0→(h,w)}
            Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
              Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)}
                Sequence
                  Filter{S_1(h,w)}
                  Filter{S_2(h,w,kh,kw)}
                    Band{S_2→(kh,kw)}
                  Filter{S_3(h,w)}
                  Filter{S_4(h,w)}
```

- This schedule tree does not manage the multi-directional memory hierarchy of Ascend.

Fusion When Forking Data and Intra-Tile Rescheduling

```
Domain
  Sequence
    Filter{$S_0(h,w)$}
      Mark{"skipped"}    /* The nodes below will not be scanned by code generator. */
        Band{$S_0 \to (h,w)$}
    Filter{$S_1(h,w)$; $S_2(h,w,kh,kw)$; $S_3(h,w)$; $S_4(h,w)$}
      Band{$S_1 \to (h/32,w/32)$;$S_2 \to (h/32,w/32)$; $S_3 \to (h/32,w/32)$; $S_4 \to (h/32,w/32)$}
        Extension   /* Introduce foreign subtree, i.e., $S_0$, to the live-out subtree.*/
          Sequence
            Filter{$S_0(h,w)$}
              Band{$S_0 \to (h,w)$}
            Filter{$S_1(h,w)$; $S_2(h,w,kh,kw)$; $S_3(h,w)$; $S_4(h,w)$}
              Band{$S_1 \to (h,w)$;$S_2 \to (h,w)$; $S_3 \to (h,w)$; $S_4 \to (h,w)$}
                Sequence
                  Filter{$S_1(h,w)$}
                  Filter{$S_2(h,w,kh,kw)$}
                    Band{$S_2 \to (kh,kw)$}
                  Filter{$S_3(h,w)$}
                  Filter{$S_4(h,w)$}
```

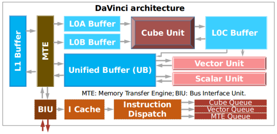


```
Domain
  Sequence
    Skipped Filter{$S_0(h,w)$}
    Filter{$S_1(h,w)$; $S_2(h,w,kh,kw)$; $S_3(h,w)$; $S_4(h,w)$}
      Band{$S_1 \to (h/32,w/32)$;$S_2 \to (h/32,w/32)$; $S_3 \to (h/32,w/32)$; $S_4 \to (h/32,w/32)$}
        Extension
          Sequence
            Filter{$S_0(h,w)$}
              Mark{"local_UB"}
                Band{$S_0 \to (h,w)$}
            Filter{$S_1(h,w)$; $S_2(h,w,kh,kw)$}
              Band{$S_1 \to (h,w)$;$S_2 \to (h,w)$}
                Sequence
                  Filter{$S_1(h,w)$}
                  Filter{$S_2(h,w,kh,kw)$}
                    Band{$S_2 \to (kh,kw)$}
            Filter{$S_3(h,w)$}
              Mark{"local_UB"}
                Band{$S_3 \to (h,w)$}
            Filter{$S_4(h,w)$}
              Mark{"local_UB"}
                Band{$S_4 \to (h,w)$}
```

- This schedule tree does not manage the multi-directional memory hierarchy of Ascend.
- We use mark nodes to let some statements flow to different buffers, and each "local_UB" filter node can be flowed to Vector/Scalar Unit.

Fusion When Forking Data and Intra-Tile Rescheduling

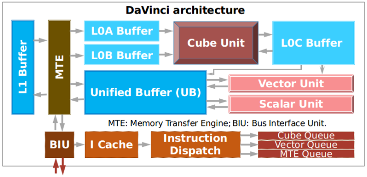
```
Domain
  Sequence
    Filter{S_0(h,w)}
      Mark{"skipped"}    /* The nodes below will not be scanned by code generator. */
        Band{S_0→(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h/32,w/32);S_2→(h/32,w/32); S_3→(h/32,w/32); S_4→(h/32,w/32)}
        Extension    /* Introduce foreign subtree, i.e., S_0, to the live-out subtree.*/
          Sequence
            Filter{S_0(h,w)}
              Band{S_0→(h,w)}
            Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
              Band{S_1→(h,w);S_2→(h,w); S_3→(h,w); S_4→(h,w)}
                Sequence
                  Filter{S_1(h,w)}
                  Filter{S_2(h,w,kh,kw)}
                    Band{S_2→(kh,kw)}
                  Filter{S_3(h,w)}
                  Filter{S_4(h,w)}
```



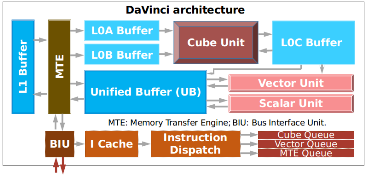
```
Domain
  Sequence
    Skipped Filter{S_0(h,w)}
    Filter{S_1(h,w); S_2(h,w,kh,kw); S_3(h,w); S_4(h,w)}
      Band{S_1→(h/32,w/32);S_2→(h/32,w/32); S_3→(h/32,w/32); S_4→(h/32,w/32)}
        Extension
          Sequence
            Filter{S_0(h,w)}
              Mark{"local_UB"}
                Band{S_0→(h,w)}
            Filter{S_1(h,w); S_2(h,w,kh,kw)}
              Band{S_1→(h,w);S_2→(h,w)}
                Sequence
                  Filter{S_1(h,w)}
                  Filter{S_2(h,w,kh,kw)}
                    Band{S_2→(kh,kw)}
            Filter{S_3(h,w)}
              Mark{"local_UB"}
                Band{S_3→(h,w)}
            Filter{S_4(h,w)}
              Mark{"local_UB"}
                Band{S_4→(h,w)}
```

- This schedule tree does not manage the multi-directional memory hierarchy of Ascend.
- We use mark nodes to let some statements flow to different buffers, and each “local_UB” filter node can be flowed to Vector/Scalar Unit.
- Intra-tile rescheduling is also performed, as a reverse process of loop fusion. A filter flowed to Cube Unit is not distributed.

Optimization of Convolution

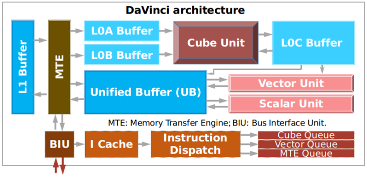


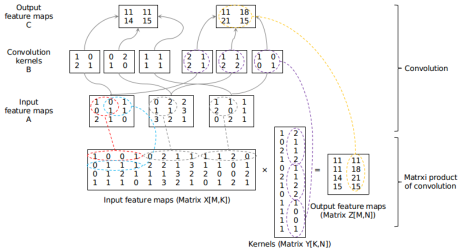
Optimization of Convolution



- The power of the Cube Unit can be fully exploited when executing matrix multiplication.

Optimization of Convolution

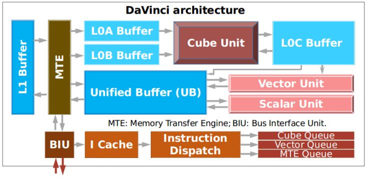


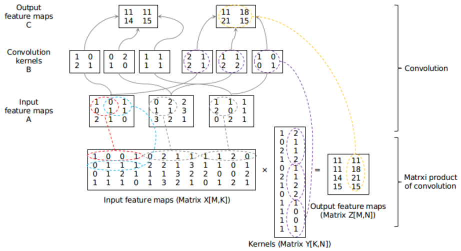


Output feature maps C
Convolution kernels B
Input feature maps A
Convolution
Input feature maps (Matrix X[M,K])
Kernels (Matrix Y[K,N])
Output feature maps (Matrix Z[M,N])
Matrxi product of convolution

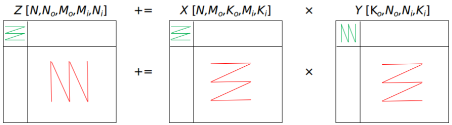
- The power of the Cube Unit can be fully exploited when executing matrix multiplication.
- We automate the *img2col* transformation [5] by grafting an external schedule and relating it using a formula (§4.5).

Optimization of Convolution




Output feature maps C
Convolution kernels B
Input feature maps A
Convolution
Input feature maps (Matrix X[M,K])
Kernels (Matrix Y[K,N])
Output feature maps (Matrix Z[M,N])
Matrxi product of convolution

- The power of the Cube Unit can be fully exploited when executing matrix multiplication.
- We automate the *img2col* transformation [5] by grafting an external schedule and relating it using a formula (§4.5).
- We also implement a fractal tiling [16] within the Cube Unit.



Other Optimizations in AKG

- Optimizations, including function inlining, common subexpression elimination, etc. are also automated as pre-processing steps (§3).

Other Optimizations in AKG

- Optimizations, including function inlining, common subexpression elimination, etc. are also automated as pre-processing steps (§3).
- We facilitate the automatic storage management of the Ascend chips using schedule trees (§4.4), like what PPCG [12] and TC [10] did.

Other Optimizations in AKG

- Optimizations, including function inlining, common subexpression elimination, etc. are also automated as pre-processing steps (§3).
- We facilitate the automatic storage management of the Ascend chips using schedule trees (§4.4), like what PPCG [12] and TC [10] did.
- We design a memory hierarchy specification language that can be generated automatically, allowing for the manual scheduling to make debugging easier (§4.6).

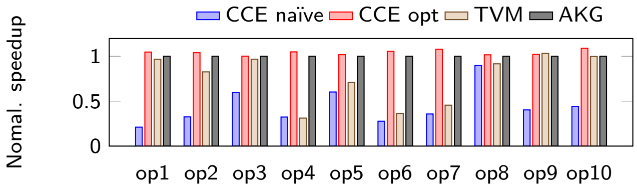
Other Optimizations in AKG

- Optimizations, including function inlining, common subexpression elimination, etc. are also automated as pre-processing steps (§3).
- We facilitate the automatic storage management of the Ascend chips using schedule trees (§4.4), like what PPCG [12] and TC [10] did.
- We design a memory hierarchy specification language that can be generated automatically, allowing for the manual scheduling to make debugging easier (§4.6).
- We exploit effective SIMD vectorization as a post-polyhedral step, maximizing the utilization of the hardware intrinsics (§5.1).
- We implement a DP-based low-level synchronization between emitted instructions, enabling efficient instruction-level pipelining (§5.2).
- We develop an auto tuning strategy to achieve better performance in practice (§5.3).

Experimental Setup

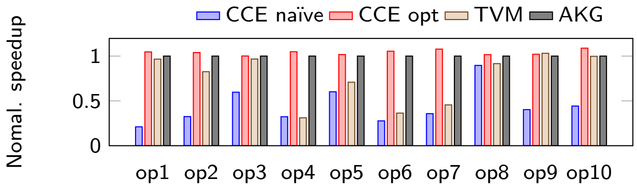
- Code is executed on the Huawei Ascend 910 chip.
- Performance is compared against (1) manually optimized CCE code written by experts, and (2) the adapted TVM schedule templates developed by the software R&D team of the chip.
- Experiment is conducted on single operators, subgraphs and end-to-end workloads.
- Each code is compiled with the same set of compilation options.

Results of Single Operators



op1: conv; op2: matmul; op3: ReLU; op4: batch matmul; op5: cast; op6: transpose; op7: one-hot; op8: add; op9: bnorm reduction; op10: bnorm update

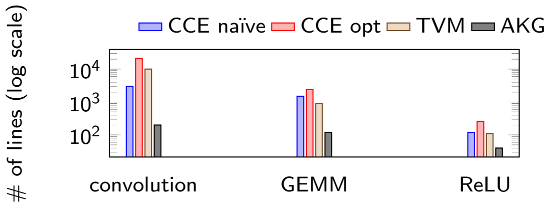
Results of Single Operators



op1: conv; op2: matmul; op3: ReLU; op4: batch matmul; op5: cast; op6: transpose; op7: one-hot; op8: add; op9: bnorm reduction; op10: bnorm update

- CCE opt is 2.8× faster than CCE naïve.
- AKG achieves the performance comparable to CCE opt, with a mean loss within 4%.
- AKG outperforms adapted TVM by 1.6× on average.

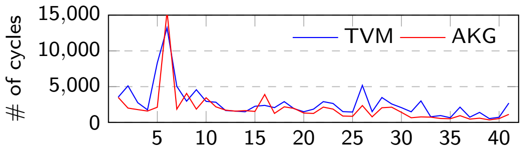
Results of Single Operators



Comparison of lines of code (lower is better).

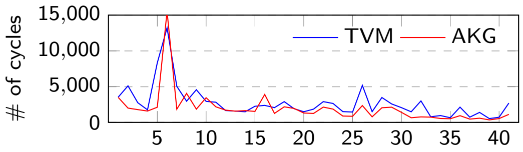
- AKG significantly reduces development efforts compared to the optimized CCE code and adapted TVM schedule templates.

Results of Single Operators



Performance of GEMM product under different shape configurations (1 μs $= 10^3$ cycles; lower is better).

Results of Single Operators



Performance of GEMM product under different shape configurations (1 μs $= 10^3$ cycles; lower is better).

- 41 different shape configurations ranging from (64,64) to (4608,4608) are used to evaluate the performance of matrix multiplication.
- AKG outperforms the adapted TVM under 29 out of the 41 shape configurations.

Results of Subgraphs

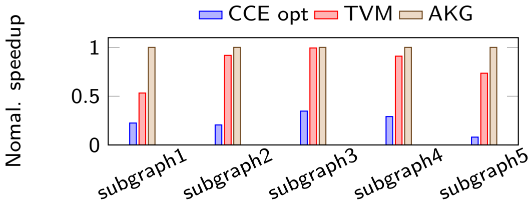
Summary of the subgraphs.

no.	# of ops	precision	batch size	input shape	output shape
1	6	FP16	16	(16,16,512,512)	(16,16,512,512)
2	21	FP16	16	(256,512,16,16)	(256,512,16,16)
3	15	FP32	16	(30522,1024)	(30522,1024)
4	11	FP32	16	(1024,1024)	(1024,1024)
5	9	FP16	16	(64,1,16,16)	(64,1,16,16)

Results of Subgraphs

Summary of the subgraphs.

no.	# of ops	precision	batch size	input shape	output shape
1	6	FP16	16	(16,16,512,512)	(16,16,512,512)
2	21	FP16	16	(256,512,16,16)	(256,512,16,16)
3	15	FP32	16	(30522,1024)	(30522,1024)
4	11	FP32	16	(1024,1024)	(1024,1024)
5	9	FP16	16	(64,1,16,16)	(64,1,16,16)

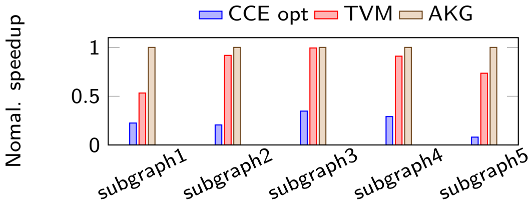


Performance of subgraphs (higher is better).

Results of Subgraphs

Summary of the subgraphs.

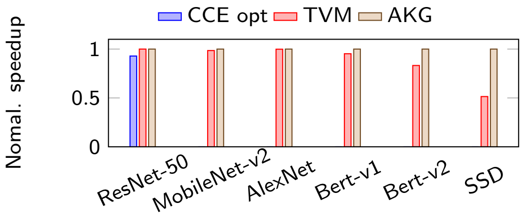
no.	# of ops	precision	batch size	input shape	output shape
1	6	FP16	16	(16,16,512,512)	(16,16,512,512)
2	21	FP16	16	(256,512,16,16)	(256,512,16,16)
3	15	FP32	16	(30522,1024)	(30522,1024)
4	11	FP32	16	(1024,1024)	(1024,1024)
5	9	FP16	16	(64,1,16,16)	(64,1,16,16)



Performance of subgraphs (higher is better).

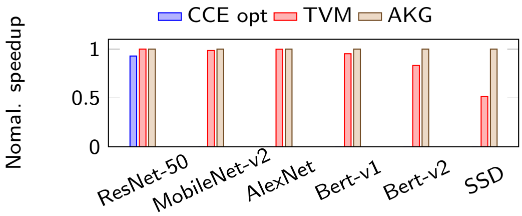
- AKG produces an average speedup of 1.3× and 5.6× over the adapted TVM and CCE opt.

Results of End-to-end Workloads



Performance of end-to-end workloads (higher is better).

Results of End-to-end Workloads



Performance of end-to-end workloads (higher is better).

- CCE opt only optimizes one end-to-end workload (ResNet-50).
- AKG performs similarly to the adapted TVM for ResNet-50, MobileNet and AlexNet, but outperforms the latter by 20.2% on Bert and SSD.
- The manual approaches take days to weeks to optimize a workload, but AKG only requires minutes to hours.

Conclusion

- AKG carefully handles the interplay between tiling and fusion using a reverse strategy [15], a patform-neutral transformation.
- AKG adopts a hierarchical fusion approach that can be adapted to other NPU architectures [6].
- AKG automates the domain-specific transformations of convolution. While the fractal tiling [16] is Ascend-specific, the *img2col* transformation [5] can be used as a general method.
- AKG also extends the expressiveness of the schedule tree representation, sharing the same objective (i.e., delivering domain-specific knowledge) with MLIR [7].

Questions & Answers

The paper is avalaible at

The code of AKG is avalaible at

References

[1] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P.
A practical automatic polyhedral parallelizer and locality optimizer.
In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI'08, ACM, pp. 101–113.

[2] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A.
Tvm: An automated end-to-end optimizing compiler for deep learning.
In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2018), OSDI'18, USENIX Association, pp. 579–594.

[3] Grosser, T., Groesslinger, A., and Lengauer, C.
Polly–performing polyhedral optimizations on a low-level intermediate representation.
Parallel Processing Letters 22, 04 (2012), 1250010.

[4] Grosser, T., Verdoolaege, S., and Cohen, A.
Polyhedral ast generation is more than scanning polyhedra.
ACM Trans. Program. Lang. Syst. 37, 4 (July 2015), 12:1–12:50.

[5] Gu, J., Liu, Y., Gao, Y., and Zhu, M.
Opencl caffe: Accelerating and enabling a cross platform machine learning framework.
In *Proceedings of the 4th International Workshop on OpenCL* (New York, NY, USA, 2016), IWOCL'16, ACM.

References

[6] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H.
In-datacenter performance analysis of a tensor processing unit.
In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA'17, ACM, pp. 1–12.

[7] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O.
Mlir: Scaling compiler infrastructure for domain specific computation.
In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2021), pp. 2–14.

[8] Liao, H., Tu, J., Xia, J., and Zhou, X.
Davinci: A scalable architecture for neural network computing.
In *2019 IEEE Hot Chips 31 Symposium (HCS)* (2019), IEEE, pp. 1–44.

[9] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S.
Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.
In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI'13, ACM, pp. 519–530.

References

[10] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., Devito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A.
The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically.
ACM Trans. Archit. Code Optim. 16, 4 (Oct. 2019).

[11] Verdoolaege, S.
Isl: An integer set library for the polyhedral model.
In *Proceedings of the Third International Congress Conference on Mathematical Software* (Berlin, Heidelberg, 2010), ICMS'10, Springer-Verlag, pp. 299–302.

[12] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F.
Polyhedral parallel code generation for cuda.
ACM Trans. Archit. Code Optim. 9, 4 (Jan. 2013), 54:1–54:23.

[13] Verdoolaege, S., and Janssens, G.
Scheduling for ppcg.
Report CW 706 (2017).

[14] Zhao, J., and Cohen, A.
Flextended tiles: A flexible extension of overlapped tiles for polyhedral compilation.
ACM Trans. Archit. Code Optim. 16, 4 (Dec. 2019).

[15] Zhao, J., and Di, P.
Optimizing the memory hierarchy by compositing automatic transformations on computations and data.
In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture* (Piscataway, NJ, USA, 2020), MICRO-53, IEEE Press, pp. 427–441.

[16] Zhao, Y., Du, Z., Guo, Q., Liu, S., Li, L., Xu, Z., Chen, T., and Chen, Y.
Cambricon-f: Machine learning computers with fractal von neumann architecture.
In *Proceedings of the 46th International Symposium on Computer Architecture* (New York, NY, USA, 2019), ISCA'19, ACM, pp. 788–801.