LLVM DEVELOPERS' MEETING 2024

MLIR and PyTorch: A Compilation Pipeline Targeting Huawei's Ascend Backend

Weizhe Chen, Alexandre Singer, Fang Gao, Kai-Ting Amy Wang

Huawei Canada Research Centre

Summary

We present an approach to compiling PyTorch code, through MLIR, to executable Ascend C code. Ascend C is a programming language oriented to operator development scenarios that support Ascend AI Processors.

This pipeline efficiently transforms high-level PyTorch code to optimized, hardware-specific Ascend C code by bridging PyTorch, MLIR, and Ascend hardware. It allows users of Ascend Processors to benefit from MLIR optimizations and Ascend compiler engineers to contribute innovations and improvements to the MLIR community.

Ascend C Programming Language

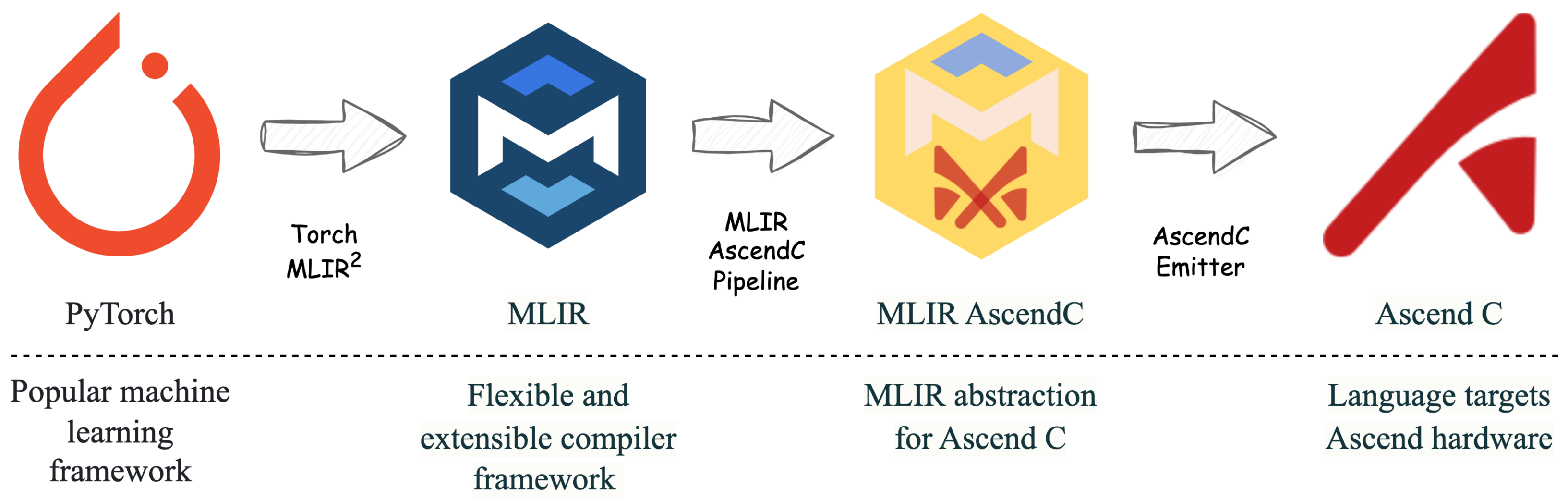
Ascend C is the new programming language launched by Huawei in 2023, targeting the Ascend architecture[1].

It aims to improve the efficiency of operator development and to help AI engineers to develop operators and tune models at a low cost.

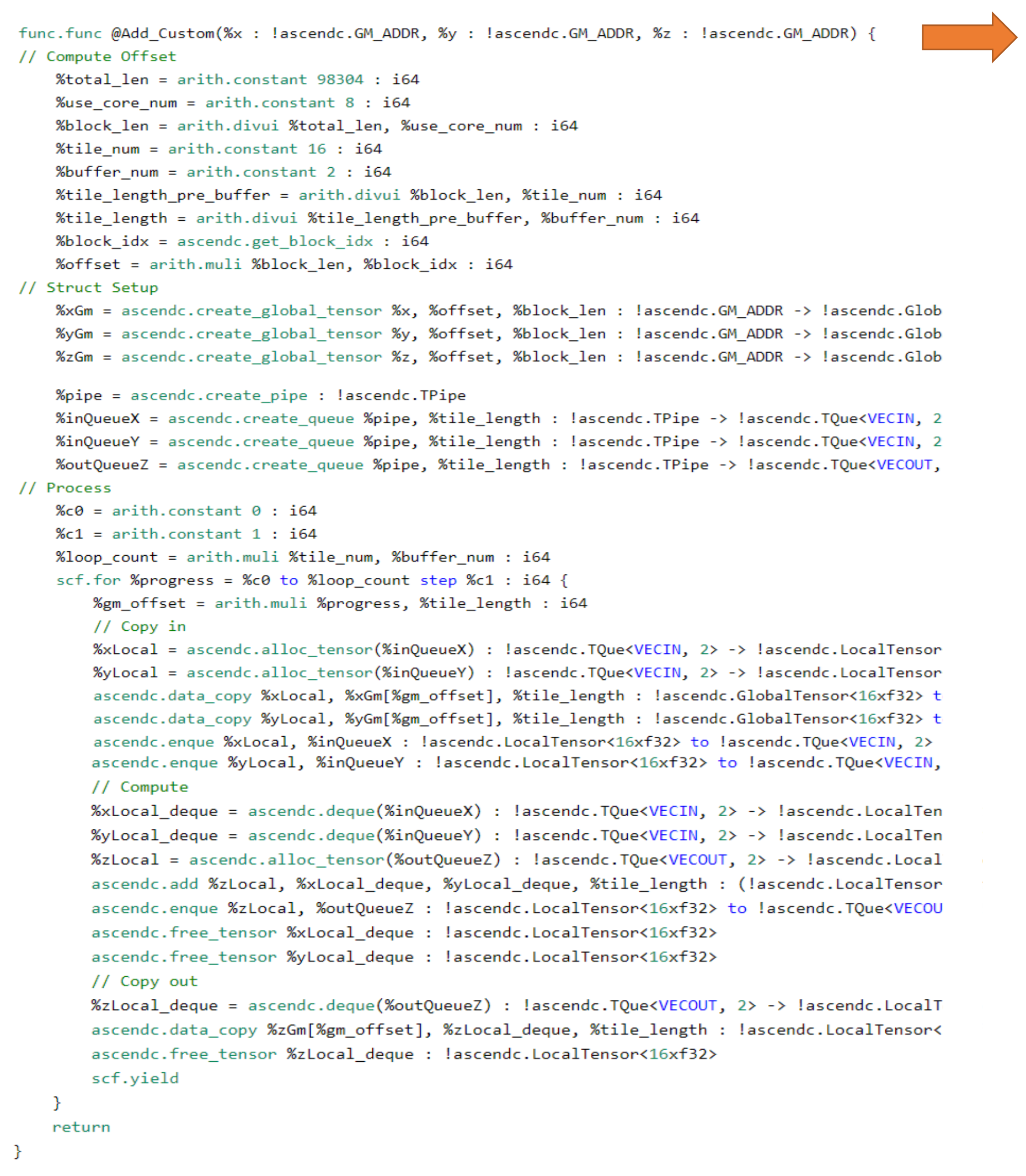
- Operator development scenarios
- Natively supports C/C++ specifications
- Automatic parallel scheduling
- Multi-layer library interface
- CPU-NPU dual debugging

```
__aicore__ inline void Compute(int32_t progress)
{
    // deque input tensors from VECIN queue
    LocalTensor<half> xLocal = inQueueX.DeQue<half>();
    LocalTensor<half> yLocal = inQueueY.DeQue<half>();
    LocalTensor<half> zLocal = outQueueZ.AllocTensor<half>();
    // call Add instr for computation
    Add(zLocal, xLocal, yLocal, TILE_LENGTH);
    // enque the output tensor to VECOUT queue
    outQueueZ.EnQue<half>(zLocal);
    // free input tensors for reuse
    inQueueX.FreeTensor(xLocal);
    inQueueY.FreeTensor(yLocal);
}
```

Goal: PyTorch → MLIR → MLIR AscendC → Ascend

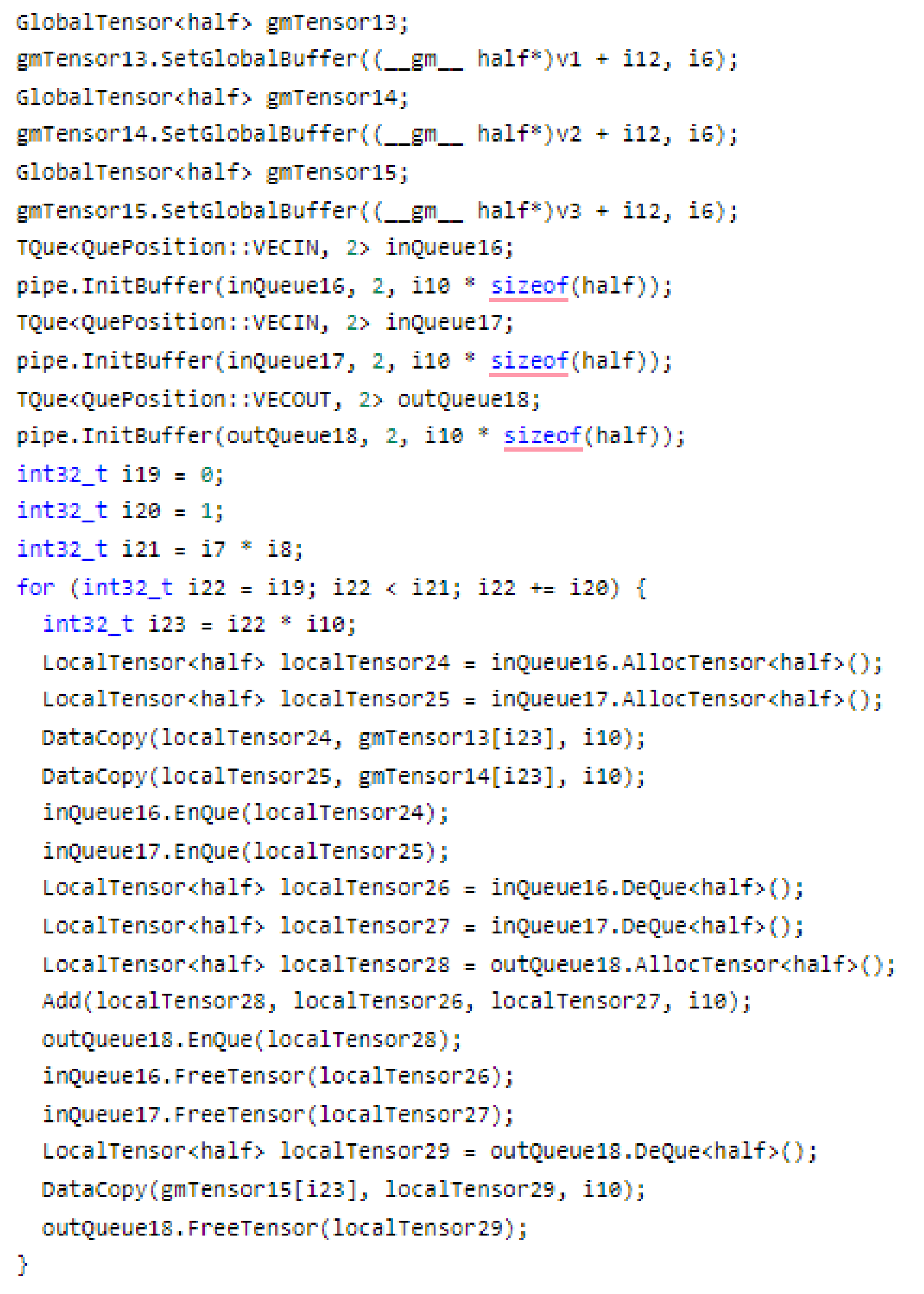


c = torch.add(a, b)



```
func.func @Add_Custom(%x : !ascendc.GM_ADDR, %y : !ascendc.GM_ADDR, %z : !ascendc.GM_ADDR) {
// Compute Offset
    %total_len = arith.constant 98304 : i64
    %use_core_num = arith.constant 8 : i64
    %block_len = arith.divui %total_len, %use_core_num : i64
    %tile_num = arith.constant 16 : i64
    %buffer_num = arith.constant 2 : i64
    %tile_length_pre_buffer = arith.divui %block_len, %tile_num : i64
    %tile_length = arith.divui %tile_length_pre_buffer, %buffer_num : i64
    %block_idx = ascendc.get_block_idx : i64
    %offset = arith.muli %block_len, %block_idx : i64
// Struct Setup
    %xGm = ascendc.create_global_tensor %x, %offset, %block_len : !ascendc.GM_ADDR -> !ascendc.Glob
    %yGm = ascendc.create_global_tensor %y, %offset, %block_len : !ascendc.GM_ADDR -> !ascendc.Glob
    %zGm = ascendc.create_global_tensor %z, %offset, %block_len : !ascendc.GM_ADDR -> !ascendc.Glob

    %pipe = ascendc.create_pipe : !ascendc.TPipe
    %inQueueX = ascendc.create_queue %pipe, %tile_length : !ascendc.TPipe -> !ascendc.TQue<VECIN, 2
    %inQueueY = ascendc.create_queue %pipe, %tile_length : !ascendc.TPipe -> !ascendc.TQue<VECIN, 2
    %outQueueZ = ascendc.create_queue %pipe, %tile_length : !ascendc.TPipe -> !ascendc.TQue<VECOUT,
// Process
    %c0 = arith.constant 0 : i64
    %c1 = arith.constant 1 : i64
    %loop_count = arith.muli %tile_num, %buffer_num : i64
    scf.for %progress = %c0 to %loop_count step %c1 : i64 {
        %gm_offset = arith.muli %progress, %tile_length : i64
        // Copy in
        %xLocal = ascendc.alloc_tensor(%inQueueX) : !ascendc.TQue<VECIN, 2> -> !ascendc.LocalTensor
        %yLocal = ascendc.alloc_tensor(%inQueueY) : !ascendc.TQue<VECIN, 2> -> !ascendc.LocalTensor
        ascendc.data_copy %xLocal, %xGm[%gm_offset], %tile_length : !ascendc.GlobalTensor<16xf32> t
        ascendc.data_copy %yLocal, %yGm[%gm_offset], %tile_length : !ascendc.GlobalTensor<16xf32> t
        ascendc.enque %xLocal, %inQueueX : !ascendc.LocalTensor<16xf32> to !ascendc.TQue<VECIN, 2>
        ascendc.enque %yLocal, %inQueueY : !ascendc.LocalTensor<16xf32> to !ascendc.TQue<VECIN,
        // Compute
        %xLocal_deque = ascendc.deque(%inQueueX) : !ascendc.TQue<VECIN, 2> -> !ascendc.LocalTen
        %yLocal_deque = ascendc.deque(%inQueueY) : !ascendc.TQue<VECIN, 2> -> !ascendc.LocalTen
        %zLocal = ascendc.alloc_tensor(%outQueueZ) : !ascendc.TQue<VECOUT, 2> -> !ascendc.Local
        ascendc.add %zLocal, %xLocal_deque, %yLocal_deque, %tile_length : (!ascendc.LocalTensor
        ascendc.enque %zLocal, %outQueueZ : !ascendc.LocalTensor<16xf32> to !ascendc.TQue<VECOU
        ascendc.free_tensor %xLocal_deque : !ascendc.LocalTensor<16xf32>
        ascendc.free_tensor %yLocal_deque : !ascendc.LocalTensor<16xf32>
        // Copy out
        %zLocal_deque = ascendc.deque(%outQueueZ) : !ascendc.TQue<VECOUT, 2> -> !ascendc.LocalT
        ascendc.data_copy %zGm[%gm_offset], %zLocal_deque, %tile_length : !ascendc.LocalTensor<
        ascendc.free_tensor %zLocal_deque : !ascendc.LocalTensor<16xf32>
        scf.yield
    }
    return
}
```

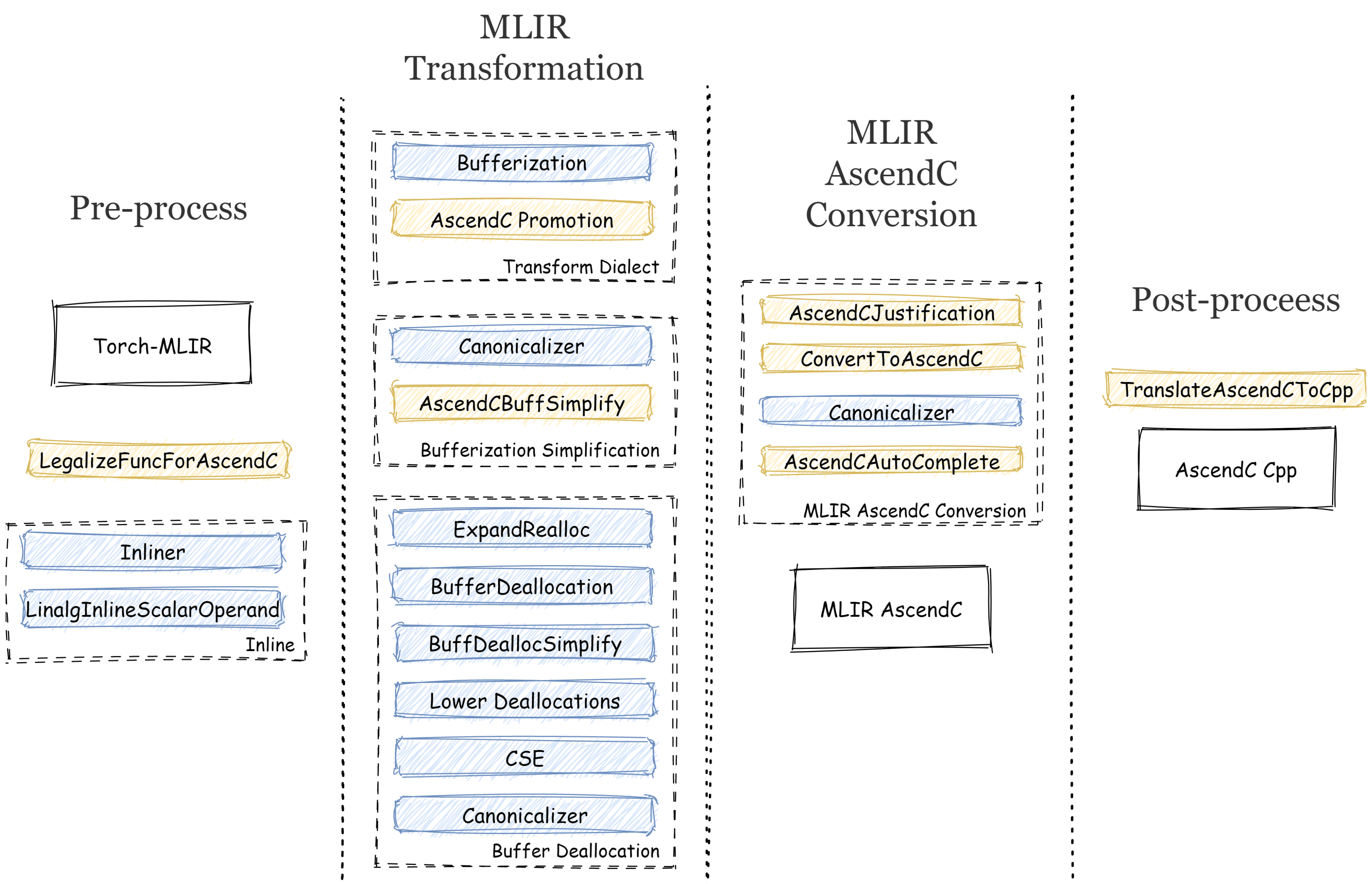


```
GlobalTensor<half> gmTensor13;
gmTensor13.SetGlobalBuffer((__gm__ half*)v1 + i12, i6);
GlobalTensor<half> gmTensor14;
gmTensor14.SetGlobalBuffer((__gm__ half*)v2 + i12, i6);
GlobalTensor<half> gmTensor15;
gmTensor15.SetGlobalBuffer((__gm__ half*)v3 + i12, i6);
TQue<QuePosition::VECIN, 2> inQueue16;
pipe.InitBuffer(inQueue16, 2, i10 * sizeof(half));
TQue<QuePosition::VECIN, 2> inQueue17;
pipe.InitBuffer(inQueue17, 2, i10 * sizeof(half));
TQue<QuePosition::VECOUT, 2> outQueue18;
pipe.InitBuffer(outQueue18, 2, i10 * sizeof(half));
int32_t i19 = 0;
int32_t i20 = 1;
int32_t i21 = i7 * i8;
for (int32_t i22 = i19; i22 < i21; i22 += i20) {
  int32_t i23 = i22 * i10;
  LocalTensor<half> localTensor24 = inQueue16.AllocTensor<half>();
  LocalTensor<half> localTensor25 = inQueue17.AllocTensor<half>();
  DataCopy(localTensor24, gmTensor13[i23], i10);
  DataCopy(localTensor25, gmTensor14[i23], i10);
  inQueue16.EnQue(localTensor24);
  inQueue17.EnQue(localTensor25);
  LocalTensor<half> localTensor26 = inQueue16.DeQue<half>();
  LocalTensor<half> localTensor27 = inQueue17.DeQue<half>();
  LocalTensor<half> localTensor28 = outQueue18.AllocTensor<half>();
  Add(localTensor28, localTensor26, localTensor27, i10);
  outQueue18.EnQue(localTensor28);
  inQueue16.FreeTensor(localTensor26);
  inQueue17.FreeTensor(localTensor27);
  LocalTensor<half> localTensor29 = outQueue18.DeQue<half>();
  DataCopy(gmTensor15[i23], localTensor29, i10);
  outQueue18.FreeTensor(localTensor29);
}
```

MLIR-AscendC Dialect

- Simplified and one-to-one abstraction of the Ascend C language.
- Fully express Ascend C operators using MLIR.
    - Type: AscendC_TPipe, AscendC_TQue, AscendC_GMADDR, AscendC_TensorType
    - Op: constructors, memory management, data process, vector computation, system variable access
    - Attribute: TPosition Enum Attribute

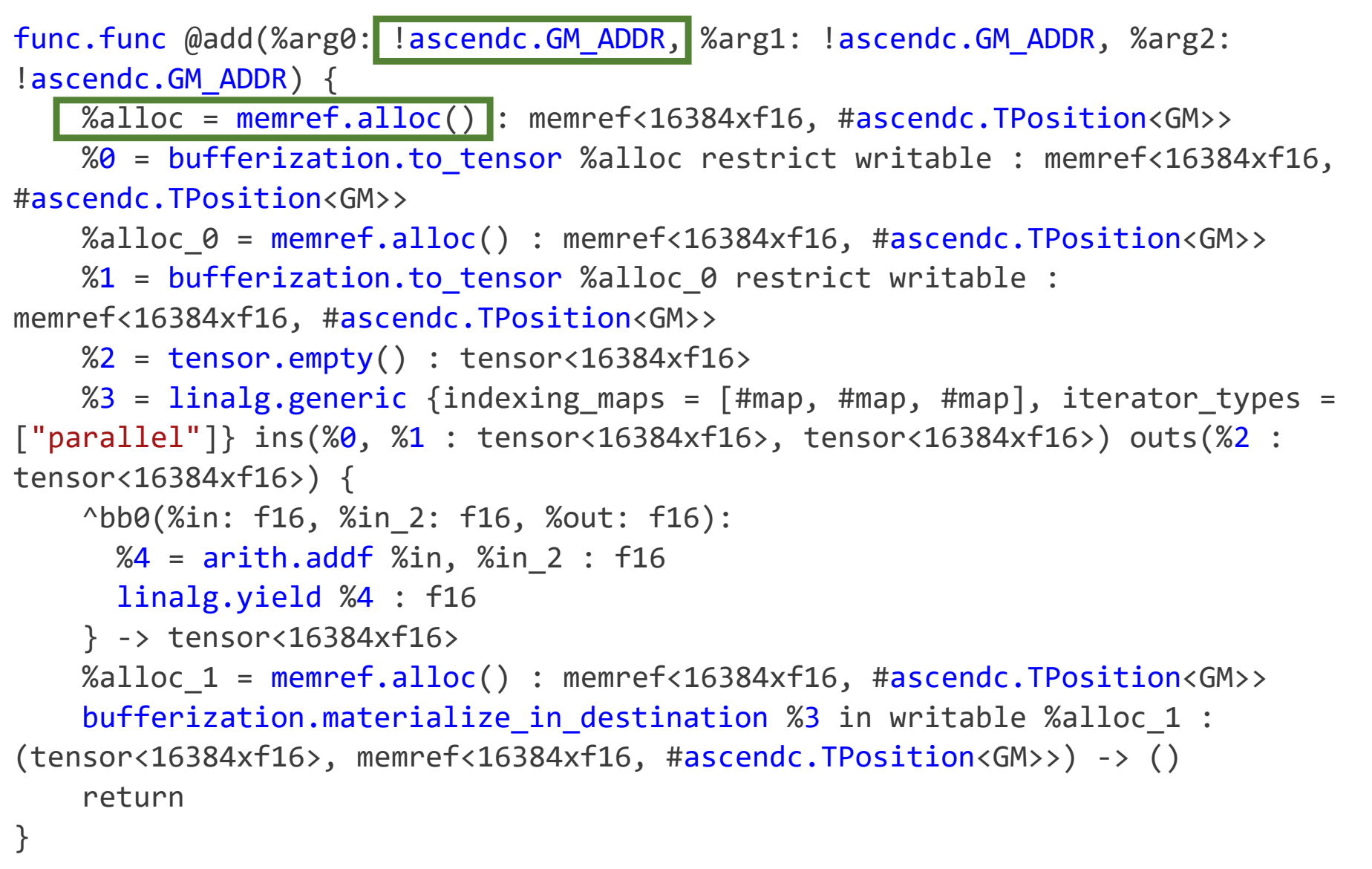
MLIR-AscendC Pipeline



1. Pre-Process

The first stage will prepare the MLIR code for further transformation and conversion.

It will create a memref operand as a replacement for the original tensor operand and combine two through inlining to prepare for bufferization. It will then modify function arguments to customized types to prepare for Ascend C conversion.

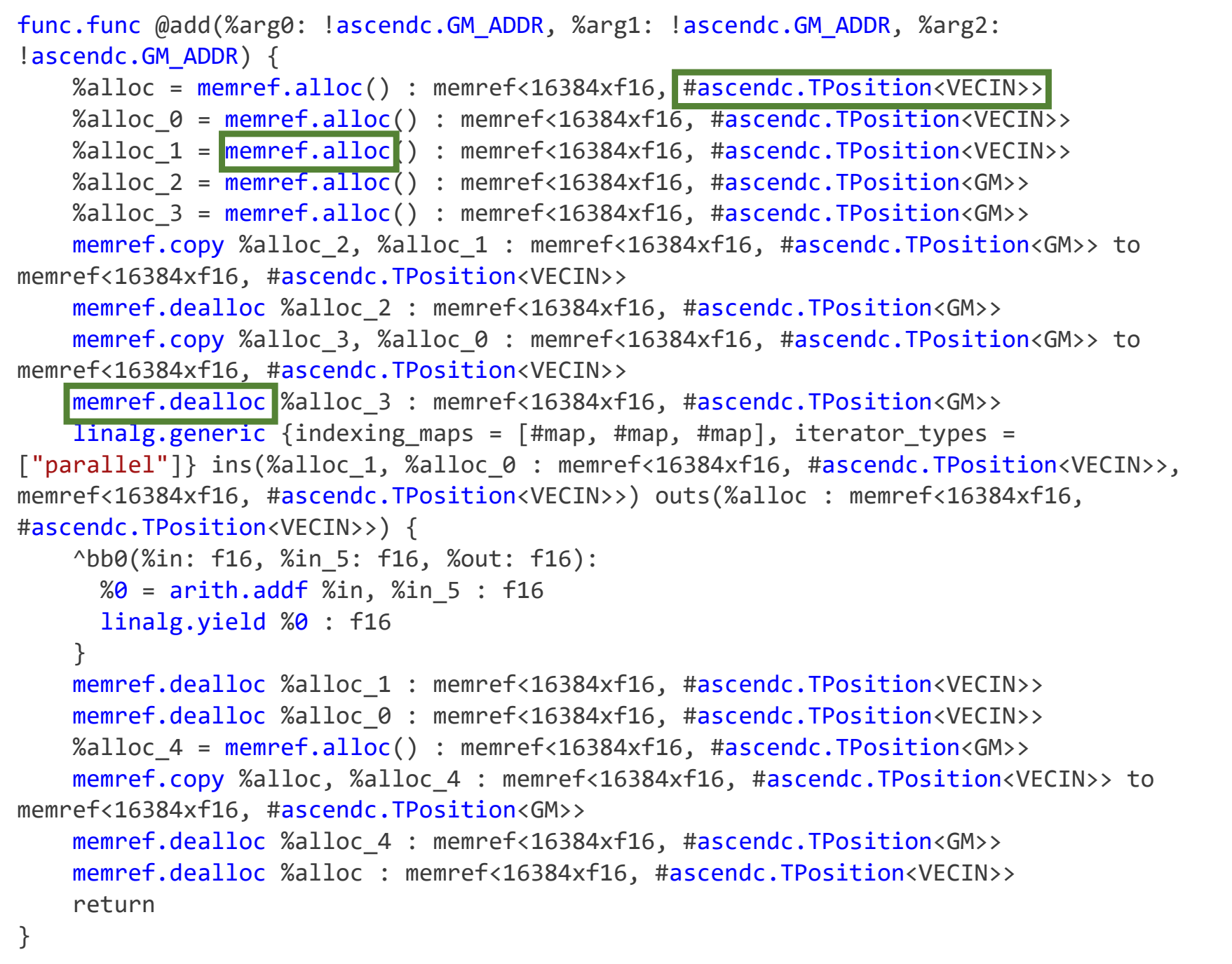


```
func.func @add(%arg0: !ascendc.GM_ADDR, %arg1: !ascendc.GM_ADDR, %arg2:
!ascendc.GM_ADDR) {
    %alloc = memref.alloc() : memref<16384xf16, #ascendc.TPosition<GM>>
    %0 = bufferization.to_tensor %alloc restrict writable : memref<16384xf16,
#ascendc.TPosition<GM>>
    %alloc_0 = memref.alloc() : memref<16384xf16, #ascendc.TPosition<GM>>
    %1 = bufferization.to_tensor %alloc_0 restrict writable :
memref<16384xf16, #ascendc.TPosition<GM>>
    %2 = tensor.empty() : tensor<16384xf16>
    %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types =
["parallel"]} ins(%0, %1 : tensor<16384xf16>, tensor<16384xf16>) outs(%2 :
tensor<16384xf16>) {
    ^bb0(%in: f16, %in_2: f16, %out: f16):
      %4 = arith.addf %in, %in_2 : f16
      linalg.yield %4 : f16
    } -> tensor<16384xf16>
    %alloc_1 = memref.alloc() : memref<16384xf16, #ascendc.TPosition<GM>>
    bufferization.materialize_in_destination %3 in writable %alloc_1 :
(tensor<16384xf16>, memref<16384xf16, #ascendc.TPosition<GM>>) -> ()
    return
}
```

2. MLIR Transformation

The second stage performs MLIR transformation including bufferization, promotion and deallocation as Ascend C has memref-like operands with customized address space.

We perform one-shot bufferize, customized promotion, and utilize the upstream pipeline for buffer deallocation to prepare for MLIR AscendC conversion.



```
func.func @add(%arg0: !ascendc.GM_ADDR, %arg1: !ascendc.GM_ADDR, %arg2:
!ascendc.GM_ADDR) {
    %alloc = memref.alloc() : memref<16384xf16, #ascendc.TPosition<VECIN>>
    %alloc_0 = memref.alloc() : memref<16384xf16, #ascendc.TPosition<VECIN>>
    %alloc_1 = memref.alloc() : memref<16384xf16, #ascendc.TPosition<VECIN>>
    %alloc_2 = memref.alloc() : memref<16384xf16, #ascendc.TPosition<GM>>
    %alloc_3 = memref.alloc() : memref<16384xf16, #ascendc.TPosition<GM>>
    memref.copy %alloc_2, %alloc_1 : memref<16384xf16, #ascendc.TPosition<GM>> to
memref<16384xf16, #ascendc.TPosition<VECIN>>
    memref.dealloc %alloc_2 : memref<16384xf16, #ascendc.TPosition<GM>>
    memref.copy %alloc_3, %alloc_0 : memref<16384xf16, #ascendc.TPosition<GM>> to
memref<16384xf16, #ascendc.TPosition<VECIN>>
    memref.dealloc %alloc_3 : memref<16384xf16, #ascendc.TPosition<GM>>
    linalg.generic {indexing_maps = [#map, #map, #map], iterator_types =
["parallel"]} ins(%alloc_1, %alloc_0 : memref<16384xf16, #ascendc.TPosition<VECIN>>,
memref<16384xf16, #ascendc.TPosition<VECIN>>) outs(%alloc : memref<16384xf16,
#ascendc.TPosition<VECIN>>) {
    ^bb0(%in: f16, %in_5: f16, %out: f16):
      %0 = arith.addf %in, %in_5 : f16
      linalg.yield %0 : f16
    }
    memref.dealloc %alloc_1 : memref<16384xf16, #ascendc.TPosition<VECIN>>
    memref.dealloc %alloc_0 : memref<16384xf16, #ascendc.TPosition<VECIN>>
    %alloc_4 = memref.alloc() : memref<16384xf16, #ascendc.TPosition<GM>>
    memref.copy %alloc, %alloc_4 : memref<16384xf16, #ascendc.TPosition<VECIN>> to
memref<16384xf16, #ascendc.TPosition<GM>>
    memref.dealloc %alloc_4 : memref<16384xf16, #ascendc.TPosition<GM>>
    memref.dealloc %alloc : memref<16384xf16, #ascendc.TPosition<VECIN>>
    return
}
```

3. MLIR AscendC Conversion

The third stage converts standard MLIR to MLIR AscendC, which adds target-specific information.

Specifically, it justifies the usage of Ascend C by inserting TPipe and adjusting TPosition. Then, it will recursively convert into MLIR AscendC and add the synchronization methods used by Ascend C.

```
func.func @add(%arg0: !ascendc.GM_ADDR, %arg1: !ascendc.GM_ADDR, %arg2:
!ascendc.GM_ADDR) {
  %c16384_i32 = arith.constant 16384 : i32
  %0 = ascendc.create_global_tensor %arg0, %c16384_i32 : !ascendc.GM_ADDR ->
!ascendc.GlobalTensor<16384xf16>
  …
  %3 = ascendc.create_pipe : !ascendc.Tpipe
  …
  %8 = ascendc.create_queue %3, %c16384_i32 : !ascendc.TPipe ->
!ascendc.TQue<VECIN, 1>
  %9 = ascendc.alloc_tensor(%8) : !ascendc.TQue<VECIN, 1> ->
!ascendc.LocalTensor<16384xf16>
  ascendc.data_copy %9, %0, %c16384_i32 : !ascendc.GlobalTensor<16384xf16> to
    !ascendc.LocalTensor<16384xf16>
  ascendc.enque %9, %8 : !ascendc.LocalTensor<16384xf16> to !ascendc.TQue<VECIN,
1>
  …
  %10 = ascendc.deque(%6) : !ascendc.TQue<VECIN, 1> ->
!ascendc.LocalTensor<16384xf16>
  …
  ascendc.add %5, %11, %10, %c16384_i32 : (!ascendc.LocalTensor<16384xf16>,
!ascendc.LocalTensor<16384xf16>) -> !ascendc.LocalTensor<16384xf16>
  …
  ascendc.data_copy %2, %12, %c16384_i32 : !ascendc.LocalTensor<16384xf16> to
!ascendc.GlobalTensor<16384xf16>
  ascendc.free_tensor(%4, %12) : !ascendc.LocalTensor<16384xf16> from
!ascendc.TQue<VECOUT, 1>
  return
}
```

4. Post-Process

The fourth stage will emit the Ascend C language, which is the backend for the pipeline.

It will extend the current EmitC[3] abilities to include Ascend C specifications and produce executable code that targets the Ascend architecture.

```
#include "kernel_operator.h"
using namespace AscendC;

extern "C" __global__ __aicore__ void add(GM_ADDR v1, GM_ADDR v2, GM_ADDR v3) {
  int32_t i4 = 16384;
  GlobalTensor<half> gmTensor5;
  gmTensor5.SetGlobalBuffer((__gm__ half*)v1, i4);
  GlobalTensor<half> gmTensor6;
  gmTensor6.SetGlobalBuffer((__gm__ half*)v2, i4);
  GlobalTensor<half> gmTensor7;
  gmTensor7.SetGlobalBuffer((__gm__ half*)v3, i4);
  TPipe pipe;
  TQue<QuePosition::VECOUT, 1> outQueue8;
  pipe.InitBuffer(outQueue8, 1, i4 * sizeof(half));
  LocalTensor<half> localTensor9 = outQueue8.AllocTensor<half>();
  TQue<QuePosition::VECIN, 1> inQueue10;
  pipe.InitBuffer(inQueue10, 1, i4 * sizeof(half));
  LocalTensor<half> localTensor11 = inQueue10.AllocTensor<half>();
  TQue<QuePosition::VECIN, 1> inQueue12;
  pipe.InitBuffer(inQueue12, 1, i4 * sizeof(half));
  LocalTensor<half> localTensor13 = inQueue12.AllocTensor<half>();
  DataCopy(localTensor13, gmTensor5, i4);
  inQueue12.EnQue(localTensor13);
  DataCopy(localTensor11, gmTensor6, i4);
  inQueue10.EnQue(localTensor11);
  LocalTensor<half> localTensor14 = inQueue10.DeQue<half>();
  LocalTensor<half> localTensor15 = inQueue12.DeQue<half>();
  Add(localTensor9, localTensor13, localTensor11, i4);
  outQueue8.EnQue(localTensor9);
  inQueue12.FreeTensor(localTensor13);
  inQueue10.FreeTensor(localTensor11);
  LocalTensor<half> localTensor16 = outQueue8.DeQue<half>();
  DataCopy(gmTensor7, localTensor9, i4);
  outQueue8.FreeTensor(localTensor9);
  return;
}
```

References

[1] Ascend. Ascend C [Computer software]. https://www.hiascend.com/en
[2] LLVM. Torch-MLIR [Computer software]. https://github.com/llvm/torch-mlir
[3] LLVM. MLIR EmitC [Computer software]. https://github.com/llvm/llvm-project/tree/main/mlir/include/mlir/Dialect/EmitC