HIVM: MLIR Dialect Stack For Huawei Atlas NPU Compilation

Huawei – 2012 Labs – CSI

- Tarasov Vladislav - Huawei
- Hugo Trachino - Huawei
- Bokhanko Andrey - Huawei
- Gribov Yuri - Huawei
- Suslov Pavel - Huawei
- Gaskov Vladimir - Huawei

Date: 2026.04.15

Security Level: Open-Source

Contents

1. Target Platform – Atlas at a glance
2. Vertical stack overview: MLIR slice
3. Down the pipeline: one kernel story

Target Platform – Atlas

Target Platform – Atlas

- Designed for AI from the ground up

Target Platform - Atlas

- Designed for AI from the ground up
- Half Of The World runs on Atlas

Target Platform - Atlas

- Designed for AI from the ground up
- Half Of The World runs on Atlas
  > Alibaba
  > Huawei
  > Tencent
  > ByteDance

Target Platform – Atlas

- Designed for AI from the ground up
- Half Of The World runs on Atlas
  - > Alibaba
  - > Huawei
  - > Tencent
  - > ByteDance
- 910A – 2019
- 910 – 2022
- 910 – 2024
- 950 – New Frontier

Target Platform – Atlas

Atlas 910A (2019)

- 256 TFLOPS

Target Platform – Atlas

Atlas 910A (2019)

- 256 TFLOPS

Atlas 910 (2022)

- 320 TFLOPS

Target Platform – Atlas

Atlas 910A (2019)

- 256 TFLOPS

Atlas 910 (2022)

- 320 TFLOPS

Atlas 910 (2024)

- 800 TFLOPS

Target Platform – Atlas

Atlas 910A (2019)
- 256 TFLOPS

Atlas 910 (2022)
- 320 TFLOPS

Atlas 910 (2024)
- 800 TFLOPS

Atlas 950 (2026)
- X PFLOPS

 Huawei Proprietary - Restricted Distribution

Target Platform – Atlas

Atlas 950 (2026)

- X PFLOPS
- Cube + Vector

GM
L2
AIC
MTE2
MTE3
L1
MTE1
L0A
MTE1
L0B
MTE1
BT
FixPipe
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

Target Platform – Atlas

Atlas 950 (2026)

- X PFLOPS
- Cube + Vector
- 2 TB/s Interconnect
- SIMD + SIMT

Target Platform – Atlas

Atlas 950 (2026)

- X PFLOPS
- Cube + Vector
- 2 TB/s Interconnect
- SIMD + SIMT
- 144 GB HBM
- ZFLOP level (cluster)

GM
L2
AIC
L1
MTE2
MTE3
MTE1
L0A
L0B
BT
FixPipe
FP
Scalar1
CUBE
L0C
AIV x 2
UB
VEC
Scalar2

Target Platform – Atlas

Ready for the real world

FP32 / HF32 / FP16 / BF16 /

FP8 / MXFP8 / HiF8 / MXFP4

Atlas 950 (2026)

- X PFLOPS
- Cube + Vector
- 2 TB/s Interconnect
- SIMD + SIMT
- 144 GB HBM
- ZFLOP level (cluster)

GM
L2
AIC
MTE2
MTE3
L1
MTE1
L0A
MTE1
L0B
MTE1
BT
FixPipe
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

Target Platform – Atlas 950

MMAD step-by-step

GM
L2
AIC
L1
MTE2
MTE3
MTE1
L0A
L0B
BT
FP
FixPipe
Scalar1
CUBE
L0C
AIV x 2
UB
VEC
Scalar2

Target Platform – Atlas 950

MMAD step-by-step

GM
L2
AIC
L1
MTE2
MTE3
MTE1
MTE1
MTE1
FixPipe
L0A
L0B
BT
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

Target Platform – Atlas 950

MMAD step-by-step

Huawei Proprietary - Restricted Distribution

Target Platform – Atlas 950

MMAD step-by-step

GM
L2
AIC
MTE2
MTE3
L1
MTE1
L0A
MTE1
L0B
MTE1
BT
FixPipe
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

Target Platform – Atlas 950

MMAD step-by-step

GM
L2
AIC
MTE2
MTE3
L1
MTE1
L0A
MTE1
L0B
MTE1
BT
FixPipe
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

Huawei Proprietary - Restricted Distribution

Target Platform – Atlas 950

MMAD step-by-step

GM

L2

AIC

L1

MTE2

MTE3

MTE1

FixPipe

L0A

L0B

BT

FP

Scalar1

CUBE

L0C

FixPipe

FixPipe

MTE1

MTE3

AIV x 2

MTE2

MTE3

UB

VEC

Scalar2

Huawei Proprietary - Restricted Distribution

Target Platform – Atlas 950

MMAD step-by-step

AIC
GM
L2
MTE2
MTE3
L1
MTE1
MTE1
MTE1
FixPipe
L0A
L0B
BT
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

Huawei Proprietary - Restricted Distribution

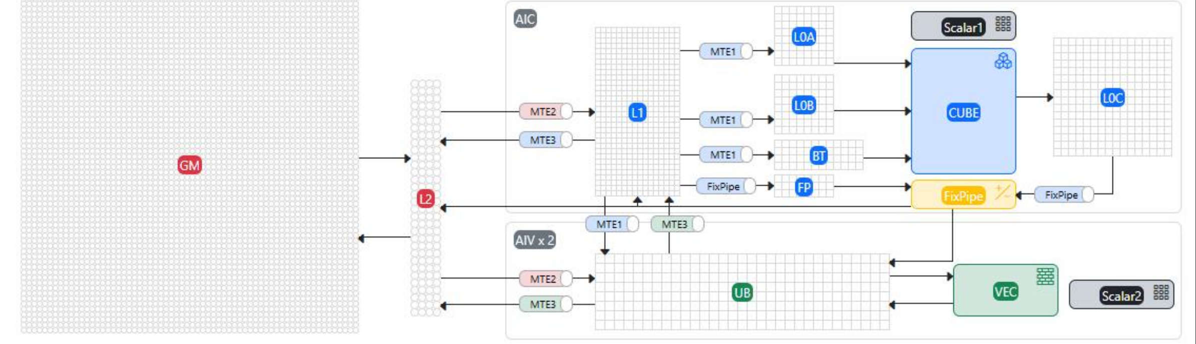
Target Platform – Atlas 950

MMAD step-by-step

GM
L2
AIC
MTE2
MTE3
L1
MTE1
L0A
MTE1
L0B
MTE1
BT
FixPipe
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

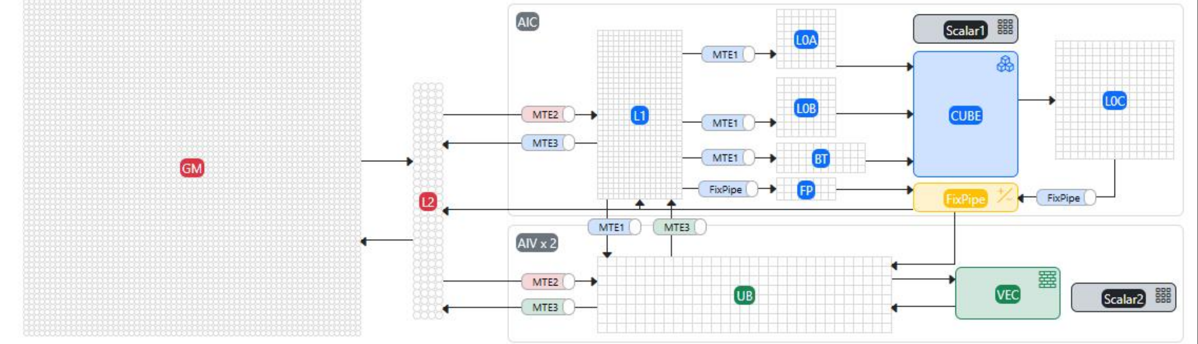
Huawei Proprietary - Restricted Distribution

Target Platform – Atlas 950



From a developer perspective

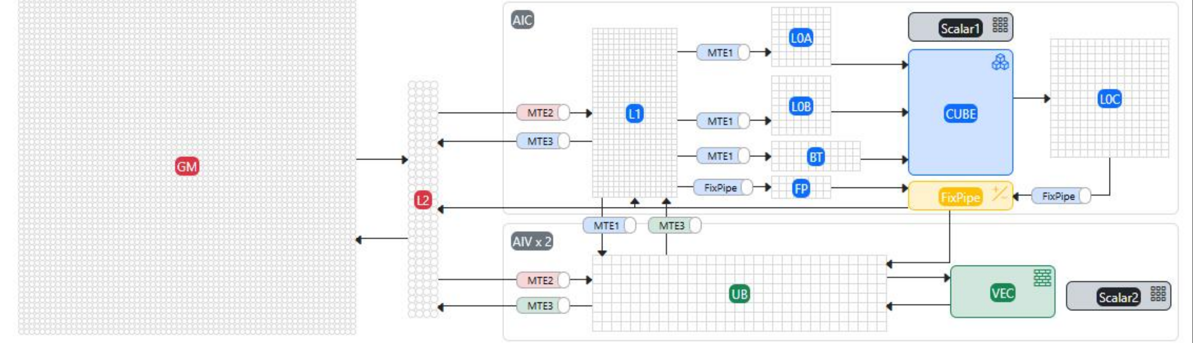
Target Platform – Atlas 950



From a developer perspective

- Many functional units

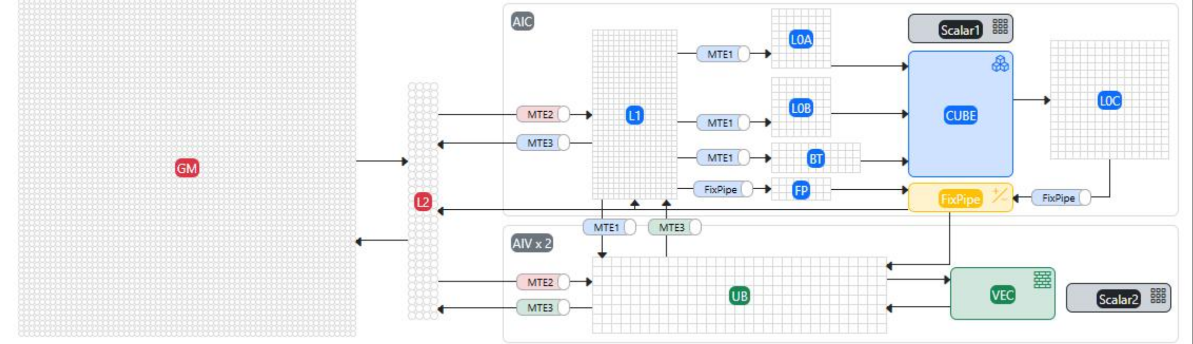
Target Platform – Atlas 950



From a developer perspective

- Many functional units
  - Cube – tensor core

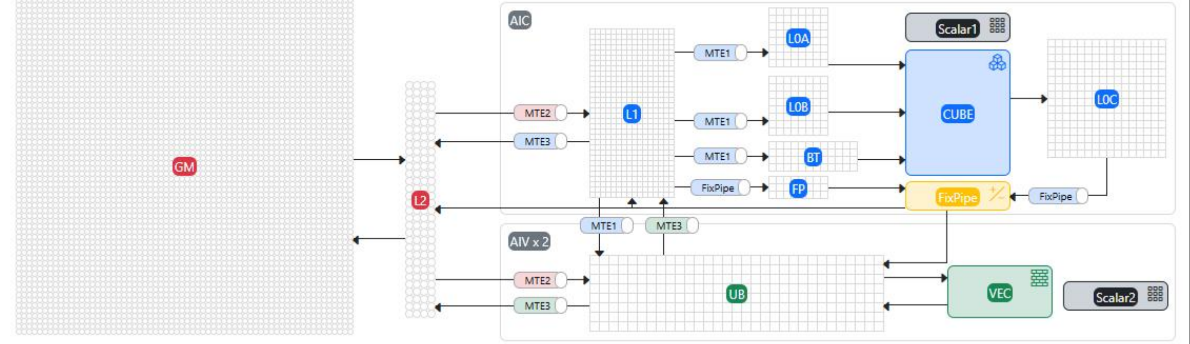
Target Platform – Atlas 950



From a developer perspective

- Many functional units
  - Cube – tensor core
  - Vector – vector operations

Target Platform – Atlas 950

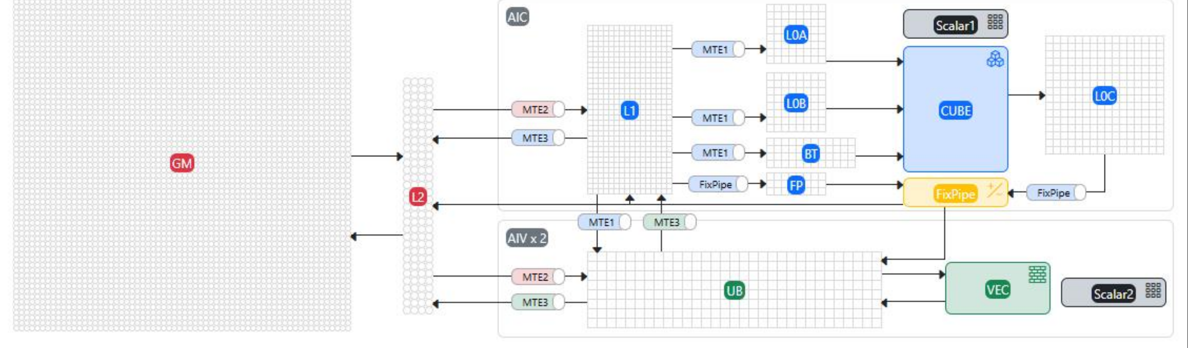


From a developer perspective

- Many functional units
  - Cube – tensor core
  - Vector – vector operations
  - Scalar – issue, control, dispatch

Huawei Proprietary - Restricted Distribution

Target Platform – Atlas 950



From a developer perspective

- Many functional units
  - Cube – tensor core
  - Vector – vector operations
  - Scalar – issue, control, dispatch

  - Complex memory and cache hierarchy

 Huawei Proprietary - Restricted Distribution

Target Platform – Atlas 950

From a developer perspective

- Many functional units
  - Cube – tensor core
  - Vector – vector operations
  - Scalar – issue, control, dispatch

  - Complex memory and cache hierarchy
  - All transfers – async DMA with explicit unit synchronization via set/wait flags

Target Platform – Atlas 950

From a developer perspective

- Many functional units
  - Cube – tensor core
  - Vector – vector operations
  - Scalar – issue, control, dispatch

  - Complex memory and cache hierarchy
  - All transfers – async DMA with explicit unit synchronization via set/wait flags

- Low-level programming is a bit complicated

Vertical stack overview

Please meet the Xiao

Vertical stack overview

Please meet the Xiao

His Torch-written model works

Vertical stack overview

Please meet the Xiao

His Torch-written model works

He want it to **actually** work

Huawei Proprietary - Restricted Distribution

Vertical stack overview

Please meet the Miao

Vertical stack overview

Please meet the Miao

He have the solution!

Vertical stack overview

Torch

Torch, compatible with Xiao's previous workaround

https://gitcode.com/Ascend/pytorch

Vertical stack overview

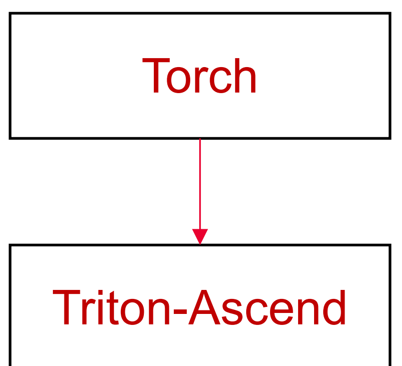
Torch

Torch, compatible with Xiao's previous workaround

With custom **inductor** backend

https://gitcode.com/Ascend/pytorch

Vertical stack overview



Torch, compatible with Xiao's previous workaround

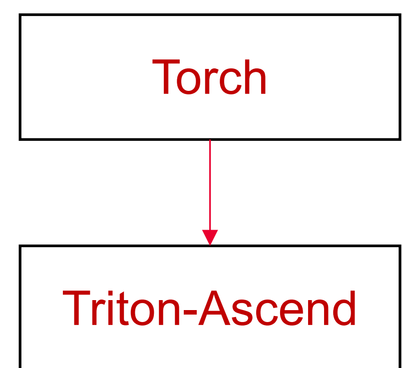
With custom **inductor** chip-aware backend
Invokes the Triton-Ascend

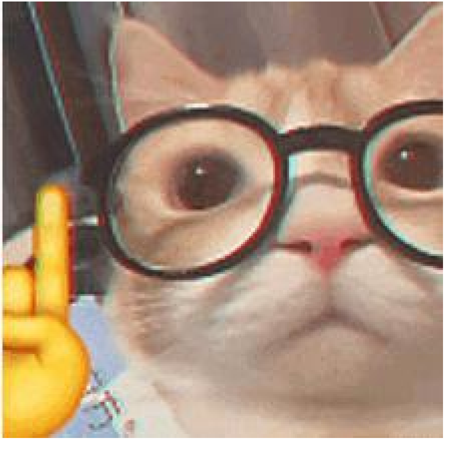
https://gitcode.com/Ascend/pytorch

Vertical stack overview

Triton-Ascend

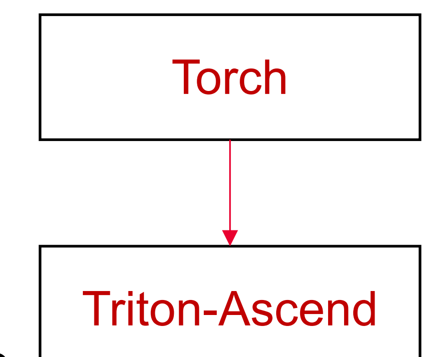
https://gitcode.com/Ascend/triton-ascend





Huawei Proprietary - Restricted Distribution

Vertical stack overview



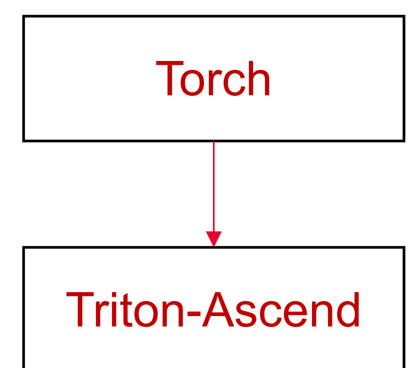
Triton-Ascend

Open-Sourced and widely used well-known Triton language

https://gitcode.com/Ascend/triton-ascend

Huawei Proprietary - Restricted Distribution

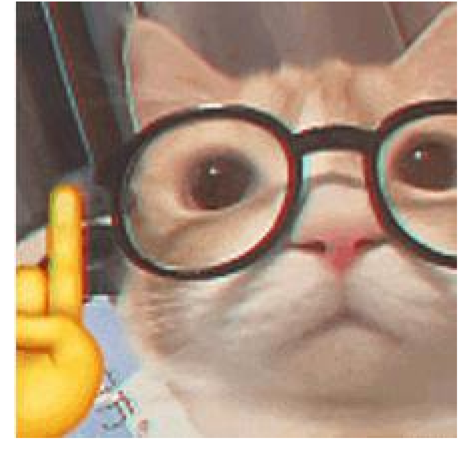
Vertical stack overview



Triton-Ascend

Open-Sourced and widely used well-known Triton language
With backend for Atlas chips

https://gitcode.com/Ascend/triton-ascend



Huawei Proprietary - Restricted Distribution

Vertical stack overview

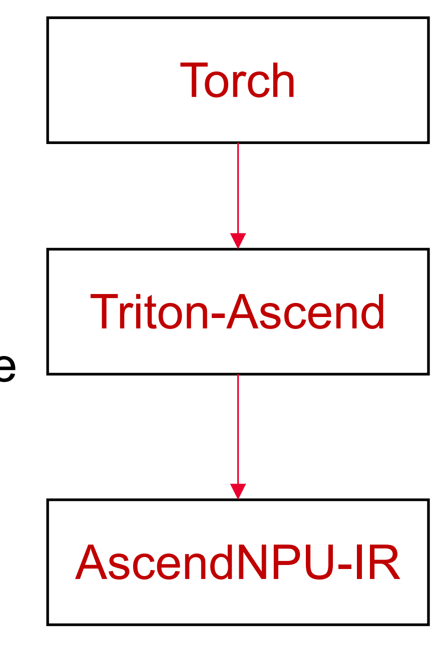
Triton-Ascend

Open-Sourced and widely used well-known Triton language

With backend for Atlas chips

Produces MLIR for the AscendNPU-IR

https://gitcode.com/Ascend/triton-ascend

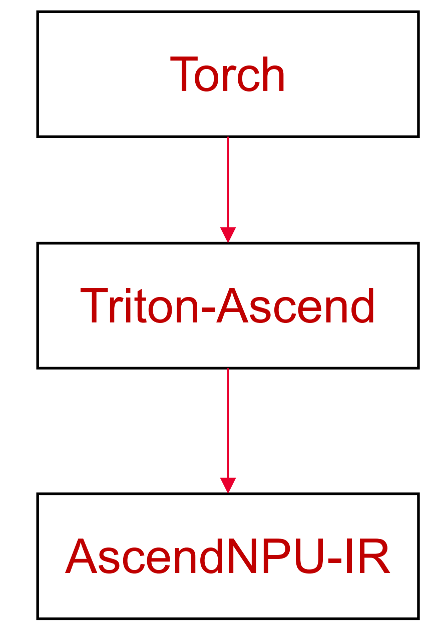


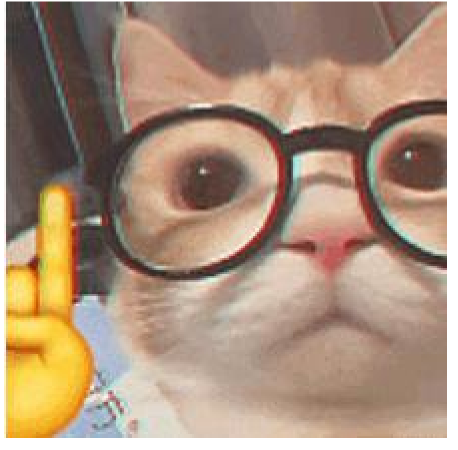
Vertical stack overview

AscendNPU-IR

Optimizing Atlas-aware MLIR Compiler

https://gitcode.com/Ascend/AscendNPU-IR





Huawei Proprietary - Restricted Distribution

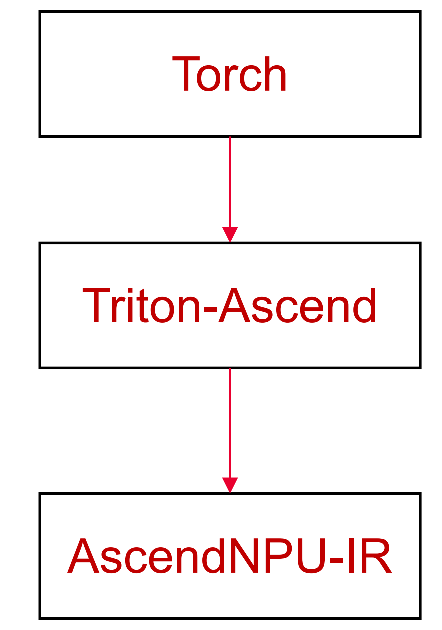
Vertical stack overview

AscendNPU-IR

Optimizing Atlas-aware MLIR Compiler

First-class citizen in this talk

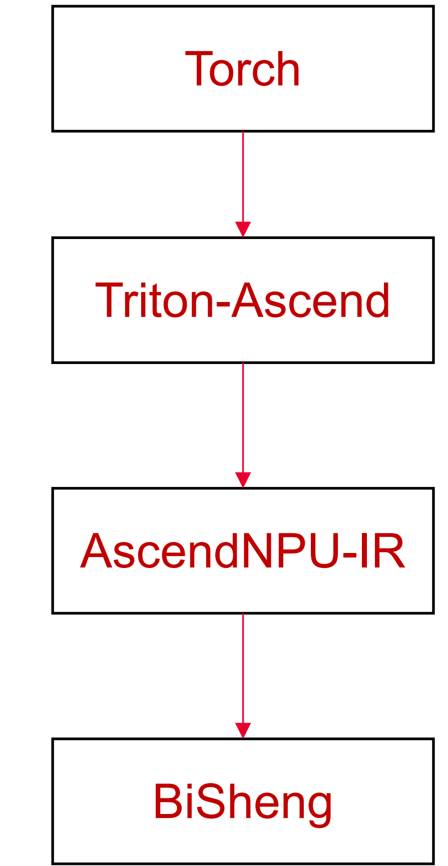
https://gitcode.com/Ascend/AscendNPU-IR



Vertical stack overview

AscendNPU-IR

Optimizing Atlas-aware MLIR Compiler
First-class citizen in this talk
Produces low-level IR for the BiSheng compiler
https://gitcode.com/Ascend/AscendNPU-IR

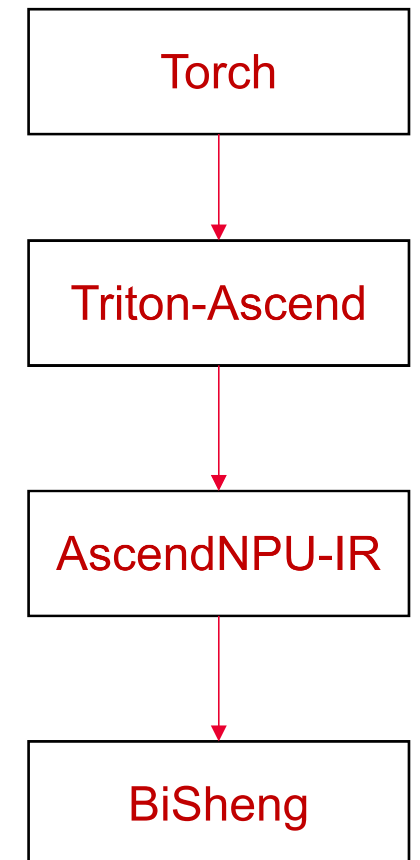


 Huawei Proprietary - Restricted Distribution

Vertical stack overview

BiSheng compiler

Optimizing backend for Atlas chips



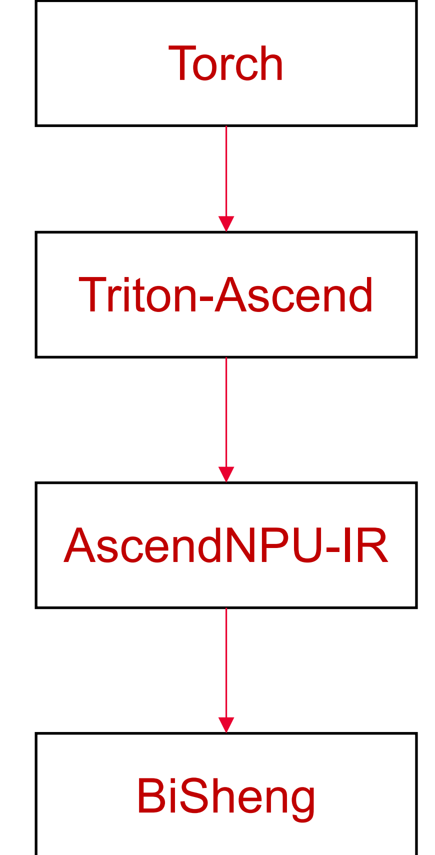
Huawei Proprietary - Restricted Distribution

Vertical stack overview

BiSheng compiler

Optimizing backend for Atlas chips

Not **yet** open-sourced



Huawei Proprietary - Restricted Distribution

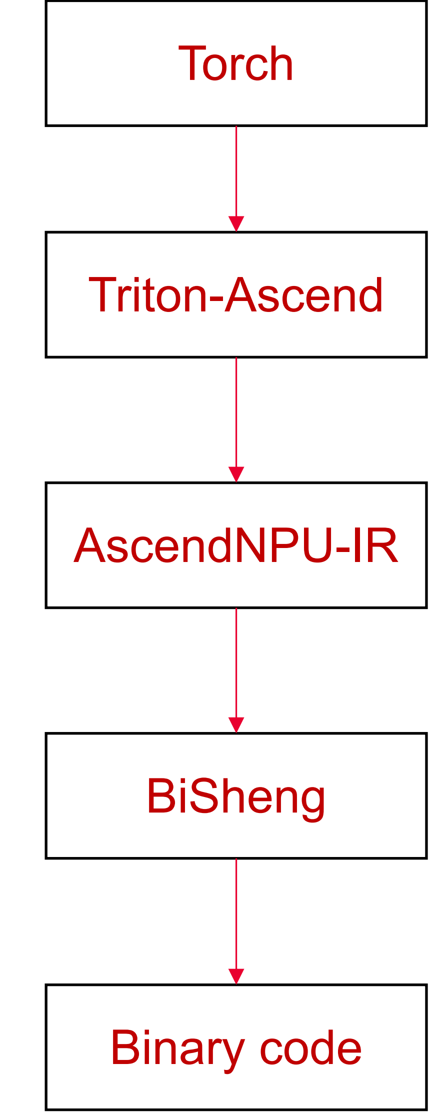
Vertical stack overview

BiSheng compiler

Optimizing backend for Atlas chips

Not **yet** open-sourced

Produces highly-optimized binary code



Vertical stack overview

BiSheng compiler

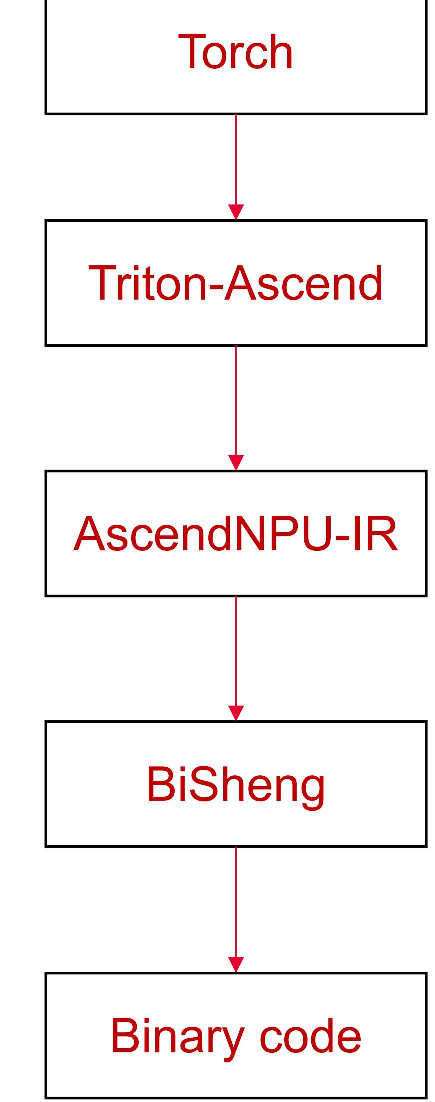
Optimizing backend for Atlas chips

Not **yet** open-sourced

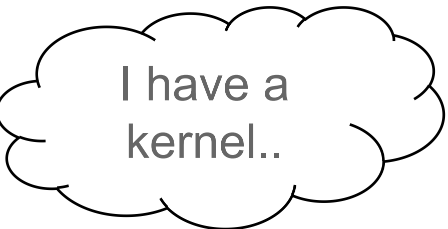
Produces highly-optimized binary code

Binary releases are fully-available with CANN

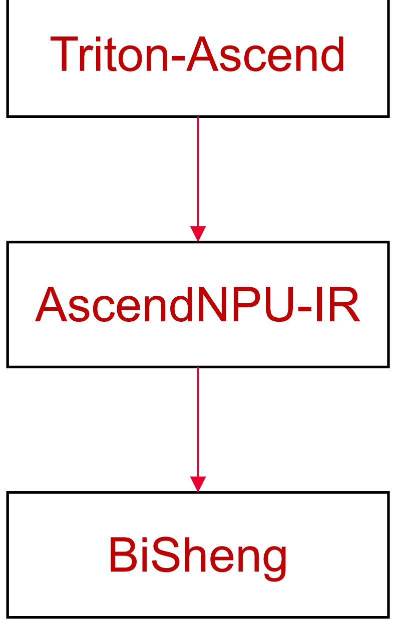
https://www.hiascend.com/cann/download



Down the pipeline: one kernel story




```python
@triton.jit
def kernel(
    a_ptr, b_ptr, c_ptr,
    stride_am, stride_ak,
    stride_bk, stride_bn,
    stride_cm, stride_cn,
    alpha, beta,
    BLOCK_M: tl.constexpr,
    BLOCK_N: tl.constexpr,
    BLOCK_K: tl.constexpr,
):
    pid = tl.program_id(0)
    offs_m = tl.arange(0, BLOCK_M)[:, None]
    offs_n = tl.arange(0, BLOCK_N)[None, :]
    offs_k = tl.arange(0, BLOCK_K)
    a_ptrs = a_ptr + offs_m * stride_am + offs_k[None, :] * stride_ak
    b_ptrs = b_ptr + offs_k[:, None] * stride_bk + offs_n * stride_bn
    a = tl.load(a_ptrs)
    b = tl.load(b_ptrs)
    acc = tl.dot(a, b)
    acc = acc * alpha
    acc = acc + beta
    acc = tl.maximum(acc, 0.0)
    c_ptrs = c_ptr + offs_m * stride_cm + offs_n * stride_cn
    tl.store(c_ptrs, acc)
```



$$C = \max\big(\alpha(AB) + \beta,\ 0\big)$$

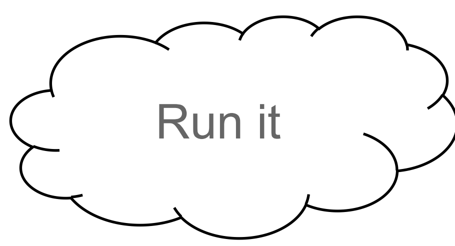
Huawei Proprietary - Restricted Distribution

Down the pipeline: one kernel story

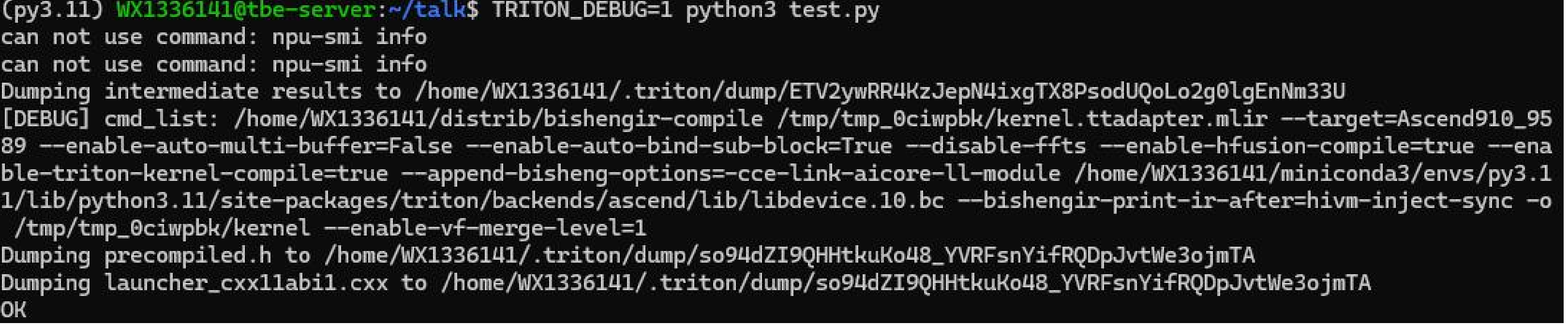
```python
def main():
    device = "npu"
    M = N = K = 16

    a = torch.randn((M, K), device=device, dtype=torch.float16)
    b = torch.randn((K, N), device=device, dtype=torch.float16)
    c = torch.empty((M, N), device=device, dtype=torch.float32)

    alpha = 0.5
    beta = 1.0

    grid = (1,)

    kernel[grid](
        a, b, c,
        a.stride(0), a.stride(1),
        b.stride(0), b.stride(1),
        c.stride(0), c.stride(1),
        alpha, beta,
        BLOCK_M=16, BLOCK_N=16, BLOCK_K=16,
    )

    ref = torch.relu(torch.matmul(a.float(), b.float()) * alpha + beta)
    torch.testing.assert_close(c, ref, atol=1e-3, rtol=1e-3)
    print("OK")
    print(c)


if __name__ == "__main__":
    main()
```

Let me see how it works..

Down the pipeline: one kernel story



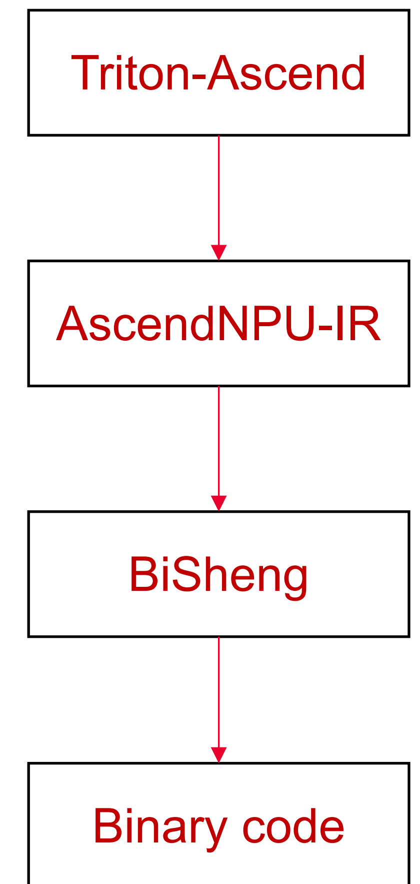
Triton-Ascend



Huawei Proprietary - Restricted Distribution

Down the pipeline: one kernel story

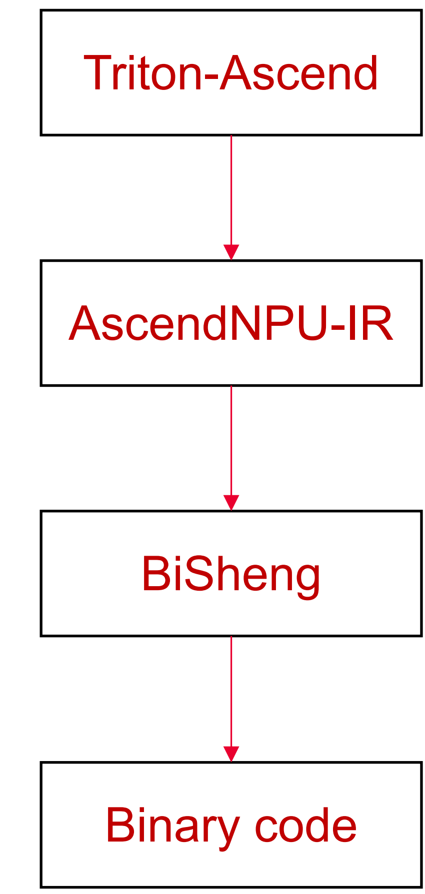
```
(py3.11) WX1336141@tbe-server:~/talk$ cp /home/WX1336141/.triton/dump/ETV2ywRR4KzJepN4ixgTX8Pso
dUQoLo2g0lgEnNm33U/kernel.ttadapter.mlir ./
(py3.11) WX1336141@tbe-server:~/talk$ /home/WX1336141/distrib/bishengir-compile kernel.ttadapte
r.mlir --target=Ascend910_9589 --enable-auto-multi-buffer=False --enable-auto-bind-sub-block=Tr
ue --disable-ffts --enable-hfusion-compile=true --enable-triton-kernel-compile=true '--append-b
isheng-options=-cce-link-aicore-ll-module /home/WX1336141/miniconda3/envs/py3.11/lib/python3.11
/site-packages/triton/backends/ascend/lib/libdevice.10.bc' -o kernel --enable-vf-merge-level=1
--mlir-print-ir-after-all >& log.txt
```



Huawei Proprietary - Restricted Distribution

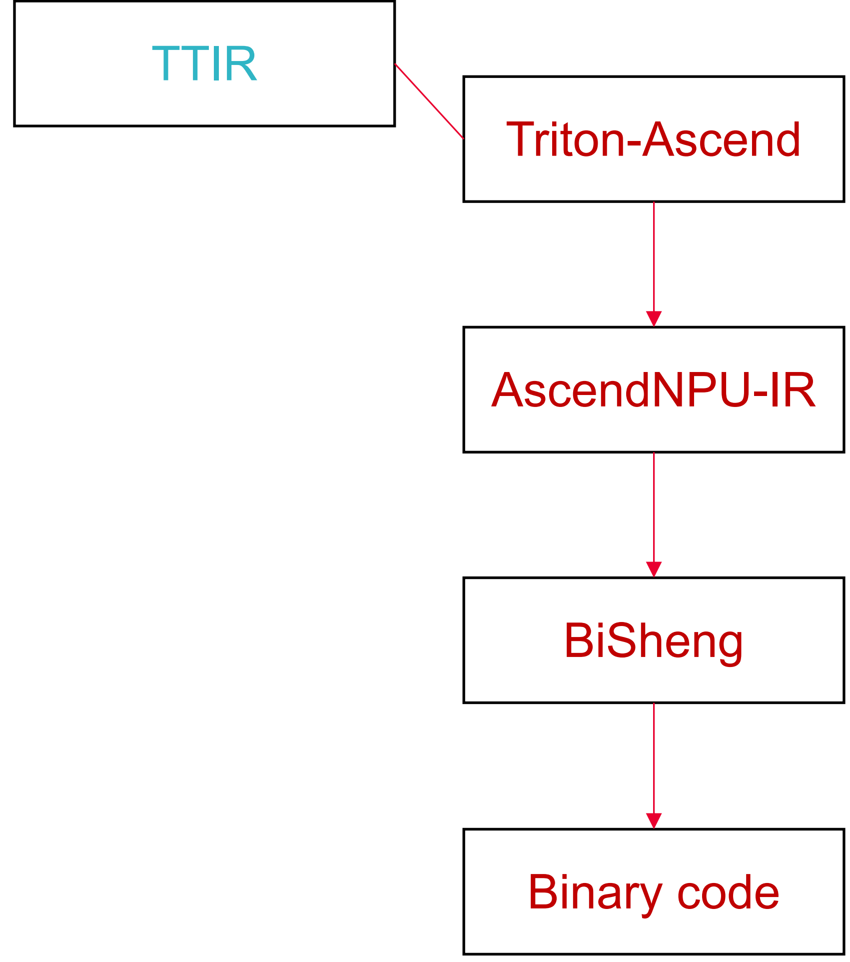
Down the pipeline: one kernel story

We'll look at the compilation pipeline for the Atlas 950 chip, but the pipeline for 910 is similar, though the behavior of some stages may differ.

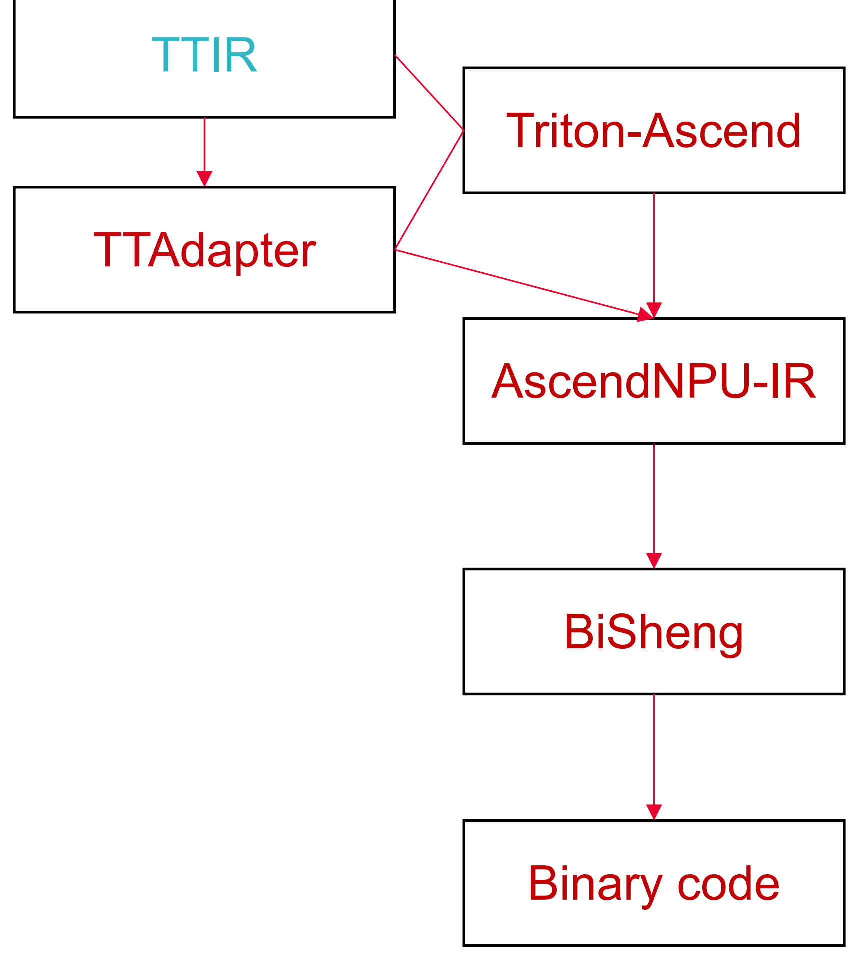


Huawei Proprietary - Restricted Distribution

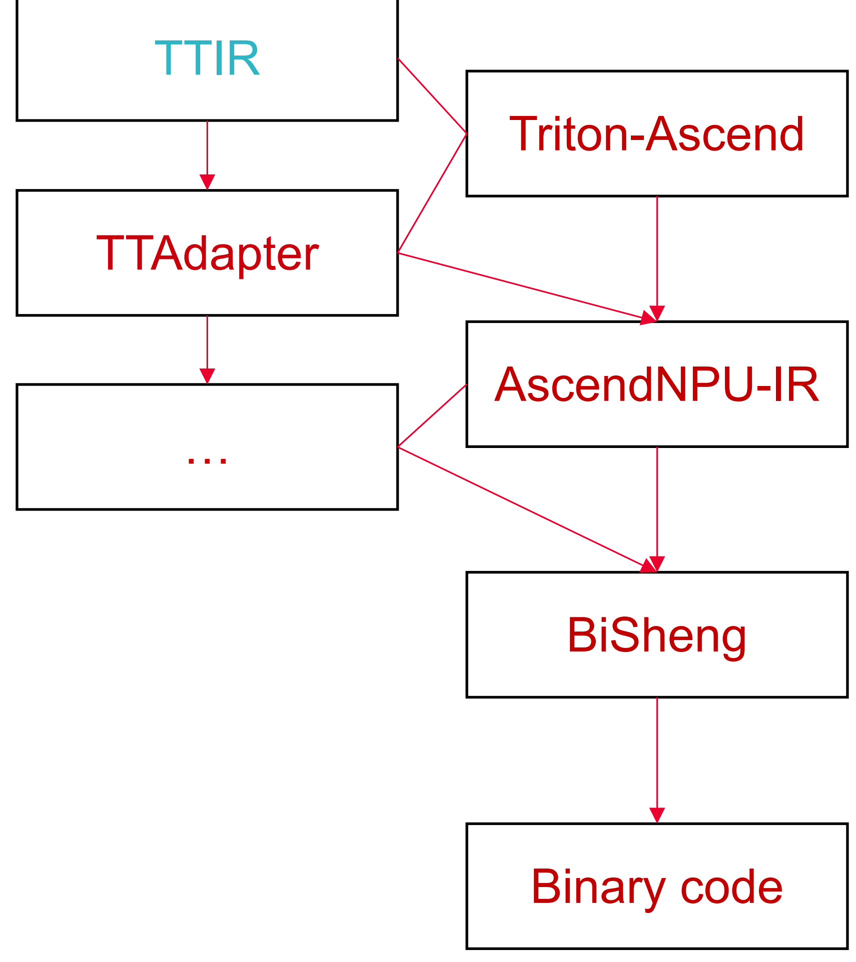
Down the pipeline: one kernel story



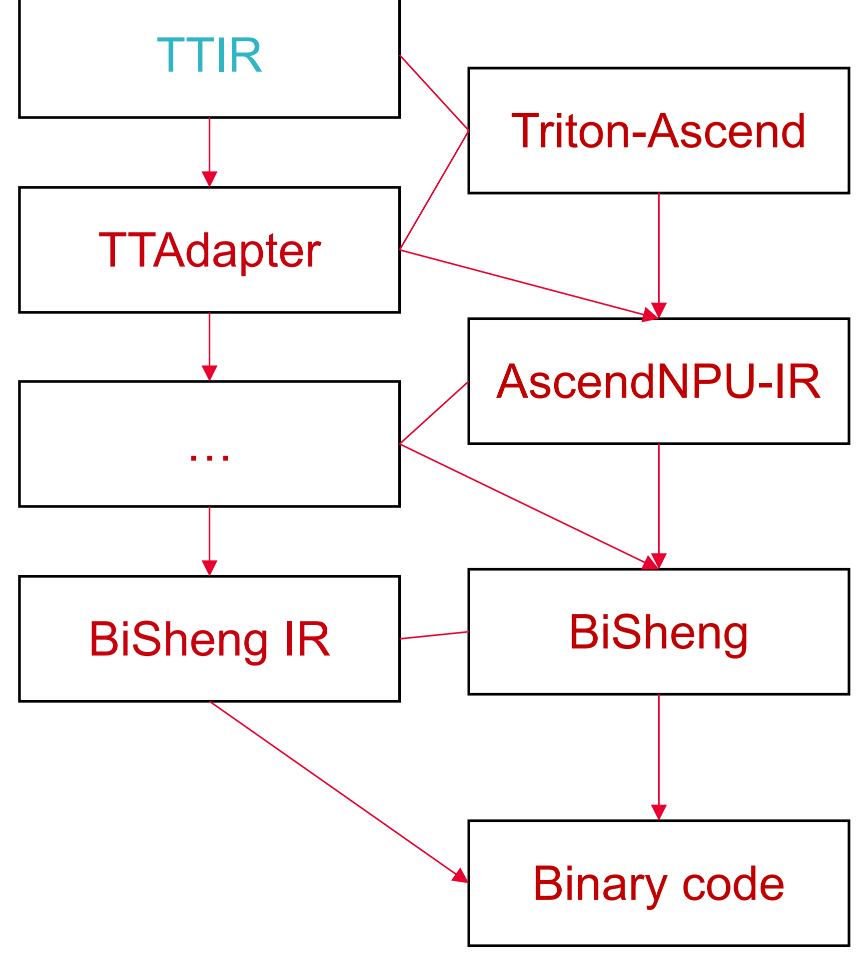
Down the pipeline: one kernel story



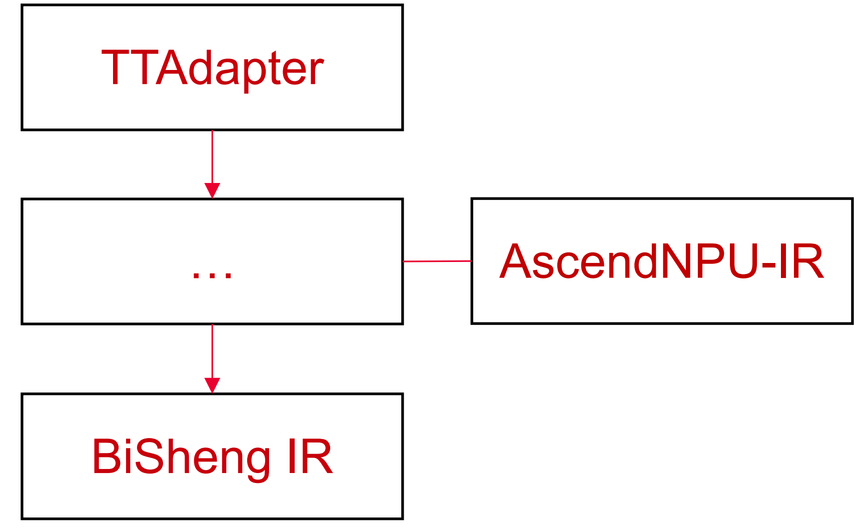
Down the pipeline: one kernel story



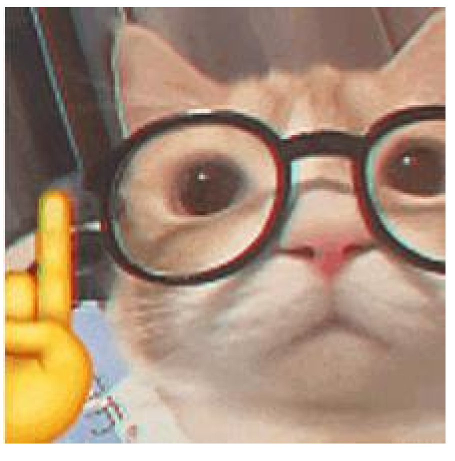
Down the pipeline: one kernel story



Down the pipeline: one kernel story



The Key – layers and dialects fusion



Flexible nonlinear lowering

Linalg

Tensor

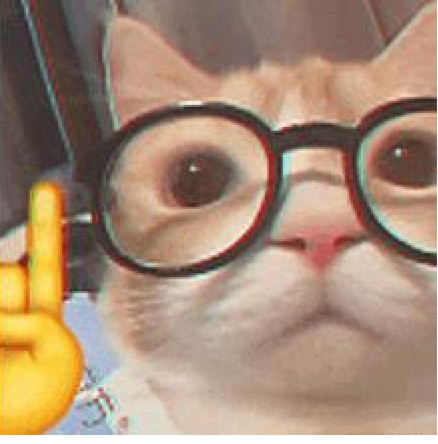
MemRef

Arith

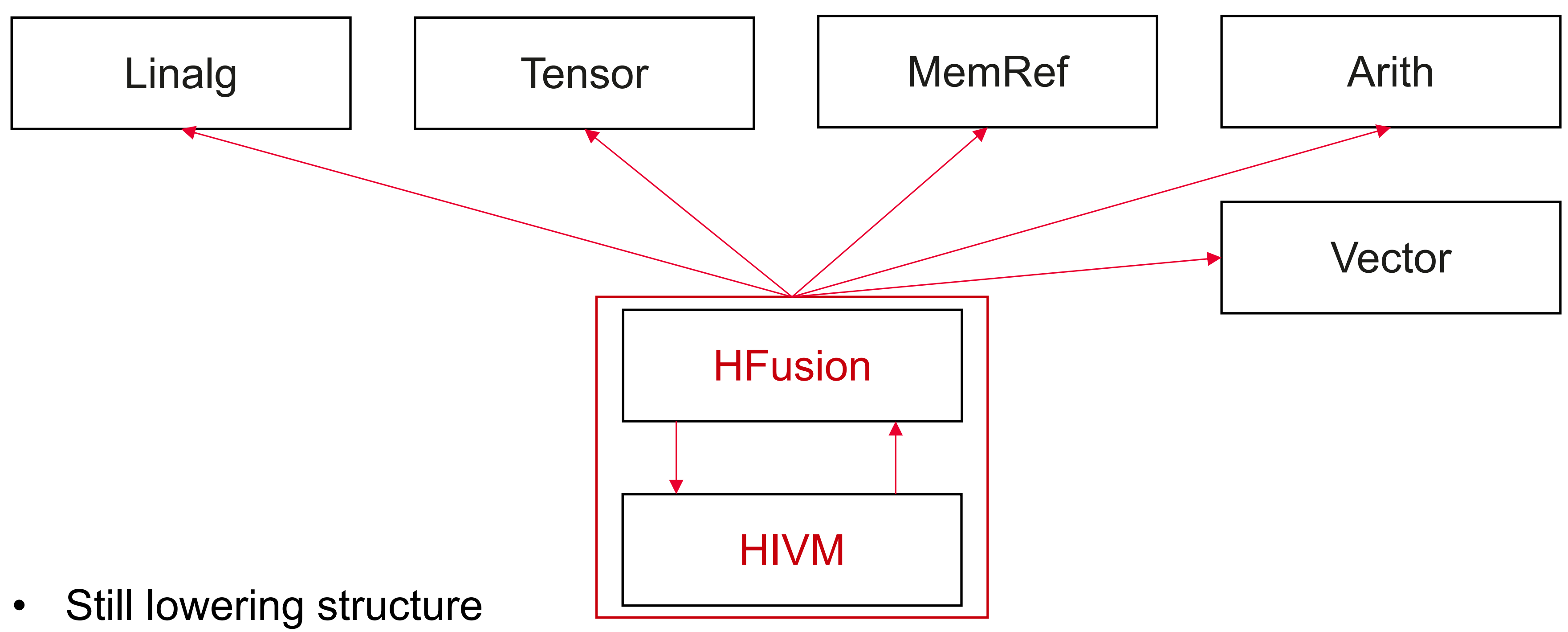
HFusion

Vector

HIVM

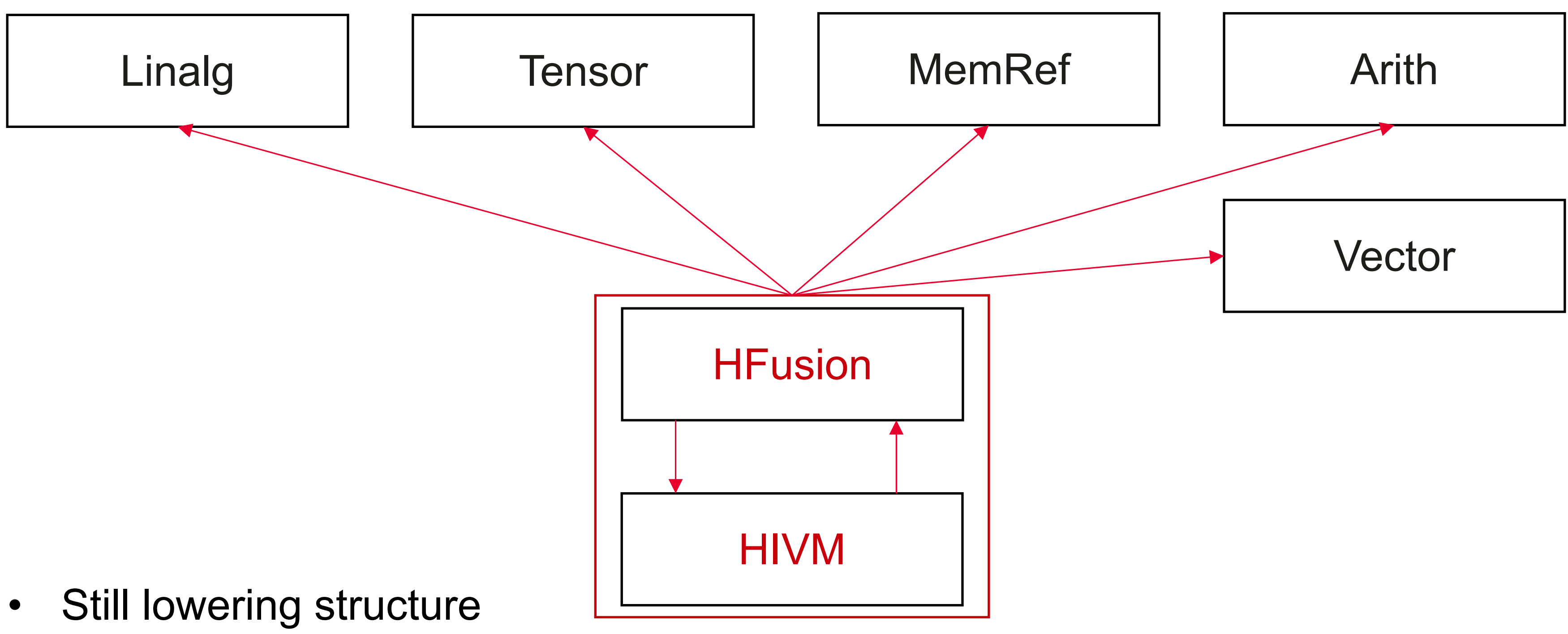


Flexible nonlinear lowering



- Still lowering structure

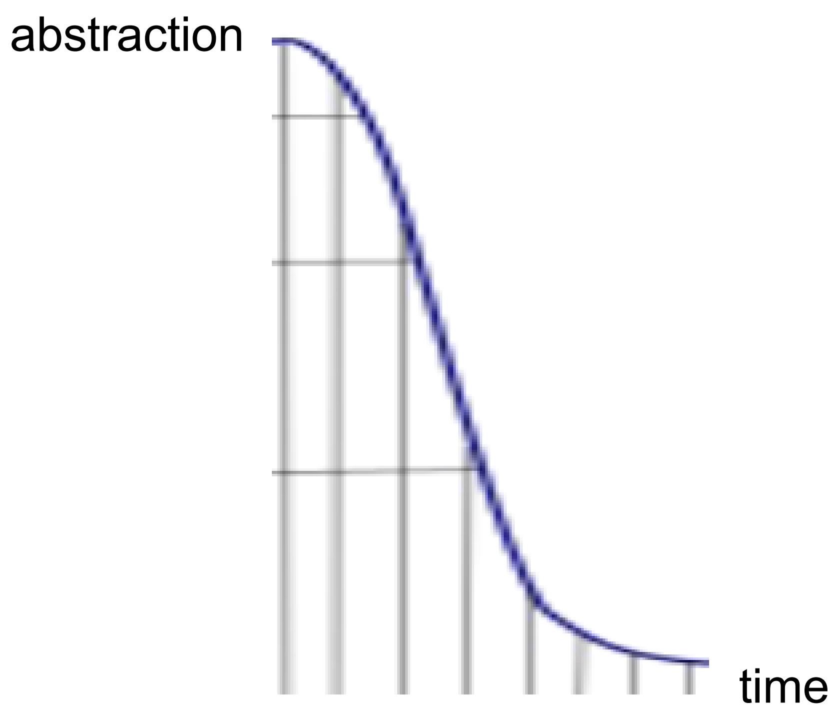
Flexible nonlinear lowering



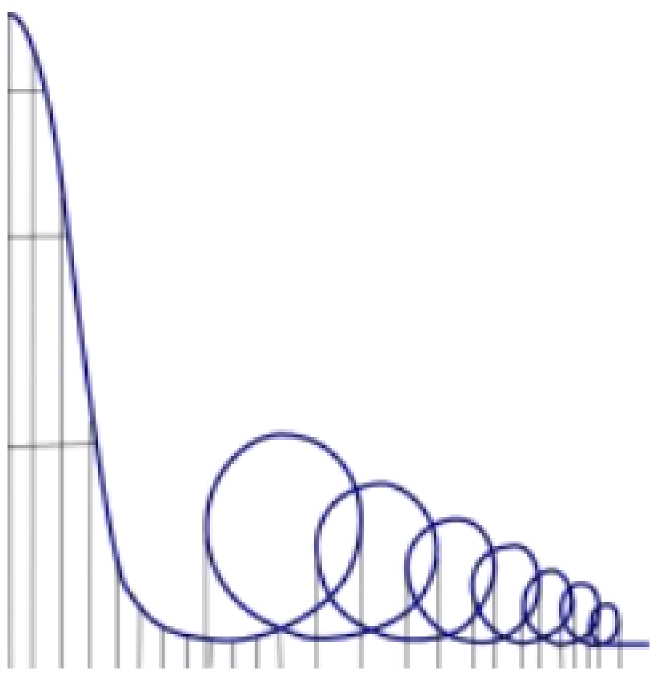
- Still lowering structure
- But with mixture of a different dialects on each level
  - Ability to reuse optimizations
  - Deep inter-dialect integration and communication

Huawei Proprietary - Restricted Distribution

So the pipeline does not look like this



But rather like this



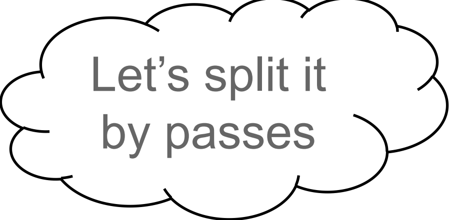
https://en.wikipedia.org/wiki/Euthanasia_Coaster

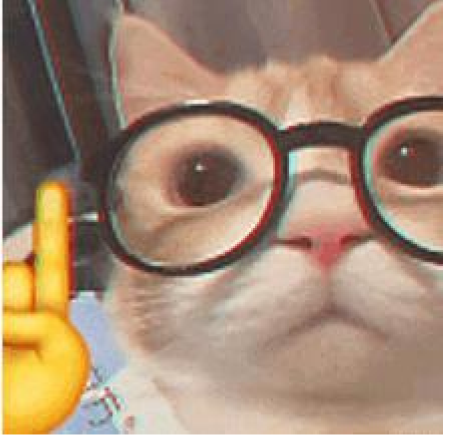
Down the pipeline: one kernel story

```
 func.func @kernel(%arg0: memref<?xi8> loc("test.py":8:0), %arg1: memref<?xi8> loc("test.py":8:0), %arg2:
 memref<?xf16> {tt.divisibility = 16 : i32, tt.tensor_kind = 0 : i32} loc("test.py":8:0), %arg3: memref<?x
f16> {tt.divisibility = 16 : i32, tt.tensor_kind = 0 : i32} loc("test.py":8:0), %arg4: memref<?xf32> {tt.d
ivisibility = 16 : i32, tt.tensor_kind = 1 : i32} loc("test.py":8:0), %arg5: i32 {tt.divisibility = 16 : i
32} loc("test.py":8:0), %arg6: i32 {tt.divisibility = 16 : i32} loc("test.py":8:0), %arg7: i32 {tt.divisib
ility = 16 : i32} loc("test.py":8:0), %arg8: f32 loc("test.py":8:0), %arg9: f32 loc("test.py":8:0), %arg10
: i32 loc("test.py":8:0), %arg11: i32 loc("test.py":8:0), %arg12: i32 loc("test.py":8:0), %arg13: i32 loc(
"test.py":8:0), %arg14: i32 loc("test.py":8:0), %arg15: i32 loc("test.py":8:0)) attributes {SyncBlockLockA
rgIdx = 0 : i64, WorkspaceArgIdx = 1 : i64, global_kernel = "local", mix_mode = "mix", parallel_mode = "si
md"} {
    %cst = arith.constant 0.000000e+00 : f32 loc(#loc1)
    %0 = tensor.empty() : tensor<16x16xf32> loc(#loc2)
    %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32> loc(#loc1)
    %2 = arith.index_cast %arg5 : i32 to index loc(#loc)
    %reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%2, 1] :
memref<?xf16> to memref<16x16xf16, strided<[?, 1]>> loc(#loc3)
    %3 = arith.index_cast %arg6 : i32 to index loc(#loc)
    %reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1]
: memref<?xf16> to memref<16x16xf16, strided<[?, 1]>> loc(#loc4)
    %alloc = memref.alloc() : memref<16x16xf16> loc(#loc5)
    memref.copy %reinterpret_cast, %alloc : memref<16x16xf16, strided<[?, 1]>> to memref<16x16xf16> loc(#l
oc5)
    %4 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16> loc(#loc5)
    %alloc_1 = memref.alloc() : memref<16x16xf16> loc(#loc6)
    memref.copy %reinterpret_cast_0, %alloc_1 : memref<16x16xf16, strided<[?, 1]>> to memref<16x16xf16> lo
c(#loc6)
    %5 = bufferization.to_tensor %alloc_1 restrict writable : memref<16x16xf16> loc(#loc6)
    %6 = linalg.matmul {input_precision = "ieee"} ins(%4, %5 : tensor<16x16xf16>, tensor<16x16xf16>) outs(
%1 : tensor<16x16xf32>) -> tensor<16x16xf32> loc(#loc7)
    %7 = linalg.fill ins(%arg8 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32> loc(#loc2)
    %8 = arith.mulf %6, %7 : tensor<16x16xf32> loc(#loc2)
    %9 = linalg.fill ins(%arg9 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32> loc(#loc8)
    %10 = arith.addf %8, %9 : tensor<16x16xf32> loc(#loc8)
    %11 = arith.maxnumf %10, %1 : tensor<16x16xf32> loc(#loc1)
    %12 = arith.index_cast %arg7 : i32 to index loc(#loc)
    %reinterpret_cast_2 = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%12, 1]
 : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>> loc(#loc9)
    bufferization.materialize_in_destination %11 in writable %reinterpret_cast_2 : (tensor<16x16xf32>, mem
ref<16x16xf32, strided<[?, 1]>>) -> () loc(#loc10)
    return loc(#loc)
  } loc(#loc)
} loc(#loc)
```

Huawei Proprietary - Restricted Distribution

Down the pipeline





Down the pipeline: HFusion dialect

Both tensor and memref-oriented vocabulary to clearly express and perform in-kernel high level operations fusion, planning and normalization

Down the pipeline: HFusion dialect

Both tensor and memref-oriented vocabulary to clearly express and perform in-kernel high level operations fusion, planning and normalization

```
%0 = tensor.empty() : tensor<4x2x64xf16>
%1 = tensor.empty() : tensor<4x2x64xf16>
%2 = hfusion.interleave %0, %1 : tensor<4x2x64xf16>, tensor<4x2x64xf16> -> tensor<4x2x128xf16>
```


Huawei Proprietary - Restricted Distribution

Down the pipeline: HFusion dialect

Both tensor and memref-oriented vocabulary to clearly express and perform in-kernel high level operations fusion, planning and normalization

```
%0 = tensor.empty() : tensor<4x2x64xf16>
%1 = tensor.empty() : tensor<4x2x64xf16>
%2 = hfusion.interleave %0, %1 : tensor<4x2x64xf16>, tensor<4x2x64xf16> -> tensor<4x2x128xf16>
```

```
%0 = tensor.empty() : tensor<4096x32x1xf16>
%1 = hfusion.deinterleave %arg0 channel<1> : tensor<4096x32x2xf16> -> tensor<4096x32x1xf16>
```


Huawei Proprietary - Restricted Distribution

Down the pipeline: HFusion dialect

Both tensor and memref-oriented vocabulary to clearly express and perform in-kernel high level operations fusion, planning and normalization

```
%0 = tensor.empty() : tensor<4x2x64xf16>
%1 = tensor.empty() : tensor<4x2x64xf16>
%2 = hfusion.interleave %0, %1 : tensor<4x2x64xf16>, tensor<4x2x64xf16> -> tensor<4x2x128xf16>
```

```
%0 = tensor.empty() : tensor<4096x32x1xf16>
%1 = hfusion.deinterleave %arg0 channel<1> : tensor<4096x32x2xf16> -> tensor<4096x32x1xf16>
```

```
%21 = hfusion.elementwise_binary {fun = #hfusion.binary_fn<powf>}
   ins(%18, %20 : tensor<60x70x768xf32>, tensor<64x70x768xf32>)
   outs(%0 : tensor<64x70x768xf32>) -> tensor<64x70x768xf32>
```


Huawei Proprietary - Restricted Distribution

Down the pipeline: HFusion dialect

Both tensor and memref-oriented vocabulary to clearly express and perform in-kernel high level operations fusion, planning and normalization

```
%0 = tensor.empty() : tensor<4x2x64xf16>
%1 = tensor.empty() : tensor<4x2x64xf16>
%2 = hfusion.interleave %0, %1 : tensor<4x2x64xf16>, tensor<4x2x64xf16> -> tensor<4x2x128xf16>
```

```
%0 = tensor.empty() : tensor<4096x32x1xf16>
%1 = hfusion.deinterleave %arg0 channel<1> : tensor<4096x32x2xf16> -> tensor<4096x32x1xf16>
```

```
%21 = hfusion.elementwise_binary {fun = #hfusion.binary_fn<powf>}
   ins(%18, %20 : tensor<60x70x768xf32>, tensor<64x70x768xf32>)
   outs(%0 : tensor<64x70x768xf32>) -> tensor<64x70x768xf32>
```

```
%0 = tensor.empty() : tensor<4x8x8xf32>
%1 = hfusion.flip %0 : tensor<4x8x8xf32> flip axis = 1 -> tensor<4x8x8xf32>
```

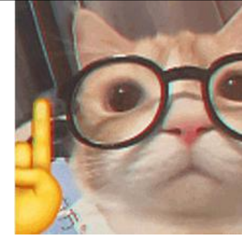
Down the pipeline: HFusion dialect

While still allowing to easily mix it with normal linalg code at any stages

```
%1 = hfusion.deinterleave %arg0 channel<1> : tensor<4096x32x2xf16> -> tensor<4096x32x1xf16>
%2 = linalg.reduce
   ins(%1 : tensor<4096x32x1xf16>)
   outs(%0 : tensor<32x1xf16) dimensions = [0]
   (%in: f16, %init: f16) {
      %31 = arith.addf %in, %init : f16
      linalg.yield %31 : f16
   }
```

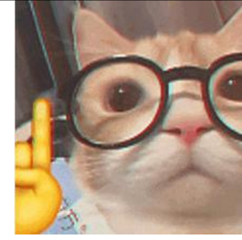

Huawei Proprietary - Restricted Distribution

Down the pipeline: (the way to) HFusion



```
func.func @kernel(...) {
  %cst = arith.constant 0.000000e+00 : f32
  %0 = tensor.empty() : tensor<16x16xf32>
  %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %2 = arith.index_cast %arg5 : i32 to index
  %reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%2, 1]
    : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
  %3 = arith.index_cast %arg6 : i32 to index
  %reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1]
    : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
  %alloc = memref.alloc() : memref<16x16xf16>
  memref.copy %reinterpret_cast, %alloc : memref<16x16xf16, strided<[?, 1]>> to memref<16x16xf16>
  %4 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
  %alloc_1 = memref.alloc() : memref<16x16xf16>
  memref.copy %reinterpret_cast_0, %alloc_1 : memref<16x16xf16, strided<[?, 1]>> to memref<16x16xf16>
  %5 = bufferization.to_tensor %alloc_1 restrict writable : memref<16x16xf16>
  %6 = linalg.matmul {input_precision = "ieee"}
    ins(%4, %5 : tensor<16x16xf16>, tensor<16x16xf16>)
    outs(%1 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %7 = linalg.fill ins(%arg8 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %8 = arith.mulf %6, %7 : tensor<16x16xf32>
  %9 = linalg.fill ins(%arg9 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %10 = arith.addf %8, %9 : tensor<16x16xf32>
  %11 = arith.maxnumf %10, %1 : tensor<16x16xf32>
  %12 = arith.index_cast %arg7 : i32 to index
  %reinterpret_cast_2 = memref.reinterpret_cast %arg4 to offset: [0],
    sizes: [16, 16], strides: [%12, 1] : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
  bufferization.materialize_in_destination %11 in writable %reinterpret_cast_2
    : (tensor<16x16xf32>, memref<16x16xf32, strided<[?, 1]>>) -> ()
  return
}
```

Down the pipeline: (the way to) HFusion



```
func.func @kernel(...) {
  ...
  %8 = arith.mulf %6, %7 : tensor<16x16xf32>
  %9 = linalg.fill ins(%arg9 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %10 = arith.addf %8, %9 : tensor<16x16xf32>
  %11 = arith.maxnumf %10, %1 : tensor<16x16xf32>
  %12 = arith.index_cast %arg7 : i32 to index
  %reinterpret_cast_2 = memref.reinterpret_cast %arg4 to offset: [0],
    sizes: [16, 16], strides: [%12, 1] : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
  bufferization.materialize_in_destination %11 in writable %reinterpret_cast_2
    : (tensor<16x16xf32>, memref<16x16xf32, strided<[?, 1]>>) -> ()
  return
}
```

- Normalize arith ops to their linalg / hfusion form

 Huawei Proprietary - Restricted Distribution

Down the pipeline: (the way to) HFusion

```
func.func @kernel() {
  ...
  %8 = tensor.empty() : tensor<16x16xf32>
  %9 = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
    ins(%6, %7 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%8 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %10 = linalg.fill ins(%arg9 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %11 = tensor.empty() : tensor<16x16xf32>
  %12 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
    ins(%9, %10 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%11 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %13 = tensor.empty() : tensor<16x16xf32>
  %14 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<maxnumf>}
    ins(%12, %1 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%13 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %15 = arith.index_cast %arg7 : i32 to index
  %reinterpret_cast_2 = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%15, 1]
    : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
  bufferization.materialize_in_destination %14 in writable %reinterpret_cast_2
    : (tensor<16x16xf32>, memref<16x16xf32, strided<[?, 1]>>) -> ()
  return
}
```

- Normalize arith ops to their linalg / hfusion forms

Down the pipeline: (the way to) HFusion

```
func.func @kernel() {
  ...
  %8 = tensor.empty() : tensor<16x16xf32>
  %9 = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
    ins(%6, %7 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%8 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %10 = linalg.fill ins(%arg9 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %11 = tensor.empty() : tensor<16x16xf32>
  %12 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
    ins(%9, %10 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%11 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %13 = tensor.empty() : tensor<16x16xf32>
  %14 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<maxnumf>}
    ins(%12, %1 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%13 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %15 = arith.index_cast %arg7 : i32 to index
  %reinterpret_cast_2 = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%15, 1]
    : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
  bufferization.materialize_in_destination %14 in writable %reinterpret_cast_2
    : (tensor<16x16xf32>, memref<16x16xf32, strided<[?, 1]>>) -> ()
  return
}
```

- Normalize arith ops to their linalg / hfusion forms
- And canonicalize a bit

Huawei Proprietary - Restricted Distribution

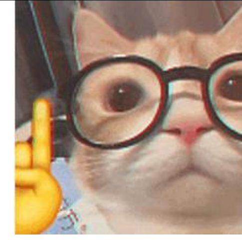
Down the pipeline: (the way to) HFusion

```
func.func @kernel(...) {
  ...
  %8 = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
    ins(%6, %7 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %9 = linalg.fill ins(%arg9 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %10 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
    ins(%8, %9 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %11 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<maxnumf>}
    ins(%10, %1 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %12 = arith.index_cast %arg7 : i32 to index
  %reinterpret_cast_2 = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%12, 1]
    : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
  bufferization.materialize_in_destination %11 in writable %reinterpret_cast_2
    : (tensor<16x16xf32>, memref<16x16xf32, strided<[?, 1]>>) -> ()
  return
}
```

- Normalize arith ops to their linalg / hfusion forms
- And canonicalize a bit

Huawei Proprietary - Restricted Distribution

Down the pipeline: HFusion



```
func.func @kernel(...) {
  %cst = arith.constant 0.000000e+00 : f32
  %0 = tensor.empty() : tensor<16x16xf32>
  %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %2 = arith.index_cast %arg5 : i32 to index
  %reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%2, 1]
    : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
  %3 = arith.index_cast %arg6 : i32 to index
  %reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1]
    : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
  %alloc = memref.alloc() : memref<16x16xf16>
  memref.copy %reinterpret_cast, %alloc : memref<16x16xf16, strided<[?, 1]>> to memref<16x16xf16>
  %4 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
  %alloc_1 = memref.alloc() : memref<16x16xf16>
  memref.copy %reinterpret_cast_0, %alloc_1 : memref<16x16xf16, strided<[?, 1]>> to memref<16x16xf16>
  %5 = bufferization.to_tensor %alloc_1 restrict writable : memref<16x16xf16>
  %6 = linalg.matmul {input_precision = "ieee"}
    ins(%4, %5 : tensor<16x16xf16>, tensor<16x16xf16>)
    outs(%1 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %7 = linalg.fill ins(%arg8 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %8 = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
    ins(%6, %7 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %9 = linalg.fill ins(%arg9 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %10 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
    ins(%8, %9 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %11 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<maxnumf>}
    ins(%10, %1 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %12 = arith.index_cast %arg7 : i32 to index
  %reinterpret_cast_2 = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%12, 1]
    : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
  bufferization.materialize_in_destination %11 in writable %reinterpret_cast_2
    : (tensor<16x16xf32>, memref<16x16xf32, strided<[?, 1]>>) -> ()
  return
}
```

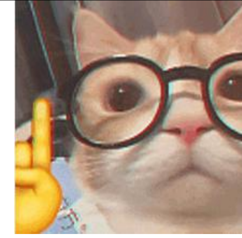
The kernel is still compact and expressive, but in HFusion-convenient form

Down the pipeline: HFusion

But some of the later canonicalization steps can make the code bigger...

Huawei Proprietary - Restricted Distribution

Down the pipeline: HFusion



```
func.func @kernel(...) {
  ...
  %11 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<maxnumf>}
    ins(%10, %1 : tensor<16x16xf32>, tensor<16x16xf32>)
    outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  ...
}
```

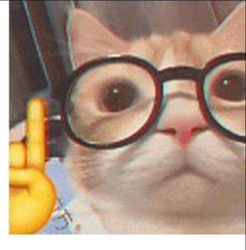

Huawei Proprietary - Restricted Distribution

Down the pipeline: HFusion

```
func.func @kernel(...) {
  ...
  %11 = tensor.empty() : tensor<16x16xi32>
  %12 = linalg.fill ins(%c2147483647_i32 : i32) outs(%11 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %13 = tensor.empty() : tensor<16x16xi32>
  %14 = hfusion.bitcast ins(%10 : tensor<16x16xf32>) outs(%13 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %15 = tensor.empty() : tensor<16x16xi32>
  %16 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<vand>}
   ins(%14, %12 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%15 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %17 = tensor.empty() : tensor<16x16xi32>
  %18 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
   ins(%16, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%17 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %19 = linalg.elemwise_binary {fun = #linalg.binary_fn<min_signed>}
   ins(%18, %c1_i32 : tensor<16x16xi32>, i32) outs(%18 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %20 = linalg.elemwise_binary {fun = #linalg.binary_fn<max_signed>}
   ins(%19, %c0_i32 : tensor<16x16xi32>, i32) outs(%19 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %21 = tensor.empty() : tensor<16x16xf32>
  %22 = hfusion.cast {
   cast = #hfusion.type_fn<cast_signed>, enable_overflow = true, enable_saturate = false,
   round_mode = #hfusion.round_mode<rint>, unsigned_mode = #hfusion.unsigned_mode<si2si>}
   ins(%20 : tensor<16x16xi32>) outs(%21 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %23 = tensor.empty() : tensor<16x16xi1>
  %24 = hfusion.compare {compare_fn = #hfusion.compare_fn<vne>}
   ins(%22, %cst_0 : tensor<16x16xf32>, f32) outs(%23 : tensor<16x16xi1>) -> tensor<16x16xi1>
  %25 = tensor.empty() : tensor<16x16xf32>
  %26 = hfusion.select
   ins(%24, %cst, %10 : tensor<16x16xi1>, f32, tensor<16x16xf32>)
   outs(%25 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %27 = tensor.empty() : tensor<16x16xi32>
```

(Part 1) It’s still a high-level dialect, so we can afford things like this here

Down the pipeline: HFusion



```
  %28 = linalg.fill ins(%c2147483647_i32 : i32)
   outs(%27 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %29 = tensor.empty() : tensor<16x16xi32>
  %30 = hfusion.bitcast ins(%1 : tensor<16x16xf32>) outs(%29 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %31 = tensor.empty() : tensor<16x16xi32>
  %32 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<vand>}
   ins(%30, %28 : tensor<16x16xi32>, tensor<16x16xi32>)
   outs(%31 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %33 = tensor.empty() : tensor<16x16xi32>
  %34 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
   ins(%32, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%33 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %35 = linalg.elemwise_binary {fun = #linalg.binary_fn<min_signed>}
   ins(%34, %c1_i32 : tensor<16x16xi32>, i32) outs(%34 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %36 = linalg.elemwise_binary {fun = #linalg.binary_fn<max_signed>}
   ins(%35, %c0_i32 : tensor<16x16xi32>, i32) outs(%35 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %37 = tensor.empty() : tensor<16x16xf32>
  %38 = hfusion.cast {
   cast = #hfusion.type_fn<cast_signed>, enable_overflow = true, enable_saturate = false,
   round_mode = #hfusion.round_mode<rint>, unsigned_mode = #hfusion.unsigned_mode<si2si>}
   ins(%36 : tensor<16x16xi32>) outs(%37 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %39 = tensor.empty() : tensor<16x16xi1>
  %40 = hfusion.compare {compare_fn = #hfusion.compare_fn<vne>}
   ins(%38, %cst_0 : tensor<16x16xf32>, f32) outs(%39 : tensor<16x16xi1>) -> tensor<16x16xi1>
  %41 = tensor.empty() : tensor<16x16xf32>
  %42 = hfusion.select
   ins(%40, %cst, %1 : tensor<16x16xi1>, f32, tensor<16x16xf32>) outs(%41 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %43 = tensor.empty() : tensor<16x16xf32>
  %44 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<maxf>}
   ins(%26, %42 : tensor<16x16xf32>, tensor<16x16xf32>) outs(%43 : tensor<16x16xf32>) -> tensor<16x16xf32>
  ...
}
```

(Part 2) It’s still a high-level dialect, so we can afford things like this here

Down the pipeline: HFusion

Because the later steps will cover our back again, keeping the essential changes and making it as compact as possible again

Down the pipeline: HFusion

Because the later steps will cover our back again, keeping the essential changes and making it as compact as possible again

Here comes the first hardware-aware optimizations and canonicalization steps, such as broadcast inlining

Down the pipeline: HFusion

```
func.func @kernel() {
  ...
  %7 = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>}
    ins(%6, %arg8 : tensor<16x16xf32>, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %8 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
    ins(%7, %arg9 : tensor<16x16xf32>, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %9 = tensor.empty() : tensor<16x16xi32>
  %10 = hfusion.bitcast
    ins(%8 : tensor<16x16xf32>) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %11 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<vand>}
    ins(%10, %c2147483647_i32 : tensor<16x16xi32>, i32) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %12 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
    ins(%11, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %13 = linalg.elemwise_binary {fun = #linalg.binary_fn<min_signed>}
    ins(%12, %c1_i32 : tensor<16x16xi32>, i32) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %14 = linalg.elemwise_binary {fun = #linalg.binary_fn<max_signed>}
    ins(%13, %c0_i32 : tensor<16x16xi32>, i32) outs(%13 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %15 = hfusion.cast {
    cast = #hfusion.type_fn<cast_signed>, enable_overflow = true, enable_saturate = false,
    round_mode = #hfusion.round_mode<rint>, unsigned_mode = #hfusion.unsigned_mode<si2si>}
    ins(%14 : tensor<16x16xi32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %16 = tensor.empty() : tensor<16x16xi1>
  %17 = hfusion.compare {compare_fn = #hfusion.compare_fn<vne>}
    ins(%15, %cst_0 : tensor<16x16xf32>, f32) outs(%16 : tensor<16x16xi1>) -> tensor<16x16xi1>
```

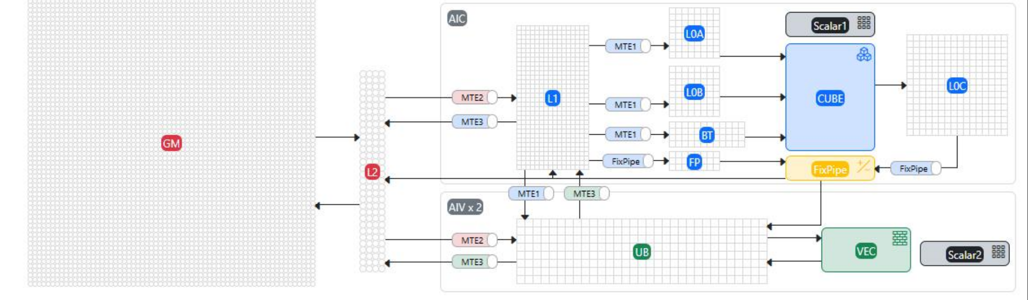

Huawei Proprietary - Restricted Distribution

Down the pipeline: HFusion

```
  %18 = hfusion.select
   ins(%17, %cst, %8 : tensor<16x16xi1>, f32, tensor<16x16xf32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %19 = hfusion.bitcast ins(%1 : tensor<16x16xf32>) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %20 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<vand>}
   ins(%19, %c2147483647_i32 : tensor<16x16xi32>, i32) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %21 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>}
   ins(%20, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %22 = linalg.elemwise_binary {fun = #linalg.binary_fn<min_signed>}
   ins(%21, %c1_i32 : tensor<16x16xi32>, i32) outs(%21 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %23 = linalg.elemwise_binary {fun = #linalg.binary_fn<max_signed>}
   ins(%22, %c0_i32 : tensor<16x16xi32>, i32) outs(%22 : tensor<16x16xi32>) -> tensor<16x16xi32>
  %24 = hfusion.cast {
   cast = #hfusion.type_fn<cast_signed>, enable_overflow = true, enable_saturate = false,
   round_mode = #hfusion.round_mode<rint>, unsigned_mode = #hfusion.unsigned_mode<si2si>}
   ins(%23 : tensor<16x16xi32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %25 = hfusion.compare {compare_fn = #hfusion.compare_fn<vne>}
   ins(%24, %cst_0 : tensor<16x16xf32>, f32) outs(%16 : tensor<16x16xi1>) -> tensor<16x16xi1>
  %26 = hfusion.select
   ins(%25, %cst, %cst_0 : tensor<16x16xi1>, f32, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %27 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<maxf>}
   ins(%18, %26 : tensor<16x16xf32>, tensor<16x16xf32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
  %28 = arith.index_cast %arg7 : i32 to index
  ...
}
```


Huawei Proprietary - Restricted Distribution

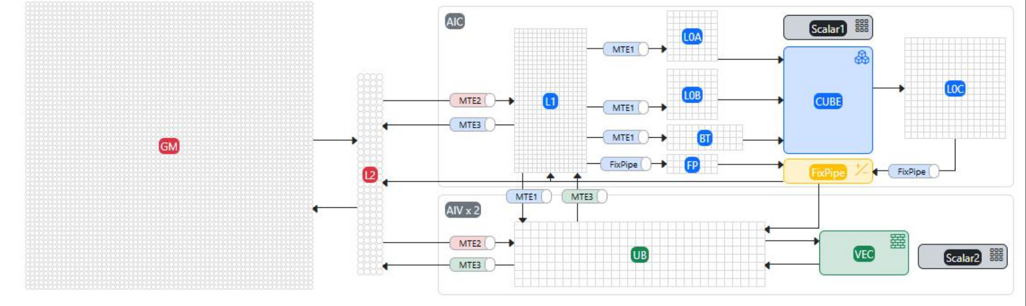
Down the pipeline: HIVM dialect



- HIVM.h
- HIVMAttrs.td
- HIVMBase.td
- HIVMDMAOps.td
- HIVMDoc.td
- HIVMImpl.h
- HIVMInterfaces.h
- HIVMInterfaces.td
- HIVMIntrinOps.td
- HIVMMacroOps.td
- HIVMOps.td
- HIVMSynchronizationOps.td
- HIVMTraits.h
- HIVMTraits.td
- HIVMVectorize.h
- HIVMVectorOps.td

Huawei Proprietary - Restricted Distribution

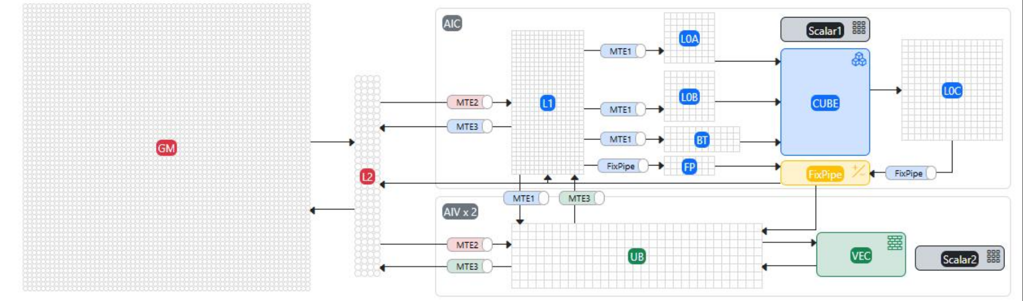
Down the pipeline: HIVM dialect



Multilevel hardware-aware dialect

- HIVM.h
- HIVMAttrs.td
- HIVMBase.td
- HIVMDMAOps.td
- HIVMDoc.td
- HIVMImpl.h
- HIVMInterfaces.h
- HIVMInterfaces.td
- HIVMIntrinOps.td
- HIVMMacroOps.td
- HIVMOps.td
- HIVMSynchronizationOps.td
- HIVMTraits.h
- HIVMTraits.td
- HIVMVectorize.h
- HIVMVectorOps.td

Down the pipeline: HIVM dialect



Multilevel hardware-aware dialect

- With explicit control

```
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
```

- HIVM.h
- HIVMAttrs.td
- HIVMBase.td
- HIVMDMAOps.td
- HIVMDoc.td
- HIVMImpl.h
- HIVMInterfaces.h
- HIVMInterfaces.td
- HIVMIntrinOps.td
- HIVMMacroOps.td
- HIVMOps.td
- HIVMSynchronizationOps.td
- HIVMTraits.h
- HIVMTraits.td
- HIVMVectorize.h
- HIVMVectorOps.td

Huawei Proprietary - Restricted Distribution

Down the pipeline: HIVM dialect

Multilevel hardware-aware dialect

- With explicit control

```
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
```

- But still enough high-level to work with memrefs or tensors *while being able to meet with real hardware operations*

```
%expanded_22 = tensor.expand_shape %54 [[0, 1]] output_shape [1, 32] : tensor<32xi64> into tensor<1x32xi64>
%55 = hivm.hir.vbrc ins(%expanded_22 : tensor<1x32xi64>) outs(%51 : tensor<64x32xi64>) broadcast_dims = [0]
%56 = hivm.hir.vcmp ins(%52, %55: tensor<64x32xi64>, tensor<64x32xi64>) outs(%48 : tensor<64x32xi1>) compare_mode = <ge>
%57 = hivm.hir.vmul ins(%45, %21 : tensor<64x32xf32>, f32) outs(%3 : tensor<64x32xf32>) -> tensor<64x32xf32>
%58 = hivm.hir.vsel ins(%56, %5, %4 : tensor<64x32xi1>, tensor<64x32xf32>, tensor<64x32xf32>) outs(%3 : tensor<64x32xf32>)
%59 = hivm.hir.vadd ins(%57, %58 : tensor<64x32xf32>, tensor<64x32xf32>) outs(%3 : tensor<64x32xf32>) -> tensor<64x32xf32>
```

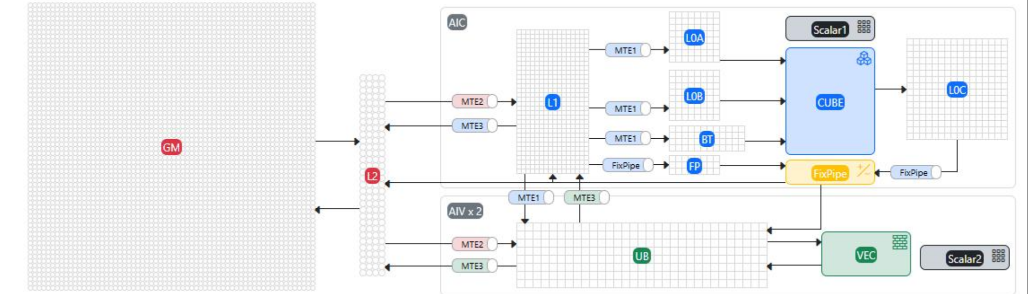
```
hivm.hir.wait_flag [#hivm.pipe<PIPE_MTE1>, #hivm.pipe<PIPE_M>, #hivm.event<EVENT_ID0>]
```

HFusion & HIVM, mix

The last Hfusion-only layer

```
...
%1 = linalg.fill ins(%cst_0 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
...
%6 = linalg.matmul {input_precision = "ieee"} ins(%4, %5 : tensor<16x16xf16>, tensor<16x16xf16>) outs(%1 : tensor<16x16xf32>) -> tensor<16x16xf32>
%7 = linalg.elemwise_binary {fun = #linalg.binary_fn<mul>} ins(%6, %arg8 : tensor<16x16xf32>, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%8 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>} ins(%7, %arg9 : tensor<16x16xf32>, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%9 = tensor.empty() : tensor<16x16xi32>
%10 = hfusion.bitcast ins(%8 : tensor<16x16xf32>) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32>
%11 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<vand>} ins(%10, %c2147483647_i32 : tensor<16x16xi32>, i32) outs(%9 : tensor<16x16xi32>) -> tensor<16x16
%12 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>} ins(%11, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32
%13 = linalg.elemwise_binary {fun = #linalg.binary_fn<min_signed>} ins(%12, %c1_i32 : tensor<16x16xi32>, i32) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%14 = linalg.elemwise_binary {fun = #linalg.binary_fn<max_signed>} ins(%13, %c0_i32 : tensor<16x16xi32>, i32) outs(%13 : tensor<16x16xi32>) -> tensor<16x16xi32>
%15 = hfusion.cast {cast = #hfusion.type_fn<cast_signed>, enable_overflow = true, enable_saturate = false, round_mode = #hfusion.round_mode<rint>, unsigned_mode
%16 = tensor.empty() : tensor<16x16xi1>
%17 = hfusion.compare {compare_fn = #hfusion.compare_fn<vne>} ins(%15, %cst_0 : tensor<16x16xf32>, f32) outs(%16 : tensor<16x16xi1>) -> tensor<16x16xi1>
%18 = hfusion.select ins(%17, %cst, %8 : tensor<16x16xi1>, f32, tensor<16x16xf32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%19 = hfusion.bitcast ins(%1 : tensor<16x16xf32>) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32>
%20 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<vand>} ins(%19, %c2147483647_i32 : tensor<16x16xi32>, i32) outs(%9 : tensor<16x16xi32>) -> tensor<16x16
%21 = linalg.elemwise_binary {fun = #linalg.binary_fn<add>} ins(%20, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%9 : tensor<16x16xi32>) -> tensor<16x16xi32
%22 = linalg.elemwise_binary {fun = #linalg.binary_fn<min_signed>} ins(%21, %c1_i32 : tensor<16x16xi32>, i32) outs(%21 : tensor<16x16xi32>) -> tensor<16x16xi32>
%23 = linalg.elemwise_binary {fun = #linalg.binary_fn<max_signed>} ins(%22, %c0_i32 : tensor<16x16xi32>, i32) outs(%22 : tensor<16x16xi32>) -> tensor<16x16xi32>
%24 = hfusion.cast {cast = #hfusion.type_fn<cast_signed>, enable_overflow = true, enable_saturate = false, round_mode = #hfusion.round_mode<rint>, unsigned_mode
%25 = hfusion.compare {compare_fn = #hfusion.compare_fn<vne>} ins(%24, %cst_0 : tensor<16x16xf32>, f32) outs(%16 : tensor<16x16xi1>) -> tensor<16x16xi1>
%26 = hfusion.select ins(%25, %cst, %cst_0 : tensor<16x16xi1>, f32, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%27 = hfusion.elemwise_binary {fun = #hfusion.binary_fn<maxf>} ins(%18, %26 : tensor<16x16xf32>, tensor<16x16xf32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x
%28 = arith.index_cast %arg7 : i32 to index
...
```

HFusion & HIVM, mix



And here comes the HIVM

HFusion & HIVM, mix

And here comes the HIVM

```
%1 = hivm.hir.vbrc ins(%cst_0 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
...
%7 = hivm.hir.mmadL1 ins(%4, %5, %true, %c0, %c0, %c0 : tensor<16x16xf16>, tensor<16x16xf16>, i1, index, index, index) outs(%6 : tensor<16x16xf32>) -> tensor<16
%8 = hivm.hir.vmul ins(%7, %arg8 : tensor<16x16xf32>, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%9 = hivm.hir.vadd ins(%8, %arg9 : tensor<16x16xf32>, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%10 = tensor.empty() : tensor<16x16xi32>
%11 = hivm.hir.bitcast %9 : tensor<16x16xf32> -> tensor<16x16xi32>
%12 = tensor.empty() : tensor<16x16xi32>
%13 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%14 = hivm.hir.vand ins(%11, %13 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%10 : tensor<16x16xi32>) -> tensor<16x16xi32>
%15 = hivm.hir.vadd ins(%14, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%10 : tensor<16x16xi32>) -> tensor<16x16xi32>
%16 = hivm.hir.vmin ins(%15, %c1_i32 : tensor<16x16xi32>, i32) outs(%15 : tensor<16x16xi32>) -> tensor<16x16xi32>
%17 = hivm.hir.vmax ins(%16, %c0_i32 : tensor<16x16xi32>, i32) outs(%16 : tensor<16x16xi32>) -> tensor<16x16xi32>
%18 = hivm.hir.vcast ins(%17 : tensor<16x16xi32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%19 = tensor.empty() : tensor<16x16xi1>
%20 = hivm.hir.vcmp ins(%18, %cst_0 : tensor<16x16xf32>, f32) outs(%19 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%21 = hivm.hir.vsel ins(%20, %cst, %9 : tensor<16x16xi1>, f32, tensor<16x16xf32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%22 = hivm.hir.bitcast %1 : tensor<16x16xf32> -> tensor<16x16xi32>
%23 = tensor.empty() : tensor<16x16xi32>
%24 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%23 : tensor<16x16xi32>) -> tensor<16x16xi32>
%25 = hivm.hir.vand ins(%22, %24 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%10 : tensor<16x16xi32>) -> tensor<16x16xi32>
%26 = hivm.hir.vadd ins(%25, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%10 : tensor<16x16xi32>) -> tensor<16x16xi32>
%27 = hivm.hir.vmin ins(%26, %c1_i32 : tensor<16x16xi32>, i32) outs(%26 : tensor<16x16xi32>) -> tensor<16x16xi32>
%28 = hivm.hir.vmax ins(%27, %c0_i32 : tensor<16x16xi32>, i32) outs(%27 : tensor<16x16xi32>) -> tensor<16x16xi32>
%29 = hivm.hir.vcast ins(%28 : tensor<16x16xi32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%30 = hivm.hir.vcmp ins(%29, %cst_0 : tensor<16x16xf32>, f32) outs(%19 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%31 = hivm.hir.vsel ins(%30, %cst, %cst_0 : tensor<16x16xi1>, f32, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%32 = hivm.hir.vmax ins(%21, %31 : tensor<16x16xf32>, tensor<16x16xf32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
```

HFusion & HIVM, mix

And here comes the HIVM. V stands for Vector, MMAD - matmul

```
%1 = hivm.hir.vbrc ins(%cst_0 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
...
%7 = hivm.hir.mmadL1 ins(%4, %5, %true, %c0, %c0, %c0 : tensor<16x16xf16>, tensor<16x16xf16>, i1, index, index, index) outs(%6 : tensor<16x16xf32>) -> tensor<16
%8 = hivm.hir.vmul ins(%7, %arg8 : tensor<16x16xf32>, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%9 = hivm.hir.vadd ins(%8, %arg9 : tensor<16x16xf32>, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%10 = tensor.empty() : tensor<16x16xi32>
%11 = hivm.hir.bitcast %9 : tensor<16x16xf32> -> tensor<16x16xi32>
%12 = tensor.empty() : tensor<16x16xi32>
%13 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%14 = hivm.hir.vand ins(%11, %13 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%10 : tensor<16x16xi32>) -> tensor<16x16xi32>
%15 = hivm.hir.vadd ins(%14, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%10 : tensor<16x16xi32>) -> tensor<16x16xi32>
%16 = hivm.hir.vmin ins(%15, %c1_i32 : tensor<16x16xi32>, i32) outs(%15 : tensor<16x16xi32>) -> tensor<16x16xi32>
%17 = hivm.hir.vmax ins(%16, %c0_i32 : tensor<16x16xi32>, i32) outs(%16 : tensor<16x16xi32>) -> tensor<16x16xi32>
%18 = hivm.hir.vcast ins(%17 : tensor<16x16xi32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%19 = tensor.empty() : tensor<16x16xi1>
%20 = hivm.hir.vcmp ins(%18, %cst_0 : tensor<16x16xf32>, f32) outs(%19 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%21 = hivm.hir.vsel ins(%20, %cst, %9 : tensor<16x16xi1>, f32, tensor<16x16xf32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%22 = hivm.hir.bitcast %1 : tensor<16x16xf32> -> tensor<16x16xi32>
%23 = tensor.empty() : tensor<16x16xi32>
%24 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%23 : tensor<16x16xi32>) -> tensor<16x16xi32>
%25 = hivm.hir.vand ins(%22, %24 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%10 : tensor<16x16xi32>) -> tensor<16x16xi32>
%26 = hivm.hir.vadd ins(%25, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%10 : tensor<16x16xi32>) -> tensor<16x16xi32>
%27 = hivm.hir.vmin ins(%26, %c1_i32 : tensor<16x16xi32>, i32) outs(%26 : tensor<16x16xi32>) -> tensor<16x16xi32>
%28 = hivm.hir.vmax ins(%27, %c0_i32 : tensor<16x16xi32>, i32) outs(%27 : tensor<16x16xi32>) -> tensor<16x16xi32>
%29 = hivm.hir.vcast ins(%28 : tensor<16x16xi32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%30 = hivm.hir.vcmp ins(%29, %cst_0 : tensor<16x16xf32>, f32) outs(%19 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%31 = hivm.hir.vsel ins(%30, %cst, %cst_0 : tensor<16x16xi1>, f32, f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%32 = hivm.hir.vmax ins(%21, %31 : tensor<16x16xf32>, tensor<16x16xf32>) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
```

HFusion & HIVM, mix

Explicit DMA transfers

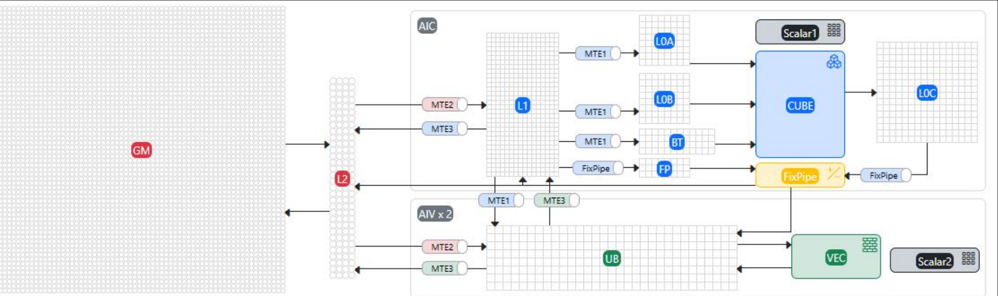
```
%1 = hivm.hir.vbrc ins(%cst_0 : f32) outs(%0 : tensor<16x16xf32>) -> tensor<16x16xf32>
%2 = arith.index_cast %arg5 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%2, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%3 = arith.index_cast %arg6 : i32 to index
%reinterpret_cast_1 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%alloc = memref.alloc() : memref<16x16xf16>
memref.copy %reinterpret_cast, %alloc : memref<16x16xf16, strided<[?, 1]>> to memref<16x16xf16>
%4 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
%alloc_2 = memref.alloc() : memref<16x16xf16>
memref.copy %reinterpret_cast_1, %alloc_2 : memref<16x16xf16, strided<[?, 1]>> to memref<16x16xf16>
%5 = bufferization.to_tensor %alloc_2 restrict writable : memref<16x16xf16>
```

HFusion & HIVM, mix

GM to UB

```
%3 = hivm.hir.vbrc ins(%cst_0 : f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%4 = arith.index_cast %arg5 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%4, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%5 = arith.index_cast %arg6 : i32 to index
%reinterpret_cast_1 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%5, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%alloc = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc : memref<16x16xf16>)
  eviction_policy = <EvictFirst>
%6 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
%alloc_2 = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast_1 : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc_2 : memref<16x16xf16>)
  eviction_policy = <EvictFirst>
%7 = bufferization.to_tensor %alloc_2 restrict writable : memref<16x16xf16>
```

HFusion & HIVM, mix

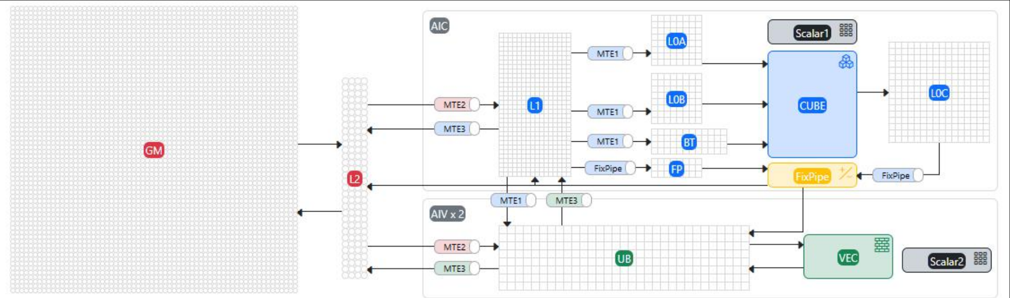


Looking on the architecture and IR, MMAD is still very high-level

```
%alloc_2 = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast_1 : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc_2 : memref<16x16xf16>) eviction_policy = <EvictFirst>
%7 = bufferization.to_tensor %alloc_2 restrict writable : memref<16x16xf16>
%8 = tensor.empty() : tensor<16x16xf32>
%9 = hivm.hir.mmadL1 ins(%6, %7, %true, %c0, %c0, %c0 : tensor<16x16xf16>, tensor<16x16xf16>, i1, index, index, index)
  outs(%8 : tensor<16x16xf32>) -> tensor<16x16xf32>
%10 = hivm.hir.vmul ins(%9, %arg8 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%11 = hivm.hir.vadd ins(%10, %arg9 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
```


Huawei Proprietary - Restricted Distribution

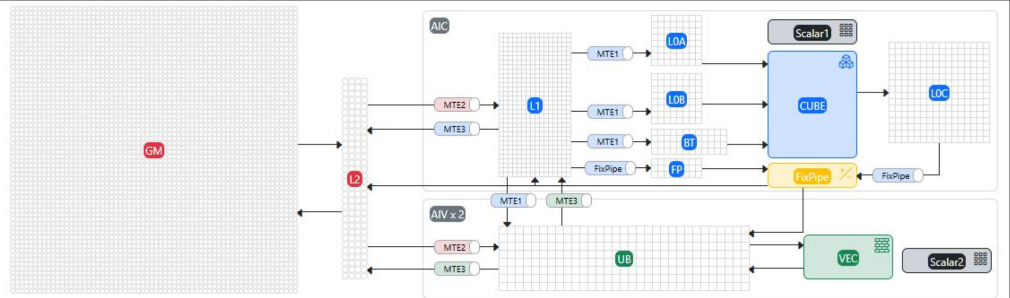
HFusion & HIVM, mix



Fixpipe, made explicit

```
%alloc = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc : memref<16x16xf16>) eviction_policy = <EvictFirst>
%6 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
%alloc_2 = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast_1 : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc_2 : memref<16x16xf16>) eviction_policy = <EvictFirst>
%7 = bufferization.to_tensor %alloc_2 restrict writable : memref<16x16xf16>
%8 = tensor.empty() : tensor<16x16xf32>
● %9 = hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true}
  ins(%6, %7, %true, %c16, %c16, %c16 : tensor<16x16xf16>, tensor<16x16xf16>, i1, index, index, index) outs(%8 : tensor<16x16xf32>) -> tensor<16x16xf32>
%alloc_3 = memref.alloc() : memref<16x16xf32, #hivm.address_space<ub>>
%memspacecast = memref.memory_space_cast %alloc_3 : memref<16x16xf32, #hivm.address_space<ub>> to memref<16x16xf32>
● hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%9 : tensor<16x16xf32>) outs(%alloc_3 : memref<16x16xf32, #hivm.address_space<ub>>)
%10 = bufferization.to_tensor %memspacecast restrict writable : memref<16x16xf32>
%11 = hivm.hir.vmul ins(%10, %arg8 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%12 = hivm.hir.vadd ins(%11, %arg9 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
```

HFusion & HIVM, mix

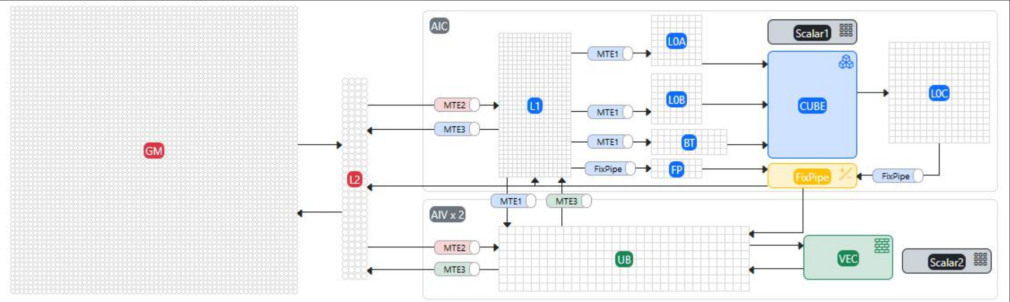


Instructions need to be bound to a specific core type

```
%2 = tensor.empty() : tensor<16x16xf32>
● %3 = hivm.hir.vbrc ins(%cst_0 : f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%4 = arith.index_cast %arg5 : i32 to index
...
%alloc = memref.alloc() : memref<16x16xf16>
● hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc : memref<16x16xf16>) eviction_policy = <EvictFirst>
%6 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
%alloc_2 = memref.alloc() : memref<16x16xf16>
● hivm.hir.load ins(%reinterpret_cast_1 : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc_2 : memref<16x16xf16>) eviction_policy = <EvictFirst>
...
%9 = hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%6, %7, %true, %c16, %c16, %c16 : tensor<16x16xf16>, tensor<16x16xf16>, i1, inde
%alloc_3 = memref.alloc() : memref<16x16xf32, #hivm.address_space<ub>>
%memspacecast = memref.memory_space_cast %alloc_3 : memref<16x16xf32, #hivm.address_space<ub>> to memref<16x16xf32>
● hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%9 : tensor<16x16xf32>) outs(%alloc_3 : memref<16x16xf32, #hivm.address_space<ub>>)
...
● %16 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%15 : tensor<16x16xi32>) -> tensor<16x16xi32>
...
● hivm.hir.store ins(%35 : tensor<16x16xf32>) outs(%reinterpret_cast_4 : memref<16x16xf32, strided<[?, 1]>>)
```


Huawei Proprietary - Restricted Distribution

HFusion & HIVM, mix



Instructions need to be bound to a specific core type

```
%2 = tensor.empty() : tensor<16x16xf32>
%3 = hivm.hir.vbrc {hivm.tcore_type = #hivm.tcore_type<VECTOR>} ins(%cst_0 : f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%4 = arith.index_cast %arg5 : i32 to index
...
%alloc = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc : memref<16x16xf16>)
  {hivm.tcore_type = #hivm.tcore_type<CUBE>} eviction_policy = <EvictFirst>
%6 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
%alloc_2 = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast_1 : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc_2 : memref<16x16xf16>)
  {hivm.tcore_type = #hivm.tcore_type<CUBE>} eviction_policy = <EvictFirst>
...
%9 = hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%6, %7, %true, %c16, %c16, %c16 : tensor<16x16xf16>, tensor<16x16xf16>, i1, ind
%alloc_3 = memref.alloc() : memref<16x16xf32, #hivm.address_space<ub>>
%memspacecast = memref.memory_space_cast %alloc_3 : memref<16x16xf32, #hivm.address_space<ub>> to memref<16x16xf32>
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%9 : tensor<16x16xf32>) outs(%alloc_3 : memref<16x16xf32, #hivm.address_space<ub>>)
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
%10 = bufferization.to_tensor %memspacecast restrict writable : memref<16x16xf32>
hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
...
hivm.hir.store ins(%35 : tensor<16x16xf32>) outs(%reinterpret_cast_4 : memref<16x16xf32, strided<[?, 1]>>)
  {hivm.tcore_type = #hivm.tcore_type<VECTOR>}
```

HFusion & HIVM, mix

```
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
%0 = arith.muli %arg10, %arg11 : i32
%1 = arith.muli %0, %arg12 : i32
annotation.mark %1 {logical_block_num} : i32
%2 = tensor.empty() : tensor<16x16xf32>
%3 = hivm.hir.vbrc {hivm.tcore_type = #hivm.tcore_type<VECTOR>} ins(%cst_0 : f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%4 = arith.index_cast %arg5 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%4, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%5 = arith.index_cast %arg6 : i32 to index
%reinterpret_cast_1 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%5, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%alloc = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc : memref<16x16xf16>) {hivm.tcore_type = #hivm.tcore_type<CUBE>} eviction_pol
%6 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
%alloc_2 = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast_1 : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc_2 : memref<16x16xf16>) {hivm.tcore_type = #hivm.tcore_type<CUBE>} evictior
%7 = bufferization.to_tensor %alloc_2 restrict writable : memref<16x16xf16>
%8 = tensor.empty() : tensor<16x16xf32>
%9 = hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%6, %7, %true, %c16, %c16, %c16 : tensor<16x16xf16>, tensor<16x16xf16>, i1, ind
%alloc_3 = memref.alloc() : memref<16x16xf32, #hivm.address_space<ub>>
%memspacecast = memref.memory_space_cast %alloc_3 : memref<16x16xf32, #hivm.address_space<ub>> to memref<16x16xf32>
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%9 : tensor<16x16xf32>) outs(%alloc_3 : memref<16x16xf32, #hivm.address_space<ub>>)
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
%10 = bufferization.to_tensor %memspacecast restrict writable : memref<16x16xf32>
hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
%11 = hivm.hir.vmul ins(%10, %arg8 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%12 = hivm.hir.vadd ins(%11, %arg9 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
```

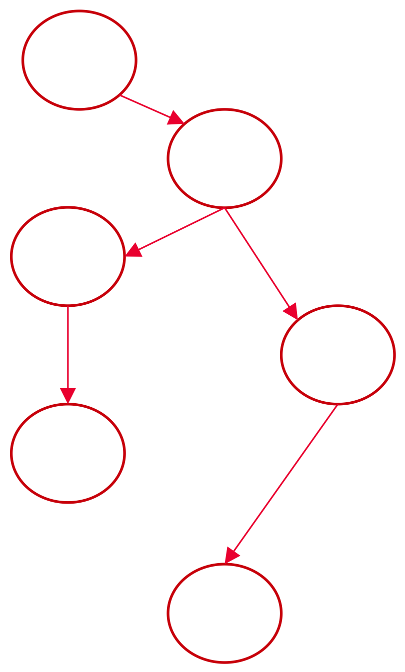
All kernel code together for the last time (1/2)

HFusion & HIVM, mix

```
%13 = tensor.empty() : tensor<16x16xi32>
%14 = hivm.hir.bitcast %12 : tensor<16x16xf32> -> tensor<16x16xi32>
%15 = tensor.empty() : tensor<16x16xi32>
%16 = hivm.hir.vbrc {hivm.tcore_type = #hivm.tcore_type<VECTOR>} ins(%c2147483647_i32 : i32) outs(%15 : tensor<16x16xi32>) -> tensor<16x16xi32>
%17 = hivm.hir.vand ins(%14, %16 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%13 : tensor<16x16xi32>) -> tensor<16x16xi32>
%18 = hivm.hir.vadd ins(%17, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%13 : tensor<16x16xi32>) -> tensor<16x16xi32>
%19 = hivm.hir.vmin ins(%18, %c1_i32 : tensor<16x16xi32>, i32) outs(%18 : tensor<16x16xi32>) -> tensor<16x16xi32>
%20 = hivm.hir.vmax ins(%19, %c0_i32 : tensor<16x16xi32>, i32) outs(%19 : tensor<16x16xi32>) -> tensor<16x16xi32>
%21 = hivm.hir.vcast ins(%20 : tensor<16x16xi32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%22 = tensor.empty() : tensor<16x16xi1>
%23 = hivm.hir.vcmp ins(%21, %cst_0 : tensor<16x16xf32>, f32) outs(%22 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%24 = hivm.hir.vsel ins(%23, %cst, %12 : tensor<16x16xi1>, f32, tensor<16x16xf32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%25 = hivm.hir.bitcast %3 : tensor<16x16xf32> -> tensor<16x16xi32>
%26 = tensor.empty() : tensor<16x16xi32>
%27 = hivm.hir.vbrc {hivm.tcore_type = #hivm.tcore_type<VECTOR>} ins(%c2147483647_i32 : i32) outs(%26 : tensor<16x16xi32>) -> tensor<16x16xi32>
%28 = hivm.hir.vand ins(%25, %27 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%13 : tensor<16x16xi32>) -> tensor<16x16xi32>
%29 = hivm.hir.vadd ins(%28, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%13 : tensor<16x16xi32>) -> tensor<16x16xi32>
%30 = hivm.hir.vmin ins(%29, %c1_i32 : tensor<16x16xi32>, i32) outs(%29 : tensor<16x16xi32>) -> tensor<16x16xi32>
%31 = hivm.hir.vmax ins(%30, %c0_i32 : tensor<16x16xi32>, i32) outs(%30 : tensor<16x16xi32>) -> tensor<16x16xi32>
%32 = hivm.hir.vcast ins(%31 : tensor<16x16xi32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%33 = hivm.hir.vcmp ins(%32, %cst_0 : tensor<16x16xf32>, f32) outs(%22 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%34 = hivm.hir.vsel ins(%33, %cst, %cst_0 : tensor<16x16xi1>, f32, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%35 = hivm.hir.vmax ins(%24, %34 : tensor<16x16xf32>, tensor<16x16xf32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%36 = arith.index_cast %arg7 : i32 to index
%reinterpret_cast_4 = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%36, 1] : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
hivm.hir.store ins(%35 : tensor<16x16xf32>) outs(%reinterpret_cast_4 : memref<16x16xf32, strided<[?, 1]>>) {hivm.tcore_type = #hivm.tcore_type<VECTOR>}
```

All kernel code together for the last time (2/2)

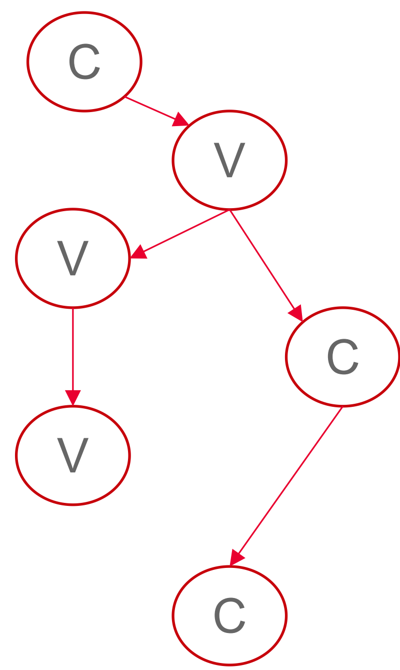
Split mix kernel



Mark Cube and Vector ops

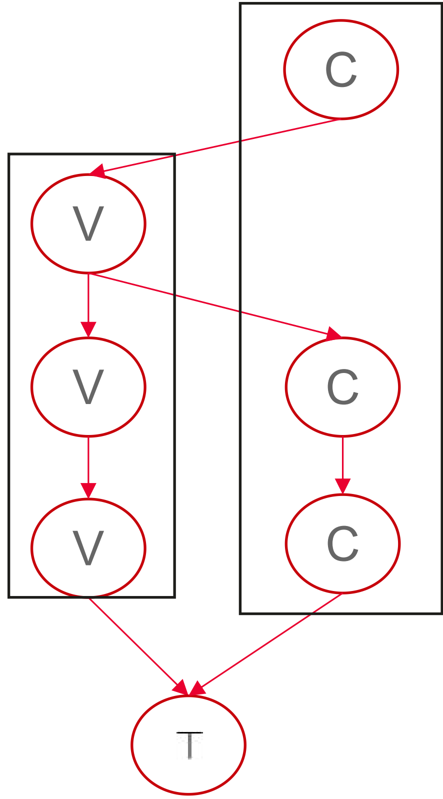
 Huawei Proprietary - Restricted Distribution

Split mix kernel



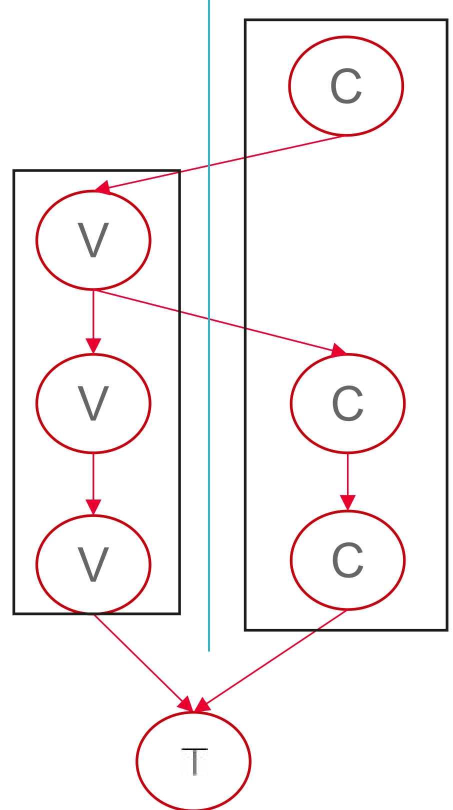
Mark Cube and Vector ops

Split mix kernel



Combine into cube and vector groups

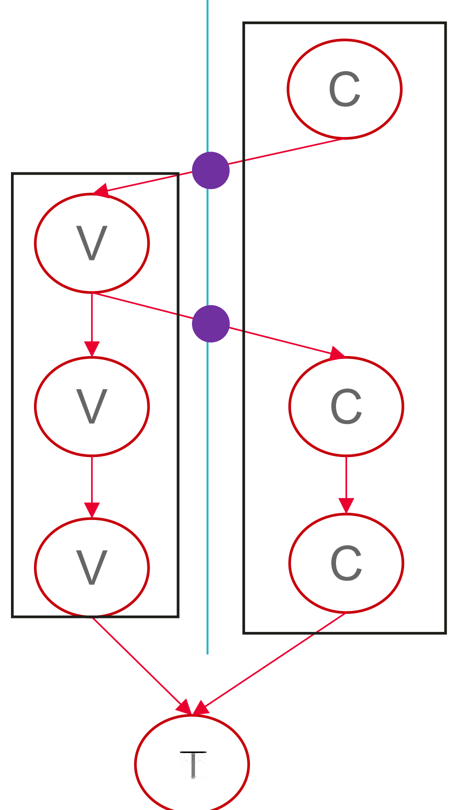
Split mix kernel



Detect required communication points between

AIC and AIV

Split mix kernel

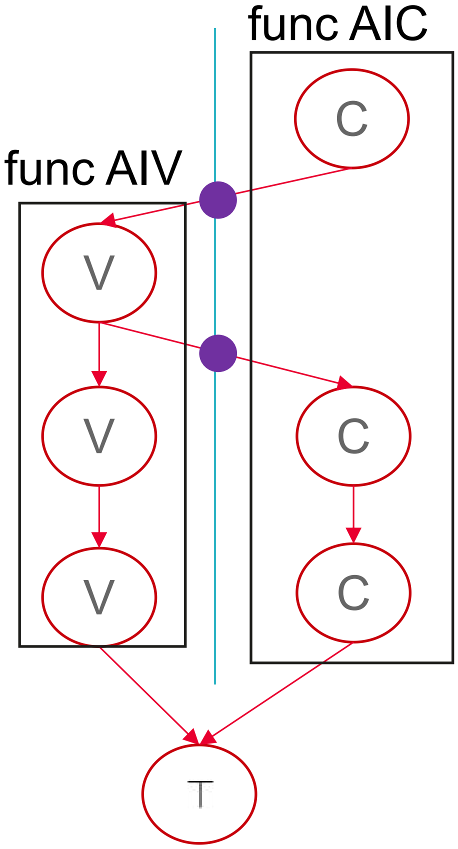


Detect required communication points between AIC and AIV

Adding synchronization

GM
L2
AIC
MTE2
MTE3
L1
MTE1
L0A
MTE1
L0B
MTE1
BT
FixPipe
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

Split mix kernel



Then, split to AIC & AIV functions and insert sync actions later

Split AIC / AIV

```
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
%0 = arith.muli %arg10, %arg11 : i32
%1 = arith.muli %0, %arg12 : i32
annotation.mark %1 {logical_block_num} : i32
%2 = arith.index_cast %arg5 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%2, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%3 = arith.index_cast %arg6 : i32 to index
%reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%alloc = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc : memref<16x16xf16>) eviction_policy = <EvictFirst>
%4 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
%alloc_1 = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast_0 : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc_1 : memref<16x16xf16>) eviction_policy = <EvictFirst>
%5 = bufferization.to_tensor %alloc_1 restrict writable : memref<16x16xf16>
%6 = tensor.empty() : tensor<16x16xf32>
%7 = hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%4, %5, %true, %c16, %c16, %c16 : tensor<16x16xf16>, tensor<16x16xf16>, i1, ind
%alloc_2 = memref.alloc() : memref<16x16xf32, #hivm.address_space<ub>>
annotation.mark %alloc_2 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>} : memref<16x16xf32, #hivm.address_space<ub>>
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%7 : tensor<16x16xf32>) outs(%alloc_2 : memref<16x16xf32, #hivm.address_space<ub>>)
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
```

AIC part

Split AIC / AIV

```
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
%0 = arith.muli %arg10, %arg11 : i32
%1 = arith.muli %0, %arg12 : i32
annotation.mark %1 {logical_block_num} : i32
%2 = tensor.empty() : tensor<16x16xf32>
%3 = hivm.hir.vbrc ins(%cst_0 : f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%4 = arith.index_cast %arg5 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%4, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%5 = arith.index_cast %arg6 : i32 to index
%reinterpret_cast_1 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%5, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
%alloc = memref.alloc() : memref<16x16xf16>
%6 = bufferization.to_tensor %alloc restrict writable : memref<16x16xf16>
%alloc_2 = memref.alloc() : memref<16x16xf16>
%7 = bufferization.to_tensor %alloc_2 restrict writable : memref<16x16xf16>
%8 = tensor.empty() : tensor<16x16xf32>
%alloc_3 = memref.alloc() : memref<16x16xf32, #hivm.address_space<ub>>
annotation.mark %alloc_3 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>} : memref<16x16xf32, #hivm.address_space<ub>>
%memspacecast = memref.memory_space_cast %alloc_3 : memref<16x16xf32, #hivm.address_space<ub>> to memref<16x16xf32>
%9 = bufferization.to_tensor %memspacecast restrict writable : memref<16x16xf32>
hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
```

AIV part (1/2)

Split AIC / AIV

```
%10 = hivm.hir.vmul ins(%9, %arg8 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> te
%11 = hivm.hir.vadd ins(%10, %arg9 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%12 = tensor.empty() : tensor<16x16xi32>
%13 = hivm.hir.bitcast %11 : tensor<16x16xf32> -> tensor<16x16xi32>
%14 = tensor.empty() : tensor<16x16xi32>
%15 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%14 : tensor<16x16xi32>) -> tensor<16x16xi32>
%16 = hivm.hir.vand ins(%13, %15 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%17 = hivm.hir.vadd ins(%16, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%18 = hivm.hir.vmin ins(%17, %c1_i32 : tensor<16x16xi32>, i32) outs(%17 : tensor<16x16xi32>) -> tensor<16x16xi32>
%19 = hivm.hir.vmax ins(%18, %c0_i32 : tensor<16x16xi32>, i32) outs(%18 : tensor<16x16xi32>) -> tensor<16x16xi32>
%20 = hivm.hir.vcast ins(%19 : tensor<16x16xi32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%21 = tensor.empty() : tensor<16x16xi1>
%22 = hivm.hir.vcmp ins(%20, %cst_0 : tensor<16x16xf32>, f32) outs(%21 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%23 = hivm.hir.vsel ins(%22, %cst, %11 : tensor<16x16xi1>, f32, tensor<16x16xf32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%24 = hivm.hir.bitcast %3 : tensor<16x16xf32> -> tensor<16x16xi32>
%25 = tensor.empty() : tensor<16x16xi32>
%26 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%25 : tensor<16x16xi32>) -> tensor<16x16xi32>
%27 = hivm.hir.vand ins(%24, %26 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%28 = hivm.hir.vadd ins(%27, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%29 = hivm.hir.vmin ins(%28, %c1_i32 : tensor<16x16xi32>, i32) outs(%28 : tensor<16x16xi32>) -> tensor<16x16xi32>
%30 = hivm.hir.vmax ins(%29, %c0_i32 : tensor<16x16xi32>, i32) outs(%29 : tensor<16x16xi32>) -> tensor<16x16xi32>
%31 = hivm.hir.vcast ins(%30 : tensor<16x16xi32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%32 = hivm.hir.vcmp ins(%31, %cst_0 : tensor<16x16xf32>, f32) outs(%21 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%33 = hivm.hir.vsel ins(%32, %cst, %cst_0 : tensor<16x16xi1>, f32, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%34 = hivm.hir.vmax ins(%23, %33 : tensor<16x16xf32>, tensor<16x16xf32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%35 = arith.index_cast %arg7 : i32 to index
%reinterpret_cast_4 = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%35, 1] : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
hivm.hir.store ins(%34 : tensor<16x16xf32>) outs(%reinterpret_cast_4 : memref<16x16xf32, strided<[?, 1]>>)
```

AIV part (2/2)

HUAWEI

HFusion & HIVM, pre-vectorization

```
%10 = hivm.hir.vmul ins(%9, %arg8 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> te
%11 = hivm.hir.vadd ins(%10, %arg9 : tensor<16x16xf32>, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%12 = tensor.empty() : tensor<16x16xi32>
%13 = hivm.hir.bitcast %11 : tensor<16x16xf32> -> tensor<16x16xi32>
%14 = tensor.empty() : tensor<16x16xi32>
%15 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%14 : tensor<16x16xi32>) -> tensor<16x16xi32>
%16 = hivm.hir.vand ins(%13, %15 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%17 = hivm.hir.vadd ins(%16, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%18 = hivm.hir.vmin ins(%17, %c1_i32 : tensor<16x16xi32>, i32) outs(%17 : tensor<16x16xi32>) -> tensor<16x16xi32>
%19 = hivm.hir.vmax ins(%18, %c0_i32 : tensor<16x16xi32>, i32) outs(%18 : tensor<16x16xi32>) -> tensor<16x16xi32>
%20 = hivm.hir.vcast ins(%19 : tensor<16x16xi32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%21 = tensor.empty() : tensor<16x16xi1>
%22 = hivm.hir.vcmp ins(%20, %cst_0 : tensor<16x16xf32>, f32) outs(%21 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%23 = hivm.hir.vsel ins(%22, %cst, %11 : tensor<16x16xi1>, f32, tensor<16x16xf32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%24 = hivm.hir.bitcast %3 : tensor<16x16xf32> -> tensor<16x16xi32>
%25 = tensor.empty() : tensor<16x16xi32>
%26 = hivm.hir.vbrc ins(%c2147483647_i32 : i32) outs(%25 : tensor<16x16xi32>) -> tensor<16x16xi32>
%27 = hivm.hir.vand ins(%24, %26 : tensor<16x16xi32>, tensor<16x16xi32>) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%28 = hivm.hir.vadd ins(%27, %c-2139095040_i32 : tensor<16x16xi32>, i32) outs(%12 : tensor<16x16xi32>) -> tensor<16x16xi32>
%29 = hivm.hir.vmin ins(%28, %c1_i32 : tensor<16x16xi32>, i32) outs(%28 : tensor<16x16xi32>) -> tensor<16x16xi32>
%30 = hivm.hir.vmax ins(%29, %c0_i32 : tensor<16x16xi32>, i32) outs(%29 : tensor<16x16xi32>) -> tensor<16x16xi32>
%31 = hivm.hir.vcast ins(%30 : tensor<16x16xi32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%32 = hivm.hir.vcmp ins(%31, %cst_0 : tensor<16x16xf32>, f32) outs(%21 : tensor<16x16xi1>) compare_mode = <ne> -> tensor<16x16xi1>
%33 = hivm.hir.vsel ins(%32, %cst, %cst_0 : tensor<16x16xi1>, f32, f32) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%34 = hivm.hir.vmax ins(%23, %33 : tensor<16x16xf32>, tensor<16x16xf32>) outs(%2 : tensor<16x16xf32>) -> tensor<16x16xf32>
%35 = arith.index_cast %arg7 : i32 to index
%reinterpret_cast_4 = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%35, 1] : memref<?xf32> to memref<16x16xf32, strided<[?, 1]>>
hivm.hir.store ins(%34 : tensor<16x16xf32>) outs(%reinterpret_cast_4 : memref<16x16xf32, strided<[?, 1]>>)
```

AIV part

HUAWEI

HFusion & HIVM, pre-vectorization

```
scf.for %arg13 = %c0 to %c2 step %c1 {
  %18 = affine.apply #map()[%arg13]
  hivm.hir.set_ctrl false at ctrl[60]
  hivm.hir.set_ctrl true at ctrl[48]
  annotation.mark %1 {logical_block_num} : i32
  %alloc = memref.alloc() : memref<8x16xf32, #hivm.address_space<ub>>
  annotation.mark %alloc {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm
  %memspacecast = memref.memory_space_cast %alloc : memref<8x16xf32, #hivm.address_space<ub>> to memref<8x16xf32>
  %19 = bufferization.to_tensor %memspacecast restrict writable : memref<8x16xf32>
  hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
  %20 = linalg.fill ins(%arg8 : f32) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %21 = linalg.mul ins(%19, %20 : tensor<8x16xf32>, tensor<8x16xf32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %22 = linalg.fill ins(%arg9 : f32) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %23 = linalg.add ins(%21, %22 : tensor<8x16xf32>, tensor<8x16xf32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %24 = hivm.hir.bitcast %23 : tensor<8x16xf32> -> tensor<8x16xi32>
  %mapped_1 = linalg.map { arith.andi } ins(%24, %5 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%4 : tensor<8x16xi32>)
  %25 = linalg.add ins(%mapped_1, %7 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%4 : tensor<8x16xi32>) -> tensor<8x16xi32>
  %26 = linalg.min ins(%25, %9 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%25 : tensor<8x16xi32>) -> tensor<8x16xi32>
  %27 = linalg.max ins(%26, %11 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%26 : tensor<8x16xi32>) -> tensor<8x16xi32>
  %28 = hfusion.cast {round_mode = #hfusion.round_mode<rint>} ins(%27 : tensor<8x16xi32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %29 = hfusion.compare {compare_fn = #hfusion.compare_fn<vne>} ins(%28, %cst : tensor<8x16xf32>, f32) outs(%14 : tensor<8x16xi1>) -> tensor<8x16xi1>
  %30 = linalg.select ins(%29, %cst_0, %23 : tensor<8x16xi1>, f32, tensor<8x16xf32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %31 = linalg.max ins(%30, %16 : tensor<8x16xf32>, tensor<8x16xf32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %subview = memref.subview %reinterpret_cast[%18, 0] [8, 16] [1, 1] {to_be_bubbled_slice} : memref<16x16xf32, strided<[?, 1]>> to memref<8x16xf32, strided<[?, 1], of
  hivm.hir.store ins(%31 : tensor<8x16xf32>) outs(%subview : memref<8x16xf32, strided<[?, 1], offset: ?>>) {tiled_op}
} {map_for_to_forall, mapping = [#hivm.sub_block<x>]}
```

Expose the loop structure with basic tiling

HFusion & HIVM, pre-vectorization

```
scf.for %arg13 = %c0 to %c2 step %c1 {
  %18 = affine.apply #map()[%arg13]
  hivm.hir.set_ctrl false at ctrl[60]
  hivm.hir.set_ctrl true at ctrl[48]
  annotation.mark %1 {logical_block_num} : i32
  %alloc = memref.alloc() : memref<8x16xf32, #hivm.address_space<ub>>
  annotation.mark %alloc {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm
  %memspacecast = memref.memory_space_cast %alloc : memref<8x16xf32, #hivm.address_space<ub>> to memref<8x16xf32>
  %19 = bufferization.to_tensor %memspacecast restrict writable : memref<8x16xf32>
  hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
● %20 = linalg.fill ins(%arg8 : f32) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
● %21 = linalg.mul ins(%19, %20 : tensor<8x16xf32>, tensor<8x16xf32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
● %22 = linalg.fill ins(%arg9 : f32) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
● %23 = linalg.add ins(%21, %22 : tensor<8x16xf32>, tensor<8x16xf32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %24 = hivm.hir.bitcast %23 : tensor<8x16xf32> -> tensor<8x16xi32>
  %mapped_1 = linalg.map { arith.andi } ins(%24, %5 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%4 : tensor<8x16xi32>)
● %25 = linalg.add ins(%mapped_1, %7 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%4 : tensor<8x16xi32>) -> tensor<8x16xi32>
● %26 = linalg.min ins(%25, %9 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%25 : tensor<8x16xi32>) -> tensor<8x16xi32>
● %27 = linalg.max ins(%26, %11 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%26 : tensor<8x16xi32>) -> tensor<8x16xi32>
● %28 = hfusion.cast {round_mode = #hfusion.round_mode<rint>} ins(%27 : tensor<8x16xi32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
● %29 = hfusion.compare {compare_fn = #hfusion.compare_fn<vne>} ins(%28, %cst : tensor<8x16xf32>, f32) outs(%14 : tensor<8x16xi1>) -> tensor<8x16xi1>
● %30 = linalg.select ins(%29, %cst_0, %23 : tensor<8x16xi1>, f32, tensor<8x16xf32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
● %31 = linalg.max ins(%30, %16 : tensor<8x16xf32>, tensor<8x16xf32>) outs(%2 : tensor<8x16xf32>) -> tensor<8x16xf32>
  %subview = memref.subview %reinterpret_cast[%18, 0] [8, 16] [1, 1] {to_be_bubbled_slice} : memref<16x16xf32, strided<[?, 1]>> to memref<8x16xf32, strided<[?, 1], of
  hivm.hir.store ins(%31 : tensor<8x16xf32>) outs(%subview : memref<8x16xf32, strided<[?, 1], offset: ?>>) {tiled_op}
} {map_for_to_forall, mapping = [#hivm.sub_block<x>]}
```

Ops can be combined

Huawei Proprietary - Restricted Distribution

HFusion & HIVM, pre-vectorization

```
scf.for %arg13 = %c0 to %c2 step %c1 {
  %9 = affine.apply affine_map<()[s0] -> (s0 * 8)>()[%arg13]
  hivm.hir.set_ctrl false at ctrl[60]
  hivm.hir.set_ctrl true at ctrl[48]
  annotation.mark %1 {logical_block_num} : i32
  %alloc = memref.alloc() : memref<8x16xf32, #hivm.address_space<ub>>
  annotation.mark %alloc {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm
  %memspacecast = memref.memory_space_cast %alloc : memref<8x16xf32, #hivm.address_space<ub>> to memref<8x16xf32>
  %10 = bufferization.to_tensor %memspacecast restrict writable : memref<8x16xf32>
  hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
● %11 = linalg.generic {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>, affine_map<(d0, d1) -> ()>, affine_map<(d0, d1) -> ()>, affine_map<(d0, d1) -> (d0, d1)>], itera
  ^bb0(%in: f32, %in_2: f32, %in_3: f32, %out: f32):
    %14 = arith.mulf %in, %in_2 : f32
    %15 = arith.addf %14, %in_3 : f32
    linalg.yield %15 : f32
  } -> tensor<8x16xf32>
  %12 = hivm.hir.bitcast %11 : tensor<8x16xf32> -> tensor<8x16xi32>
  %mapped_1 = linalg.map { arith.andi } ins(%12, %5 : tensor<8x16xi32>, tensor<8x16xi32>) outs(%4 : tensor<8x16xi32>)
● %13 = linalg.generic {indexing_maps = [affine_map<(d0, d1) -> (d0, d1)>, affine_map<(d0, d1) -> (d0, d1)>, affine_map<(d0, d1) -> (d0, d1)>, affine_map<(d0, d1) -> (d
  ^bb0(%in: i32, %in_2: f32, %in_3: f32, %out: f32):
    %14 = arith.addi %in, %c-2139095040_i32 : i32
    %15 = arith.minsi %14, %c1_i32 : i32
    %16 = arith.maxsi %15, %c0_i32 : i32
    %17 = arith.cmpi ne, %16, %c0_i32 : i32
    %18 = arith.select %17, %cst, %in_2 : f32
    %19 = arith.maximumf %18, %in_3 : f32
    linalg.yield %19 : f32
  } -> tensor<8x16xf32>
  %subview = memref.subview %reinterpret_cast[%9, 0] [8, 16] [1, 1] {to_be_bubbled_slice} : memref<16x16xf32, strided<[?, 1]>> to memref<8x16xf32, strided<[?, 1], off
  hivm.hir.store ins(%13 : tensor<8x16xf32>) outs(%subview : memref<8x16xf32, strided<[?, 1], offset: ?>>) {tiled_op}
} {map_for_to_forall, mapping = [#hivm.sub_block<x>]}
```

High-level linalg generic representation required for vectorization

Vectorization at a glance

Vectorization at a glance

- Objective
  - > Transform scalar/tensor-style generalized linalg computations to MLIR vectorization-friendly fused loops

Vectorization at a glance

- Objective
  - > Transform scalar/tensor-style generalized linalg computations to MLIR vectorization-friendly fused loops
  - > Combine fusing, tiling and vectorization in one pipeline

HFusion & HIVM, vectorization

```
scf.for %arg13 = %c0 to %c2 step %c1 {
  %8 = affine.apply #map1()[%arg13]
  hivm.hir.set_ctrl false at ctrl[60]
  hivm.hir.set_ctrl true at ctrl[48]
  annotation.mark %1 {logical_block_num} : i32
  %alloc = memref.alloc() : memref<8x16xf32, #hivm.address_space<ub>>
  annotation.mark %alloc {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm
  %memspacecast = memref.memory_space_cast %alloc : memref<8x16xf32, #hivm.address_space<ub>> to memref<8x16xf32>
  %9 = bufferization.to_tensor %memspacecast restrict writable : memref<8x16xf32>
  hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
● %10 = scf.execute_region -> tensor<8x16xf32> {
    %13 = func.call @kernel_mix_aiv_outlined_vf_0(%9, %arg8, %arg9, %c0, %c16, %c64, %c8, %c1, %2) {hivm.vector_function, no_inline} : (tensor<8x16xf32>, f32, f32
    scf.yield %13 : tensor<8x16xf32>
  }
  %11 = hivm.hir.bitcast %10 : tensor<8x16xf32> -> tensor<8x16xi32>
● %12 = scf.execute_region -> tensor<8x16xf32> {
    %13 = func.call @kernel_mix_aiv_outlined_vf_1(%11, %4#1, %3, %10, %6, %c0, %c8, %c1, %2) {hivm.vector_function, no_inline} : (tensor<8x16xi32>, tensor<8x16xi
    scf.yield %13 : tensor<8x16xf32>
  }
  %subview = memref.subview %reinterpret_cast[%8, 0] [8, 16] [1, 1] {to_be_bubbled_slice} : memref<16x16xf32, strided<[?, 1]>> to memref<8x16xf32, strided<[?, 1], off
  hivm.hir.store ins(%12 : tensor<8x16xf32>) outs(%subview : memref<8x16xf32, strided<[?, 1], offset: ?>>) {tiled_op}
} {map_for_to_forall, mapping = [#hivm.sub_block<x>]}
```

Vectorize generic blocks and outline to vector functions

Huawei Proprietary - Restricted Distribution

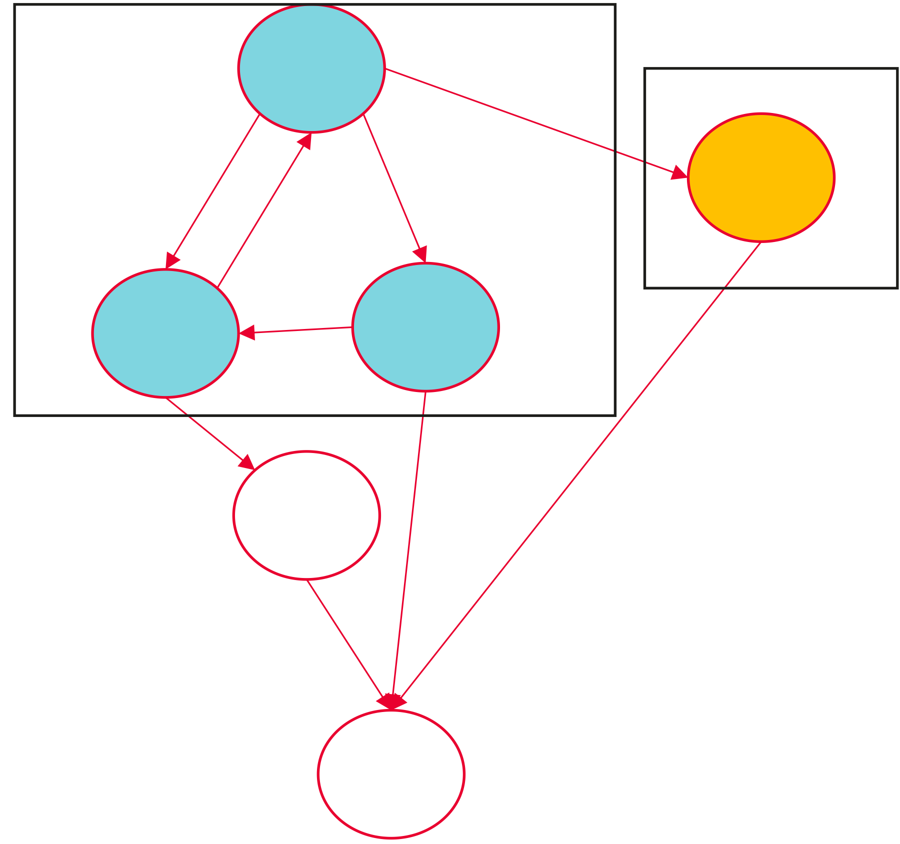
HFusion & HIVM, vectorization

```
func.func @kernel_mix_aiv_outlined_vf_0(...) {
  ...
  %0 = scf.for %arg9 = %c0 to %c8 step %c1 iter_args(%arg10 = %arg8) -> (tensor<8x16xf32>) {
   %1 = scf.for %arg11 = %c0 to %c16 step %c64 iter_args(%arg12 = %arg10) -> (tensor<8x16xf32>) {
    %2 = affine.min #map(%arg11)
    %extracted_slice = tensor.extract_slice %arg0[%arg9, %arg11] [1, %2] [1, 1] : tensor<8x16xf32> to tensor<1x?xf32>
    %extracted_slice_0 = tensor.extract_slice %arg12[%arg9, %arg11] [1, %2] [1, 1] : tensor<8x16xf32> to tensor<1x?xf32>
    %c1_1 = arith.constant 1 : index
    %c1_2 = arith.constant 1 : index
    %dim = tensor.dim %extracted_slice, %c1_2 : tensor<1x?xf32>
    %c0_3 = arith.constant 0 : index
    %cst = arith.constant 0.000000e+00 : f32
    %3 = vector.create_mask %c1_1, %dim : vector<1x64xi1>
    %4 = vector.mask %3 { vector.transfer_read %extracted_slice[%c0_3, %c0_3], %cst {in_bounds = [true, true]} : tensor<1x?xf32>, vector<1x64xf32> } : vector<1x64xi
    %cst_4 = arith.constant 0.000000e+00 : f32
    %5 = vector.mask %3 { vector.transfer_read %extracted_slice_0[%c0_3, %c0_3], %cst_4 {in_bounds = [true, true]} : tensor<1x?xf32>, vector<1x64xf32> } : vector<1x
    %6 = vector.broadcast %arg1 : f32 to vector<1x64xf32>
    %7 = arith.mulf %4, %6 : vector<1x64xf32>
    %8 = vector.broadcast %arg2 : f32 to vector<1x64xf32>
    %9 = arith.addf %7, %8 : vector<1x64xf32>
    %c0_5 = arith.constant 0 : index
    %10 = vector.mask %3 { vector.transfer_write %9, %extracted_slice_0[%c0_5, %c0_5] {in_bounds = [true, true]} : vector<1x64xf32>, tensor<1x?xf32> } : vector<1x64
    %inserted_slice = tensor.insert_slice %10 into %arg12[%arg9, %arg11] [1, %2] [1, 1] : tensor<1x?xf32> into tensor<8x16xf32>
    scf.yield %inserted_slice : tensor<8x16xf32>
   }
   scf.yield %1 : tensor<8x16xf32>
  }
  return %0 : tensor<8x16xf32>
}
```

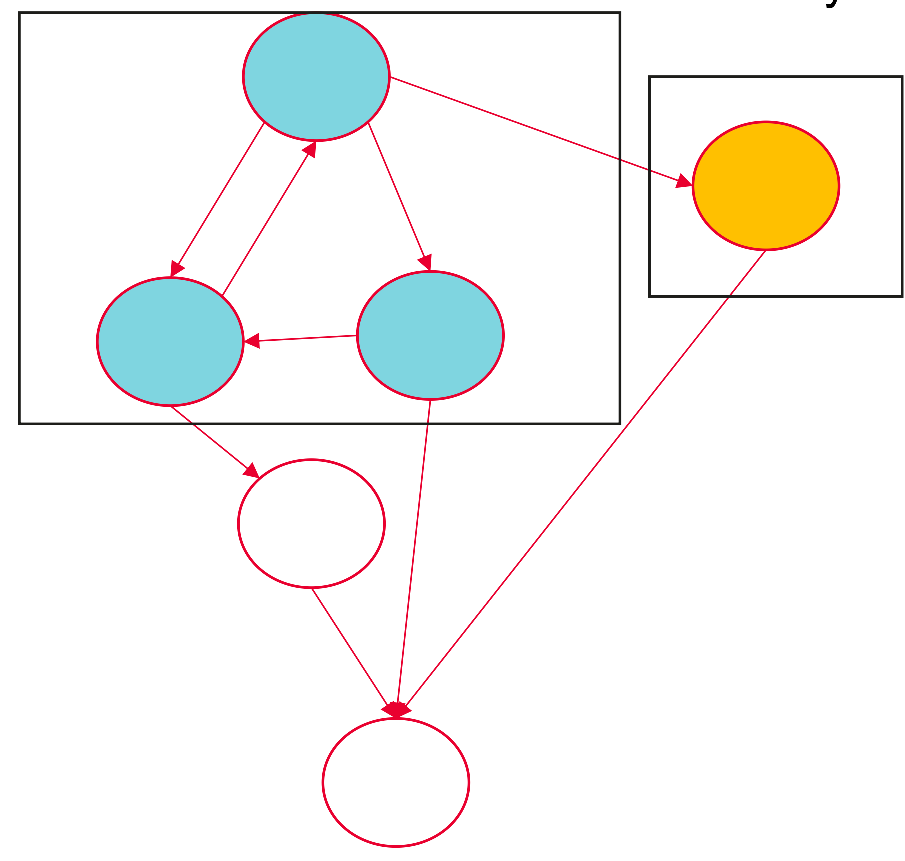
Vectorize generic blocks and outline to vector functions

Vectorization at a glance

Identify vectorizable CFG structures



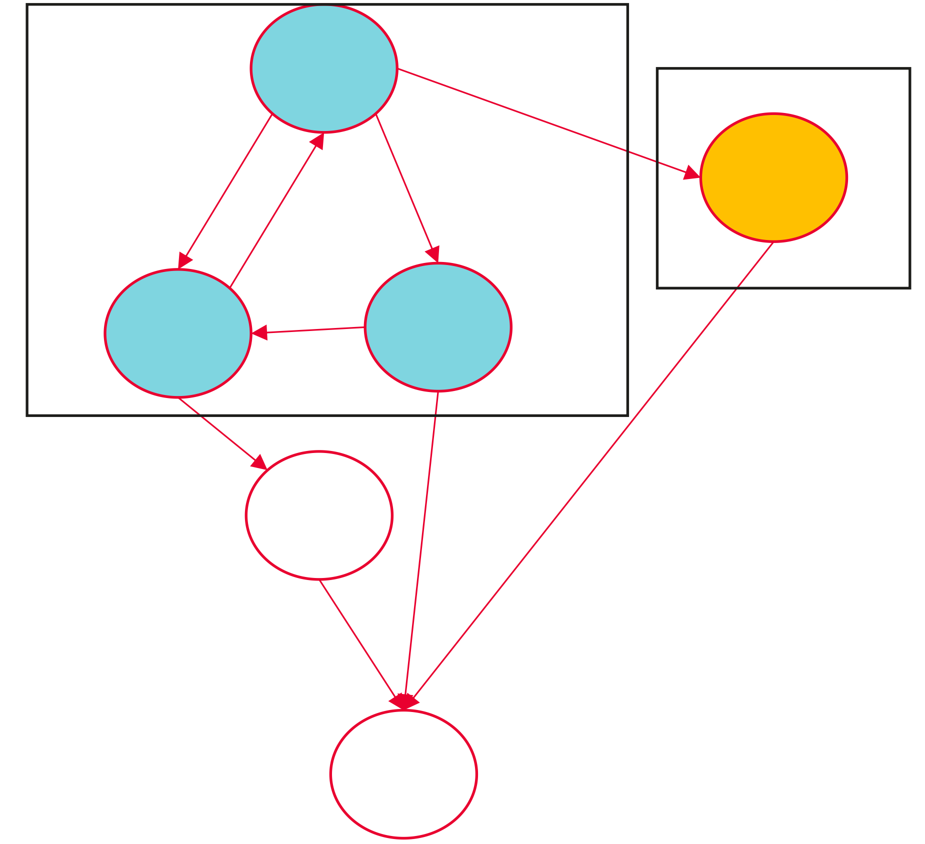
Vectorization at a glance



Identify vectorizable CFG structures

- Collect Linalg / HFusion / HIVM ops

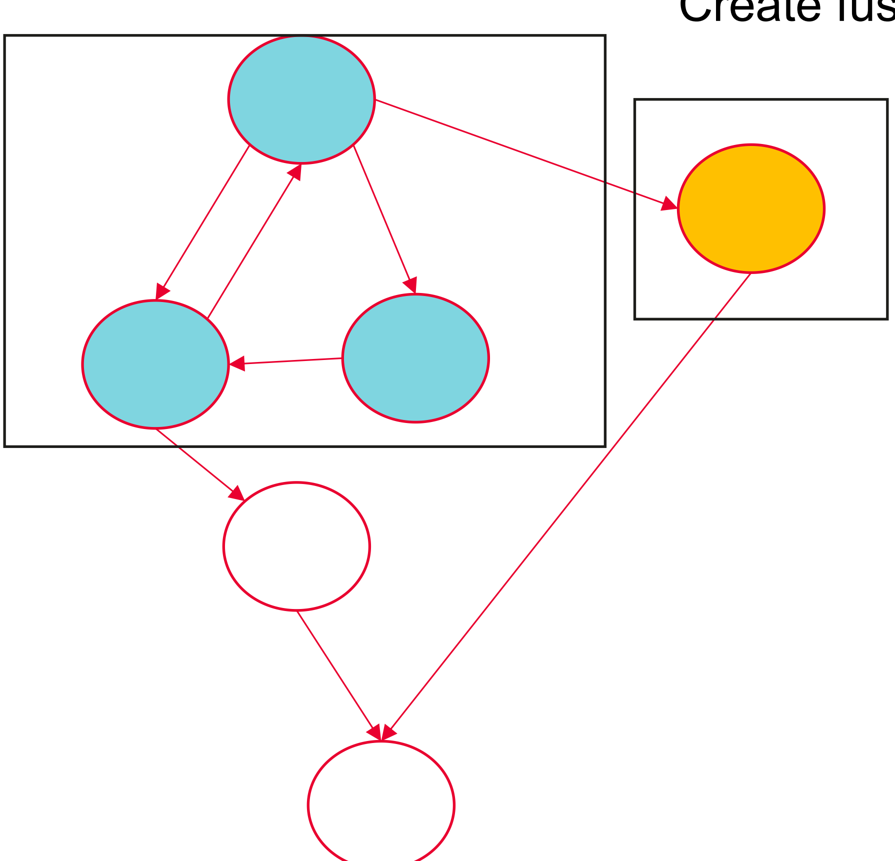
Vectorization at a glance



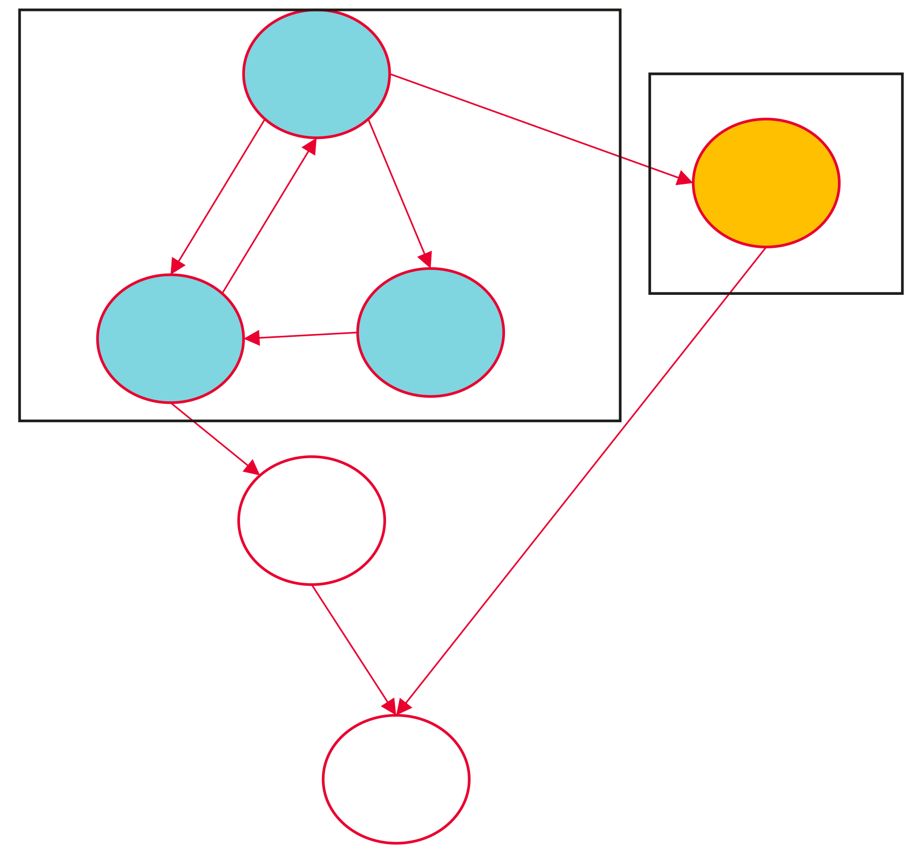
- Collect Linalg / HFusion / HIVM ops
- Attach per-op metadata
  - Shape
  - Loop rank
  - Reduction info
  - Element width

Vectorization at a glance

Create fusion groups

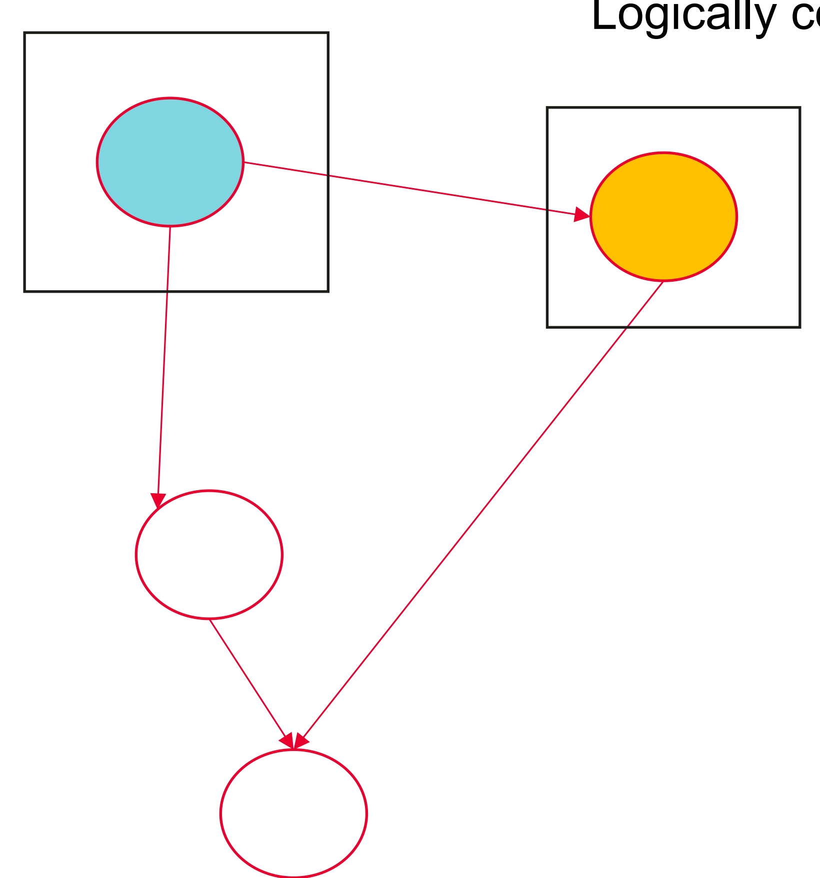


Vectorization at a glance



- Start from leaf nodes
- Merge compatible nodes

Vectorization at a glance



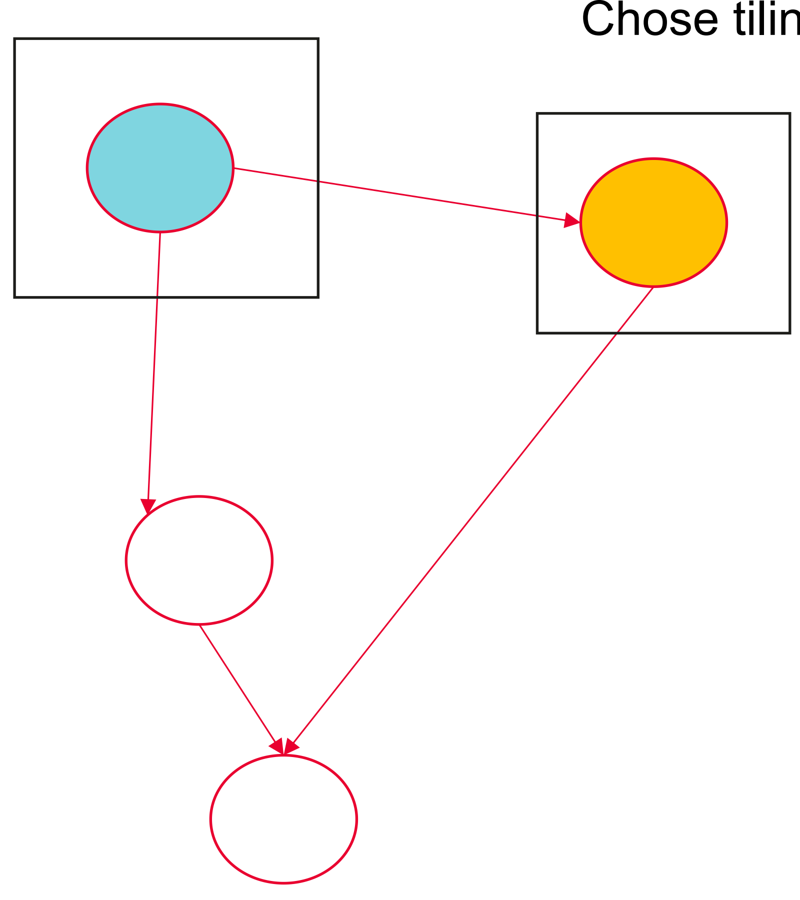
Logically combine, chose tiling strategy

Size is derived from

- Iteration rank
- Tensor shape
- Reduction dims
- Element width
- Target vector length

Huawei Proprietary - Restricted Distribution

Vectorization at a glance



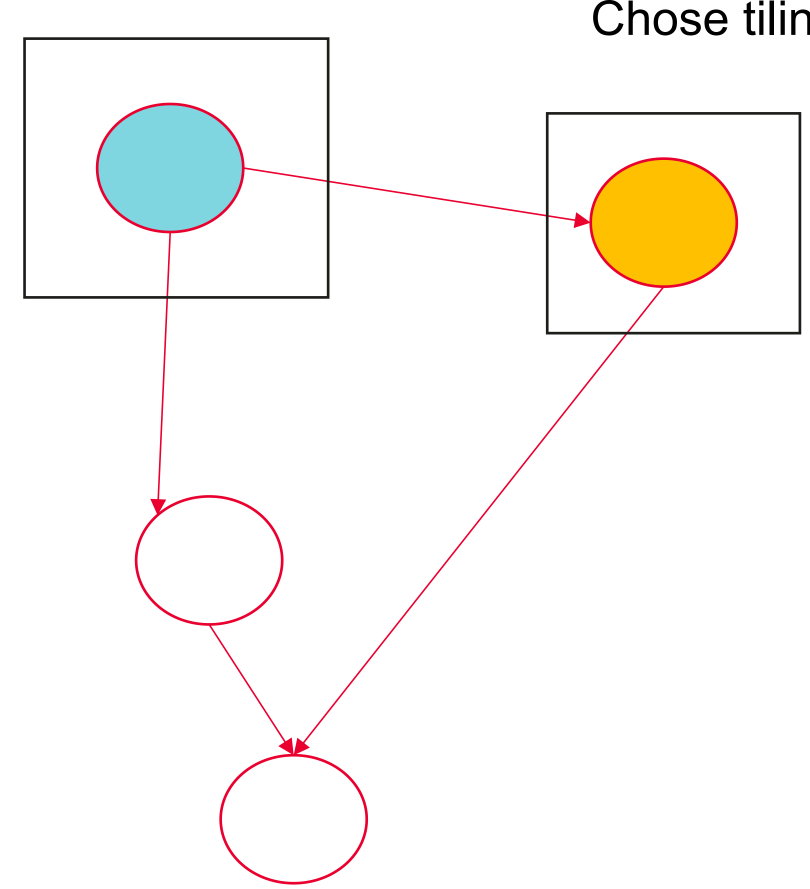
Chose tiling strategy

Tiling behavior:

- Standard case: along the inner axis
- Transpose or reduce case: multi-axis tiling

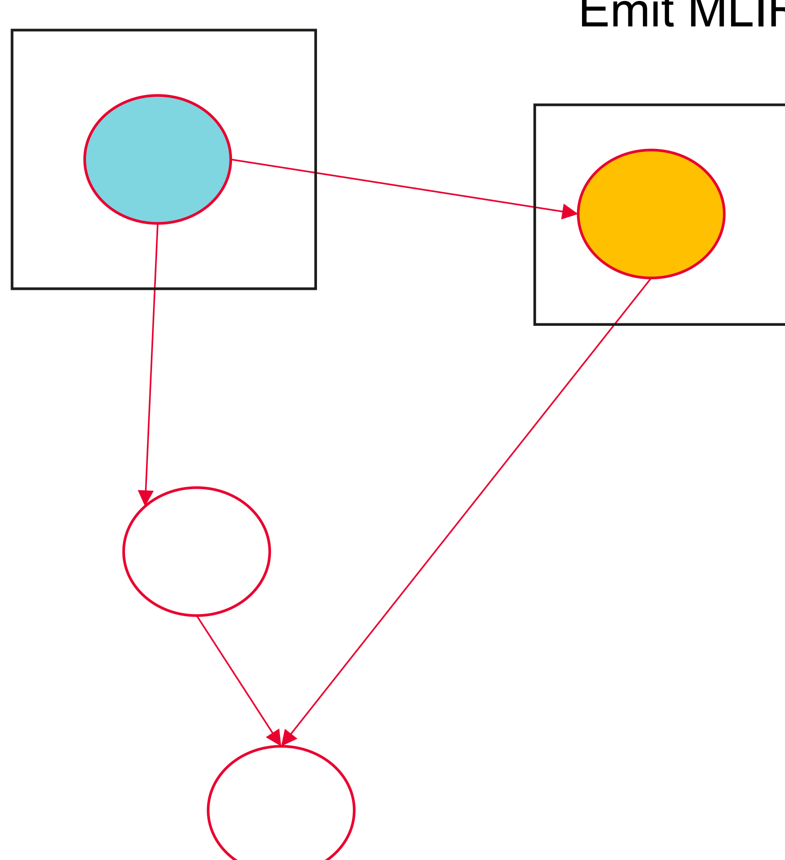
Vectorization at a glance

Chose tiling strategy



Outcome: each fused op gets a tile size

Vectorization at a glance



Emit MLIR transform sequence, apply it and outline vector functions

HFusion & HIVM, vectorization

```
%2 = tensor.empty() : tensor<8x16xf32>
%3 = tensor.empty() : tensor<8x16xi32>
%4:2 = call @kernel_mix_aiv_outlined_vf_2(%2, %3) {hivm.vector_function, no_inline} : (tensor<8x16xf32>, tensor<8x16xi32>) -> (tensor<8x16xf32>, tensor<8x16xi32>)
%5 = hivm.hir.bitcast %4#0 : tensor<8x16xf32> -> tensor<8x16xi32>
%6 = tensor.empty() : tensor<8x16xi32>
%7 = tensor.empty() : tensor<8x16xf32>
%8 = call @kernel_mix_aiv_outlined_vf_3(%5, %4#1, %6, %7) {hivm.vector_function, no_inline} : (tensor<8x16xi32>, tensor<8x16xi32>, tensor<8x16xi32>, tensor<8x16x
%9 = arith.index_cast %arg7 : i32 to index
```

Time for high-level tensor representation is over

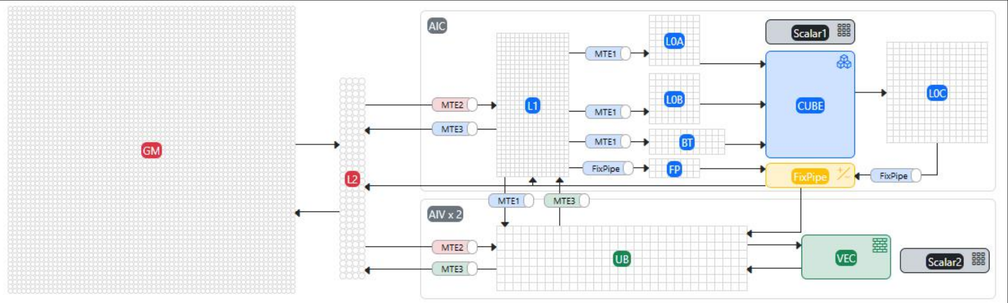
Low-level HIVM

```
● %alloc = memref.alloc() {alignment = 64 : i64} : memref<8x16xf32>
● %alloc_0 = memref.alloc() {alignment = 64 : i64} : memref<8x16xi32>
  %2:2 = call @kernel_mix_aiv_outlined_vf_2(%alloc, %alloc_0) {hivm.vector_function, no_inline} : (memref<8x16xf32>, memref<8x16xi32>) -> (memref<8x16xf32>, memre
  %3 = hivm.hir.bitcast %2#0 : memref<8x16xf32> -> memref<8x16xi32>
  %alloc_1 = memref.alloc() {alignment = 64 : i64} : memref<8x16xf32>
  %4 = call @kernel_mix_aiv_outlined_vf_3(%3, %2#1, %alloc_1) {hivm.vector_function, no_inline} : (memref<8x16xi32>, memref<8x16xi32>, memref<8x16xf32>) -> mem
  %5 = arith.index_cast %arg7 : i32 to index
```

Bufferize tensors

Huawei Proprietary - Restricted Distribution

Low-level HIVM

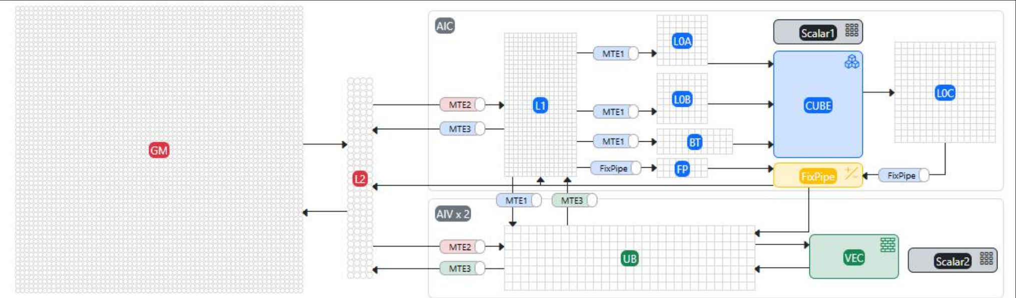



```
%0 = scf.for %arg4 = %c0 to %c8 step %c1 iter_args(%arg5 = %arg3) -> (memref<8x16xf32>) {
  %subview = memref.subview %arg0[%arg4, 0] [1, 16] [1, 1] : memref<8x16xf32> to memref<1x16xf32, strided<[16, 1], offset: ?>>
  %subview_0 = memref.subview %arg5[%arg4, 0] [1, 16] [1, 1] : memref<8x16xf32> to memref<1x16xf32, strided<[16, 1], offset: ?>>
  %1 = vector.constant_mask [1, 16] : vector<1x64xi1>
  %2 = vector.transfer_read %subview[%c0, %c0], %cst, %1 {in_bounds = [true, true]} : memref<1x16xf32, strided<[16, 1], offset: ?>>, vector<1x64xf32>
  %3 = vector.broadcast %arg1 : f32 to vector<1x64xf32>
  %4 = arith.mulf %2, %3 : vector<1x64xf32>
  %5 = vector.broadcast %arg2 : f32 to vector<1x64xf32>
  %6 = arith.addf %4, %5 : vector<1x64xf32>
  vector.transfer_write %6, %subview_0[%c0, %c0], %1 {in_bounds = [true, true]} : vector<1x64xf32>, memref<1x16xf32, strided<[16, 1], offset: ?>>
  %subview_1 = memref.subview %arg5[%arg4, 0] [1, 16] [1, 1] : memref<8x16xf32> to memref<1x16xf32, strided<[16, 1], offset: ?>>
  memref.copy %subview_0, %subview_1 : memref<1x16xf32, strided<[16, 1], offset: ?>> to memref<1x16xf32, strided<[16, 1], offset: ?>>
  scf.yield %arg5 : memref<8x16xf32>
}
```

Cycle low-level canonicalization

 Huawei Proprietary - Restricted Distribution

Low-level HIVM

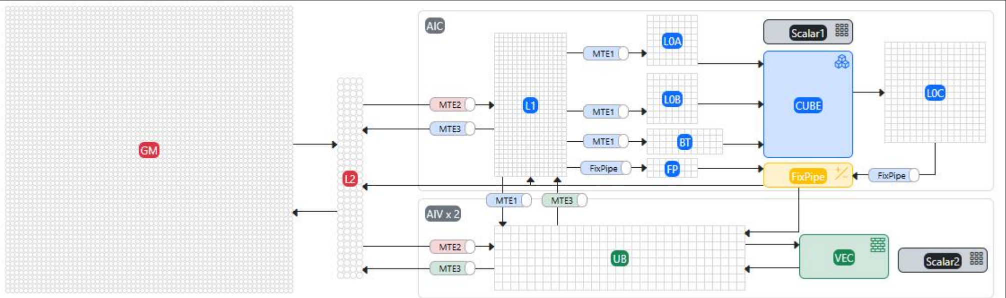



```
scf.for %arg4 = %c0 to %c8 step %c1 {
  %subview = memref.subview %arg0[%arg4, 0] [1, 16] [1, 1] : memref<8x16xf32, #hivm.address_space<ub>> to memref<1x16xf32, strided<[16, 1], offset: ?>, #hivm.addre
  %subview_0 = memref.subview %arg3[%arg4, 0] [1, 16] [1, 1] : memref<8x16xf32, #hivm.address_space<ub>> to memref<1x16xf32, strided<[16, 1], offset: ?>, #hivm.ad
  %0 = vector.constant_mask [1, 16] : vector<1x64xi1>
  %1 = vector.transfer_read %subview[%c0, %c0], %cst, %0 {in_bounds = [true, true]} : memref<1x16xf32, strided<[16, 1], offset: ?>, #hivm.address_space<ub>>, vector<
  %2 = vector.broadcast %arg1 : f32 to vector<1x64xf32>
  %3 = arith.mulf %1, %2 : vector<1x64xf32>
  %4 = vector.broadcast %arg2 : f32 to vector<1x64xf32>
  %5 = arith.addf %3, %4 : vector<1x64xf32>
  vector.transfer_write %5, %subview_0[%c0, %c0], %0 {in_bounds = [true, true]} : vector<1x64xf32>, memref<1x16xf32, strided<[16, 1], offset: ?>, #hivm.address_space<
}
```

Cycle low-level canonicalization

 Huawei Proprietary - Restricted Distribution

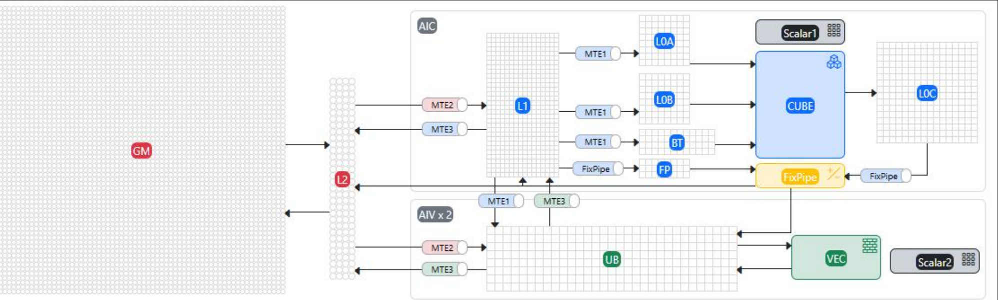
Low-level HIVM




```
%reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf16> to memref<16x16xf16, strided<[?, 1]>>
● %alloc = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc : memref<16x16xf16>) eviction_policy = <EvictFirst>
● %alloc_1 = memref.alloc() : memref<16x16xf16>
hivm.hir.load ins(%reinterpret_cast_0 : memref<16x16xf16, strided<[?, 1]>>) outs(%alloc_1 : memref<16x16xf16>) eviction_policy = <EvictFirst>
● %alloc_2 = memref.alloc() {alignment = 64 : i64} : memref<16x16xf32>
hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%alloc, %alloc_1, %true, %c16, %c16, %c16 : memref<16x16xf16>, memref<16x16xf16>, i1
● %alloc_3 = memref.alloc() : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %alloc_3 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%alloc_2 : memref<16x16xf32>) outs(%alloc_3 : memref<8x16xf32, #hivm.address_space<ub>>) dual_dst_
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
```

Infer memory spaces

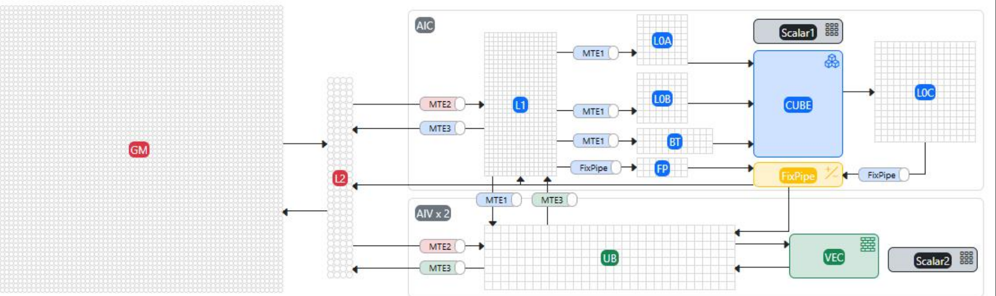
Low-level HIVM




```
%reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf16, #hivm.address_space<gm>> to memref<16x16xf16, st
%alloc = memref.alloc() : memref<16x16xf16, #hivm.address_space<cbuf>>
hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%alloc : memref<16x16xf16, #hivm.address_space<cbuf>>) e
%alloc_1 = memref.alloc() : memref<16x16xf16, #hivm.address_space<cbuf>>
hivm.hir.load ins(%reinterpret_cast_0 : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%alloc_1 : memref<16x16xf16, #hivm.address_space<cbuf
%alloc_2 = memref.alloc() {alignment = 64 : i64} : memref<16x16xf32, #hivm.address_space<cc>>
hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%alloc, %alloc_1, %true, %c16, %c16, %c16 : memref<16x16xf16, #hivm.address_space<c
%alloc_3 = memref.alloc() : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %alloc_3 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%alloc_2 : memref<16x16xf32, #hivm.address_space<cc>>) outs(%alloc_3 : memref<8x16xf32, #hivm.addre
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
```

Infer memory spaces

Low-level HIVM

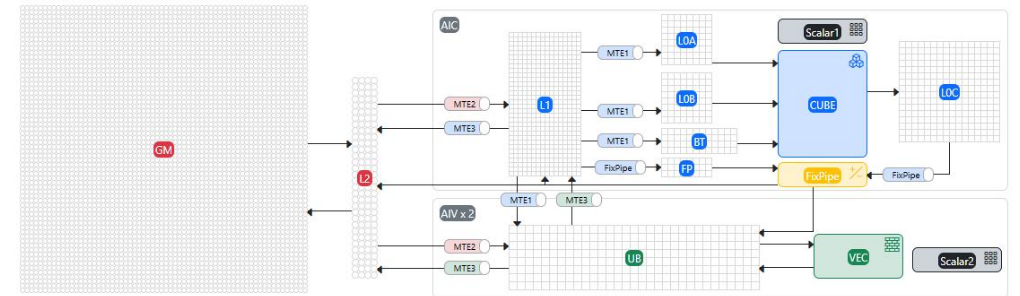



```
%reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf16, #hivm.address_space<gm>> to memref<16x16xf16, st
%alloc = memref.alloc() : memref<16x16xf16, #hivm.address_space<cbuf>>
hivm.hir.load ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%alloc : memref<16x16xf16, #hivm.address_space<cbuf>>) e
%alloc_1 = memref.alloc() : memref<16x16xf16, #hivm.address_space<cbuf>>
hivm.hir.load ins(%reinterpret_cast_0 : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%alloc_1 : memref<16x16xf16, #hivm.address_space<cbuf
%alloc_2 = memref.alloc() {alignment = 64 : i64} : memref<16x16xf32, #hivm.address_space<cc>>
hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%alloc, %alloc_1, %true, %c16, %c16, %c16 : memref<16x16xf16, #hivm.address_space<c
%alloc_3 = memref.alloc() : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %alloc_3 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%alloc_2 : memref<16x16xf32, #hivm.address_space<cc>>) outs(%alloc_3 : memref<8x16xf32, #hivm.addr
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
```

Plan memory (AIC)

Huawei Proprietary - Restricted Distribution

Low-level HIVM

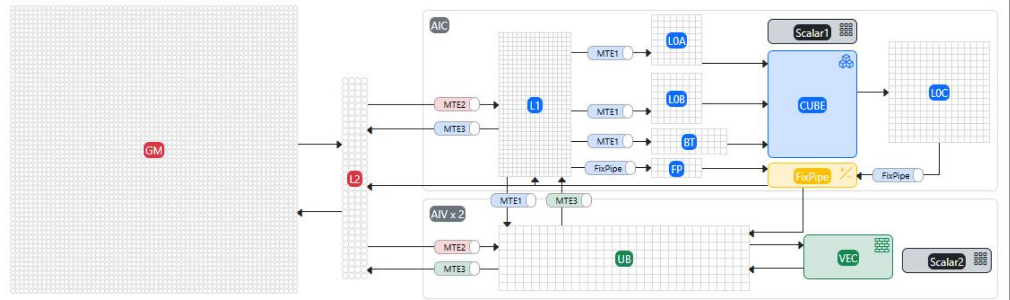



```
%c1536_i64 = arith.constant 1536 : i64
%c512_i64 = arith.constant 512 : i64
%c0_i64 = arith.constant 0 : i64
...
%reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf16, #hivm.address_space<gm>> to memref<16x16xf16, s
%4 = hivm.hir.pointer_cast(%c0_i64) : memref<1x1x16x16xf16, #hivm.address_space<cbuf>>
%cast = memref.cast %4 : memref<1x1x16x16xf16, #hivm.address_space<cbuf>> to memref<?x?x?x?xf16, #hivm.address_space<cbuf>>
hivm.hir.nd2nz {dst_continuous} ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%cast : memref<?x?x?x?xf16, #hivm.add
%5 = hivm.hir.pointer_cast(%c512_i64) : memref<1x1x16x16xf16, #hivm.address_space<cbuf>>
%cast_1 = memref.cast %5 : memref<1x1x16x16xf16, #hivm.address_space<cbuf>> to memref<?x?x?x?xf16, #hivm.address_space<cbuf>>
hivm.hir.nd2nz {dst_continuous} ins(%reinterpret_cast_0 : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%cast_1 : memref<?x?x?x?xf16, #hivm
%6 = hivm.hir.pointer_cast(%c0_i64) : memref<1x1x16x16xf32, #hivm.address_space<cc>>
%cast_2 = memref.cast %6 : memref<1x1x16x16xf32, #hivm.address_space<cc>> to memref<?x?x?x?xf32, #hivm.address_space<cc>>
hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%cast, %cast_1, %true, %c16, %c16, %c16 : memref<?x?x?x?xf16, #hivm.address_space<
%7 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %7 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm.add
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%cast_2 : memref<?x?x?x?xf32, #hivm.address_space<cc>>) outs(%7 : memref<8x16xf32, #hivm.address
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
```

Plan memory (AIC)

Huawei Proprietary - Restricted Distribution

Low-level HIVM




```
%alloc = memref.alloc() {alignment = 64 : i64} : memref<8x16xf32, #hivm.address_space<ub>>
%alloc_0 = memref.alloc() {alignment = 64 : i64} : memref<8x16xi32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_2(%alloc, %alloc_0) {hivm.vector_function, no_inline} : (memref<8x16xf32, #hivm.address_space<ub>>, memref<8x16xi32, #hivm.add
%2 = hivm.hir.bitcast %alloc : memref<8x16xf32, #hivm.address_space<ub>> -> memref<8x16xi32, #hivm.address_space<ub>>
%alloc_1 = memref.alloc() {alignment = 64 : i64} : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_3(%2, %alloc_0, %alloc_1) {hivm.vector_function, no_inline} : (memref<8x16xi32, #hivm.address_space<ub>>, memref<8x16xi32, #hi
%3 = arith.index_cast %arg7 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf32, #hivm.address_space<gm>> to memref<16x16xf32, stri
%4 = hivm.hir.get_sub_block_idx -> i64
%5 = arith.index_cast %4 : i64 to index
%6 = affine.apply #map()[%5]
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
annotation.mark %1 {logical_block_num} : i32
%alloc_2 = memref.alloc() : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %alloc_2 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hiv
%memspacecast = memref.memory_space_cast %alloc_2 : memref<8x16xf32, #hivm.address_space<ub>> to memref<8x16xf32, #hivm.address_space<ub>>
hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
%alloc_3 = memref.alloc() {alignment = 64 : i64} : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_0(%memspacecast, %arg8, %arg9, %alloc_3) {hivm.vector_function, no_inline} : (memref<8x16xf32, #hivm.address_space<ub>>, f32,
%7 = hivm.hir.bitcast %alloc_3 : memref<8x16xf32, #hivm.address_space<ub>> -> memref<8x16xi32, #hivm.address_space<ub>>
%alloc_4 = memref.alloc() {alignment = 64 : i64} : memref<8x16xf32, #hivm.address_space<ub>>
```

Plan memory (AIV)

Huawei Proprietary - Restricted Distribution

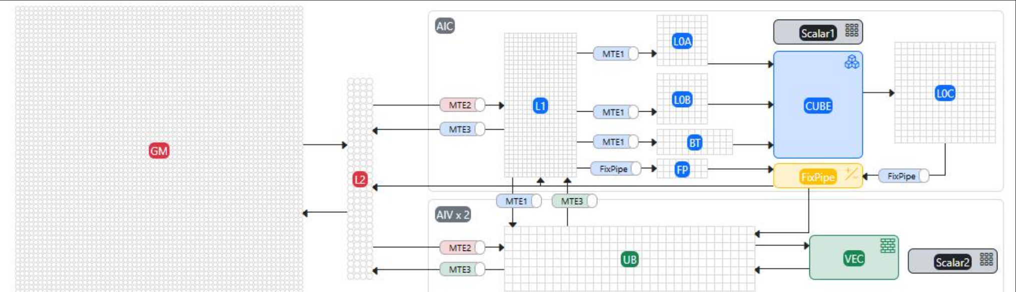
Low-level HIVM

```
%c1536_i64 = arith.constant 1536 : i64
%c1024_i64 = arith.constant 1024 : i64
%c512_i64 = arith.constant 512 : i64
%c0_i64 = arith.constant 0 : i64
...
%2 = hivm.hir.pointer_cast(%c0_i64) : memref<8x16xf32, #hivm.address_space<ub>>
%3 = hivm.hir.pointer_cast(%c512_i64) : memref<8x16xi32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_2(%2, %3) {hivm.vector_function, no_inline} : (memref<8x16xf32, #hivm.address_space<ub>>, memref<8x16xi32, #hivm.address_spa
%4 = hivm.hir.bitcast %2 : memref<8x16xf32, #hivm.address_space<ub>> -> memref<8x16xi32, #hivm.address_space<ub>>
%5 = hivm.hir.pointer_cast(%c1024_i64) : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_3(%4, %3, %5) {hivm.vector_function, no_inline} : (memref<8x16xi32, #hivm.address_space<ub>>, memref<8x16xi32, #hivm.address_
%6 = arith.index_cast %arg7 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%6, 1] : memref<?xf32, #hivm.address_space<gm>> to memref<16x16xf32, stri
%7 = hivm.hir.get_sub_block_idx -> i64
%8 = arith.index_cast %7 : i64 to index
%9 = affine.apply #map()[%8]
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
annotation.mark %1 {logical_block_num} : i32
%10 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %10 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm.ad
hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
%11 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_0(%10, %arg8, %arg9, %11) {hivm.vector_function, no_inline} : (memref<8x16xf32, #hivm.address_space<ub>>, f32, f32, memref<8x1
%12 = hivm.hir.bitcast %11 : memref<8x16xf32, #hivm.address_space<ub>> -> memref<8x16xi32, #hivm.address_space<ub>>
%13 = hivm.hir.pointer_cast(%c512_i64) : memref<8x16xf32, #hivm.address_space<ub>>
```

Plan memory (AIV)

Huawei Proprietary - Restricted Distribution

Low-level HIVM




```
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
%0 = arith.muli %arg10, %arg11 : i32
%1 = arith.muli %0, %arg12 : i32
annotation.mark %1 {logical_block_num} : i32
%2 = arith.index_cast %arg5 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%2, 1] : memref<?xf16, #hivm.address_space<gm>> to memref<16x16xf16, stri
%3 = arith.index_cast %arg6 : i32 to index
%reinterpret_cast_0 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf16, #hivm.address_space<gm>> to memref<16x16xf16, s
%4 = hivm.hir.pointer_cast(%c0_i64) : memref<1x1x16x16xf16, #hivm.address_space<cbuf>>
%cast = memref.cast %4 : memref<1x1x16x16xf16, #hivm.address_space<cbuf>> to memref<?x?x?x?xf16, #hivm.address_space<cbuf>>
hivm.hir.nd2nz {dst_continuous} ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%cast : memref<?x?x?x?xf16, #hivm.add
%5 = hivm.hir.pointer_cast(%c512_i64) : memref<1x1x16x16xf16, #hivm.address_space<cbuf>>
%cast_1 = memref.cast %5 : memref<1x1x16x16xf16, #hivm.address_space<cbuf>> to memref<?x?x?x?xf16, #hivm.address_space<cbuf>>
hivm.hir.nd2nz {dst_continuous} ins(%reinterpret_cast_0 : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%cast_1 : memref<?x?x?x?xf16, #hivm
%6 = hivm.hir.pointer_cast(%c0_i64) : memref<1x1x16x16xf32, #hivm.address_space<cc>>
%cast_2 = memref.cast %6 : memref<1x1x16x16xf32, #hivm.address_space<cc>> to memref<?x?x?x?xf32, #hivm.address_space<cc>>
hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%cast, %cast_1, %true, %c16, %c16, %c16 : memref<?x?x?x?xf16, #hivm.address_space
%7 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %7 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm.add
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%cast_2 : memref<?x?x?x?xf32, #hivm.address_space<cc>>) outs(%7 : memref<8x16xf32, #hivm.address
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
```

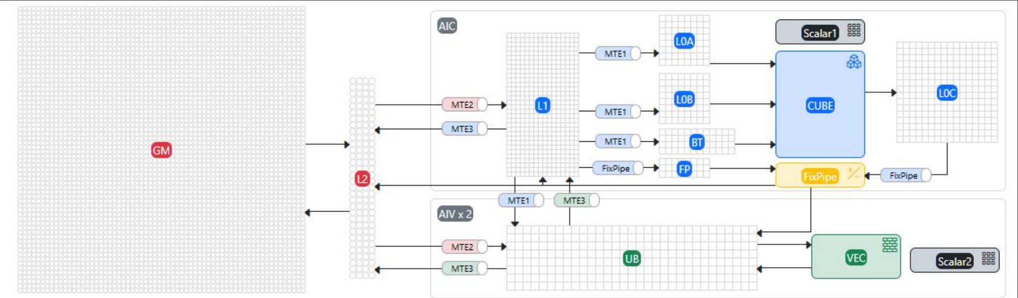
Inject Sync (AIC)

Low-level HIVM

```
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
%0 = arith.muli %arg10, %arg11 : i32
%1 = arith.muli %0, %arg12 : i32
annotation.mark %1 {logical_block_num} : i32
%2 = arith.index_cast %arg5 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg2 to offset: [0], sizes: [16, 16], strides: [%2, 1] : memref<?xf16, #hivm.address_space<gm>> to memref<16x16xf16, stri
%3 = arith.index_cast %arg6 : i32 to index
%reinterpret_cast_1 = memref.reinterpret_cast %arg3 to offset: [0], sizes: [16, 16], strides: [%3, 1] : memref<?xf16, #hivm.address_space<gm>> to memref<16x16xf16, s
%4 = hivm.hir.pointer_cast(%c0_i64_0) : memref<1x1x16x16xf16, #hivm.address_space<cbuf>>
%cast = memref.cast %4 : memref<1x1x16x16xf16, #hivm.address_space<cbuf>> to memref<?x?x?x?xf16, #hivm.address_space<cbuf>>
hivm.hir.set_flag[<PIPE_M>, <PIPE_MTE1>, <EVENT_ID0>]
hivm.hir.set_flag[<PIPE_M>, <PIPE_MTE1>, <EVENT_ID1>]
hivm.hir.nd2nz {dst_continuous} ins(%reinterpret_cast : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%cast : memref<?x?x?x?xf16, #hivm.add
hivm.hir.set_flag[<PIPE_MTE2>, <PIPE_MTE1>, <EVENT_ID1>]
%5 = hivm.hir.pointer_cast(%c512_i64) : memref<1x1x16x16xf16, #hivm.address_space<cbuf>>
%cast_2 = memref.cast %5 : memref<1x1x16x16xf16, #hivm.address_space<cbuf>> to memref<?x?x?x?xf16, #hivm.address_space<cbuf>>
hivm.hir.nd2nz {dst_continuous} ins(%reinterpret_cast_1 : memref<16x16xf16, strided<[?, 1]>, #hivm.address_space<gm>>) outs(%cast_2 : memref<?x?x?x?xf16, #hivm
hivm.hir.set_flag[<PIPE_MTE2>, <PIPE_MTE1>, <EVENT_ID0>]
%6 = hivm.hir.pointer_cast(%c0_i64_0) : memref<1x1x16x16xf32, #hivm.address_space<cc>>
%cast_3 = memref.cast %6 : memref<1x1x16x16xf32, #hivm.address_space<cc>> to memref<?x?x?x?xf32, #hivm.address_space<cc>>
hivm.hir.mmadL1 {already_set_real_mkn, fixpipe_already_inserted = true} ins(%cast, %cast_2, %true, %c16, %c16, %c16 : memref<?x?x?x?xf16, #hivm.address_space
hivm.hir.set_flag[<PIPE_M>, <PIPE_FIX>, <EVENT_ID0>]
%7 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %7 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm.add
hivm.hir.wait_flag[<PIPE_M>, <PIPE_FIX>, <EVENT_ID0>]
hivm.hir.fixpipe {dma_mode = #hivm.dma_mode<nz2nd>} ins(%cast_3 : memref<?x?x?x?xf32, #hivm.address_space<cc>>) outs(%7 : memref<8x16xf32, #hivm.address
hivm.hir.wait_flag[<PIPE_M>, <PIPE_MTE1>, <EVENT_ID0>]
hivm.hir.wait_flag[<PIPE_M>, <PIPE_MTE1>, <EVENT_ID1>]
hivm.hir.pipe_barrier[<PIPE_ALL>]
hivm.hir.sync_block_set[<CUBE>, <PIPE_FIX>, <PIPE_S>] flag = 0
```

Inject Sync (AIC)

Low-level HIVM



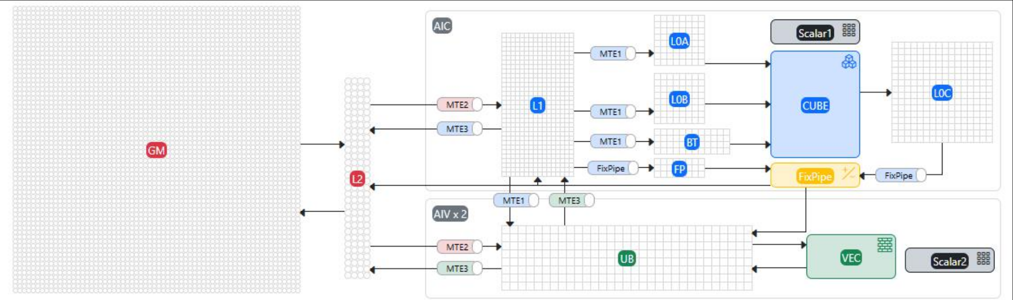

```
%2 = hivm.hir.pointer_cast(%c0_i64) : memref<8x16xf32, #hivm.address_space<ub>>
%3 = hivm.hir.pointer_cast(%c512_i64) : memref<8x16xi32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_2(%2, %3) {hivm.vector_function, no_inline} : (memref<8x16xf32, #hivm.address_space<ub>>, memref<8x16xi32, #hivm.address_space
%4 = hivm.hir.bitcast %2 : memref<8x16xf32, #hivm.address_space<ub>> -> memref<8x16xi32, #hivm.address_space<ub>>
%5 = hivm.hir.pointer_cast(%c1024_i64) : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_3(%4, %3, %5) {hivm.vector_function, no_inline} : (memref<8x16xi32, #hivm.address_space<ub>>, memref<8x16xi32, #hivm.address_s
%6 = arith.index_cast %arg7 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%6, 1] : memref<?xf32, #hivm.address_space<gm>> to memref<16x16xf32, stride
%7 = hivm.hir.get_sub_block_idx -> i64
%8 = arith.index_cast %7 : i64 to index
%9 = affine.apply #map()[%8]
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
annotation.mark %1 {logical_block_num} : i32
%10 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %10 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm.add
hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
%11 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_0(%10, %arg8, %arg9, %11) {hivm.vector_function, no_inline} : (memref<8x16xf32, #hivm.address_space<ub>>, f32, f32, memref<8x16
%12 = hivm.hir.bitcast %11 : memref<8x16xf32, #hivm.address_space<ub>> -> memref<8x16xi32, #hivm.address_space<ub>>
%13 = hivm.hir.pointer_cast(%c512_i64) : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_1(%12, %3, %11, %5, %13) {hivm.vector_function, no_inline} : (memref<8x16xi32, #hivm.address_space<ub>>, memref<8x16xi32, #hiv
%subview = memref.subview %reinterpret_cast[%9, 0] [8, 16] [1, 1] {to_be_bubbled_slice} : memref<16x16xf32, strided<[?, 1]>, #hivm.address_space<gm>> to memref<8
hivm.hir.store ins(%13 : memref<8x16xf32, #hivm.address_space<ub>>) outs(%subview : memref<8x16xf32, strided<[?, 1], offset: ?>, #hivm.address_space<gm>>) {tiled_
```


Huawei Proprietary - Restricted Distribution

Inject Sync (AIV)

HUAWEI

Low-level HIVM




```
%2 = hivm.hir.pointer_cast(%c0_i64) : memref<8x16xf32, #hivm.address_space<ub>>
%3 = hivm.hir.pointer_cast(%c512_i64) : memref<8x16xi32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_2(%2, %3) {hivm.vector_function, no_inline} : (memref<8x16xf32, #hivm.address_space<ub>>, memref<8x16xi32, #hivm.address_spa
%4 = hivm.hir.bitcast %2 : memref<8x16xf32, #hivm.address_space<ub>> -> memref<8x16xi32, #hivm.address_space<ub>>
%5 = hivm.hir.pointer_cast(%c1024_i64) : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_3(%4, %3, %5) {hivm.vector_function, no_inline} : (memref<8x16xi32, #hivm.address_space<ub>>, memref<8x16xi32, #hivm.address
%6 = arith.index_cast %arg7 : i32 to index
%reinterpret_cast = memref.reinterpret_cast %arg4 to offset: [0], sizes: [16, 16], strides: [%6, 1] : memref<?xf32, #hivm.address_space<gm>> to memref<16x16xf32, str
%7 = hivm.hir.get_sub_block_idx -> i64
%8 = arith.index_cast %7 : i64 to index
%9 = affine.apply affine_map<()[s0] -> (s0 * 8)>()[%8]
hivm.hir.set_ctrl false at ctrl[60]
hivm.hir.set_ctrl true at ctrl[48]
annotation.mark %1 {logical_block_num} : i32
%10 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
annotation.mark %10 {effects = ["write", "read"], hivm.tightly_coupled_buffer = #hivm.tightly_coupled_buffer<0>, hivm.tiling_dim = 0 : index} : memref<8x16xf32, #hivm.a
hivm.hir.sync_block_wait[<VECTOR>, <PIPE_FIX>, <PIPE_S>] flag = 0
%11 = hivm.hir.pointer_cast(%c1536_i64) : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_0(%10, %arg8, %arg9, %11) {hivm.vector_function, no_inline} : (memref<8x16xf32, #hivm.address_space<ub>>, f32, f32, memref<8x
%12 = hivm.hir.bitcast %11 : memref<8x16xf32, #hivm.address_space<ub>> -> memref<8x16xi32, #hivm.address_space<ub>>
%13 = hivm.hir.pointer_cast(%c512_i64) : memref<8x16xf32, #hivm.address_space<ub>>
call @kernel_mix_aiv_outlined_vf_1(%12, %3, %11, %5, %13) {hivm.vector_function, no_inline} : (memref<8x16xi32, #hivm.address_space<ub>>, memref<8x16xi32, #h
hivm.hir.set_flag[<PIPE_V>, <PIPE_MTE3>, <EVENT_ID0>]
%subview = memref.subview %reinterpret_cast[%9, 0] [8, 16] [1, 1] {to_be_bubbled_slice} : memref<16x16xf32, strided<[?, 1]>, #hivm.address_space<gm>> to memref
hivm.hir.wait_flag[<PIPE_V>, <PIPE_MTE3>, <EVENT_ID0>]
hivm.hir.store ins(%13 : memref<8x16xf32, #hivm.address_space<ub>>) outs(%subview : memref<8x16xf32, strided<[?, 1], offset: ?>, #hivm.address_space<gm>>) {tile
hivm.hir.pipe_barrier[<PIPE_ALL>]
```

Inject Sync (AIV)

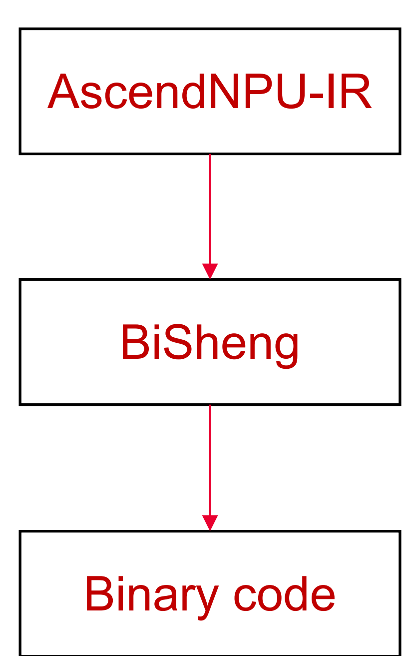
HUAWEI

Target Platform – Atlas

GM
L2
AIC
MTE2
MTE3
L1
MTE1
MTE1
MTE1
FixPipe
L0A
L0B
BT
FP
Scalar1
CUBE
L0C
FixPipe
FixPipe
MTE1
MTE3
AIV x 2
MTE2
MTE3
UB
VEC
Scalar2

That's the bottom of the open-sourced stack

- Some parts of the BiSheng Compiler for Atlas 950 soon will be open-sourced
- Binary releases are fully-available with CANN
  - https://www.hiascend.com/cann/download



Huawei Proprietary - Restricted Distribution

Thank you.

Bring digital to every person, home and organization for a fully connected, intelligent world.

Copyright©2018 Huawei Technologies Co., Ltd.
All Rights Reserved.

The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future product portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.