

Home	
Governance	
Users of MLIR	
MLIR Related Publications	
Talks	
Deprecations & Current Refactoring	
Getting Started	+
Code Documentation	-
Bindings	+
Tools	+
Action: Tracing and Debugging MLIR-based Compilers	
Bufferization	
Data Layout Modeling	
Defining Dialects	+
Diagnostic Infrastructure	
Dialect Conversion	
Dialects	+
Interfaces	
LLVM IR Target	
MLIR Bytecode Format	
MLIR C API	
MLIR Language Reference	
MLIR Release Notes	
Operation Canonicalization	
Ownership-based Buffer Deallocation	
Pass Infrastructure	

Passes

This document describes the available MLIR passes and their contracts.

- [General Transformation Passes](#)
 - [-bubble-down-memory-space-casts](#)
 - [-canonicalize](#)
 - [-composite-fixed-point-pass](#)
 - [-control-flow-sink](#)
 - [-cse](#)
 - [-generate-runtime-verification](#)
 - [-inline](#)
 - [-loop-invariant-code-motion](#)
 - [-loop-invariant-subset-hoisting](#)
 - [-mem2reg](#)
 - [-print-ir](#)
 - [-print-op-stats](#)
 - [-remove-dead-values](#)
 - [-sccp](#)
 - [-snapshot-op-locations](#)
 - [-sroa](#)
 - [-strip-debuginfo](#)
 - [-symbol-dce](#)
 - [-symbol-privatize](#)
 - [-topological-sort](#)
 - [-trivial-dce](#)
 - [-view-op-graph](#)
- [Bufferization Passes](#)
 - [-buffer-deallocation-simplification](#)
 - [-buffer-hoisting](#)



Passes	
Pattern Rewriting : Generic DAG-to-DAG Rewriting	
Pattern Search	
PDDL - PDL Language	
Quantization	
Rationale	+
Remark Infrastructure	
Shape Inference	
SPIR-V Dialect to LLVM Dialect conversion manual	
Symbols and Symbol Tables	
Table-driven Declarative Rewrite Rule (DRR)	
Tokens	
Traits	+
Tutorials	+

- [-buffer-loop-hoisting](#)
- [-buffer-results-to-out-params](#)
- [-bufferization-lower-deallocations](#)
- [-drop-equivalent-buffer-results](#)
- [-eliminate-empty-tensors](#)
- [-empty-tensor-to-alloc-tensor](#)
- [-one-shot-bufferize](#)
- [-optimize-allocation-liveness](#)
- [-ownership-based-buffer-deallocation](#)
- [-promote-buffers-to-stack](#)
- [-static-memory-planner-analysis](#)
- [Conversion Passes](#)
 - [-acc-to-llvm](#)
 - [-arm-neon-2d-to-intr](#)
 - [-convert-affine-for-to-gpu](#)
 - [-convert-amdgpu-to-roccl](#)
 - [-convert-arith-to-amdgpu](#)
 - [-convert-arith-to-apfloat](#)
 - [-convert-arith-to-arm-sme](#)
 - [-convert-arith-to-emitc](#)
 - [-convert-arith-to-llvm](#)
 - [-convert-arith-to-spirv](#)
 - [-convert-arm-sme-to-llvm](#)
 - [-convert-arm-sme-to-scf](#)
 - [-convert-async-to-llvm](#)
 - [-convert-bufferization-to-memref](#)
 - [-convert-cf-to-llvm](#)
 - [-convert-cf-to-spirv](#)
 - [-convert-complex-to-libm](#)
 - [-convert-complex-to-llvm](#)
 - [-convert-complex-to-roccl-library-calls](#)
 - [-convert-complex-to-spirv](#)
 - [-convert-complex-to-standard](#)
 - [-convert-func-to-emitc](#)
 - [-convert-func-to-llvm](#)

- [-convert-func-to-spirv](#)
- [-convert-gpu-to-llvm-spv](#)
- [-convert-gpu-to-nvvm](#)
- [-convert-gpu-to-rocdl](#)
- [-convert-gpu-to-spirv](#)
- [-convert-index-to-llvm](#)
- [-convert-index-to-spirv](#)
- [-convert-linalg-to-std](#)
- [-convert-math-to-apfloat](#)
- [-convert-math-to-emitc](#)
- [-convert-math-to-funcs](#)
- [-convert-math-to-libm](#)
- [-convert-math-to-llvm](#)
- [-convert-math-to-nvvm](#)
- [-convert-math-to-rocdl](#)
- [-convert-math-to-spirv](#)
- [-convert-math-to-xevm](#)
- [-convert-memref-to-emitc](#)
- [-convert-memref-to-spirv](#)
- [-convert-nvgpu-to-nvvm](#)
- [-convert-nvvm-to-llvm](#)
- [-convert-openacc-to-scf](#)
- [-convert-openmp-to-llvm](#)
- [-convert-parallel-loops-to-gpu](#)
- [-convert-pdl-to-pdl-interp](#)
- [-convert-scf-to-cf](#)
- [-convert-scf-to-emitc](#)
- [-convert-scf-to-openmp](#)
- [-convert-scf-to-spirv](#)
- [-convert-shape-constraints](#)
- [-convert-shape-to-std](#)
- [-convert-shard-to-mpi](#)
- [-convert-spirv-to-llvm](#)
- [-convert-tensor-to-linalg](#)
- [-convert-tensor-to-spirv](#)

- [-convert-to-emitc](#)
- [-convert-to-llvm](#)
- [-convert-ub-to-llvm](#)
- [-convert-ub-to-spirv](#)
- [-convert-vector-to-amx](#)
- [-convert-vector-to-arm-sme](#)
- [-convert-vector-to-gpu](#)
- [-convert-vector-to-llvm](#)
- [-convert-vector-to-scf](#)
- [-convert-vector-to-spirv](#)
- [-convert-vector-to-xegpu](#)
- [-convert-xegpu-to-xevm](#)
- [-convert-xevm-to-llvm](#)
- [-finalize-memref-to-llvm](#)
- [-gpu-to-llvm](#)
- [-lift-cf-to-scf](#)
- [-lower-affine](#)
- [-lower-host-to-llvm](#)
- [-map-memref-spirv-storage-class](#)
- [-raise-scf-to-affine](#)
- [-raise-wasm-mlir](#)
- [-reconcile-unrealized-casts](#)
- [-set-llvm-module-datalayout](#)
- [-tosa-to-arith](#)
- [-tosa-to-linalg](#)
- [-tosa-to-linalg-named](#)
- [-tosa-to-mlprogram](#)
- [-tosa-to-scf](#)
- [-tosa-to-spirv-tosa](#)
- [-tosa-to-spirv-tosa-mark-graph-constants](#)
- [-tosa-to-tensor](#)
- ['acc' Dialect Passes](#)
 - [-acc-bind-routine](#)
 - [-acc-cg-to-gpu](#)
 - [-acc-compute-lowering](#)



- [-acc-declare-gpu-module-insertion](#)
- [-acc-emit-remarks-data](#)
- [-acc-emit-remarks-loop](#)
- [-acc-emit-remarks-private](#)
- [-acc-if-clause-lowering](#)
- [-acc-implicit-data](#)
- [-acc-implicit-declare](#)
- [-acc-implicit-routine](#)
- [-acc-legalize-serial](#)
- [-acc-loop-tiling](#)
- [-acc-recipe-materialization](#)
- [-acc-routine-lowering](#)
- [-acc-routine-to-gpu-func](#)
- [-acc-specialize-for-device](#)
- [-acc-specialize-for-host](#)
- [-offload-livein-value-canonicalization](#)
- [-offload-target-verifier](#)
- [-openacc-legalize-data-values](#)
- ['affine' Dialect Passes](#)
 - [-affine-data-copy-generate](#)
 - [-affine-expand-index-ops](#)
 - [-affine-expand-index-ops-as-affine](#)
 - [-affine-fold-memref-alias-ops](#)
 - [-affine-loop-coalescing](#)
 - [-affine-loop-fusion](#)
 - [-affine-loop-invariant-code-motion](#)
 - [-affine-loop-normalize](#)
 - [-affine-loop-tile](#)
 - [-affine-loop-unroll](#)
 - [-affine-loop-unroll-jam](#)
 - [-affine-parallelize](#)
 - [-affine-pipeline-data-transfer](#)
 - [-affine-raise-from-memref](#)
 - [-affine-scalrep](#)
 - [-affine-simplify-min-max](#)



- [-affine-simplify-structures](#)
 - [-affine-simplify-with-bounds](#)
 - [-affine-super-vectorize](#)
- [‘amdgpu’ Dialect Passes](#)
 - [-amdgpu-emulate-atomics](#)
 - [-amdgpu-maskedload-to-load](#)
 - [-amdgpu-resolve-strided-metadata](#)
- [‘arith’ Dialect Passes](#)
 - [-arith-emulate-unsupported-floats](#)
 - [-arith-emulate-wide-int](#)
 - [-arith-expand](#)
 - [-arith-int-range-narrowing](#)
 - [-arith-unsigned-when-equivalent](#)
 - [-int-range-optimizations](#)
- [‘arm_sme’ Dialect Passes](#)
 - [-arm-sme-outer-product-fusion](#)
 - [-arm-sme-vector-legalization](#)
 - [-enable-arm-streaming](#)
 - [-test-arm-sme-tile-allocation](#)
- [‘arm_sve’ Dialect Passes](#)
 - [-arm-sve-legalize-vector-storage](#)
- [‘async’ Dialect Passes](#)
 - [-async-func-to-async-runtime](#)
 - [-async-parallel-for](#)
 - [-async-runtime-policy-based-ref-counting](#)
 - [-async-runtime-ref-counting](#)
 - [-async-runtime-ref-counting-opt](#)
 - [-async-to-async-runtime](#)
- [‘emitc’ Dialect Passes](#)
 - [-form-expressions](#)
 - [-mlgo-add-reflection-map](#)
 - [-wrap-emitc-func-in-class](#)
- [‘func’ Dialect Passes](#)
 - [-duplicate-function-elimination](#)
- [‘gpu’ Dialect Passes](#)



- [-gpu-async-region](#)
- [-gpu-decompose-memrefs](#)
- [-gpu-eliminate-barriers](#)
- [-gpu-kernel-outlining](#)
- [-gpu-launch-sink-index-computations](#)
- [-gpu-map-parallel-loops](#)
- [-gpu-module-to-binary](#)
- [-nvvm-attach-target](#)
- [-rocdl-attach-target](#)
- [-spirv-attach-target](#)
- [-xevm-attach-target](#)
- ['linalg' Dialect Passes](#)
 - [-convert-elementwise-to-linalg](#)
 - [-convert-linalg-to-affine-loops](#)
 - [-convert-linalg-to-loops](#)
 - [-convert-linalg-to-parallel-loops](#)
 - [-linalg-block-pack-matmul](#)
 - [-linalg-fold-into-elementwise](#)
 - [-linalg-fold-unit-extent-dims](#)
 - [-linalg-fuse-elementwise-ops](#)
 - [-linalg-generalize-named-ops](#)
 - [-linalg-inline-scalar-operands](#)
 - [-linalg-morph-ops](#)
 - [-linalg-specialize-generic-ops](#)
 - [-simplify-depthwise-conv](#)
- ['llvm' Dialect Passes](#)
 - [-ensure-debug-info-scope-on-llvm-func](#)
 - [-llvm-add-comdats](#)
 - [-llvm-legalize-for-export](#)
 - [-llvm-request-c-wrappers](#)
 - [-llvm-use-default-visibility](#)
- ['llvm' Target Passes](#)
 - [-llvm-target-to-data-layout](#)
 - [-llvm-target-to-target-features](#)
- ['math' Dialect Passes](#)



- [-math-expand-ops](#)
- [-math-extend-to-supported-types](#)
- [-math-sincos-fusion](#)
- [-math-uplift-to-fma](#)
- [‘memref’ Dialect Passes](#)
 - [-expand-realloc](#)
 - [-expand-strided-metadata](#)
 - [-flatten-memref](#)
 - [-fold-memref-alias-ops](#)
 - [-memref-elide-reinterpret-cast](#)
 - [-memref-emulate-wide-int](#)
 - [-memref-expand](#)
 - [-normalize-memrefs](#)
 - [-reify-result-shapes](#)
 - [-resolve-ranked-shaped-type-result-dims](#)
 - [-resolve-shaped-type-result-dims](#)
- [‘omp’ Dialect Passes](#)
 - [-omp-host-op-filter](#)
 - [-omp-mark-declare-target](#)
 - [-omp-offload-privatization-prepare](#)
 - [-omp-stack-to-shared](#)
- [‘shard’ Dialect Passes](#)
 - [-shard-partition](#)
 - [-shard-simplify](#)
 - [-sharding-propagation](#)
- [‘ml_program’ Dialect Passes](#)
 - [-mlprogram-pipeline-globals](#)
- [‘nvgpu’ Dialect Passes](#)
 - [-nvgpu-optimize-shared-memory](#)
- [‘nvvm’ Dialect Passes](#)
 - [-llvm-optimize-for-nvvm-target](#)
- [‘quant’ Dialect Passes](#)
 - [-lower-quant-ops](#)
 - [-normalize-quant-types](#)
 - [-strip-func-quant-types](#)



- [Reducer Passes](#)
 - [-opt-reduction-pass](#)
 - [-reduction-tree](#)
- [‘scf’ Dialect Passes](#)
 - [-scf-for-loop-canonicalization](#)
 - [-scf-for-loop-peeling](#)
 - [-scf-for-loop-range-folding](#)
 - [-scf-for-loop-specialization](#)
 - [-scf-for-to-while](#)
 - [-scf-forall-to-for](#)
 - [-scf-forall-to-parallel](#)
 - [-scf-parallel-for-to-nested-fors](#)
 - [-scf-parallel-loop-fusion](#)
 - [-scf-parallel-loop-specialization](#)
 - [-scf-parallel-loop-tiling](#)
 - [-test-scf-parallel-loop-collapsing](#)
- [‘shape’ Dialect Passes](#)
 - [-outline-shape-computation](#)
 - [-remove-shape-constraints](#)
 - [-shape-to-shape-lowering](#)
- [‘sparse_tensor’ Dialect Passes](#)
 - [-lower-sparse-foreach-to-scf](#)
 - [-lower-sparse-iteration-to-scf](#)
 - [-lower-sparse-ops-to-foreach](#)
 - [-pre-sparsification-rewrite](#)
 - [-sparse-assembler](#)
 - [-sparse-buffer-rewrite](#)
 - [-sparse-gpu-codegen](#)
 - [-sparse-reinterpret-map](#)
 - [-sparse-space-collapse](#)
 - [-sparse-storage-specifier-to-llvm](#)
 - [-sparse-tensor-codegen](#)
 - [-sparse-tensor-conversion](#)
 - [-sparse-vectorization](#)
 - [-sparsification](#)



- [-sparsification-and-bufferization](#)
 - [-stage-sparse-qps](#)
- [‘spv’ Dialect Passes](#)
 - [-decorate-spirv-composite-type-layout](#)
 - [-spirv-canonicalize-gl](#)
 - [-spirv-lower-abi-attrs](#)
 - [-spirv-promote-to-replicated-constants](#)
 - [-spirv-rewrite-inserts](#)
 - [-spirv-unify-aliased-resource](#)
 - [-spirv-update-vce](#)
 - [-spirv-webgpu-prepare](#)
- [‘tensor’ Dialect Passes](#)
 - [-fold-tensor-subset-ops](#)
 - [-scalarize-single-element-tensor-return](#)
- [‘transform’ Dialect Passes](#)
 - [-transform-dialect-check-uses](#)
 - [-transform-infer-effects](#)
 - [-transform-interpreter](#)
 - [-transform-preload-library](#)
- [‘vector’ Dialect Passes](#)
 - [-lower-vector-mask](#)
 - [-lower-vector-multi-reduction](#)
 - [-lower-vector-to-from-elements-to-shuffle-tree](#)
- [TOSA Dialect Passes](#)
 - [-tosa-arith-const-to-tosa-const](#)
 - [-tosa-attach-target](#)
 - [-tosa-convert-integer-type-to-signless](#)
 - [-tosa-downgrade-1-1-to-1-0](#)
 - [-tosa-experimental-input-shape](#)
 - [-tosa-infer-shapes](#)
 - [-tosa-layerwise-constant-fold](#)
 - [-tosa-make-broadcastable](#)
 - [-tosa-narrow-f64-to-f32](#)
 - [-tosa-narrow-i64-to-i32](#)
 - [-tosa-optional-decompositions](#)



- [-tosa-reduce-transposes](#)
- [-tosa-validate](#)
- [XeGPU Dialect Passes](#)
 - [-xegpu-blocking](#)
 - [-xegpu-optimize-peephole](#)
 - [-xegpu-propagate-layout](#)
 - [-xegpu-sg-to-lane-distribute](#)
 - [-xegpu-vector-linearize](#)
 - [-xegpu-wg-to-sg-distribute](#)

General Transformation Passes ¶

-bubble-down-memory-space-casts ¶

Bubbles down memory-space cast operations.

This pass tries to iteratively bubble down all possible memory-space cast operations. It is important to note that the determination of which casts are bubbled down is based on the interfaces

`MemorySpaceCastConsumerOpInterface`, and `MemorySpaceCastOpInterface`, and not the pass. The pass only looks for operations implementing the `MemorySpaceCastConsumerOpInterface` interface, and invoking the interface methods to perform the bubbling down.

Example:

```
func.func @op_with_cast_sequence(%arg0: memref<4x4xf32, 1>, %arg1: index, %arg2: f32) ->
memref<16xf32> {
  %memspacecast = memref.memory_space_cast %arg0 : memref<4x4xf32, 1> to memref<4x4xf32>
  %c0 = arith.constant 0 : index
  %c4 = arith.constant 4 : index
  %expanded = memref.expand_shape %memspacecast [[0], [1, 2]] output_shape [4, 2, 2] :
memref<4x4xf32> into memref<4x2x2xf32>
  %collapsed = memref.collapse_shape %expanded [[0, 1, 2]] : memref<4x2x2xf32> into
memref<16xf32>
  %loaded = memref.load %collapsed[%c0] : memref<16xf32>
  %added = arith.addf %loaded, %arg2 : f32
  memref.store %added, %collapsed[%c0] : memref<16xf32>
```



```

    %atomic_result = memref.atomic_rmw addf %arg2, %collapsed[%c4] : (f32, memref<16xf32>) ->
    f32
    return %collapsed : memref<16xf32>
}
// mlir-opt --bubble-down-memory-space-casts
func.func @op_with_cast_sequence(%arg0: memref<4x4xf32, 1>, %arg1: index, %arg2: f32) ->
memref<16xf32> {
    %c4 = arith.constant 4 : index
    %c0 = arith.constant 0 : index
    %expand_shape = memref.expand_shape %arg0 [[0], [1, 2]] output_shape [4, 2, 2] :
memref<4x4xf32, 1> into memref<4x2x2xf32, 1>
    %collapse_shape = memref.collapse_shape %expand_shape [[0, 1, 2]] : memref<4x2x2xf32, 1>
into memref<16xf32, 1>
    %memspacecast = memref.memory_space_cast %collapse_shape : memref<16xf32, 1> to
memref<16xf32>
    %0 = memref.load %collapse_shape[%c0] : memref<16xf32, 1>
    %1 = arith.addf %0, %arg2 : f32
    memref.store %1, %collapse_shape[%c0] : memref<16xf32, 1>
    %2 = memref.atomic_rmw addf %arg2, %collapse_shape[%c4] : (f32, memref<16xf32, 1>) -> f32
    return %memspacecast : memref<16xf32>
}

```

-canonicalize ¶

Canonicalize operations

This pass performs various types of canonicalizations over a set of operations by iteratively applying the canonicalization patterns of all loaded dialects until either a fixpoint is reached or the maximum number of iterations/rewrites is exhausted. Canonicalization is best-effort and does not guarantee that the entire IR is in a canonical form after running this pass. See [Operation Canonicalization](#) for more details.

Options ¶

-top-down	: Seed the worklist in general top-down order
-region-simplify	: Perform control flow optimizations to the region tree
-max-iterations	: Max. iterations between applying patterns / simplifying regions
-max-num-rewrites	: Max. number of pattern rewrites within an iteration

```
-test-convergence      : Test only: Fail pass on non-convergence to detect cyclic pattern
-cse-between-iterations : Run full CSE between each pattern-application iteration. CSE-
driven changes trigger extra iterations, so this may push the iteration count up to max-
iterations and affect convergence under test-convergence.
-filter-dialects       : If non-empty, only collect canonicalization patterns from the
dialects with the given namespaces. The listed dialects are force-loaded into the context
as dependent dialects.
-disable-patterns      : Labels of patterns that should be filtered out during application
-enable-patterns       : Labels of patterns that should be used during application, all
other patterns are filtered out
```

-composite-fixed-point-pass ¶

Composite fixed point pass

Composite pass runs provided set of passes until fixed point or maximum number of iterations reached.

Options ¶

```
-name          : Composite pass display name
-pipeline      : Composite pass inner pipeline
-max-iterations : Maximum number of iterations if inner pipeline
```

-control-flow-sink ¶

Sink operations into conditional blocks

This pass implements control-flow sink on operations that implement `RegionBranchOpInterface` by moving dominating operations whose only uses are in a conditionally-executed regions into those regions so that executions paths where their results are not needed do not perform unnecessary computations.

This is similar (but opposite) to loop-invariant code motion, which hoists operations out of regions executed more than once. The implementation of control-flow sink uses a simple and conservative cost model: operations are never duplicated and are only moved into singly-executed regions.

It is recommended to run canonicalization first to remove unreachable blocks: ops in unreachable blocks may prevent other operations from being sunk as they may contain uses of their results



Statistics ¶

```
num-sunk : Number of operations sunk
```

-cse ¶

Eliminate common sub-expressions

This pass implements a generalized algorithm for common sub-expression elimination. This pass relies on information provided by the `Memory SideEffect` interface to identify when it is safe to eliminate operations. See [Common subexpression elimination](#) for more general details on this optimization.

Statistics ¶

```
num-cse'd : Number of operations CSE'd  
num-dce'd : Number of operations DCE'd
```

-generate-runtime-verification ¶

Generate additional runtime op verification checks

This pass generates op-specific runtime checks using the `RuntimeVerifiableOpInterface`. It can be run for debugging purposes after passes that are suspected to introduce faulty IR.

Options ¶

```
-verbose-level : Verbosity level for runtime verification messages: 0 = Minimum (only  
source location), 1 = Detailed (include full operation details, names, types, shapes, etc.)
```

-inline ¶

Inline function calls

Options ¶



```
-default-pipeline : The optimizer pipeline used for callables that do not have a
dedicated optimizer pipeline in opPipelineList
-op-pipelines      : Callable operation specific optimizer pipelines (in the form of
`dialect.op(pipeline)`)
-max-iterations    : Maximum number of iterations when inlining within an SCC
-inlining-threshold : If the ratio between the number of the operations in the callee and
the number of the operations in the caller exceeds this value (in percentage), then the
callee is not inlined even if it is legal to inline it
```

-loop-invariant-code-motion ¶

Hoist loop invariant instructions outside of the loop

-loop-invariant-subset-hoisting ¶

Hoist loop invariant subset ops outside of the loop

-mem2reg ¶

Promotes memory slots into values.

This pass removes loads out of and stores into a memory slot, and turns them into direct uses of SSA values.

This is done generically using the `PromotableAllocationOpInterface`, `PromotableOpInterface`, `PromotableMemOpInterface` and `PromotableRegionOpInterface` interfaces.

This pass will attempt to compute which definitions of the content of the memory slot reach operations that use the memory slot pointer. It will rewire or remove operations that use the slot pointer so they no longer use it. If any of this is not possible, the IR will be left without mutation.

Options ¶

```
-region-simplify : Perform control flow optimizations to the region tree
```

Statistics ¶



```
promoted slots : Total amount of memory slot promoted  
new block args : Total amount of new block argument inserted in blocks
```

-print-ir ¶

Print IR on the debug stream

Print the entire IR on the debug stream. This is meant for debugging purposes to inspect the IR at a specific point in the pipeline.

Options ¶

```
-label : Label
```

-print-op-stats ¶

Print statistics of operations

Options ¶

```
-json : print the stats as JSON
```

-remove-dead-values ¶

Remove dead values

The goal of this pass is optimization (reducing runtime) by removing unnecessary instructions. Unlike other passes that rely on local information gathered from patterns to accomplish optimization, this pass uses a full analysis of the IR, specifically, liveness analysis, and is thus more powerful.

Currently, this pass performs the following optimizations: (A) Removes function arguments that are not live, (B) Removes function return values that are not live across all callers of the function, (C) Removes unnecessary operands, results, region arguments, and region terminator operands of region branch ops, and, (D) Removes simple and region branch ops that have all non-live results and don't affect memory in any way.



Here, a “simple op” refers to an op that isn’t a symbol op, symbol-user op, region branch op, branch op, region branch terminator op, or return-like.

It is noteworthy that we do not refer to non-live values as “dead” in this file to avoid confusing it with dead code analysis’s “dead”, which refers to unreachable code (code that never executes on hardware) while “non-live” refers to code that executes on hardware but is unnecessary. Thus, while the removal of dead code helps little in reducing runtime, removing non-live values should theoretically have significant impact (depending on the amount removed).

It is also important to note that unlike other passes (like `canonicalize`) that apply op-specific optimizations through patterns, this pass uses different interfaces to handle various types of ops and tries to cover all existing ops through these interfaces.

It is because of its reliance on (a) liveness analysis and (b) interfaces that makes it so powerful that it can optimize ops that don't have a canonicalizer and even when an op does have a canonicalizer, it can perform more aggressive optimizations, as observed in the test files associated with this pass.

Example of optimization (A):-

```
int add_2_to_y(int x, int y) {  
    return 2 + y  
}  
  
print(add_2_to_y(3, 4))  
print(add_2_to_y(5, 6))
```

becomes

```
int add_2_to_y(int y) {  
    return 2 + y  
}  
  
print(add_2_to_y(4))  
print(add_2_to_y(6))
```

Example of optimization (B):-



```
int, int get_incremented_values(int y) {
    store y somewhere in memory
    return y + 1, y + 2
}
```

```
y1, y2 = get_incremented_values(4)
y3, y4 = get_incremented_values(6)
print(y2)
```

becomes

```
int get_incremented_values(int y) {
    store y somewhere in memory
    return y + 2
}
```

```
y2 = get_incremented_values(4)
y4 = get_incremented_values(6)
print(y2)
```

Example of optimization (C):-

Assume only %result1 is live here. Then,

```
%result1, %result2, %result3 = scf.while (%arg1 = %operand1, %arg2 = %operand2) {
    %terminator_operand2 = add %arg2, %arg2
    %terminator_operand3 = mul %arg2, %arg2
    %terminator_operand4 = add %arg1, %arg1
    scf.condition(%terminator_operand1) %terminator_operand2, %terminator_operand3,
    %terminator_operand4
} do {
^bb0(%arg3, %arg4, %arg5):
    %terminator_operand6 = add %arg4, %arg4
    %terminator_operand5 = add %arg5, %arg5
```



```
scf.yield %terminator_operand5, %terminator_operand6
}
```

becomes

```
%result1, %result2 = scf.while (%arg2 = %operand2) {
  %terminator_operand2 = add %arg2, %arg2
  %terminator_operand3 = mul %arg2, %arg2
  scf.condition(%terminator_operand1) %terminator_operand2, %terminator_operand3
} do {
^bb0(%arg3, %arg4):
  %terminator_operand6 = add %arg4, %arg4
  scf.yield %terminator_operand6
}
```

It is interesting to see that %result2 won't be removed even though it is not live because %terminator_operand3 forwards to it and cannot be removed. And, that is because it also forwards to %arg4, which is live.

Example of optimization (D):-

```
int square_and_double_of_y(int y) {
  square = y ^ 2
  double = y * 2
  return square, double
}

sq, do = square_and_double_of_y(5)
print(do)
```

becomes

```
int square_and_double_of_y(int y) {
  double = y * 2
  return double
}
```



```
}

do = square_and_double_of_y(5)
print(do)
```

Note: If `canonicalize` is set to “false”, this pass does not remove any block arguments / op results from ops that implement the `RegionBranchOpInterface`. Instead, it just sets dead operands to “ub.poison”.

Options ¶

```
-canonicalize : Canonicalize region branch ops
```

-sccp ¶

Sparse Conditional Constant Propagation

This pass implements a general algorithm for sparse conditional constant propagation. This algorithm detects values that are known to be constant and optimistically propagates this throughout the IR. Any values proven to be constant are replaced, and removed if possible.

This implementation is based on the algorithm described by Wegman and Zadeck in [“Constant Propagation with Conditional Branches”](#) (1991).

-snapshot-op-locations ¶

Generate new locations from the current IR

This pass allows for generating new locations from the IR during any stage of compilation, by snapshotting the IR to a file and using that file to generate new locations for the operations.

Depending on the value of the `tag` option, different resulting locations may be generated:

- If unset, the original location of the operation is replaced.

Example:

```
// old:
... loc("original_source.cpp":1:1)
```



```
// new:
... loc("snapshot_source.mlir":10:10)
```

- If set, the new location is fused with the original location in the form of a [Name_Location](#) with the specified tag.

Example:

```
// old:
... loc("original_source.cpp":1:1)

// new:
... loc(fused["original_source.cpp":1:1, "snapshot"("snapshot_source.mlir":10:10)])
```

Options ¶

```
-filename      : The filename to print the generated IR
-tag           : A tag to use when fusing the new locations with the original. If
unset, the locations are replaced.
-print-debuginfo : Print debug info in MLIR output
-print-op-generic : Print the generic op form
-print-local-scope : Print with local scope and inline information (eliding aliases for
attributes, types, and locations
-pretty-debuginfo : Print pretty debug info in MLIR output
```

-sroa ¶

Scalar Replacement of Aggregates

Scalar Replacement of Aggregates. Replaces allocations of aggregates into independant allocations of its elements.

Allocators must implement `DestructurableAllocationOpInterface` to provide the list of memory slots for which destructuring should be attempted.



This pass will only be applied if all accessors of the aggregate implement the `DestructurableAccessorOpInterface`. If the accessors provide a view into the struct, users of the view must ensure it is used in a type-safe manner and within bounds by implementing `TypeSafeOpInterface`.

Statistics ¶

```
destructured slots      : Total amount of memory slots destructured
slots with memory benefit : Total amount of memory slots in which the destructured size was
smaller than the total size after eliminating unused fields
max subelement number   : Maximal number of sub-elements a successfully destructured slot
initially had
```

`-strip-debuginfo` ¶

Strip debug info from all operations

This pass strips the IR of any location information, by replacing all operation locations with [unknown](#).

`-symbol-dce` ¶

Eliminate dead symbols

This pass deletes all symbols that are found to be unreachable. This is done by computing the set of operations that are known to be live, propagating that liveness to other symbols, and then deleting all symbols that are not within this live set. Live symbols are those that have a [visibility](#) that extends beyond the IR, e.g. `public`, or those that are referenced by live symbols or other non-Symbol operations.

For example, consider the following input:

```
func.func private @dead_private_function()
func.func private @live_private_function()

// Note: The `public` isn't necessary here, as this is the default.
func.func public @public_function() {
  "foo.return"() {uses = [@live_private_function]} : () -> ()
}
```



A known live function, `public_function`, contains a reference to an otherwise non-live function `live_private_function`. After running `symbol-dce`, only these two symbols should remain, as the final symbol `dead_private_function` is not visible outside of the current IR and there are no links to known-live operations. After running, we get the expected:

```
func.func private @live_private_function()

func.func public @public_function() {
  "foo.return"() {uses = [@live_private_function]} : () -> ()
}
```

See [Symbols and SymbolTables](#) for more information on `Symbols`.

Statistics ¶

`num-dce'd` : Number of symbols DCE'd

-symbol-privatize ¶

Mark symbols private

This pass marks all top-level symbols of the operation run as `private` except if listed in `exclude` pass option.

Options ¶

`-exclude` : Comma separated list of symbols that should not be marked private

-topological-sort ¶

Sort regions without SSA dominance in topological order

Recursively sorts all nested regions without SSA dominance in topological order. The main purpose is readability, as well as potentially processing of certain transformations and analyses. The function sorts the operations in all nested regions such that, as much as possible, all users appear after their producers.



This sort is stable. If the block is already topologically sorted, the IR is not changed. Operations that form a cycle are moved to the end of the regions in a stable order.

-trivial-dce ¶

Remove trivially dead operations and blocks

This pass eliminates only trivially dead operations; that is, side-effect-free operations with no users. By default, it also removes trivially dead blocks; that is, blocks that are unreachable from their region entry block. The `remove-blocks` option can be disabled to preserve unreachable blocks.

This pass does not run a liveness analysis and does not remove dead use-def cycles.

By default, this pass recursively visits nested regions. The `recursive` option can be disabled to restrict the pass to only the immediate regions nested under the current operation.

Options ¶

```
-recursive      : Recursively visit nested regions
-remove-blocks  : Remove unreachable blocks
```

-view-op-graph ¶

Print Graphviz visualization of an operation

This pass prints a Graphviz graph of a module.

- Operations are represented as nodes;
- Uses (data flow) as edges;
- Control flow as dashed edges;
- Regions/blocks as subgraphs.

By default, only data flow edges are printed.

Note: See <https://www.graphviz.org/doc/info/lang.html> for more information about the Graphviz DOT language.

Options ¶



```
-max-label-len      : Limit attribute/type length to number of chars
-print-attrs        : Print attributes of operations
-print-control-flow-edges : Print control flow edges
-print-data-flow-edges  : Print data flow edges
-print-result-types   : Print result types of operations
```

Bufferization Passes ¶

`-buffer-deallocation-simplification` ¶

Optimizes `bufferization.dealloc` operation for more efficient codegen

This pass uses static alias analysis to reduce the number of alias checks required at runtime. Such checks are sometimes necessary to make sure that memrefs aren't deallocated before their last usage (use after free) or that some memref isn't deallocated twice (double free).

`-buffer-hoisting` ¶

Optimizes placement of allocation operations by moving them into common dominators and out of nested regions

This pass implements an approach to aggressively move allocations upwards into common dominators and out of nested regions.

`-buffer-loop-hoisting` ¶

Optimizes placement of allocation operations by moving them out of loop nests

This pass implements an approach to aggressively move allocations upwards out of loop nests. It does not move allocations into common dominators.

`-buffer-results-to-out-params` ¶

Converts memref-typed function results to out-params

Some calling conventions prefer to pass output memrefs as "out params". The conversion to this calling convention must be done as an atomic transformation of the entire program (hence this is a module pass).



For example, if a call is rewritten, the callee needs to be rewritten otherwise the IR will end up invalid. Thus, this transformation require an atomic change to the entire program (e.g. the whole module).

This pass is expected to run immediately after bufferization is finished. At that point, tensor-typed results will have been converted to memref-typed results, and can be consistently converted to out params.

All memref-typed results are appended to the function argument list.

The main issue with this pass (and the out-param calling convention) is that buffers for results need to be allocated in the caller. This currently only works for static shaped memrefs.

If the hoist-static-allocs option is on, the pass tries to eliminate the allocation for the returned memref and avoid the memory-copy if possible. This optimization applies on the returned memref which has static shape and is allocated by memref.alloc in the function. It will use the memref given in function argument to replace the allocated memref.

Options ¶

```
-add-result-attr      : Add the attribute 'bufferize.result' to all output parameters.
-hoist-static-allocs   : Hoist static allocations to call sites.
-hoist-dynamic-allocs : Hoist dynamic allocations to call sites.
-modify-public-functions : Modify function signatures of public functions.
```

-bufferization-lower-deallocations ¶

Lowers `bufferization.dealloc` operations to `memref.dealloc` operations

This pass lowers `bufferization.dealloc` operations to the `memref` dialect. It can be applied to a `builtin.module` or operations implementing the `FunctionOpInterface`. For the latter, only simple `dealloc` operations can be lowered because the library function necessary for the fully generic lowering cannot be inserted. In this case, an error will be emitted. Next to `memref.dealloc` operations, it may also emit operations from the `arith`, `scf`, and `func` dialects to build conditional deallocations and library functions to avoid code-size blow-up.

-drop-equivalent-buffer-results ¶

Remove `MemRef` return values that are equivalent to a `bbArg`



This pass removes MemRef return values from functions if they are equivalent to a function bbArg. In that case, the return value is redundant and the respective CallOp operand can be used at the call site.

Note: If a bbArg buffer is not returned directly but casted to beforehand, the buffer is still considered equivalent.

Options ¶

`-modify-public-functions` : Modify function signatures of public functions.

`-eliminate-empty-tensors` ¶

Try to eliminate all `tensor.empty` ops.

Try to eliminate “`tensor.empty`” ops inside `op`. This transformation looks for subset ops that insert a tensor that originates from a “`tensor.empty`” (as per the reverse use-def chain). Such “`tensor.empty`” ops are replaced with the destination subset.

E.g.:

```
%0 = tensor.empty() : tensor<10xf32>
%1 = linalg.fill ... outs(%0 : tensor<10xf32>)
%2 = tensor.insert_slice %1 into %t ...
```

In the above example, the subset op is “`tensor.insert_slice`”. When tracing back the reverse use-def chain of a the source, we end up at a “`tensor.empty`” op. The “`tensor.empty`” op is replaced with a “`tensor.extract_slice`” op.

`-empty-tensor-to-alloc-tensor` ¶

Replace all `empty` ops by `alloc_tensor` ops.

`tensor.empty` ops return a tensor of unspecified contents who's only purpose is to carry the tensor shape. This pass converts such ops to `bufferization.alloc_tensor` ops, which bufferize to buffer allocations.

`-one-shot-bufferize` ¶

One-Shot Bufferize



This pass bufferizes all ops that implement `BufferizableOpInterface`. It first performs an in-placeability analysis on SSA use-def chains of tensor values to determine which `OpOperands` may bufferize in-place, i.e., without inserting a buffer copy. It then rewrites the IR, inserting a buffer allocation and copy for each `OpOperand` that was decided to bufferize out-of-place.

One-Shot Bufferize (and `BufferizableOpInterface`) was designed for ops that are in destination-passing style. When bufferizing such ops, it is possible to reuse the buffer of a tensor `OpOperand` for a tensor `OpResult`. In essence, a possible destination of an operation is already passed as an SSA value.

`tensor.insert` is an example for an op in destination-passing style. E.g., when bufferizing `%t0 = tensor.insert %f into %dest[%idx]`, `buffer(%t0)` is identical to `buffer(%dest)` in the absence of RaW conflicts. As a counter example, `tensor.generate` is not in destination-passing style and always results in a new buffer allocation.

One-Shot Bufferize does not deallocate any buffers that it allocates. The `-buffer-deallocation-pipeline` pipeline should be run after One-Shot Bufferize to insert the deallocation operations necessary to eliminate memory leaks.

One-Shot Bufferize will by default reject IR that contains non-bufferizable op, i.e., ops that do not implement `BufferizableOpInterface`. Such IR can be allowed with `allow-unknown-ops=1`. In that case, `to_buffer` and `to_tensor` ops will be generated at the bufferization boundary. This is useful for compatibility with existing partial bufferization passes: These can bufferize the remaining IR after running One-Shot Bufferize.

Note: Running One-Shot Bufferize after a partial bufferization pass is currently not supported. Running partial bufferization passes after running One-Shot Bufferize is supported and the recommended way to gradually migrate from partial bufferization to One-Shot Bufferize.

With `dialect-filter`, bufferization can be restricted to a set of dialects. If no filter is specified, all ops that implement `BufferizableOpInterface` are bufferized. Ops from the `std` dialect are an exception: These ops are always ignored, even if no filter is specified. When specifying a dialect filter and `allow-unknown-ops` is not turned on, bufferization would fail when encountering an op that is not included in the filter (even if it is bufferizable).

One-Shot Bufferize will by default assume memref types with fully dynamic layout maps when a precise layout cannot be inferred. E.g., this is the case when wrapping a non-bufferizable op in `to_buffer/to_tensor` ops. This behavior can be overridden with `unknown-type-conversion`. Valid values are `fully-dynamic-layout-map` and `identity-layout-map`.



For testing/debugging purposes, `test-analysis-only=1` `print-conflicts=1` prints analysis results and explains why an OpOperand was decided to bufferize out-of-place. This is useful for understanding why One-Shot Bufferize chose to insert a certain buffer copy.

`bufferize-function-boundaries` is an experimental flag for bufferizing `FuncOp`, `ReturnOp` and `CallOp`. This feature is still under development and supports only simple cases at the moment. In particular:

- Recursive or circular function call graphs are not supported.
- External functions (without bodies) that return a tensor are not supported.
- Function with multiple blocks or multiple `ReturnOps` are not supported.
- Layout maps on function signatures can be controlled with a separate `function-boundary-type-conversion` option, which is similar to `unknown-type-conversion` but supports an additional `infer-layout-map` option. `fully-dynamic-layout-map` and `identity-layout-map` ensure that function signatures bufferize to easily predictable types, potentially at the cost of additional casts and copies, respectively. When layout maps are inferred, function return types may be more precise, but less predictable. Function argument types cannot be inferred and always have fully dynamic layout maps with `infer-layout-map`.

One-Shot Bufferize implements the following contract around function calls: The buffer of function arguments is always writable (unless annotated with `bufferization.writable = false`). A buffer copy may be inserted at the call site where necessary. Alias sets and equivalence info is propagated through function calls. Whenever a function is bufferized, all other functions that are being called were already analyzed and bufferized, so exact alias and equivalence information is available. This is why recursive function calls are not yet supported.

One-Shot Bufferize gathers additional information during the analysis phase when function boundary bufferization is activated. E.g., whether a function argument is read/written and which returned values are aliasing/equivalent. For debugging purposes, such information can be printed with `test-analysis-only`.

The order in which ops are analyzed is important. The analysis is greedy and ops that are analyzed earlier are more likely to bufferize in-place. The heuristic can be set with `analysis-heuristic`. At the moment, the following heuristics are available:

- `bottom-up` (default): Analyze ops from bottom to top.
- `top-down`: Analyze ops from top to bottom.
- `fuzzer`: Randomize the ordering of ops with `analysis-fuzzer-seed`.
- `bottom-up-from-terminators`: Traverse the reverse use-def chains of tensor IR, starting from region branch terminators (bottom-up). Nested regions are traversed before enclosing regions. Analyze the traversed ops first, then analyze the remaining ops bottom-up. This heuristic is useful for bufferizing loop



constructs. One-Shot Bufferize currently supports only such IR where yielded tensor values bufferize to equivalent region iter_args, and first analyzing all ops on the path from the “yielding” op to the beginning of the loop body makes it more likely for the region iter_args and yielded values to bufferize to equivalent buffers.

Options ¶

```
-allow-return-allocs-from-loops : Allows returning/yielding new allocations from a loop.
-allow-unknown-ops              : Allows unknown (not bufferizable) ops in the input IR.
-analysis-fuzzer-seed          : Test only: Analyze ops in random order with a given
seed (fuzzer)
-analysis-heuristic             : Heuristic that control the IR traversal during
analysis
-bufferize-function-boundaries  : Bufferize function boundaries (experimental).
-check-parallel-regions        : Account for parallel regions in RaW analysis.
-copy-before-write              : Skip the analysis. Make a buffer copy on every write.
-dialect-filter                 : Restrict bufferization to ops from these dialects.
-dump-alias-sets               : Test only: Annotate tensor IR with alias sets
-no-analysis-func-filter        : Skip analysis of functions with these symbol names. Set
copyBeforeWrite to true when bufferizing them.
-function-boundary-type-conversion : Controls layout maps when bufferizing function
signatures.
-must-infer-memory-space        : The memory space of an memref types must always be
inferred. If unset, a default memory space of 0 is used otherwise.
-use-encoding-for-memory-space  : Use the Tensor encoding attribute for the memory
space. Exclusive to the 'must-infer-memory-space' option
-test-analysis-only             : Test only: Only run inplaceability analysis and
annotate IR
-print-conflicts                : Test only: Annotate IR with RaW conflicts. Requires
test-analysis-only.
-unknown-type-conversion        : Controls layout maps for non-inferable memref types.
-buffer-alignment               : Sets the alignment of newly allocated buffers.
```

Statistics ¶



```

num-buffer-alloc      : Number of buffer allocations
num-tensor-in-place   : Number of in-place tensor OpOperands
num-tensor-out-of-place : Number of out-of-place tensor OpOperands

```

-optimize-allocation-liveness ¶

This pass optimizes the liveness of temp allocations in the input function

This pass will find all operations that have a memory allocation effect. It will search for the corresponding deallocation and move it right after the last user of the allocation. This will optimize the liveness of the allocations.

The pass is expected to run after the deallocation pipeline.

-ownership-based-buffer-deallocation ¶

Adds all required dealloc operations for all allocations in the input program

This pass implements an algorithm to automatically introduce all required deallocation operations for all buffers in the input program. This ensures that the resulting program does not have any memory leaks.

The Buffer Deallocation pass operates on the level of operations implementing the `FunctionOpInterface`. Such operations can take `MemRefs` as arguments, but also return them. To ensure compatibility among all functions (including external ones), some rules have to be enforced. They are just assumed to hold for all external functions. Functions for which the definition is available ideally also already adhere to the ABI. Otherwise, all `MemRef` write operations in the input IR must dominate all `MemRef` read operations in the input IR. Then, the pass may modify the input IR by inserting `bufferization.clone` operations such that the output IR adheres to the function boundary ABI:

- When a `MemRef` is passed as a function argument, ownership is never acquired. It is always the caller's responsibility to deallocate such `MemRefs`.
- Returning a `MemRef` from a function always passes ownership to the caller, i.e., it is also the caller's responsibility to deallocate `MemRefs` returned from a called function.
- A function must not return a `MemRef` with the same allocated base buffer as one of its arguments (in this case a copy has to be created). Note that in this context two subviews of the same buffer that don't overlap are also considered an alias.

It is recommended to bufferize all operations first such that no tensor values remain in the IR once this pass is applied. That way all allocated `MemRefs` will be properly deallocated without any additional manual work.



Otherwise, the pass that bufferizes the remaining tensors is responsible to add the corresponding deallocation operations. Note that this pass does not consider any values of tensor type and assumes that MemRef values defined by `bufferization.to_buffer` do not return ownership and do not have to be deallocated. `bufferization.to_tensor` operations are handled similarly to `bufferization.clone` operations with the exception that the result value is not handled because it's a tensor (not a MemRef).

Input

```
#map0 = affine_map<(d0) -> (d0)>
module {
  func.func @condBranch(%arg0: i1,
                        %arg1: memref<2xf32>,
                        %arg2: memref<2xf32>) {
    cf.cond_br %arg0, ^bb1, ^bb2
  ^bb1:
    cf.br ^bb3(%arg1 : memref<2xf32>)
  ^bb2:
    %0 = memref.alloc() : memref<2xf32>
    linalg.generic {
      indexing_maps = [#map0, #map0],
      iterator_types = ["parallel"]}
    outs(%arg1, %0 : memref<2xf32>, memref<2xf32>) {
      ^bb0(%gen1_arg0: f32, %gen1_arg1: f32):
        %tmp1 = exp %gen1_arg0 : f32
        linalg.yield %tmp1 : f32
    }
    cf.br ^bb3(%0 : memref<2xf32>)
  ^bb3(%1: memref<2xf32>):
    "memref.copy"(%1, %arg2) : (memref<2xf32>, memref<2xf32>) -> ()
    return
  }
}
```

Output

```
#map = affine_map<(d0) -> (d0)>
```



```

module {
  func.func @condBranch(%arg0: i1,
                        %arg1: memref<2xf32>,
                        %arg2: memref<2xf32>) {
    %false = arith.constant false
    %true = arith.constant true
    cf.cond_br %arg0, ^bb1, ^bb2
  ^bb1: // pred: ^bb0
    cf.br ^bb3(%arg1, %false : memref<2xf32>, i1)
  ^bb2: // pred: ^bb0
    %alloc = memref.alloc() : memref<2xf32>
    linalg.generic {
      indexing_maps = [#map, #map],
      iterator_types = ["parallel"]}
    outs(%arg1, %alloc : memref<2xf32>, memref<2xf32>)
    ^bb0(%out: f32, %out_0: f32):
      %2 = math.exp %out : f32
      linalg.yield %2, %out_0 : f32, f32
    }
    cf.br ^bb3(%alloc, %true : memref<2xf32>, i1)
  ^bb3(%0: memref<2xf32>, %1: i1): // 2 preds: ^bb1, ^bb2
    memref.copy %0, %arg2 : memref<2xf32> to memref<2xf32>
    %base_buffer, %offset, %sizes, %strides =
      memref.extract_strided_metadata %0 :
        memref<2xf32> -> memref<f32>, index, index, index
    bufferization.dealloc (%base_buffer : memref<f32>) if (%1)
    return
  }
}

```

The `private-function-dynamic-ownership` pass option allows the pass to add additional arguments to private functions to dynamically give ownership of MemRefs to callees. This can enable earlier deallocations and allows the pass to by-pass the function boundary ABI and thus potentially leading to fewer MemRef clones being inserted. For example, the private function

```

func.func private @passthrough(%memref: memref<2xi32>) -> memref<2xi32> {

```



```
    return %memref : memref<2xi32>
}
```

would be converted to

```
func.func private @passthrough(%memref: memref<2xi32>,
                               %ownership: i1) -> (memref<2xi32>, i1) {
    return %memref, %ownership : memref<2xi32>, i1
}
```

and thus allows the returned MemRef to alias with the MemRef passed as argument (which would otherwise be forbidden according to the function boundary ABI).

Options ¶

`-private-function-dynamic-ownership` : Allows to add additional arguments to private functions to dynamically pass ownership of memrefs to callees. This can enable earlier deallocations.

`-promote-buffers-to-stack` ¶

Promotes heap-based allocations to automatically managed stack-based allocations

This pass implements a simple algorithm to convert heap-based memory allocations to stack-based ones. It uses a built-in heuristic to decide whether it makes sense to convert an allocation. Furthermore, dynamic shaped buffers that are limited by the rank of the tensor can be converted. They are only transformed if they are considered to be small.

Options ¶

`-max-alloc-size-in-bytes` : Maximal size in bytes to promote allocations to stack.
`-max-rank-of-allocated-memref` : Maximal memref rank to promote dynamic buffers.

`-static-memory-planner-analysis` ¶



Transforms same-block alloc/dealloc pairs into static arena allocation with computed offsets

This pass identifies and transforms `memref.alloc` / `memref.dealloc` pairs in the same basic block into a single static arena allocation.

For each `memref.alloc` the pass checks a conservative eligibility envelope:

- Static memref shape.
- The allocation lives directly in the function's entry block (allocations nested in a region are skipped for now).
- Every `memref.dealloc` that may free the buffer is anchored in the entry block. Deallocs reached indirectly through `arith.select`, `scf.if` results, `cf` branches, or view ops are handled via the shared `BufferViewFlowAnalysis`; a dealloc nested inside an entry-block op (e.g. an `scf.if` body) is accepted and bounds the lifetime conservatively.

Eligible allocations are packed into a single arena using a configurable planning algorithm (see the `algorithm` option). The arena is an i8 byte buffer (`memref<Nxi8>`) that can hold allocations of different element types. Each original allocation is replaced with a `memref.view` operation that creates a typed view into the arena at the computed offset.

Ineligible allocations are skipped and retain their original `alloc/dealloc` operations. Skip reasons are reported via op remarks. Aggregate counts are reported as pass statistics.

This pass is expected to run after the deallocation pipeline (`ownership-based-buffer-deallocation` followed by `bufferization-lower-deallocations`).

Example transformation:

```
// Before:
func.func @example() {
  %0 = memref.alloc() : memref<1024xf32>
  "some.use"(%0) : (memref<1024xf32>) -> ()
  memref.dealloc %0 : memref<1024xf32>
  return
}

// After:
func.func @example() {
  %arena = memref.alloc() {alignment = 1 : i64} : memref<4096xi8>
  %c0 = arith.constant 0 : index
```



```
%0 = memref.view %arena[%c0][] : memref<4096xi8> to memref<1024xf32>
"some.use"(%0) : (memref<1024xf32>) -> ()
return
}
```

Options ¶

-arena-mode : Arena allocation mode: 'allocate' creates arena via AllocOp, 'arg' extracts arena from function arguments

-algorithm : Memory planning algorithm to use.

Statistics ¶

num-eligible : Number of alloc/dealloc pairs eligible for static planning

num-skipped-dynamic : Number of allocations skipped: dynamic shape

num-skipped-nested : Number of allocations skipped: nested in loop or conditional

num-skipped-no-dealloc : Number of allocations skipped: no unique same-block dealloc

num-skipped-escaping : Number of allocations skipped: cross-block alias or escaping use

Conversion Passes ¶

-acc-to-llvm ¶

Lower OpenACC operations to LLVM dialect.

-arm-neon-2d-to-intr ¶

Convert Arm NEON structured ops to intrinsics

Creates a pass to lower Arm NEON 2D ops to intrinsics, i.e. equivalent ops operating on flattened 1D vectors and mapping more directly to the corresponding Arm NEON instruction.

-convert-affine-for-to-gpu ¶

Convert top-level AffineFor Ops to GPU kernels



Options ¶

```
-gpu-block-dims : Number of GPU block dimensions for mapping
-gpu-thread-dims : Number of GPU thread dimensions for mapping
```

-convert-amdgpu-to-rocdl ¶

Convert AMDGPU dialect to ROC DL dialect

This pass converts supported AMDGPU ops to ROC DL dialect intrinsics.

Options ¶

```
-chipset : Chipset that these operations will run on
```

-convert-arith-to-amdgpu ¶

Convert Arith operations to AMDGPU-specific implementations

Convert arith operations (currently extf and truncf on 8-bit floats) to operations in the amdgpu dialect. This pass is done in two steps in order to avoid running a notional arith-to-rocdl and arith-to-llvm simultaneously.

Options ¶

```
-chipset : Chipset that these operations will run on
-saturate-fp8-truncf : Use saturating truncation for 8-bit float types
-allow-packed-f16-round-to-zero : Whether we should allow f32->f16 packed round-to-zero conversion
```

-convert-arith-to-apfloat ¶

Convert Arith ops to APFloat runtime library calls

This pass converts supported Arith ops to APFloat-based runtime library calls (APFloatWrappers.cpp). APFloat is a software implementation of floating-point arithmetic operations.



`-convert-arith-to-arm-sme` ¶

Convert Arith dialect to ArmSME dialect

`-convert-arith-to-emitc` ¶

Convert Arith dialect to EmitC dialect

`-convert-arith-to-llvm` ¶

Convert Arith dialect to LLVM dialect

This pass converts supported Arith ops to LLVM dialect instructions.

Options ¶

`-index-bitwidth` : Bitwidth of the index type, 0 to use size of machine word

`-convert-arith-to-spirv` ¶

Convert Arith dialect to SPIR-V dialect

Options ¶

`-emulate-lt-32-bit-scalar-types` : Emulate narrower scalar types with 32-bit ones if not supported by the target

`-emulate-unsupported-float-types` : Emulate unsupported float types by representing them with integer types of same bit width

`-convert-arm-sme-to-llvm` ¶

Lower the operations from the ArmSME dialect into the LLVM dialect

Options ¶

`-dump-tile-live-ranges` : Dump the live ranges of SME tiles (for debugging)

`-convert-arm-sme-to-scf` ¶

Lower the operations from the ArmSME dialect into the SCF dialect

`-convert-async-to-llvm` ¶

Convert the operations from the async dialect into the LLVM dialect

Convert `async.execute` operations to LLVM coroutines and use async runtime API to execute them.

`-convert-bufferization-to-memref` ¶

Convert operations from the Bufferization dialect to the MemRef dialect

This pass converts bufferization operations into memref operations.

In the current state, this pass only transforms a `bufferization.clone` operation into `memref.alloc` and `memref.copy` operations and `bufferization.dealloc` operations (the same way as the `-bufferization-lower-deallocations` pass). The conversion of `clone` operations is needed, since some clone operations could remain after applying several transformation processes. Currently, only `canonicalize` transforms clone operations or even eliminates them. This can lead to errors if any clone op survived after all conversion passes (starting from the bufferization dialect) are performed.

See: <https://llvm.discourse.group/t/bufferization-error-related-to-memref-clone/4665>

To avoid these errors, this pass can be performed as a last clean-up pass to transform remaining operations and to proceed in other dialects (memref e.g.).

Note that this pass only transforms the operation without any further analyses. This pass does not consider any memory analysis or optimization and hence does not resolve any memory leaks.

`-convert-cf-to-llvm` ¶

Convert ControlFlow operations to the LLVM dialect

Convert ControlFlow operations into LLVM IR dialect operations.

If other operations are present and their results are required by the LLVM IR dialect operations, the pass will fail. Any LLVM IR operations or types already present in the IR will be kept as is.



Options ¶

`-index-bitwidth` : Bitwidth of the index type, 0 to use size of machine word

`-convert-cf-to-spirv` ¶

Convert ControlFlow dialect to SPIR-V dialect

Options ¶

`-emulate-lt-32-bit-scalar-types` : Emulate narrower scalar types with 32-bit ones if not supported by the target

`-emulate-unsupported-float-types` : Emulate unsupported float types by representing them with integer types of same bit width

`-convert-complex-to-libm` ¶

Convert Complex dialect to libm calls

This pass converts supported Complex ops to libm calls.

`-convert-complex-to-llvm` ¶

Convert Complex dialect to LLVM dialect

Options ¶

`-complex-range` : Control the intermediate calculation of complex number division

`-convert-complex-to-rocdl-library-calls` ¶

Convert Complex dialect to ROC DL library calls

This pass converts supported Complex ops to calls to the AMD device library.

`-convert-complex-to-spirv` ¶



Convert Complex dialect to SPIRV dialect

-convert-complex-to-standard ¶

Convert Complex dialect to standard dialect

Options ¶

-complex-range : Control the intermediate calculation of complex number division

-convert-func-to-emitc ¶

Convert Func dialect to EmitC dialect

Options ¶

-lower-to-cpp : Target C++ (true) instead of C (false)

-convert-func-to-llvm ¶

Convert from the Func dialect to the LLVM dialect

Convert Func dialect operations into the LLVM IR dialect operations.

Input invariant ¶

- no tensor types;
- all vector are one-dimensional;
- all blocks are reachable by following the successors of the first basic block;

If other operations are present and their results are required by the LLVM IR dialect operations, the pass will fail. Any LLVM IR operations or types already present in the IR will be kept as is.

An LLVM datalayout string can be attached as an attribute to the module on which the pass anchors. Such an attribute is attached by calling the `set-module-datalayout` pass. If present, an `llvm::DataLayout` object is created from this attribute and used in the conversion to LLVM.

Output IR ¶



Functions converted to LLVM IR. Function arguments types are converted one-to-one. Function results are converted one-to-one and, in case more than 1 value is returned, packed into an LLVM IR struct type. Function calls and returns are updated accordingly. Block argument types are updated to use LLVM IR types.

Function discardable attributes ¶

Discardable attributes on `func.func` are lowered as follows.

- **LLVM `llvm.func` properties.** Each inherent attribute defined on `llvm.func` (ODS properties such as `target_cpu`, `linkage`, `vscale_range`, `passthrough`, and so on) must be attached to `func.func` using the `llvm.` prefix (for example `llvm.target_cpu`, `llvm.vscale_range`). The pass strips that prefix, validates the attribute value the same way as for `llvm.func`, and fills the corresponding fields on the generated `llvm.func`. Values that fail validation make conversion fail.
- **Unprefixed legacy names.** A discardable attribute whose name equals the bare ODS property name (without `llvm.`) is **not** forwarded: it is dropped. Only the explicit `llvm.*` spelling is lowered into `llvm.func` properties so that front ends cannot accidentally rely on ambiguous short names.
- **Opaque pass-through.** Any other discardable attribute is copied onto the `llvm.func` unchanged, so arbitrary metadata can survive the conversion. That includes names that start with `llvm.` but are **not** inherent `llvm.func` properties (for example dialect-specific markers): they are not interpreted as properties and are forwarded as discardable attributes on the result.
- **`func.varargs`.** This attribute is interpreted when converting the function type (variadic LLVM signature). It is not an LLVM IR dialect property and is handled separately from the `llvm.*` property mapping above.
- **`llvm.readnone`.** If present, the pass also sets `memory_effects` on the `llvm.func` to read-none semantics, in addition to any other attribute handling.

Options ¶

```
-use-bare-ptr-memref-call-conv : Replace FuncOp's MemRef arguments with bare pointers to
the MemRef element types
-index-bitwidth                : Bitwidth of the index type, 0 to use size of machine word
```

-convert-func-to-spirv ¶

Convert *Func* dialect to *SPIR-V* dialect

Options ¶



```
-emulate-lt-32-bit-scalar-types : Emulate narrower scalar types with 32-bit ones if not
supported by the target
-emulate-unsupported-float-types : Emulate unsupported float types by representing them
with integer types of same bit width
```

-convert-gpu-to-llvm-spv ¶

Generate LLVM operations to be ingested by a SPIR-V backend for gpu operations

Options ¶

```
-use-64bit-index : Use 64-bit integers to convert index types
```

-convert-gpu-to-nvvm ¶

Generate NVVM operations for gpu operations

Options ¶

```
-index-bitwidth           : Bitwidth of the index type, 0 to use size of machine word
-has-redux                 : Target gpu supports redux
-use-bare-ptr-memref-call-conv : Replace memref arguments in GPU functions with bare
pointers. All memrefs must have static shape.
-allow-pattern-rollback    : Experimental performance flag to disallow pattern rollback
-allowed-dialects          : Run conversion patterns of only the specified dialects
```

-convert-gpu-to-rocdl ¶

Generate ROC DL operations for gpu operations

Options ¶

```
-chipset                  : Chipset that these operations will run on
-index-bitwidth           : Bitwidth of the index type, 0 to use size of machine word
-use-bare-ptr-memref-call-conv : Replace memref arguments in GPU functions with bare
```



```
pointers.All memrefs must have static shape
-runtime                               : Runtime code will be run on (default is Unknown, can also
use HIP or OpenCL)
-allowed-dialects                     : Run conversion patterns of only the specified dialects
```

-convert-gpu-to-spirv ⓘ

Convert GPU dialect to SPIR-V dialect

This pass converts supported GPU device ops to SPIR-V ops. It does not handle GPU host ops.

A `gpu.func` op can have parameters to pass in resources. But in SPIR-V entry functions cannot take parameters; they use descriptors to access resources. By default, parameters to a `gpu.func` op will be converted to global variables. These global variables will be assigned sequential binding numbers following their order in the original `gpu.func` op, starting from 0, in set 0. One can attach `spirv.interface_var_abi` to those parameters to control the set and binding if wanted.

Options ⓘ

```
-use-64bit-index : Use 64-bit integers to convert index types
```

-convert-index-to-llvm ⓘ

Lower the index dialect to the LLVM dialect.

This pass lowers Index dialect operations to LLVM dialect operations. Operation conversions are 1-to-1 except for the exotic divides: `ceildivs`, `ceildivu`, and `floordivs`, which expand to series of LLVM operations. Importantly, the index bitwidth should be correctly set to the target pointer width via `index-bitwidth`.

Options ⓘ

```
-index-bitwidth : Bitwidth of the index type, 0 to use size of machine word
```

-convert-index-to-spirv ⓘ

Lower the index dialect to the spirv dialect.



This pass lowers Index dialect operations to SPIR-V dialect operations. Operation conversions are 1-to-1 except for the exotic divides: `ceildivs`, `ceildivu`, and `floordivs`. The index bitwidth will be 32 or 64 as specified by `use-64bit-index`.

Options ¶

```
-use-64bit-index : Use 64-bit integers to convert index types
```

`-convert-linalg-to-std` ¶

Convert the operations from the linalg dialect into the Standard dialect

`-convert-math-to-apfloat` ¶

Convert Math ops to APFloat runtime library calls

This pass converts supported Math ops to APFloat-based runtime library calls (`APFloatWrappers.cpp`). APFloat is a software implementation of floating-point mathmatic operations.

`-convert-math-to-emitc` ¶

Convert some Math operations to EmitC call_opaque operations

This pass converts supported Math ops to `call_opaque` ops targeting `libc/libm` functions. Unlike `convert-math-to-funcs` pass, converting to `call_opaque` ops allows to overload the same function with different argument types.

Options ¶

```
-language-target : Select the language standard target for callees (c99 or cpp11).
```

`-convert-math-to-funcs` ¶

Convert Math operations to calls of outlined implementations.

This pass converts supported Math ops to calls of compiler generated functions implementing these operations in software. The LLVM dialect is used for LinkonceODR linkage of the generated functions.



Options ¶

`-min-width-of-fpowi-exponent` : Convert FPowI only if the width of its exponent's integer type is greater than or equal to this value

`-convert-ctlz` : Convert `math.ctlz` to a software implementation. Enable for targets that do not natively support `ctlz`.

`-convert-math-to-libm` ¶

Convert Math dialect to libm calls

This pass converts supported Math ops to libm calls.

`-convert-math-to-llvm` ¶

Convert Math dialect to LLVM dialect

Options ¶

`-approximate-log1p` : Enable approximation of `Log1p`.

`-convert-math-to-nvvm` ¶

Convert Math dialect to CUDA libdevice calls

This pass converts supported Math ops to CUDA libdevice calls.

`-convert-math-to-rocdl` ¶

Convert Math dialect to ROC DL library calls

This pass converts supported Math ops to ROC DL library calls.

The `chipset` option specifies the target AMDGPU architecture. If the `chipset` is empty, none of the chipset-dependent patterns are added, and the pass will not attempt to parse the `chipset`.

Options ¶

```
-chipset : Chipset that these operations will run on
```

-convert-math-to-spirv ¶

Convert Math dialect to SPIR-V dialect

-convert-math-to-xevm ¶

Convert (fast) math operations to native XeVM/SPIRV equivalents

This pass converts supported math ops marked with the `afn fastmath` flag to function calls for OpenCL native math intrinsics. These intrinsics are typically mapped directly to native device instructions, often resulting in better performance. However, the precision/error of these intrinsics are implementation-defined, and thus math ops are only converted when they have the `afn fastmath` flag enabled. However, if the `convertToOCL` flag is set then all supported math ops would be lowered to OpenCL math intrinsics.

Options ¶

```
-convert-arith : Convert supported Arith ops (e.g. arith.divf) as well.
```

```
-convert-to-ocl : Convert supported Math ops to OCL intrinsics.
```

-convert-memref-to-emitc ¶

Convert MemRef dialect to EmitC dialect

Options ¶

```
-lower-to-cpp : Target C++ (true) instead of C (false)
```

-convert-memref-to-spirv ¶

Convert MemRef dialect to SPIR-V dialect

Options ¶

```
-bool-num-bits    : The number of bits to store a boolean value
-use-64bit-index  : Use 64-bit integers to convert index types
```

-convert-nvgpu-to-nvvm ⓘ

Convert NVGPU dialect to NVVM dialect

This pass converts supported NVGPU ops to NVVM dialect intrinsics.

-convert-nvvm-to-llvm ⓘ

Convert NVVM to PTX with Inline Assembly in LLVM dialect

This pass generates PTX instructions using inline assembly for NVVM operations implements `BasicPtxBuilderInterface`.

-convert-openacc-to-scf ⓘ

Convert the OpenACC ops to OpenACC with SCF dialect

-convert-openmp-to-llvm ⓘ

Convert the OpenMP ops to OpenMP ops with LLVM dialect

-convert-parallel-loops-to-gpu ⓘ

Convert mapped scf.parallel ops to gpu launch operations

Creates a pass that converts `scf.parallel` operations into a `gpu.launch` operation. The mapping of loop dimensions to launch dimensions is derived from mapping attributes. See `ParallelToGpuLaunchLowering::matchAndRewrite` for a description of the used attributes.

-convert-pdl-to-pdl-interp ⓘ

Convert PDL ops to PDL interpreter ops

-convert-scf-to-cf ⓘ

Convert SCF dialect to ControlFlow dialect, replacing structured control flow with a CFG



Options ¶

`-allow-pattern-rollback` : Experimental performance flag to disallow pattern rollback

`-convert-scf-to-emitc` ¶

Convert SCF dialect to EmitC dialect, maintaining structured control flow

`-convert-scf-to-openmp` ¶

Convert SCF parallel loop to OpenMP parallel + workshare constructs.

Options ¶

`-num-threads` : Number of threads to use

`-convert-scf-to-spirv` ¶

Convert SCF dialect to SPIR-V dialect.

Converts SCF ops into SPIR-V structured control flow ops. SPIR-V structured control flow ops do not support yielding values. So for SCF ops yielding values, SPIR-V variables are created for holding the values and load/store operations are emitted for updating them.

`-convert-shape-constraints` ¶

Convert shape constraint operations to the standard dialect

This pass eliminates shape constraints from the program, converting them to eager (side-effecting) error handling code.

This pass is separate from the regular `convert-shape-to-standard`, despite converting between the same dialects, because converting shape constraints can happen at a different part of the program than general shape computation lowering.

`-convert-shape-to-std` ¶

Convert operations from the shape dialect into the standard dialect

-convert-shard-to-mpi ¶

Convert Shard dialect to MPI dialect.

This pass lowers communication operations from the Shard dialect to the MPI dialect. If the module contains the DLT attribute “MPI:comm_world-rank”, its integer value is used as the rank instead of calling MPI_Comm_rank. This enables optimizations such as constant shape propagation and fusion, since shard and partition sizes can be determined from the rank. For some operations the conversion may require intermediate memref allocations. For compatibility with the buffer deallocation pipeline, these allocations are only deallocated when the lowered operations return buffers. When the operation was defined in tensor-land, no explicit deallocation is performed. This means that the deallocation must be handled by different means, e.g. by the deallocation pipeline.

-convert-spirv-to-llvm ¶

Convert SPIR-V dialect to LLVM dialect

See <https://mlir.llvm.org/docs/SPIRVToLLVMDialectConversion/> for more details.

Options ¶

-client-api : Derive StorageClass to address space mapping from the client API

-convert-tensor-to-linalg ¶

Convert some Tensor dialect ops to Linalg dialect

-convert-tensor-to-spirv ¶

Convert Tensor dialect to SPIR-V dialect

Options ¶

-emulate-lt-32-bit-scalar-types : Emulate narrower scalar types with 32-bit ones if not supported by the target

```
-emulate-unsupported-float-types : Emulate unsupported float types by representing them
with integer types of same bit width
```

-convert-to-emitc ¶

Convert to EmitC dialect via dialect interfaces

This is a generic pass to convert to the EmitC dialect, it uses the `ConvertToEmitCPatternInterface` dialect interface to delegate to dialects the injection of conversion patterns.

Options ¶

```
-filter-dialects : Test conversion patterns of only the specified dialects
-lower-to-cpp    : Target C++ (true) instead of C (false)
```

-convert-to-llvm ¶

Convert to LLVM via dialect interfaces found in the input IR

This is a generic pass to convert to LLVM, it uses the `ConvertToLLVMPatternInterface` dialect interface to delegate to dialects the injection of conversion patterns.

If `dynamic` is set to `true`, the pass will look for `ConvertToLLVMAttrInterface` attributes and use them to further configure the conversion process. This option also uses the `DataLayoutAnalysis` analysis to configure the type converter. Enabling this option incurs in extra overhead.

Options ¶

```
-filter-dialects      : Test conversion patterns of only the specified dialects
-dynamic              : Use op conversion attributes to configure the conversion
-allow-pattern-rollback : Experimental performance flag to disallow pattern rollback
```

-convert-ub-to-llvm ¶

Convert UB dialect to LLVM dialect

This pass converts supported UB ops to LLVM dialect instructions.



Options ¶

```
-index-bitwidth : Bitwidth of the index type, 0 to use size of machine word
```

-convert-ub-to-spirv ¶

Convert UB dialect to SPIR-V dialect

This pass converts supported UB ops to SPIR-V dialect ops.

-convert-vector-to-amx ¶

Lower the operations from the vector dialect into the X86 dialect AMX operations

-convert-vector-to-arm-sme ¶

Lower the operations from the vector dialect into the ArmSME dialect

Pass that converts vector dialect operations into equivalent ArmSME dialect operations.

-convert-vector-to-gpu ¶

Lower the operations from the vector dialect into the GPU dialect

Options ¶

```
-use-nvgpu : convert to NvGPU ops instead of GPU dialect ops
```

-convert-vector-to-llvm ¶

Lower the operations from the vector dialect into the LLVM dialect

Convert operations from the vector dialect into the LLVM IR dialect operations. The lowering pass provides several options to control the kinds of optimizations that are allowed. It also provides options that enable the use of one or more architectural-specific dialects (X86, ArmNeon, ArmSVE, etc.) in combination with the architectural-neutral vector dialect lowering.

Options ¶

```

-reassociate-fp-reductions : Allows llvm to reassociate floating-point reductions for
speed
-force-32bit-vector-indices : Allows compiler to assume vector indices fit in 32-bit if
that yields faster code
-use-vector-alignment      : Use the preferred alignment of a vector type in load/store
operations instead of the alignment of the element type of the memref. This flag is
intended for use with hardware which requires vector alignment, or in application contexts
where it is known all vector access are naturally aligned. If operations have an alignment
attribute set, the alignment attribute takes priority over this option
-enable-gep-inbounds-nuw   : Emit inbounds|nuw flags on GEPs generated by
vector.load/store lowering. Assumes 0 <= idx < dim_size and non-negative strides for all
dimensions. Setting this when those conditions do not hold is undefined behavior under the
LLVM lowering.
-enable-arm-neon           : Enables the use of ArmNeon dialect while lowering the vector
dialect.
-enable-arm-sve            : Enables the use of ArmSVE dialect while lowering the vector
dialect.
-enable-arm-i8mm           : Enables the use of Arm FEAT_I8MM instructions while lowering
the vector dialect.
-enable-arm-bf16           : Enables the use of Arm FEAT_BF16 instructions while lowering
the vector dialect.
-enable-x86                : Enables the use of X86 dialect while lowering the vector
dialect.
-vector-contract-lowering  : control the lowering of `vector.contract` operations.
-vector-transpose-lowering : control the lowering of `vector.transpose` operations.

```

-convert-vector-to-scf ¶

Lower the operations from the vector dialect into the SCF dialect

Options ¶

```

-full-unroll      : Perform full unrolling when converting vector transfers to SCF
-target-rank      : Target vector rank to which transfer ops should be lowered

```



```
-lower-tensors : Lower transfer ops that operate on tensors
-lower-scalable : Add scalable vector specific lowerings (that introduce loops)
```

-convert-vector-to-spirv ¶

Convert Vector dialect to SPIR-V dialect

-convert-vector-to-xegpu ¶

Lower the operations from the vector dialect into the XeGPU dialect

-convert-xegpu-to-xevm ¶

Convert XeGPU to XeVM dialect

Options ¶

```
-use-64bit-index : Use 64-bit integers to convert index types
```

-convert-xevm-to-llvm ¶

Convert XeVM to LLVM dialect

-finalize-memref-to-llvm ¶

Finalize MemRef dialect to LLVM dialect conversion

Finalize the conversion of the operations from the MemRef dialect to the LLVM dialect. This conversion will not convert some complex MemRef operations. Make sure to run `expand-strided-metadata` beforehand for these.

Options ¶

```
-use-aligned-alloc      : Use aligned_alloc in place of malloc for heap allocations
-index-bitwidth         : Bitwidth of the index type, 0 to use size of machine word
-use-generic-functions  : Use generic allocation and deallocation functions instead of the
                           classic 'malloc', 'aligned_alloc' and 'free' functions
```



-gpu-to-llvm ¶

Convert GPU dialect to LLVM dialect with GPU runtime calls

Creates a pass to convert a GPU operations into a sequence of GPU runtime calls.

This pass does not generate code to call GPU runtime APIs directly but instead uses a small wrapper library that exports a stable and conveniently typed ABI on top of GPU runtimes such as CUDA or ROCm (HIP).

Options ¶

```
-use-bare-pointers-for-host    : Use bare pointers to pass memref arguments to host
                                functions. All memrefs must have static shape.
-use-bare-pointers-for-kernels : Use bare pointers to pass memref arguments to kernels. The
                                kernel must use the same setting for this option.
-interperse-sizes-for-kernels : Inserts a size_t argument following each memref argument,
                                containing the static size in bytes of the buffer. Incompatible arguments are rejected.
                                This is intended for use by the Vulkan runtime with the kernel bare pointer calling
                                convention, to enable dynamic binding of buffers as arguments without static type info.
```

-lift-cf-to-scf ¶

Lift ControlFlow dialect to SCF dialect

Lifts ControlFlow operations to SCF dialect operations.

This pass is prefixed with “lift” instead of “convert” as it is not always guaranteed to replace all ControlFlow ops. If a region contains only a single kind of return-like operation, all ControlFlow operations will be replaced successfully. Otherwise a single ControlFlow switch branching to one block per return-like operation kind remains.

This pass may need to create unreachable terminators in case of infinite loops, which is only supported for ‘func.func’ for now. If you potentially have infinite loops inside CFG regions not belonging to ‘func.func’, consider using `transformCFGToSCF` function directly with corresponding `CFGToSCFInterface::createUnreachableTerminator` implementation.

-lower-affine ¶



Lower Affine operations to a combination of Arith and SCF operations

Convert operations from the affine dialect into operations from the SCF and standard dialects.

`affine.for` operations are converted to `scf.for` operations that are free of certain structural restrictions (on their bounds and step). `affine.if` is similarly converted to the `scf.if` operation. `affine.apply` operations are converted into sequences of primitive arithmetic operations from the `arith` dialect that have the same effect, using operands of the `index` type. Consequently, named maps and sets there are no longer in use may be removed from the module.

For example, `%r = affine.apply affine_map<(d0, d1)[s0] -> (d0 + 2*d1 + s0)>(%d0, %d1)[%s0]` can be converted into:

```
%d0 = <...>
%d1 = <...>
%s0 = <...>
%0 = arith.constant 2 : index
%1 = arith.muli %0, %d1
%2 = arith.addi %d0, %1
%r = arith.addi %2, %s0
```

Input invariant ¶

- no Tensor types;

These restrictions may be lifted in the future.

Output IR ¶

Functions with `affine.for` and `affine.if` operations eliminated. These functions may contain operations from the Standard dialect in addition to those already present before the pass.

Invariants ¶

- Functions without a body are not modified.
- The semantics of the other functions is preserved.
- Individual operations other than those mentioned above are not modified if they do not depend on the loop iterator value or on the result of `affine.apply`.



`-lower-host-to-llvm` ¶

Lowers the host module code and `gpu.Launch_func` to LLVM

Creates a pass to emulate `gpu.launch_func` call in LLVM dialect and lower the host module code to LLVM.

This transformation creates a sequence of global variables that are later linked to the variables in the kernel module, and a series of copies to/from them to emulate the memory transfer from the host or to the device sides. It also converts the remaining Arithmetic, Func, and MemRef dialects into LLVM dialect, emitting C wrappers.

`-map-memref-spirv-storage-class` ¶

Map numeric MemRef memory spaces to SPIR-V storage classes

Options ¶

`-client-api` : The client API to use for populating mappings

`-raise-scf-to-affine` ¶

Raise SCF operations to affine operations (best effort)

Raising to the affine dialect enables affine analyses and transforms such as loop tiling, unrolling, vectorization, and dependence analysis. Currently only `scf.for` is handled; loops that cannot be raised are left unchanged.

A `scf.for` is raised when its lower and upper bounds are valid affine dimensions (or an `affine.max` / `affine.min` respectively) and its step is a valid affine symbol. Two cases are handled:

- Constant step: raised directly, preserving the step and any `affine.min` / `affine.max` bounds as the loop's bounds.
- Dynamic (symbolic) step: the loop is normalized to unit step and the original induction variable is reconstructed in the body. An `affine.min` upper bound is supported here; an `affine.max` lower bound is not, as it cannot be raised soundly without a constant step.

Bounds and step of non-`index` type are cast to `index` first if the source type allows lossless conversion (otherwise we do not touch the loop). In any case we preserve semantics.

This pass does not modify memory accesses; use `--affine-raise-from-memref` to convert `memref.load` / `memref.store`.

-raise-wasm-mlir ¶

Convert Wasm dialect to a group of dialect as a bridge to LLVM MLIR conversion

-reconcile-unrealized-casts ¶

Simplify and eliminate unrealized conversion casts

Eliminate `unrealized_conversion_cast` operations, commonly introduced by partial dialect conversions, that transitively convert a value to another value of the same type, that is:

```
%0 = "producer.op"() : () -> !type.A
%1 = unrealized_conversion_cast %0 : !type.A to !type.B
%2 = unrealized_conversion_cast %1 : !type.B to !type.C
%3 = unrealized_conversion_cast %2 : !type.C to !type.A
"consumer.op"(%3) : (!type.A) -> ()
```

Such situations appear when the consumer operation is converted by one pass and the producer operation is converted by another pass, each of which produces an unrealized cast. This pass can be used to clean up the IR.

-set-llvm-module-datalayout ¶

Attach a datalayout string as a module attribute

Verify that the `dataLayout` string is a valid LLVM datalayout string and attach it as an attribute `LLVMDialect::getDataLayoutAttrName()` to the module, overriding the existing one.

Options ¶

```
-data-layout : String description (LLVM format) of the data layout that is expected on the
produced module
```

-tosa-to-arith ¶

Lower TOSA to the Arith dialect



Pass that converts TOSA operations to the equivalent operations using the operations in the Arith dialect. The ApplyScale operator is optionally included as it is often preserved until the final invocation.

Options ¶

```
-include-apply-rescale : Whether to include the lowering for tosa.apply_rescale to arith
-use-32-bit           : Whether to prioritize lowering to 32-bit operations
```

-tosa-to-linalg ¶

Lower TOSA to LinAlg on tensors

Pass that converts TOSA operations to the equivalent operations using the tensor operations in LinAlg.

Options ¶

```
-disable-tosa-decompositions : Disable tosa decompositions pass
-aggressive-reduce-constant  : Always perform the reduce constant optimization
```

-tosa-to-linalg-named ¶

Lower TOSA to LinAlg named operations

Pass that converts TOSA operations to the equivalent operations using the Linalg named operations.

Options ¶

```
-prefer-conv2d-kernel-layout-hwcf : Prefer generating linalg.conv_2d_nhwc_hwcf over
linalg.conv_2d_nhwc_fhwc
```

-tosa-to-mlprogram ¶

Lower TOSA to the MLProgram dialect

Pass that converts TOSA's variable operator operations to the equivalent MLProgram operations.



-tosa-to-scf ¶*Lower TOSA to the SCF dialect*

Pass that converts TOSA's control flow operations to the equivalent SCF operations.

-tosa-to-spirv-tosa ¶*Lower TOSA IR to SPIR-V Graph/TOSA operations*

Converts TOSA programs to the SPIR-V Graph/TOSA representation by wrapping converted functions in `spirv.module` and `spirv.ARM.Graph`, lowering supported TOSA ops to `spirv.Tosa.*`, and rewriting TOSA tensor and shape types to the corresponding SPIR-V ARM tensor types.

Options ¶

`-custom-op-domain-to-opcode` : Map TOSA custom op domains to `spirv.ExperimentalML.Call` opcodes. Entries use `<domain>:<opcode>`. The option can be specified multiple times; if a domain appears more than once, the last mapping wins.

-tosa-to-spirv-tosa-mark-graph-constants ¶*Mark large TOSA constants as SPIR-V Graph constants*

Marks large `tosa.const` and `tosa.const_shape` operations so the TOSA-to-SPIR-V Graph/TOSA conversion lowers them to `spirv.ARM.GraphConstant` instead of inlining them as `spirv.Constant`.

-tosa-to-tensor ¶*Lower TOSA to the Tensor dialect*

Pass that converts TOSA operations to the equivalent operations using the operations in the Tensor dialect.

‘acc’ Dialect Passes ¶**-acc-bind-routine ¶***Apply bind clause to function calls in ACC compute regions*

For calls inside offload regions that target a function with an `acc routine` directive and a `bind(name)` clause, rewrite the call to use the bound symbol so device code calls the correct call target.

Options ¶

`-device-type` : Target device type. One use case is ensuring that `device_type-specific` clauses are considered. Another is device-specific specializations.

`-acc-cg-to-gpu` ¶

Lower `acc.compute_region` to `gpu.launch`

Generate GPU code for `acc.compute_region` loop nests by converting them to `gpu.launch` and mapping `acc.par_dims` parallelism to GPU block and thread dimensions.

Options ¶

`-device-type` : Target device type. One use case is ensuring that `device_type-specific` clauses are considered. Another is device-specific specializations.

`-max-workgroup-shared-memory` : Maximum workgroup shared memory budget in bytes.

`-max-thread-private-stack` : Maximum thread-private stack allocation in bytes.

`-subgroup-size` : Subgroup size used for block-dimension alignment and reductions.

`-acc-compute-lowering` ¶

Lower ACC compute constructs to `acc.compute_region`

Converts ACC frontend compute constructs (`acc.parallel`, `acc.kernels`, `acc.serial`) to `acc.compute_region` wrapped in `acc.kernel_environment`. Converts `acc.loop` to SCF `parallel/for` loops with parallel dimension annotations.

The pass applies two phases of pattern rewrites:

1. Loop conversion: `acc.loop` is converted to `scf.parallel` or `scf.for` while the parent compute construct is still present (needed to determine loop conversion strategy).



2. Compute construct conversion: `acc.parallel`, `acc.kernels`, and `acc.serial` are replaced by `acc.kernel_environment` containing `acc.compute_region`.

Options ¶

`-device-type` : Target device type. One use case is ensuring that `device_type`-specific clauses are considered. Another is device-specific specializations.

`-acc-declare-gpu-module-insertion` ¶

Copy globals with `acc.declare` into the GPU module

Copies globals that have the `acc.declare` attribute into the GPU module so that device code can reference them.

Options ¶

`-cuda-unified` : Emit declaration-only device globals for CUDA unified memory.

`-acc-emit-remarks-data` ¶

Emit OpenACC data-mapping remarks

This pass emits remarks describing the data clauses associated with OpenACC data constructs and unstructured data directives.

`-acc-emit-remarks-loop` ¶

Emit OpenACC loop parallelism mapping remarks

This pass emits optimization remarks describing how loops inside OpenACC compute regions are mapped to parallelism levels (gang, worker, vector, sequential) and the corresponding GPU parallel dimensions.

The pass walks `acc.compute_region` operations that originated from OpenACC compute constructs and reports remarks for each loop carrying an `acc.par_dims` attribute.

Options ¶



`-gpu-dim-separator` : Separator between GPU index prefix and axis name in remarks

`-acc-emit-remarks-private` ¶

Emit OpenACC private and firstprivate remarks

This pass emits optimization remarks describing the private and firstprivate variables associated with OpenACC compute and loop constructs.

The pass walks compute constructs and `acc.loop` operations and, for each, reports the implicit and explicit private and firstprivate variables it carries.

`-acc-if-clause-lowering` ¶

Lower if clauses in ACC compute constructs

This pass lowers OpenACC compute constructs (parallel, kernels, serial) with `if` clauses using region specialization. It creates two execution paths: device execution when the condition is true, host execution when false.

When an ACC compute construct has an `if` clause, the construct should only execute on the device when the condition is true. If the condition is false, the code should execute on the host instead. This pass transforms:

```
acc.parallel if(%cond) { ... }
```

Into:

```
scf.if %cond {
  // Device path: clone data ops, compute construct without if, exit ops
  acc.parallel { ... }
} else {
  // Host path: original region body with ACC ops converted to host
}
```

The transformation handles:

- Data entry operations (`acc.copyin`, `acc.create`, etc.) are cloned to device path



- Data exit operations (acc.copyout, acc.delete, etc.) are cloned to device path
- The host path uses `populateACCHostFallbackPatterns` to convert ACC ops

`-acc-implicit-data` ¶

Generate implicit data attributes for OpenACC compute constructs

This pass implements the OpenACC specification for “Variables with Implicitly Determined Data Attributes” (OpenACC 3.4 spec, section 2.6.2).

The pass automatically generates data clause operations for variables used within OpenACC compute constructs (parallel, kernels, serial) that do not already have explicit data clauses. The semantics follow these rules:

1. If there is a default(none) clause visible, no implicit data actions apply.
2. An aggregate variable (arrays, derived types, etc.) will be treated as:
 - In a present clause when default(present) is visible.
 - In a copy clause otherwise.
3. A scalar variable will be treated as if it appears in:
 - A copy clause if the compute construct is a kernels construct.
 - A firstprivate clause otherwise (parallel, serial).

Options ¶

`-enable-implicit-reduction-copy` : Enable applying implicit copy in lieu of implicit firstprivate for reduction variables. This allows uniform treatment of reduction variables between combined constructs (e.g., 'parallel loop') and separate constructs (e.g., 'parallel' followed by 'loop'), where the OpenACC spec requires copy semantics for the former but firstprivate would normally apply for the latter.

`-ignore-default-none` : When disabled (default), if the region has a default(none) clause, this pass does not generate implicit data attributes. When enabled, implicit data is generated even with default(none). This is useful when the frontend has robust default(none) verification: user variables are already verified, so any remaining live-ins are variables that still need mapping handling.

`-acc-implicit-declare` ¶

Applies implicit acc declare to globals referenced in compute and routine acc regions



This pass applies implicit `acc declare` actions to global variables referenced in OpenACC compute regions and routine functions.

The pass performs the following actions:

1. Hoists address-of operations for non-constant globals out of OpenACC regions when they can be implicitly mapped rather than declared.
2. Collects global symbols referenced in:
 - OpenACC compute constructs (parallel, kernels, serial)
 - Functions marked with `acc routine`
 - Initialization regions of existing `acc declare` globals
 - Private/firstprivate/reduction recipe operations
3. Marks collected globals with the `acc.declare` attribute using the `copyin` data clause.

The pass avoids unnecessary `declare` marking by:

- Skipping function symbols (which use `acc routine` instead)
- Hoisting non-constant global references that can use implicit mapping
- Only processing symbols that are not already valid in device regions

`-acc-implicit-routine` ¶

Generate implicit `acc routine` for functions in `acc` regions

This pass implements the implicit rules described in OpenACC specification for `Routine Directive` (OpenACC 3.4 spec, section 2.15.1).

“If no explicit routine directive applies to a procedure whose definition appears in the program unit being compiled, then the implementation applies an implicit routine directive to that procedure if any of the following conditions holds:

- The procedure is called or its address is accessed in a compute region.”

The specification further states: “When the implementation applies an implicit routine directive to a procedure, it must recursively apply implicit routine directives to other procedures for which the above rules specify relevant dependencies. Such dependencies can form a cycle, so the implementation must take care to avoid infinite recursion.”

This pass implements these requirements by:



1. Walking through all OpenACC compute constructs and functions already marked with `acc routine` in the module and identifying function calls within these regions.
2. Creating implicit `acc.routine` operations for functions that don't already have routine declarations.
3. Recursively walking through all existing `acc routine` and creating implicit routine operations for function calls within these routines, while avoiding infinite recursion through proper tracking.

Options ¶

`-device-type` : Target device type. One use case is ensuring that `device_type`-specific clauses are considered. Another is device-specific specializations.

`-acc-legalize-serial` ¶

Legalize OpenACC serial constructs

This pass converts `acc.serial` constructs into `acc.parallel` constructs with `num_gangs(1)`, `num_workers(1)`, and `vector_length(1)`.

This transformation simplifies processing of `acc` regions by unifying the handling of serial and parallel constructs. Since an OpenACC serial region executes sequentially (like a parallel region with a single gang, worker, and vector), this conversion is semantically equivalent while enabling code reuse in later compilation stages.

`-acc-loop-tiling` ¶

Tile OpenACC loops with tile clauses

This pass implements loop tiling transformations for OpenACC loops that have tile clauses. The pass transforms loops with `tile(size1, size2, ...)` clauses into tiled loop nests.

For a 2-level nested loop with `tile(T1, T2)`, the transformation produces:

- Outer tile loops that iterate over tiles
- Inner element loops that iterate within each tile

Example transformation:

```
// Before:
#pragma acc loop tile(32, 32)
```

```

for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    A[i][j] = ...

// After:
for (i = 0; i < N; i += 32)          // tile loop 1
  for (j = 0; j < M; j += 32)        // tile loop 2
    for (ii = i; ii < min(N, i+32); ii++) // element loop 1
      for (jj = j; jj < min(M, j+32); jj++) // element loop 2
        A[ii][jj] = ...

```

The pass handles:

- Constant tile sizes
- Wildcard tile sizes ('*') which use a default tile size
- Collapsed loops with tile counts exceeding collapse count
- Proper handling of loop attributes (gang, worker, vector)

Options ¶

`-default-tile-size` : Default tile size to use for wildcard ('*') tile sizes

`-acc-recipe-materialization` ¶

Materialize OpenACC private, firstprivate and reduction recipes

Materializes OpenACC privatization, firstprivate and reduction recipes by cloning init, copy, combiner, and destroy into the construct. Replaces recipe references with materialized values (including `acc.reduction_init` and `acc.reduction_combine_region` for reductions) and removes unused recipe symbols.

`-acc-routine-lowering` ¶

Specialize acc routine functions for device

This pass handles `acc routine` directive by creating specialized functions with appropriate parallelism information that can be used for eventual creation of device function.



For each `acc.routine` that is not bound by name, the pass creates a new function (the “device” copy) whose body is a single `acc.compute_region` containing a clone of the original (host) function body. Parallelism is expressed by one `acc.par_width` derived from the routine’s clauses (seq, vector, worker, gang). The pass does not use `acc.kernel_environment`. It sets `acc.specialized_routine` on the new function and updates the `acc.routine`’s `func_name` to point to it. Routines with `bind(name)` and external functions are skipped.

Options ¶

`-device-type` : Target device type. One use case is ensuring that `device_type`-specific clauses are considered. Another is device-specific specializations.

`-acc-routine-to-gpu-func` ¶

Move ACC routine functions into the GPU module as `gpu.func`

This pass moves functions associated with `acc routine` (and any callees that must be present on the device) into the GPU module as `gpu.func` operations.

Options ¶

`-device-type` : Target device type. One use case is ensuring that `device_type`-specific clauses are considered. Another is device-specific specializations.

`-acc-specialize-for-device` ¶

Strip OpenACC constructs inside device code

In a specialized `acc routine` or `compute construct`, many OpenACC operations do not make sense because they are host-side constructs. This pass removes or transforms these operations appropriately.

The following operations are handled:

- Data entry ops (replaced with `var`): `acc.attach`, `acc.copyin`, `acc.create`, `acc.declare_device_resident`, `acc.declare_link`, `acc.deviceptr`, `acc.get_deviceptr`, `acc.nocreate`, `acc.present`, `acc.update_device`, `acc.use_device`
- Data exit ops (erased): `acc.copyout`, `acc.delete`, `acc.detach`, `acc.update_host`
- Structured data (inline region): `acc.data`, `acc.host_data`, `acc.kernel_environment`



- Unstructured data (erased): `acc.enter_data`, `acc.exit_data`, `acc.update`, `acc.declare_enter`, `acc.declare_exit`
- Compute constructs (inline region): `acc.parallel`, `acc.serial`, `acc.kernels`
- Runtime ops (erased): `acc.init`, `acc.shutdown`, `acc.set`, `acc.wait` (folded if known): `acc.on_device`

Options ¶

`-the-device-types` : List of integers which represent the device we are specializing for

`-acc-specialize-for-host` ¶

Convert OpenACC operations for host execution

This pass converts OpenACC operations to host-compatible representations. It serves as a conversion pass that transforms ACC constructs to enable execution on the host rather than on accelerator devices.

There are two modes of operation:

1. Default mode (orphan operations only): Only orphan operations that are not allowed outside compute regions are converted. Structured/unstructured data constructs, compute constructs, and their associated data operations are NOT removed.
2. Host fallback mode (`enableHostFallback=true`): ALL ACC operations within the region are converted to host equivalents. This is used when the `if` clause evaluates to false at runtime.

The following operations are handled:

- Atomic ops: converted to load/store operations
- Loop ops: converted to `scf.for` or `scf.execute_region`
- Data entry ops (orphan): replaced with `var` operand
- In host fallback mode: all data, compute, and runtime ops are removed

Options ¶

`-enable-host-fallback` : Enable host fallback mode which converts ALL ACC operations, not just orphan operations. Use this when the `if` clause evaluates to false.

`-offload-livein-value-canonicalization` ¶



Canonicalize live-in values for regions destined for outlining

This pass canonicalizes live-in values for regions destined for outlining. It handles operations that produce synthetic types or values that cannot be passed as arguments to outlined regions.

The pass performs the following transformations:

1. **Sinking:** Operations whose results are only used inside the region are moved into the region. This reduces the number of live-in values and keeps related operations together.
2. **Rematerialization:** Operations whose results are used both inside and outside the region are cloned into the region. The uses inside the region are updated to use the cloned operation's results.

Operations are considered candidates for these transformations if they implement the `OutlineRematerializationOpInterface` or match constant patterns. These operations typically produce synthetic types (shapes, bounds, field indices) that cannot be passed as function arguments.

The pass iterates until convergence since canonicalizing one value may expose new candidates (e.g., a bounds operation's operands may themselves be constants that should be rematerialized).

Example transformation (rematerialization):

```
// Before:
%c0 = arith.constant 0 : index
%c10 = arith.constant 10 : index
%bounds = acc.bounds lowerbound(%c0 : index) upperbound(%c10 : index)
acc.parallel {
  %priv = acc.private varPtr(%ptr : ...) bounds(%bounds) -> ...
  acc.yield
}
// %bounds is also used elsewhere

// After:
%c0 = arith.constant 0 : index
%c10 = arith.constant 10 : index
%bounds = acc.bounds lowerbound(%c0 : index) upperbound(%c10 : index)
acc.parallel {
  %c0_clone = arith.constant 0 : index
  %c10_clone = arith.constant 10 : index
  %bounds_clone = acc.bounds lowerbound(%c0_clone : index) upperbound(%c10_clone : index)
```



```
%priv = acc.private varPtr(%ptr : ...) bounds(%bounds_clone) -> ...
acc.yield
}
```

Example transformation (sinking):

```
// Before:
%c0 = arith.constant 0 : index
%c10 = arith.constant 10 : index
%bounds = acc.bounds lowerbound(%c0 : index) upperbound(%c10 : index)
acc.parallel {
  %priv = acc.private varPtr(%ptr : ...) bounds(%bounds) -> ...
  acc.yield
}
// %bounds is NOT used elsewhere

// After:
acc.parallel {
  %c0 = arith.constant 0 : index
  %c10 = arith.constant 10 : index
  %bounds = acc.bounds lowerbound(%c0 : index) upperbound(%c10 : index)
  %priv = acc.private varPtr(%ptr : ...) bounds(%bounds) -> ...
  acc.yield
}
```

`-offload-target-verifier` ¶

Verify values and symbols live into offload regions for legality

This pass verifies that values and symbols used within OpenACC compute constructs and other offload regions are legal for the target execution model.

The pass performs two main checks:

1. **Live-in Value Verification:** Checks that all values that are live into an offload region are valid for use in that region. This includes checking that pointer-like and mappable types have appropriate data clauses or device attributes.

2. **Symbol Use Verification:** Checks that all symbol references within an offload region are valid for that region. This includes checking for `acc.routine` declarations and `acc.declare` attributes.

The `device_type` option notes the target execution model:

- `none, nvidia, radeon`: Device execution (GPU offload)
- `host, multicore`: Host execution

When `soft_check` is enabled, the pass only emits debug messages for illegal values/symbols instead of failing compilation. This is useful for diagnostic purposes.

Options ¶

```
-device-type : Target device type. One use case is ensuring that device_type-specific
clauses are considered. Another is device-specific specializations.
-soft-check : When true, illegal values are printed via LLVM_DEBUG instead of failing
compilation. Useful for diagnostic purposes.
```

-openacc-legalize-data-values ¶

Legalizes SSA values in compute regions with results from data clause operations

This pass replace uses of the `varPtr` in compute regions (kernels, parallel, serial) with the result of data clause operations (`accPtr`).

Options ¶

```
-host-to-device : Replace varPtr uses with accPtr if true. Replace accPtr uses
with varPtr if false
-apply-to-acc-data-construct : Replaces varPtr uses with accPtr for acc compute regions
contained within acc.data or acc.declare region.
```

‘affine’ Dialect Passes ¶

-affine-data-copy-generate ¶

Generate explicit copying for affine memory operations



Options ¶

```

-fast-mem-capacity      : Set fast memory space capacity in KiB (default: unlimited)
-fast-mem-space         : Fast memory space identifier for copy generation (default: 1)
-generate-dma           : Generate DMA instead of point-wise copy
-min-dma-transfer       : Minimum DMA transfer size supported by the target in bytes
-slow-mem-space         : Slow memory space identifier for copy generation (default: 0)
-skip-non-unit-stride-loops : Testing purposes: avoid non-unit stride loop choice depths
                           for copy placement
-tag-mem-space          : Tag memory space identifier for copy generation (default: 0)

```

-affine-expand-index-ops ¶

Lower affine operations operating on indices into more fundamental operations

-affine-expand-index-ops-as-affine ¶

Lower affine operations operating on indices into affine.apply operations

-affine-fold-memref-alias-ops ¶

Fold memref alias ops into affine memory ops

The pass folds `memref.subview`, `memref.expand_shape`, and `memref.collapse_shape` operations into affine memory operations (currently only `affine.load` and `affine.store`). This is similar to the `fold-memref-alias-ops` pass in the `memref` dialect but adds handling specific to affine operations.

-affine-loop-coalescing ¶

Coalesce nested loops with independent bounds into a single loop

-affine-loop-fusion ¶

Fuse affine loop nests

This pass performs fusion of loop nests using a slicing-based approach. The transformation works on an MLIR Block granularity and applies to all blocks of the pass is run on. It combines two fusion strategies: producer-consumer fusion and sibling fusion. Producer-consumer fusion is aimed at fusing pairs of loops where the first

one writes to a memref that the second reads. Sibling fusion targets pairs of loops that share no dependences between them but that load from the same memref. The fused loop nests, when possible, are rewritten to access significantly smaller local buffers instead of the original memref's, and the latter are often either completely optimized away or contracted. This transformation leads to enhanced locality and lower memory footprint through the elimination or contraction of temporaries/intermediate memref's. These benefits are sometimes achieved at the expense of redundant computation through a cost model that evaluates available choices such as the depth at which a source slice should be materialized in the designation slice.

Example 1: Producer-consumer fusion. Input:

```
func.func @producer_consumer_fusion(%arg0: memref<10xf32>, %arg1: memref<10xf32>) {
  %0 = memref.alloc() : memref<10xf32>
  %1 = memref.alloc() : memref<10xf32>
  %cst = arith.constant 0.000000e+00 : f32
  affine.for %arg2 = 0 to 10 {
    affine.store %cst, %0[%arg2] : memref<10xf32>
    affine.store %cst, %1[%arg2] : memref<10xf32>
  }
  affine.for %arg2 = 0 to 10 {
    %2 = affine.load %0[%arg2] : memref<10xf32>
    %3 = arith.addf %2, %2 : f32
    affine.store %3, %arg0[%arg2] : memref<10xf32>
  }
  affine.for %arg2 = 0 to 10 {
    %2 = affine.load %1[%arg2] : memref<10xf32>
    %3 = arith.mulf %2, %2 : f32
    affine.store %3, %arg1[%arg2] : memref<10xf32>
  }
  return
}
```

Output:

```
func.func @producer_consumer_fusion(%arg0: memref<10xf32>, %arg1: memref<10xf32>) {
  %0 = memref.alloc() : memref<1xf32>
  %1 = memref.alloc() : memref<1xf32>
```



```
%cst = arith.constant 0.000000e+00 : f32
affine.for %arg2 = 0 to 10 {
  affine.store %cst, %0[0] : memref<1xf32>
  affine.store %cst, %1[0] : memref<1xf32>
  %2 = affine.load %1[0] : memref<1xf32>
  %3 = arith.mulf %2, %2 : f32
  affine.store %3, %arg1[%arg2] : memref<10xf32>
  %4 = affine.load %0[0] : memref<1xf32>
  %5 = arith.addf %4, %4 : f32
  affine.store %5, %arg0[%arg2] : memref<10xf32>
}
return
}
```

Example 2: Sibling fusion. Input:

```
func.func @sibling_fusion(%arg0: memref<10x10xf32>, %arg1: memref<10x10xf32>,
                          %arg2: memref<10x10xf32>, %arg3: memref<10x10xf32>,
                          %arg4: memref<10x10xf32>) {
  affine.for %arg5 = 0 to 3 {
    affine.for %arg6 = 0 to 3 {
      %0 = affine.load %arg0[%arg5, %arg6] : memref<10x10xf32>
      %1 = affine.load %arg1[%arg5, %arg6] : memref<10x10xf32>
      %2 = arith.mulf %0, %1 : f32
      affine.store %2, %arg3[%arg5, %arg6] : memref<10x10xf32>
    }
  }
  affine.for %arg5 = 0 to 3 {
    affine.for %arg6 = 0 to 3 {
      %0 = affine.load %arg0[%arg5, %arg6] : memref<10x10xf32>
      %1 = affine.load %arg2[%arg5, %arg6] : memref<10x10xf32>
      %2 = arith.addf %0, %1 : f32
      affine.store %2, %arg4[%arg5, %arg6] : memref<10x10xf32>
    }
  }
  return
}
```

Output:

```
func.func @sibling_fusion(%arg0: memref<10x10xf32>, %arg1: memref<10x10xf32>,
                          %arg2: memref<10x10xf32>, %arg3: memref<10x10xf32>,
                          %arg4: memref<10x10xf32>) {
  affine.for %arg5 = 0 to 3 {
    affine.for %arg6 = 0 to 3 {
      %0 = affine.load %arg0[%arg5, %arg6] : memref<10x10xf32>
      %1 = affine.load %arg1[%arg5, %arg6] : memref<10x10xf32>
      %2 = arith.mulf %0, %1 : f32
      affine.store %2, %arg3[%arg5, %arg6] : memref<10x10xf32>
      %3 = affine.load %arg0[%arg5, %arg6] : memref<10x10xf32>
      %4 = affine.load %arg2[%arg5, %arg6] : memref<10x10xf32>
      %5 = arith.addf %3, %4 : f32
    }
  }
}
```



```

    affine.store %5, %arg4[%arg5, %arg6] : memref<10x10xf32>
  }
}
return
}

```

Options ¶

```

-compute-tolerance : Fractional increase in additional computation tolerated while fusing
-fast-mem-space    : Faster memory space number to promote fusion buffers to
-local-buf-threshold : Threshold size (KiB) for promoting local buffers to fast memory
space
-maximal           : Enables maximal loop fusion
-mode             : fusion mode to attempt

```

-affine-loop-invariant-code-motion ¶

Hoist loop invariant instructions outside of affine loops

-affine-loop-normalize ¶

Apply normalization transformations to affine loop-like ops

Options ¶

```

-promote-single-iter : Promote single iteration loops
-use-expensive-math  : Use Presburger arithmetic to constantize loop upper bounds (higher
analysis cost)

```

-affine-loop-tile ¶

Tile affine loop nests

Options ¶



```
-cache-size : Set size of cache to tile for in KiB (default: 512)
-separate   : Separate full and partial tiles (default: false)
-tile-size  : Use this tile size for all loops
-tile-sizes : List of tile sizes for each perfect nest (overridden by -tile-size)
```

-affine-loop-unroll ¶

Unroll affine loops

Options ¶

```
-unroll-factor      : Use this unroll factor for all loops being unrolled, set it to -1,
and it will fully unroll.
-unroll-up-to-factor : Allow unrolling up to the factor specified
-unroll-num-reps     : Unroll innermost loops repeatedly this many times
-unroll-full-threshold : Unroll all loops with trip count less than or equal to this
-cleanup-unroll      : Fully unroll the cleanup loop when possible.
```

-affine-loop-unroll-jam ¶

Unroll and jam affine loops

Options ¶

```
-unroll-jam-factor : Use this unroll jam factor for all loops (default 4)
```

-affine-parallelize ¶

Convert affine.for ops into 1-D affine.parallel

Options ¶

```
-max-nested : Maximum number of nested parallel loops to produce. Defaults to
unlimited (UINT_MAX).
```



-parallel-reductions : Whether to parallelize reduction loops. Defaults to false.

-affine-pipeline-data-transfer ¶

Pipeline non-blocking data transfers between explicitly managed levels of the memory hierarchy

This pass performs a transformation to overlap non-blocking DMA operations in a loop with computations through double buffering. This is achieved by advancing dma_start operations with respect to other operations.

Input

```
func.func @pipelinedatatransfer() {
  %0 = memref.alloc() : memref<256xf32>
  %1 = memref.alloc() : memref<32xf32, 1>
  %2 = memref.alloc() : memref<1xf32>
  %c0 = arith.constant 0 : index
  %c128 = arith.constant 128 : index
  affine.for %i0 = 0 to 8 {
    affine.dma_start %0[%i0], %1[%i0], %2[%c0], %c128 : memref<256xf32>, memref<32xf32, 1>,
    memref<1xf32>
    affine.dma_wait %2[%c0], %c128 : memref<1xf32>
    %3 = affine.load %1[%i0] : memref<32xf32, 1>
    %4 = "compute"(%3) : (f32) -> f32
    affine.store %4, %1[%i0] : memref<32xf32, 1>
  }
  return
}
```

Output

```
module {
  func.func @pipelinedatatransfer() {
    %c8 = arith.constant 8 : index
    %c0 = arith.constant 0 : index
    %0 = memref.alloc() : memref<256xf32>
    %c0_0 = arith.constant 0 : index
    %c128 = arith.constant 128 : index
```



```

%1 = memref.alloc() : memref<2x32xf32, 1>
%2 = memref.alloc() : memref<2x1xf32>
affine.dma_start %0[%c0], %1[%c0 mod 2, %c0], %2[%c0 mod 2, symbol(%c0_0)], %c128 :
memref<256xf32>, memref<2x32xf32, 1>, memref<2x1xf32>
affine.for %arg0 = 1 to 8 {
  affine.dma_start %0[%arg0], %1[%arg0 mod 2, %arg0], %2[%arg0 mod 2, symbol(%c0_0)],
%c128 : memref<256xf32>, memref<2x32xf32, 1>, memref<2x1xf32>
  %8 = affine.apply #map3(%arg0)
  %9 = affine.apply #map4(%8)
  %10 = affine.apply #map4(%8)
  affine.dma_wait %2[%8 mod 2, symbol(%c0_0)], %c128 : memref<2x1xf32>
  %11 = affine.load %1[%8 mod 2, %8] : memref<2x32xf32, 1>
  %12 = "compute"(%11) : (f32) -> f32
  affine.store %12, %1[%8 mod 2, %8] : memref<2x32xf32, 1>
}
%3 = affine.apply #map3(%c8)
%4 = affine.apply #map4(%3)
%5 = affine.apply #map4(%3)
affine.dma_wait %2[%3 mod 2, symbol(%c0_0)], %c128 : memref<2x1xf32>
%6 = affine.load %1[%3 mod 2, %3] : memref<2x32xf32, 1>
%7 = "compute"(%6) : (f32) -> f32
affine.store %7, %1[%3 mod 2, %3] : memref<2x32xf32, 1>
memref.dealloc %2 : memref<2x1xf32>
memref.dealloc %1 : memref<2x32xf32, 1>
return
}
}

```

-affine-raise-from-memref ¶

Turn some memref operators to affine operators where supported

Raise memref.load and memref.store to affine.store and affine.load, inferring the affine map of those operators if needed. This allows passes like `-affine-scalrep` to optimize those loads and stores (forwarding them or eliminating them). They can be turned back to memref dialect ops with `-lower-affine`.

-affine-scalrep ¶



Replace affine memref accesses by scalars by forwarding stores to loads and eliminating redundant loads

This pass performs store to load forwarding and redundant load elimination for affine memref accesses and potentially eliminates the entire memref if all its accesses are forwarded.

Input

```
func.func @store_load_affine_apply() -> memref<10x10xf32> {
  %cf7 = arith.constant 7.0 : f32
  %m = memref.alloc() : memref<10x10xf32>
  affine.for %i0 = 0 to 10 {
    affine.for %i1 = 0 to 10 {
      affine.store %cf7, %m[%i0, %i1] : memref<10x10xf32>
      %v0 = affine.load %m[%i0, %i1] : memref<10x10xf32>
      %v1 = arith.addf %v0, %v0 : f32
    }
  }
  return %m : memref<10x10xf32>
}
```

Output

```
module {
  func.func @store_load_affine_apply() -> memref<10x10xf32> {
    %cst = arith.constant 7.000000e+00 : f32
    %0 = memref.alloc() : memref<10x10xf32>
    affine.for %arg0 = 0 to 10 {
      affine.for %arg1 = 0 to 10 {
        affine.store %cst, %0[%arg0, %arg1] : memref<10x10xf32>
        %1 = arith.addf %cst, %cst : f32
      }
    }
    return %0 : memref<10x10xf32>
  }
}
```



-affine-simplify-min-max ¶*Simplify affine min/max/apply*

Apply the SimplifyAffineMaxOp, SimplifyAffineMinOp and SimplifyAffineApplyOp patterns in addition to AffineMin/Max canonicalization patterns until a fixed point is reached. These patterns apply ValueBoundsOp interface on AffineMin/Max ops and additional simplifications such as:

```
min(x, y, cst) / cst -> 1
```

when x, y, cst are all ≥ 0 . This is typically useful to extract more static information from IR after tiling but can also come at a cost due to Presburger-style analysis.

-affine-simplify-structures ¶*Simplify affine expressions in maps/sets and normalize memrefs***-affine-simplify-with-bounds ¶***Simplify affine index operations using value bounds analysis*

This pass simplifies `affine.delinearize_index` / `affine.linearize_index` pairs and `affine.for` by using value bounds analysis to match basis products. Unlike the built-in canonicalization patterns which only use exact `OpFoldResult` comparisons, this pass can prove equality of dynamic basis products through `ValueBoundsConstraintSet`.

-affine-super-vectorize ¶*Vectorize to a target independent n-D vector abstraction***Options ¶**

```
-virtual-vector-size : Specify an n-D virtual vector size for vectorization. This must be
greater than zero.
-test-fastest-varying : Specify a 1-D, 2-D or 3-D pattern of fastest varying memory
dimensions to match. See defaultPatterns in Vectorize.cpp for a description and examples.
This is used for testing purposes
```



```
-vectorize-reductions : Vectorize known reductions expressed via iter_args. Switched off by default.
```

‘amdgpu’ Dialect Passes ¶

-amdgpu-emulate-atomics ¶

Emulate atomic operations on chipsets that do not support them

This pass rewrites any AMDGPU-specific atomic operation that is not supported on the given `chipset` into a compare-and-swap loop.

Options ¶

```
-chipset : Chipset that these operations will run on
```

-amdgpu-maskedload-to-load ¶

Lower the operations from the vector maskedload to vector load

This pass creates a transfer read op lowering optimization. The lowering will produce a conditional check at runtime. If within bounds, a vector transfer read op will be lowered to a combination of `vector.load`, `arith.select` and `vector.broadcast`. If not, it will fallback to the default lowering of the `transfer_read` op.

This pattern will make it possible for masked `transfer_read` to be lowered towards buffer load with bounds check, allowing a more optimized global load accessing pattern compared with existing implementation of `llvm.intr.masked.load` on vectors.

-amdgpu-resolve-strided-metadata ¶

Resolve `memref.extract_strided_metadata` on AMDGPU ops

This pass rewrites `memref.extract_strided_metadata` operations targeting the AMDGPU dialect casts.

The patterns in this pass should normally be run alongside those in `-expand-strided-metadata`, and creating a pass that combines those two sets of patterns is the recommended way to use this functionality. However, this pass (which will likely need a second `-expand-strided-metadata` after it) is provided so that simple usecases do

not need to create custom passes. These patterns have not been added to `-expand-strided-metadata` to prevent the memref dialect from depending on platform-specific code.

‘arith’ Dialect Passes ¶

`-arith-emulate-unsupported-floats` ¶

Emulate operations on unsupported floats with `extf/truncf`

Emulate arith and vector floating point operations that use float types which are unsupported on a target by inserting `extf/truncf` pairs around all such operations in order to produce arithmetic that can be performed while preserving the original rounding behavior.

This pass does not attempt to reason about the operations being performed to determine when type conversions can be elided.

Options ¶

```
-source-types : MLIR types without arithmetic support on a given target
-target-type  : MLIR type to convert the unsupported source types to
```

`-arith-emulate-wide-int` ¶

*Emulate $2*N$ -bit integer operations using N -bit operations*

Emulate arith integer operations that use too wide integer types with equivalent operations on supported narrow integer types. This is done by splitting original integer values into two halves.

This pass is intended preserve semantics but not necessarily provide the most efficient implementation. TODO: Optimize op emulation.

Currently, only power-of-two integer bitwidths are supported.

Options ¶

```
-widest-int-supported : Widest integer type supported by the target
```

`-arith-expand` ¶

Legalize Arith ops to be convertible to LLVM.

Options ¶

```
-include-bf16          : Enable the BF16 expansion patterns
-include-f8e8m0        : Enable the F8E8M0 expansion patterns
-include-f4e2m1        : Enable the F4E2M1 expansion patterns
-include-flush-denormals : Enable expansion of `arith.flush_denormals` on IEEE-like
floating-point types
-include-min-max-f      : Enable expansion of the floating-point min/max ops
(maximumf/minimumf/maxnumf/minnumf). These ops also have a direct arith-to-llvm lowering to
the `llvm.intr.maximum/minimum/maxnum/minnum` intrinsics; disable this so pipelines that
run arith-to-llvm can use those single-instruction lowerings instead of the cmpf/select
expansion
-include-min-max-i      : Enable expansion of the signed/unsigned integer min/max ops
(maxsi/maxui/minsi/minui). These ops also have a direct arith-to-llvm lowering to the
`llvm.intr.smax/umax/smin/umin` intrinsics; disable this so pipelines that run arith-to-
llvm can use those single-instruction lowerings instead of the cmpi/select expansion
```

-arith-int-range-narrowing ¶

Reduce integer operations bitwidth based on integer range analysis

This pass runs integer range analysis and tries to narrow arith ops to the specified bitwidth based on its results.

bitwidthsSupported assumed to be not wider than index type. TODO: get index width from DLTI.

Options ¶

```
-int-bitwidths-supported : Integer bitwidths supported
```

-arith-unsigned-when-equivalent ¶

Replace signed ops with unsigned ones where they are proven equivalent



Replace signed ops with their unsigned equivalents when integer range analysis determines that their arguments and results are all guaranteed to be non-negative when interpreted as signed integers. When this occurs, we know that the semantics of the signed and unsigned operations are the same, since they share the same behavior when their operands and results are in the range $[0, \text{signed_max}(\text{type})]$.

The affect ops include division, remainder, shifts, min, max, and integer comparisons.

`-int-range-optimizations` ¶

Do optimizations based on integer range analysis

This pass runs integer range analysis and applies optimizations based on its results. It replaces operations with known-constant results with said constants, rewrites $(0 \leq \%x < D) \bmod D$ to $\%x$.

'arm_sme' Dialect Passes ¶

`-arm-sme-outer-product-fusion` ¶

Fuse 'arm_sme.outerproduct' operations into 2-way or 4-way widening variants

This pass fuses 'arm_sme.outerproduct' operations that are chained via the accumulator into 2-way or 4-way ArmSME outer product operations.

For example:

```
%a0_ext = arith.extf %a0 : vector<[4]xf16> to vector<[4]xf32>
%b0_ext = arith.extf %b0 : vector<[4]xf16> to vector<[4]xf32>
%a1_ext = arith.extf %a1 : vector<[4]xf16> to vector<[4]xf32>
%b1_ext = arith.extf %b1 : vector<[4]xf16> to vector<[4]xf32>

%0 = arm_sme.outerproduct %a0_ext, %b0_ext : vector<[4]xf32>, vector<[4]xf32>
%1 = arm_sme.outerproduct %a1_ext, %b1_ext acc(%0) : vector<[4]xf32>, vector<[4]xf32>
```

Becomes:

```
%a_packed = vector.interleave %a0, %a1 : vector<[4]xf16> -> vector<[8]xf16>
%b_packed = vector.interleave %b0, %b1 : vector<[4]xf16> -> vector<[8]xf16>
```



```
%0 = arm_sme.fmpa_2way %a_packed, %b_packed : vector<[8]xf16>, vector<[8]xf16> into
vector<[4]x[4]xf32>
```

For further information on the 2-way or 4-way widening ops see:

https://mlir.llvm.org/docs/Dialects/ArmSME/#arm_sme_fmopa_2way-arm_sme_fmopa_2wayop

https://mlir.llvm.org/docs/Dialects/ArmSME/#arm_sme_fmopa_4way-arm_sme_fmopa_4wayop

-arm-sme-vector-legalization ⓘ

Legalize vectors for ArmSME

This pass legalizes vector operations so that they can be lowered to ArmSME. This includes decomposing operations that operate on vector types larger than a single SME tile (e.g. `vector<[8]x[8]xf32>`) into multiple SME tile-sized operations, as well as rewrites needed to get operations into forms compatible with SME lowerings.

Note: Decomposition is currently limited to vector types that are an exact multiple of SME tiles. That is scalable in two dimensions, with both the rows and columns divisible by the SVE vector length for the element type.

-enable-arm-streaming ⓘ

Enable Armv9 Streaming SVE mode

Enables the Armv9 Streaming SVE mode [1] for func.func ops by annotating them with attributes. See options for more details.

[1] <https://developer.arm.com/documentation/ddi0616/aa>

Options ⓘ

```
-streaming-mode          : Select how streaming-mode is managed at the function-level.
-za-mode                 : Select how ZA-storage is managed at the function-level.
-if-required-by-ops      : Only apply the selected streaming/ZA modes if the function
                           contains ops that implement the ArmSMETileOpInterface.
-if-scalable-and-supported : Only apply the selected streaming/ZA modes if the function
                           contains supported scalable vector operations.
```



`-test-arm-sme-tile-allocation` ¶

Tests SME ‘virtual tile’ allocation

This pass does tile allocation for SME “virtual tiles”. It is run at the ‘func.func’ op level, and assigns tile IDs (via an attribute) to all ops that implement the `ArmSMETileOpInterface`. Note: This pass is only intended to be used for testing, tile allocation is done as part of the ArmSME to LLVM conversion (`convert-arm-sme-to-llvm`).

Options ¶

```
-dump-tile-live-ranges : Dump the live ranges of SME tiles (for debugging)
-preprocess-only       : Only preprocess IR so it is ready for tile allocation (but do not
                           allocate any tiles)
```

‘arm_sve’ Dialect Passes ¶

`-arm-sve-legalize-vector-storage` ¶

Ensures stores of SVE vector types will be legal

This pass ensures that loads, stores, and allocations of SVE vector types will be legal in the LLVM backend. It does this at the memref level, so this pass must be applied before lowering all the way to LLVM.

This pass currently addresses two issues.

Loading and storing predicate types ¶

It is only legal to load/store predicate types equal to (or greater than) a full predicate register, which in MLIR is `vector<[16]xi1>`. Smaller predicate types (`vector<[1|2|4|8]xi1>`) must be converted to/from a full predicate type (referred to as a `svbool`) before and after storing and loading respectively. This pass does this by widening allocations and inserting conversion intrinsics. Note: Non-powers-of-two masks (e.g. `vector<[7]xi1>`), which are not SVE predicates, are ignored.

For example:

```
%alloca = memref.alloca() : memref<vector<[4]xi1>>
%mask = vector.constant_mask [4] : vector<[4]xi1>
memref.store %mask, %alloca[] : memref<vector<[4]xi1>>
```

```
%reload = memref.load %alloca[] : memref<vector<[4]xi1>>
```

Becomes:

```
%alloca = memref.alloca() {alignment = 1 : i64} : memref<vector<[16]xi1>>
%mask = vector.constant_mask [4] : vector<[4]xi1>
%svbool = arm_sve.convert_to_svbool %mask : vector<[4]xi1>
memref.store %svbool, %alloca[] : memref<vector<[16]xi1>>
%reload_svbool = memref.load %alloca[] : memref<vector<[16]xi1>>
%reload = arm_sve.convert_from_svbool %reload_svbool : vector<[4]xi1>
```

Relax alignments for SVE vector allocas ¶

The storage for SVE vector types only needs to have an alignment that matches the element type (for example 4 byte alignment for `f32 s`). However, the LLVM backend currently defaults to aligning to `base size x element size` bytes. For non-legal vector types like `vector<[8]xf32>` this results in $8 \times 4 = 32$ -byte alignment, but the backend only supports up to 16-byte alignment for SVE vectors on the stack. Explicitly setting a smaller alignment prevents this issue.

‘async’ Dialect Passes ¶

-async-func-to-async-runtime ¶

Lower `async.func` operations to the explicit `async.runtime` and `async.coro` operations

-async-parallel-for ¶

Convert `scf.parallel` operations to multiple `async compute ops` executed concurrently for non-overlapping iteration ranges

Options ¶

```
-async-dispatch : Dispatch async compute tasks using recursive work splitting. If `false`
async compute tasks will be launched using simple for loop in the caller thread.
-num-workers    : The number of available workers to execute async operations. If `-1` the
```

value will be retrieved from the runtime.

`-min-task-size` : The minimum task size for sharding parallel operation.

`-async-runtime-policy-based-ref-counting` ¶

Policy based reference counting for Async runtime operations

This pass works at the async runtime abstraction level, after all `async.execute` and `async.await` operations are lowered to the async runtime API calls, and async coroutine operations.

This pass doesn't rely on reference counted values liveness analysis, and instead uses simple policy to create reference counting operations. If the program violates any of the assumptions, then this pass might lead to memory leaks or runtime errors.

The default reference counting policy assumptions:

1. Async token can be awaited or added to the group only once.
2. Async value or group can be awaited only once.

Under these assumptions reference counting only needs to drop reference:

1. After `async.runtime.await` operation for async tokens and groups (until error handling is not implemented for the sync await).
2. After `async.runtime.is_error` operation for async tokens and groups (this is the last operation in the coroutine resume function).
3. After `async.runtime.load` operation for async values.

This pass introduces significantly less runtime overhead compared to the automatic reference counting.

`-async-runtime-ref-counting` ¶

Automatic reference counting for Async runtime operations

This pass works at the async runtime abstraction level, after all `async.execute` and `async.await` operations are lowered to the async runtime API calls, and async coroutine operations.

It relies on the LLVM coroutines switched-resume lowering semantics for the correct placing of the reference counting operations.

See: <https://llvm.org/docs/Coroutines.html#switched-resume-lowering>



-async-runtime-ref-counting-opt ¶

Optimize automatic reference counting operations for theAsync runtime by removing redundant operations

-async-to-async-runtime ¶

Lower all high level async operations (e.g. `async.execute`) to the explicit `async.runtime` and `async.coro` operations

'emitc' Dialect Passes ¶**-form-expressions** ¶

Form C-style expressions from C-operator ops

The pass wraps emitc ops modelling C operators in emitc.expression ops and then folds single-use expressions into their users where possible.

-mlgo-add-reflection-map ¶

Add a reflection map and a helper method to EmitC classes for runtime field lookup.

This pass adds a `reflectionMap` field and an accompanying `getBufferForName` method to EmitC classes, enabling runtime lookup of class fields by name. Each `emitc.field` is expected to have an attribute (configured by the `included-field-attrs` option) that is an array containing a single string attribute which will be used as the reflection map key. If none of the specified attributes are present on the field, the field will be excluded from the reflection map.

Example:

```
emitc.class @MyClass {
  emitc.field @field0 : !emitc.array<1xf32> {emitc.field_ref = ["feature0"]}
  emitc.func @execute() { ... }
}
```

Becomes:

```
emitc.class @MyClass {
  emitc.field @field0 : !emitc.array<1xf32> {emitc.field_ref = ["feature0"]}
  // reflectionMap maps the string "feature0" to the address of @field0
}
```



```
emitc.field @reflectionMap : const std::map<string, char*> = {"feature0", &field0}}
emitc.func @getBufferForName(%name: string) -> char* {
    return reflectionMap.at(%name)
}
emitc.func @execute() { ... }
}
```

Options ¶

-included-field-attrs : Attribute keys used to extract field names from a field's attribute dictionary

-wrap-emitc-func-in-class ¶

Wrap functions in classes, using arguments as fields.

This pass transforms `emitc.func` operations into `emitc.class` operations. Function arguments become fields of the class, and the function body is moved to a new member method within the class. By default, this is `operator>()()`.

If the corresponding function argument has attributes (accessed via `argAttrs`), these attributes are attached to the field operation. Otherwise, the field is created without additional attributes.

Example:

```
emitc.func @model(%input_data : !emitc.array<1xf32> {emitc.opaque = "input_tensor"})
attributes { } {
    %0 = "emitc.constant"() <{value = 0 : index}> : () -> !emitc.size_t
    %1 = subscript %input_data[%0] : (!emitc.array<1xf32>, !emitc.size_t) ->
!emitc.lvalue<f32>
    return
}
// becomes
emitc.class @modelClass {
    emitc.field @input_tensor : !emitc.array<1xf32> {emitc.opaque = "input_tensor"}
    emitc.func @operator() {
        %0 = "emitc.constant"() <{value = 0 : index}> : () -> !emitc.size_t
```

```

%1 = get_field @input_tensor : !emitc.array<1xf32>
%2 = subscript %1[%0] : (!emitc.array<1xf32>, !emitc.size_t) -> !emitc.lvalue<f32>
return
}
}

```

Options ¶

```

-func-name           : The name of the newly generated member function with body matching the
                        input function.
-class-name-format : Format string for the generated class name where the function name
                        placeholder '{}' is optional.

```

‘func’ Dialect Passes ¶

-duplicate-function-elimination ¶

Deduplicate functions

Deduplicate functions that are equivalent in all aspects but their symbol name. The pass chooses one representative per equivalence class, erases the remainder, and updates function calls accordingly.

‘gpu’ Dialect Passes ¶

-gpu-async-region ¶

Make GPU ops async

-gpu-decompose-memrefs ¶

Decomposes memref index computation into explicit ops.

This pass decomposes memref index computation into explicit computations on sizes/strides, obtained from `memref.extract_memref_metadata` which it tries to place outside of `gpu.launch` body. Memrefs are then reconstructed using `memref.reinterpret_cast`. This is needed for as some targets (SPIR-V) lower memrefs to bare pointers and sizes/strides for dynamically-sized memrefs are not available inside `gpu.launch`.



-gpu-eliminate-barriers ¶

Erase unnecessary barriers

Barrier elimination pass. If a barrier does not enforce any conflicting pair of memory effects, including a pair that is enforced by another barrier, it is unnecessary and can be removed. Adapted from “High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs” by Moses, Ivanov, Domke, Endo, Doerfert, and Zinenko in PPoPP 2023 and implementation in Polygeist.

-gpu-kernel-outlining ¶

Outline `gpu.launch` bodies to kernel functions

Options ¶

```
-data-layout-str : String description of the data layout
```

-gpu-launch-sink-index-computations ¶

Sink index computations into `gpu.launch` body

-gpu-map-parallel-loops ¶

Greedily maps loops to GPU hardware dimensions.

Maps the parallel loops found in the given function to workgroups. The first loop encountered will be mapped to the global workgroup and the second loop encountered to the local workgroup. Within each mapping, the first three dimensions are mapped to x/y/z hardware ids and all following dimensions are mapped to sequential loops.

Ordering of the loop mapping against the different dimensions is controlled by the `mapping-policy` option. Two policies are supported:

1. `outermost-first` (default): the outermost loop maps to X, then Y and finally Z.
2. `innermost-first`: the innermost loop maps to X, then Y and finally Z.

Options ¶



`-mapping-policy` : Policy outlining how to assign loops to GPU dimensions. Supported values are ``outermost-first`` and ``innermost-first``.

`-gpu-module-to-binary` ¶

Transforms a GPU module into a GPU binary.

This pass searches for all nested GPU modules and serializes the module using the target attributes attached to the module, producing a GPU binary with an object for every target.

The `format` argument can have the following values:

1. `offloading, llvm`: produces an offloading representation.
2. `assembly, isa`: produces assembly code.
3. `binary, bin`: produces binaries.
4. `fatbinary, fatbin`: produces fatbinaries.

Options ¶

```
-toolkit : Toolkit path.
-l       : Extra files to link to.
-opts    : Command line options to pass to the tools.
-format   : The target representation of the compilation process.
-section  : ELF section where binary is to be located.
```

`-nvvm-attach-target` ¶

Attaches an NVVM target attribute to a GPU Module.

This pass searches for all GPU Modules in the immediate regions and attaches an NVVM target if the module matches the name specified by the `module` argument.

Example:

```
// File: in.mlir:
gpu.module @nvvm_module_1 {...}
gpu.module @nvvm_module_2 {...}
```



```

gpu.module @rocdl_module_1 {...}

// mlir-opt --nvvm-attach-target="module=nvvm.* chip=sm_90" in.mlir
gpu.module @nvvm_module_1 [#nvvm.target<chip = "sm_90">] {...}
gpu.module @nvvm_module_2 [#nvvm.target<chip = "sm_90">] {...}
gpu.module @rocdl_module_1 {...}

```

Options ¶

-module	: Regex used to identify the modules to attach the target to.
-triple	: Target triple.
-chip	: Target chip.
-features	: Target features.
-O	: Optimization level.
-fast	: Enable fast math mode.
-ftz	: Enable flush to zero for denormals.
-collect-compiler-diagnostics	: Enable collection of compiler diagnostics.
-l	: Extra bitcode libraries paths to link to.
-ptxas-cmd-options	: Command line options passed to downstream compiler
-verify-target-arch	: Enable verification of the target architecture

-rocdl-attach-target ¶

Attaches a ROCDL target attribute to a GPU Module.

This pass searches for all GPU Modules in the immediate regions and attaches a ROCDL target if the module matches the name specified by the `module` argument.

Example:

```

// File: in.mlir:
gpu.module @nvvm_module_1 {...}
gpu.module @nvvm_module_2 {...}
gpu.module @rocdl_module_1 {...}

// mlir-opt --nvvm-attach-target="module=rocdl.* chip=gfx90a" in.mlir
gpu.module @nvvm_module_1 {...}

```



```
gpu.module @nvvm_module_2 {...}
gpu.module @rocdl_module_1 [#rocdl.target<chip = "gfx90a">] {...}
```

Options ¶

```
-module      : Regex used to identify the modules to attach the target to.
-triple      : Target triple.
-chip        : Target chip.
-features    : Target features.
-abi         : ABI version.
-O           : Optimization level.
-wave64      : Use Wave64 mode.
-fast        : Enable fast relaxed math opt.
-daz         : Enable denormals are zero opt.
-finite-only : Enable finite only opt.
-unsafe-math : Enable unsafe math opt.
-correct-sqrt : Enable correct rounded sqrt.
-l           : Extra bitcode libraries paths to link to.
```

-spirv-attach-target ¶

Attaches an SPIR-V target attribute to a GPU Module.

This pass searches for all GPU Modules in the immediate regions and attaches an SPIR-V target if the module matches the name specified by the `module` argument.

Example:

```
// Given the following file: in1.mlir:
gpu.module @nvvm_module_1 {...}
gpu.module @spirv_module_1 {...}
// With
// mlir-opt --spirv-attach-target="module=spirv.* ver=v1.0 caps=Kernel" in1.mlir
// it will generate,
gpu.module @nvvm_module_1 {...}
```

```
gpu.module @spirv_module_1 [#spirv.target<#spirv.vce<v1.0, [Kernel], []>,
#spirv.resource_limits<>>] {...}
```

Options ¶

```
-module      : Regex used to identify the modules to attach the target to.
-ver         : SPIR-V Version.
-caps        : List of supported SPIR-V Capabilities
-exts        : List of supported SPIR-V Extensions
-client_api  : Client API
-vendor      : Device Vendor
-device_type : Device Type
-device_id   : Device ID
```

-xevm-attach-target ¶

Attaches a XeVM target attribute to a GPU Module.

This pass searches for all GPU Modules in the immediate regions and attaches a XeVM target if the module matches the name specified by the `module` argument.

Example:

```
// File: in.mlir:
gpu.module @nvvm_module_1 {...}
gpu.module @rocdl_module_2 {...}
gpu.module @xevm_module_3 {...}
// mlir-opt --xevm-attach-target="module=xevm.* chip=pvc" in.mlir
gpu.module @nvvm_module_1 {...}
gpu.module @rocdl_module_2 {...}
gpu.module @xevm_module_3 [#xevm.target<chip = "pvc">] {...}
```

Options ¶



```

-module      : Regex used to identify the modules to attach the target to.
-triple      : Target triple.
-chip        : Target chip.
-O           : Optimization level.
-l           : Extra bitcode libraries paths to link to.
-cmd-options : Command line options passed to downstream compiler

```

'linalg' Dialect Passes ¶

-convert-elementwise-to-linalg ¶

Convert ElementwiseMappable ops to linalg

Convert ops with the `ElementwiseMappable` trait to linalg parallel loops.

This pass only converts ops that operate on ranked tensors. It can be run on op which contains linalg ops (most commonly a `FunctionOpInterface` op).

-convert-linalg-to-affine-loops ¶

Lower the operations from the linalg dialect into affine loops

-convert-linalg-to-loops ¶

Lower the operations from the linalg dialect into loops

Lowers the `linalg` ops to loop nests using `scf.for`.

Pre-condition: the operands used by the `linalg` ops have buffer semantics, i.e., tensor operands and results must be converted to memrefs via bufferization.

-convert-linalg-to-parallel-loops ¶

Lower the operations from the linalg dialect into parallel loops

-linalg-block-pack-matmul ¶

Convert linalg matmul ops to block layout and back



Pack a matmul operation into blocked layout with two levels of subdivision:

- major 2D blocks - outer dimensions, consist of minor blocks
- minor 2D blocks - inner dimensions, consist of scalar elements

A 2D matmul $M \times N \times K$ gets reshaped into blocked 4D representation as: $[MB][NB][mb][nb] += [MB][KB][mb][kb] * [NB][KB][nb][kb]$ where the (MB, NB, KB) dimensions represent the major blocks, and the (mb, nb, kb) are the minor blocks of their respective original 2D dimensions (M, N, K).

Depending on the initial operands' data layout and the specified packing options, the major blocks dimensions might get transposed e.g., $[MB][KB] \rightarrow [KB][MB]$. The minor blocks can also be transposed e.g., $[mb][kb] \rightarrow [kb][mb]$. Any present batch dimensions remain unchanged. The final result is unpacked back to the original shape.

For example, given a matmul operation:

```
%res = linalg.matmul ins(%A, %B) outs(%C)
```

the default transformation result can be represented as:

```
%A_packed = pack %A : 2D <MxK> -> 4D <MBxKBxmbxkb>
%B_packed = pack %B : 2D <KxN> -> 4D <NBxKBxnbxkb>
%C_packed = pack %C : 2D <MxN> -> 4D <MBxNBxmbxnb>
%res_packed = linalg.mmt4d ins(%A_packed, %B_packed) outs(%C_packed)
%res = unpack %res_packed : 4D <MBxNBxmbxnb> -> 2D <MxN>
```

Options ¶

```
-block-factors          : Block factors (mb, nb, kb) for relayout; use [N] for scalable
-allow-padding          : Allow packing padding
-mnk-padded-multiples   : Next multiples of the packing sizes
-mnk-order              : Permutation of matmul (M, N, K) dimensions order
-lhs-transpose-outer-blocks : Transpose LHS outer block layout [MB][KB] -> [KB][MB]
-lhs-transpose-inner-blocks : Transpose LHS inner block layout [mb][kb] -> [kb][mb]
-rhs-transpose-outer-blocks : Transpose RHS outer block layout [KB][NB] -> [NB][KB]
-rhs-transpose-inner-blocks : Transpose RHS inner block layout [kb][nb] -> [nb][kb]
```



-linalg-fold-into-elementwise ¶*Fold transpose and broadcast ops into elementwise consumers*

Fold a transpose or broadcast that feeds a `linalg.elementwise` or an elementwise-like `linalg.generic` into its consumer. `linalg.transpose` and `linalg.broadcast` producers whose consumer indexing map is a projected permutation can be absorbed into the consumer's indexing map by composing the producer's map into it. Other operands remain untouched.

-linalg-fold-unit-extent-dims ¶*Remove unit-extent dimension in Linalg ops on tensors***Options** ¶

```
-use-rank-reducing-slices : Generate rank-reducing slices instead of reassociative reshapes
```

-linalg-fuse-elementwise-ops ¶*Fuse elementwise operations on tensors***-linalg-generalize-named-ops** ¶*Convert named ops into generic ops***-linalg-inline-scalar-operands** ¶*Inline scalar operands into linalg generic ops***-linalg-morph-ops** ¶*Convert linalg ops between forms*

Convert a linalg op from one representation to another equivalent. For example, a linalg named op `linalg.add` can also be written as an category op `linalg.elementwise`, and can also be re-written as a `linalg.generic`, giving the morphism:

```
named-op <--> category_op (elementwise, contraction, ..) <--> generic
```



Note that the set of `linalg.generic` subsumes named and category ops and therefore not all `linalg.genric` can be converted to named or category op. Similarly, catgory ops subsume named ops.

Note: Legacy converters: `--linalg-generalize-named-ops` is the path `named-op --> generic-op` `--linalg-specialize-generic-ops` is the path `named-op <-- generic-op`

Options ¶

```
-named-to-category    : convert named ops to category op e.g. `linalg.elementwise`
-category-to-generic  : convert category ops e.g. `linalg.elementwise` to `linalg.generic`
-named-to-generic     : convert named ops e.g. `linalg.add` to `linalg.generic`
-generic-to-named     : convert linalg.generic to equivalent named ops
-generic-to-category  : convert linalg.generic to equivalent category ops
-category-to-named    : convert category ops e.g. `linalg.elementwise` to equivalent named
ops
```

`-linalg-specialize-generic-ops` ¶

Convert generic ops back to named ops

`-simplify-depthwise-conv` ¶

Simplify depthwise convolution.

'llvm' Dialect Passes ¶

`-ensure-debug-info-scope-on-llvm-func` ¶

Materialize LLVM debug info subprogram attribute on every LLVMFuncOp

Having a debug info subprogram attribute on a function is required for emitting line tables from MLIR FileLocCol locations.

This is not intended to be a proper replacement for frontends to emit complete debug information, however it is a convenient way to get line tables for debugging purposes. This allow to step trough in a debugger line-by-line or get a backtrace with line numbers.

Options ¶



```
-emission-kind : Emission kind to generate debug info.
```

-llvm-add-comdats ¶

Add comdats to linkonce and linkonce_odr functions

Add an any COMDAT to every linkonce and linkonce_odr function. This is necessary on Windows to link these functions as the system linker won't link weak symbols without a COMDAT. It also provides better behavior than standard weak symbols on ELF-based platforms. This pass will still add COMDATs on platforms that do not support them, for example macOS, so should only be run when the target platform supports COMDATs.

-llvm-legalize-for-export ¶

Legalize LLVM dialect to be convertible to LLVM IR

Creates a pass that legalizes the LLVM dialect operations so that they can be translated to LLVM IR.

-llvm-request-c-wrappers ¶

Request C wrapper emission for all functions

Annotate every builtin function in the module with the LLVM dialect attribute that instructs the conversion to LLVM to emit the C wrapper for the function. This pass is expected to be applied immediately before the conversion of builtin functions to LLVM to avoid the attribute being dropped by other passes.

-llvm-use-default-visibility ¶

Update default visibility of all global values and function definitions

Update the default visibility of all global values and function definitions to `visibility`, as with `-fvisibility=` (`hidden|protected`).

Options ¶

```
-visibility : Visibility to use in place of default.
```

'llvm' Target Passes ¶



`-llvm-target-to-data-layout` ¶

Derive data layout attributes from LLVM target attributes

Derive a `DataLayoutSpecInterface`-implementing data layout attribute from the LLVM-backend target specified by the `TargetAttrInterface`-implementing attribute attached to the target op at the name `llvm.target`.

Options ¶

`-initialize-llvm-targets` : Whether to pre-load all available target machines, that LLVM is configured to support, into the `TargetRegistry`.

`-llvm-target-to-target-features` ¶

Update attached #llvm.target's features per the described target

Obtain the `TargetMachine` specified by the attached `#llvm.target`'s attributes and obtain from it the full list of features of the selected target. Updates the attached `#llvm.target` so that its features reflect the full list of features.

Options ¶

`-initialize-llvm-targets` : Whether to pre-load all available target machines, that LLVM is configured to support, into the `TargetRegistry`.

'math' Dialect Passes ¶

`-math-expand-ops` ¶

Expand math operations.

Expands some math operations into more fundamental operations, allowing them to be subsequently lowered through these. For example, hyperbolic functions are transformed into their expanded form containing only `exp` functions.

The `ops` parameter can be used to apply only a subset of all the available expansions, these must correspond to the operation mnemonic. For example, `ops=sinh,acosh` will expand only `math.sinh` and `math.acosh`



operations. If the list is empty, then all expansions are applied.

Options ¶

```
-ops : Operations to expand.
```

`-math-extend-to-supported-types` ¶

Legalize floating-point math ops on low-precision floats

On many targets, the math functions are not implemented for floating-point types less precise than IEEE single-precision (aka f32), such as half-floats, bfloat16, or 8-bit floats.

This pass explicitly legalizes these math functions by inserting `arith.extf` and `arith.truncf` pairs around said op, which preserves the original semantics while enabling lowering. The extra supported floating-point types for the target are passed as arguments. Types f64 and f32 are implicitly supported.

As an exception, this pass does not legalize `math.fma`, because that is an operation frequently implemented at low precisions.

Options ¶

```
-extra-types : MLIR types with arithmetic support on a given target (f64 and f32 are  
implicitly supported)  
-target-type : MLIR type to convert the unsupported source types to
```

`-math-sincos-fusion` ¶

Fuse sin and cos operations.

Fuse sin and cos operations into a sincos operation.

`-math-uplift-to-fma` ¶

Uplift arith ops to math.fma.

Uplift sequence of addf and mulf ops to `math.fma` if fastmath flags allows it.



‘memref’ Dialect Passes ¶

-expand-realloc ¶

Expand memref.realloc operations into its components

The `memref.realloc` operation performs a conditional allocation and copy to increase the size of a buffer if necessary. This pass converts a `realloc` operation into this sequence of simpler operations such that other passes at a later stage in the compilation pipeline do not have to consider the `realloc` operation anymore (e.g., the buffer deallocation pass and the conversion pass to LLVM).

Example of an expansion:

```
%realloc = memref.realloc %alloc (%size) : memref<?xf32> to memref<?xf32>
```

is expanded to

```
%c0 = arith.constant 0 : index
%dim = memref.dim %alloc, %c0 : memref<?xf32>
%is_old_smaller = arith.cmpi ult, %dim, %arg1
%realloc = scf.if %is_old_smaller -> (memref<?xf32>) {
  %new_alloc = memref.alloc(%size) : memref<?xf32>
  %subview = memref.subview %new_alloc[0] [%dim] [1]
  memref.copy %alloc, %subview
  memref.dealloc %alloc
  scf.yield %alloc_0 : memref<?xf32>
} else {
  %reinterpret_cast = memref.reinterpret_cast %alloc to
    offset: [0], sizes: [%size], strides: [1]
  scf.yield %reinterpret_cast : memref<?xf32>
}
```

Options ¶

```
-emit-deallocs : Emit deallocation operations for the original MemRef
```



`-expand-strided-metadata` ¶

Expand memref operations into easier to analyze constructs

The pass expands memref operations that modify the metadata of a memref (sizes, offset, strides) into a sequence of easier to analyze constructs. In particular, this pass transforms operations into explicit sequence of operations that model the effect of this operation on the different metadata. This pass uses affine constructs to materialize these effects.

Supported ops include:

- `memref.collapse_shape`
- `memref.expand_shape`
- `memref.extract_aligned_pointer_as_index`
- `memref.extract_strided_metadata`
- `memref.subview`

`-flatten-memref` ¶

Flatten a multiple dimensional memref to 1-dimensional

`-fold-memref-alias-ops` ¶

Fold memref alias ops into consumer load/store ops

The pass folds loading/storing from/to memref aliasing ops to loading/storing from/to the original memref.

`-memref-elide-reinterpret-cast` ¶

Replace ops depending on redundant `reinterpret_cast`(s) to be convertible to `EmitC`.

Replace data-movement ops that depend on redundant `memref.reinterpret_cast` operations to obtain compatible shapes with equivalent ops that operate on compatible shapes directly. This simplifies conversion to `EmitC`.

`-memref-emulate-wide-int` ¶

*Emulate $2*N$ -bit integer operations using N -bit operations*

Emulate memref integer operations that use too wide integer types with equivalent operations on supported narrow integer types. This is done by splitting original integer values into two halves.



Currently, only power-of-two integer bitwidths are supported.

Options ¶

`-widest-int-supported` : Widest integer type supported by the target

`-memref-expand` ¶

Legalize memref operations to be convertible to LLVM.

`-normalize-memrefs` ¶

Normalize memrefs

This pass transforms memref types with a non-trivial [layout map](#) into memref types with an identity layout map, e.g. $(i, j) \rightarrow (i, j)$. This pass is inter-procedural, in the sense that it can modify function interfaces and call sites that pass memref types. In order to modify memref types while preserving the original behavior, users of those memref types are also modified to incorporate the resulting layout map. For instance, an [AffineLoadOp](#) will be updated to compose the layout map with the affine expression contained in the op. Operations marked with the [MemRefsNormalizable](#) trait are expected to be normalizable. Supported operations include affine operations, `memref.alloc`, `memref.dealloc`, and `func.return`.

Given an appropriate layout map specified in the code, this transformation can express tiled or linearized access to multi-dimensional data structures, but will not modify memref types without an explicit layout map.

Currently this pass is limited to only modify functions where all memref types can be normalized. If a function contains any operations that are not `MemRefNormalizable`, then the function and any functions that call or call it will not be modified.

Input

```
#tile = affine_map<(i) -> (i floordiv 4, i mod 4)>
func.func @matmul(%A: memref<16xf64, #tile>,
                  %B: index, %C: memref<16xf64>) -> (memref<16xf64, #tile>) {
  affine.for %arg3 = 0 to 16 {
    %a = affine.load %A[%arg3] : memref<16xf64, #tile>
    %p = arith.mulf %a, %a : f64
    affine.store %p, %A[%arg3] : memref<16xf64, #tile>
```

```

}
%c = memref.alloc() : memref<16xf64, #tile>
%d = affine.load %c[0] : memref<16xf64, #tile>
return %A: memref<16xf64, #tile>
}

```

Output

```

func.func @matmul(%arg0: memref<4x4xf64>, %arg1: index, %arg2: memref<16xf64>)
-> memref<4x4xf64> {
  affine.for %arg3 = 0 to 16 {
    %3 = affine.load %arg0[%arg3 floordiv 4, %arg3 mod 4]: memref<4x4xf64>
    %4 = arith.mulf %3, %3 : f64
    affine.store %4, %arg0[%arg3 floordiv 4, %arg3 mod 4]: memref<4x4xf64>
  }
  %0 = memref.alloc() : memref<4x4xf64>
  %1 = affine.apply #map1()
  %2 = affine.load %0[0, 0] : memref<4x4xf64>
  return %arg0 : memref<4x4xf64>
}

```

Input

```

#linear8 = affine_map<(i, j) -> (i * 8 + j)>
func.func @linearize(%arg0: memref<8x8xi32, #linear8>,
                    %arg1: memref<8x8xi32, #linear8>,
                    %arg2: memref<8x8xi32, #linear8>) {
  %c8 = arith.constant 8 : index
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  affine.for %arg3 = %c0 to %c8 {
    affine.for %arg4 = %c0 to %c8 {
      affine.for %arg5 = %c0 to %c8 {
        %0 = affine.load %arg0[%arg3, %arg5] : memref<8x8xi32, #linear8>
        %1 = affine.load %arg1[%arg5, %arg4] : memref<8x8xi32, #linear8>

```



```

    %2 = affine.load %arg2[%arg3, %arg4] : memref<8x8xi32, #linear8>
    %3 = arith.muli %0, %1 : i32
    %4 = arith.addi %2, %3 : i32
    affine.store %4, %arg2[%arg3, %arg4] : memref<8x8xi32, #linear8>
  }
}
}
return
}

```

Output

```

func.func @linearize(%arg0: memref<64xi32>,
    %arg1: memref<64xi32>,
    %arg2: memref<64xi32>) {
  %c8 = arith.constant 8 : index
  %c0 = arith.constant 0 : index
  affine.for %arg3 = %c0 to %c8 {
    affine.for %arg4 = %c0 to %c8 {
      affine.for %arg5 = %c0 to %c8 {
        %0 = affine.load %arg0[%arg3 * 8 + %arg5] : memref<64xi32>
        %1 = affine.load %arg1[%arg5 * 8 + %arg4] : memref<64xi32>
        %2 = affine.load %arg2[%arg3 * 8 + %arg4] : memref<64xi32>
        %3 = arith.muli %0, %1 : i32
        %4 = arith.addi %2, %3 : i32
        affine.store %4, %arg2[%arg3 * 8 + %arg4] : memref<64xi32>
      }
    }
  }
  return
}

```

-reify-result-shapes ¶*Reifies the results of `tensor::PadOp` and `tensor::ConcatOp`.*

This pass reifies the shapes of a subset of `ReifyRankedShapedTypeOpInterface` ops with tensor results.

The pass currently only supports result shape type reification for:

- `tensor::PadOp`
- `tensor::ConcatOp` It addresses a representation gap where implicit op semantics are needed to infer static result types from dynamic operands. But it does so by using `ReifyRankedShapedTypeOpInterface` as the source of truth rather than the op itself. As a consequence, this cannot generalize today.

TODO: in the future, we should consider coupling this information with op “transfer functions” (e.g. `IndexingMapOpInterface`) to provide a source of truth that can work across result shape inference, canonicalization and op verifiers.

The pass replaces the operations with their reified versions, when more static information can be derived, and inserts casts when results shapes are updated.

Example:

```
#map = affine_map<(d0) -> (-d0 + 256)>
func.func @func(%arg0: f32, %arg1: index, %arg2: tensor<64x?x64xf32>)
  -> tensor<1x?x64xf32>
{
  %0 = affine.apply #map(%arg1)
  %extracted_slice = tensor.extract_slice %arg2[0, 0, 0] [1, %arg1, 64] [1, 1, 1]
    : tensor<64x?x64xf32> to tensor<1x?x64xf32>
  %padded = tensor.pad %extracted_slice low[0, 0, 0] high[0, %0, 0] {
    ^bb0(%arg3: index, %arg4: index, %arg5: index):
      tensor.yield %arg0 : f32
  } : tensor<1x?x64xf32> to tensor<1x?x64xf32>
  return %padded : tensor<1x?x64xf32>
}

// mlir-opt --reify-result-shapes
#map = affine_map<()[s0] -> (-s0 + 256)>
func.func @func(%arg0: f32, %arg1: index, %arg2: tensor<64x?x64xf32>)
  -> tensor<1x?x64xf32>
{
  %0 = affine.apply #map()[%arg1]
```



```

%extracted_slice = tensor.extract_slice %arg2[0, 0, 0] [1, %arg1, 64] [1, 1, 1]
    : tensor<64x?x64xf32> to tensor<1x?x64xf32>
%padded = tensor.pad %extracted_slice low[0, 0, 0] high[0, %0, 0] {
  ^bb0(%arg3: index, %arg4: index, %arg5: index):
    tensor.yield %arg0 : f32
} : tensor<1x?x64xf32> to tensor<1x256x64xf32>
%cast = tensor.cast %padded : tensor<1x256x64xf32> to tensor<1x?x64xf32>
return %cast : tensor<1x?x64xf32>
}

```

-resolve-ranked-shaped-type-result-dims ¶

Resolve memref.dim of result values of ranked shape type

The pass resolves memref.dim of result of operations that implement the `ReifyRankedShapedTypeOpInterface` in terms of shapes of its operands.

Options ¶

```

-error-on-pattern-iteration-limit : Throw an error when pattern rewriter hits iteration
limit

```

-resolve-shaped-type-result-dims ¶

Resolve memref.dim of result values

The pass resolves memref.dim of result of operations that implement the `InferShapedTypeOpInterface` or `ReifyRankedShapedTypeOpInterface` in terms of shapes of its operands.

Options ¶

```

-error-on-pattern-iteration-limit : Throw an error when pattern rewriter hits iteration
limit

```

‘omp’ Dialect Passes ¶



-omp-host-op-filter ¶

Removes all non target-related ops from host functions.

When compiling for an OpenMP offloading target, host functions holding target regions must be maintained in order for the `omp.target` ops to be translated to LLVM IR as functions.

However, these host functions might contain more host code than just what's necessary for generating code for their contained target regions on the target device, potentially causing problematic artifacts to be generated into the target device's module.

This pass strips out all non-essential operations from host functions when compiling for an OpenMP target device, leaving just `omp.target` operations, related `omp.map.info` and placeholders in the form of function arguments to keep the MLIR representation legal.

It must run on a module at a low level of abstraction (i.e. directly translatable to LLVM IR), after any passes that might produce new OpenMP operations have already completed.

-omp-mark-declare-target ¶

Marks all functions called by an OpenMP declare target function as declare target

Marks functions contained within the module as declare target if they are called from within an explicitly marked declare target function or a target region (`omp.target`).

-omp-offload-privatization-prepare ¶

Prepare OpenMP maps for privatization for deferred target tasks

When generating LLVMIR for privatized variables in an OpenMP offloading directive (eg. `omp::TargetOp`) that creates a deferred target task (when the `nowait` clause is used), we need to copy the privatized variable out of the stack of the generating task and into the heap so that the deferred target task can still access it. However, if such a privatized variable is also mapped, typically the case for allocatables, then the corresponding `omp::MapInfoOp` needs to be fixed up to map the new heap-allocated variable and not the original variable.

-omp-stack-to-shared ¶

Replaces stack allocations target devices with shared memory.



This pass replaces `llvm.alloca` operations located in a non-SPMD target region and then potentially used inside of an `omp.parallel` region with `omp.alloc_shared_mem` and `omp.free_shared_mem`. This is also done for top-level function `llvm.alloca`s used in the same way when the parent function is a target device function.

This ensures that explicit private allocations, intended to be shared across threads, use the proper memory space on a target device while supporting the case of parallel regions indirectly reached from within a target region via function calls.

‘shard’ Dialect Passes ¶

-shard-partition ¶

Partition a function into SPMD form.

This pass fits in right after a pass that annotates the function with shardings like the `ShardingPropagation` pass. It operates on a fully annotated IR.

A fully annotated IR required that all ranked tensor operands, results and block arguments are annotated with the `shard.shard` operation.

All direct descendant operations in the function must implement the `ShardingInterface` interface or all their ranked tensor operands and results must have full replication sharding.

The input IR must have sharding annotations such that each operation that implements `ShardingInterface` can handle during partition with its `partition` method. This can be achieved with the `ShardingPropagation` pass.

If the function has multiple terminating blocks, it is the responsibility of the the one who annotates the function with shardings to make sure that all returns would be consisted that is, have the same sharding.

Example:

```
shard.grid @grid_1d(shape = 2)

func.func @f(
  %arg0: tensor<2xi8>
) -> tensor<2xi8> {
  %0 = shard.shard %arg0 to <@grid_1d, [[0]]> : tensor<2xi8>
  %1 = shard.shard %0 to <@grid_1d, [[0]]> annotate_for_users: tensor<2xi8>
  %2 = tosa.abs %1 : (tensor<2xi8>) -> tensor<2xi8>
```

```

%3 = shard.shard %2 to <@grid_1d, [[0]]> : tensor<2xi8>
%4 = shard.shard %3 to <@grid_1d, [[]]> annotate_for_users: tensor<2xi8>
return %4 : tensor<2xi8>
}

```

Partitioning the above would result in

- Performing the element-wise `abs` operation on each device.
- Resharding to full replication with an all-gather.

```

shard.grid @grid_1d(shape = 2)

func.func @f(%arg0: tensor<1xi8>) -> tensor<2xi8> {
  %0 = tosa.abs %arg0 : (tensor<1xi8>) -> tensor<1xi8>
  %1 = shard.all_gather %0 on @grid_1d grid_axes = [0] gather_axis = 0 : tensor<1xi8> ->
  tensor<2xi8>
  return %1 : tensor<2xi8>
}

```

`-shard-simplify` ¶

Shard simplify patterns.

Applies simplification patterns on the Shard dialect operations. This includes:

- All-reduce endomorphism simplification, e.g. transforming `all_reduce_sum(x) + all_reduce_sum(y)` into `all_reduce_sum(x + y)`.
- Folding `AllSliceOp(AllReduceOp)` into `ReduceScatterOp` when both ops share the same grid and `grid_axes`.
- Folding static grid shapes into constants.

`-sharding-propagation` ¶

Sharding propagation

Propagates sharding information throughout the graph. After this pass, each of the operations' operands and results is annotated with a `shard.shard` operation, and the operations themselves are added with sharding option attributes.

Options ¶

`-traversal` : Traversal order to use for sharding propagation:

‘ml_program’ Dialect Passes ¶

`-mlprogram-pipeline-globals` ¶

Optimize `ml_program` global operations for read and store

`ml_program`’s load and store operations can be optimized for write-write or write-read sets of operations. This allows known tensors to not be re-read when the value is already known in IR.

The pass is designed to handle both nested regions and function calls safely.

‘nvgpu’ Dialect Passes ¶

`-nvgpu-optimize-shared-memory` ¶

Optimizes accesses to shard memory memrefs in order to reduce bank conflicts.

‘nvvm’ Dialect Passes ¶

`-llvm-optimize-for-nvvm-target` ¶

Optimize NVVM IR

‘quant’ Dialect Passes ¶

`-lower-quant-ops` ¶

Lower `quant.dcast` and `quant.qcast` ops

Lower quantization (`quant.qcast`) and dequantization (`quant.dcast`) ops into other core dialects.

The lowering process generates storage type casts in the form of `quant.scast` ops to act as an interface between the original quantized types of operands and results and their corresponding storage types used in the generated arithmetic computations.



-normalize-quant-types ¶

Normalize generic quantized types to specific quantized types

This pass converts generic quantized types in the `quant` dialect to more specific types when possible.

The following conversions are performed:

1. Sub-channel to per-axis: If the shape of the scales tensor of sub-channel quantized type has all but one non-one value, it is converted to a per-axis quantized type.

For example:

```

◦ !quant.uniform<i8:f32:{0:1}, {{2.0}, {3.0}}> -> !quant.uniform<i8:f32:0, {2.0, 3.0}>
◦ tensor<?x?x!quant.uniform<i8:f32:{0:1,1:4}, {{2.0}, {3.0}}>> -> tensor<?x?
  x!quant.uniform<i8:f32:0, {2.0, 3.0}>>

```

2. Sub-channel to per-tensor: If a sub-channel quantized type has only one scale or zero-point, it is converted to a per-tensor quantized type.

For example:

```

◦ !quant.uniform<i8:f32:{}, {{2.0}}> -> !quant.uniform<i8:f32, 2.0>
◦ tensor<?x?x!quant.uniform<i8:f32:{0:1, 0:4}, {{2.0}}>> -> tensor<?x?
  x!quant.uniform<i8:f32, 2.0>>

```

The rationale for these conversions is that the decompositions / handling of more precise quantized types tends to be more efficient than treating everything as subchannel type.

-strip-func-quant-types ¶

Strip quantized types from function headers

Identify occurrences of function arguments using a quantized type and replace them with a new value of the corresponding storage (signless integer) type. For each converted argument, a `quant.scast` op is introduced at the head of the function's entry block converting the new integer argument into the original quantized value.

Reducer Passes ¶

-opt-reduction-pass ¶

A wrapper pass that reduces the file with optimization passes



Options ¶

```
-opt-pass      : The optimization passes used for reduction, e.g., symbol-dce
-opt-pass-file : The file path containing the optimization pipeline definition
-test         : The location of the tester which tests the file interestingness
-test-arg     : arguments of the tester
```

-reduction-tree ¶

Reduce the input with reduction-tree algorithm

Options ¶

```
-traversal-mode : The graph traversal mode, the default is single-path mode
-test          : The location of the tester which tests the file interestingness
-test-arg      : arguments of the tester
```

‘scf’ Dialect Passes ¶

-scf-for-loop-canonicalization ¶

Canonicalize operations within scf.for loop bodies

-scf-for-loop-peeling ¶

Peel for loops at their upper bounds.

Options ¶

```
-peel-front : Peel the first iteration out of the loop.
-skip-partial : Do not peel loops inside of the last, partial iteration of another already peeled loop.
```

-scf-for-loop-range-folding ¶



Fold add/mul ops into loop range

-scf-for-loop-specialization ¶

Specialize for loops for vectorization

-scf-for-to-while ¶

Convert SCF for loops to SCF while loops

This pass transforms SCF.ForOp operations to SCF.WhileOp. The For loop condition is placed in the 'before' region of the while operation, and the induction variable incrementation and loop body in the 'after' region. The loop carried values of the while op are the induction variable (IV) of the for-loop + any iter_args specified for the for-loop. Any 'yield' ops in the for-loop are rewritten to additionally yield the (incremented) induction variable.

```
scf.for %i = %c0 to %arg1 step %c1 {
  %0 = arith.addi %arg2, %arg2 : i32
  memref.store %0, %arg0[%i] : memref<?xi32>
}
```

After:

```
%0 = scf.while (%i = %c0) : (index) -> index {
  %1 = arith.cmpi slt, %i, %arg1 : index
  scf.condition(%1) %i : index
} do {
  ^bb0(%i: index):
  %1 = arith.addi %i, %c1 : index
  %2 = arith.addi %arg2, %arg2 : i32
  memref.store %2, %arg0[%i] : memref<?xi32>
  scf.yield %1 : index
}
```

-scf-forall-to-for ¶

Convert SCF forall loops to SCF for loops

-scf-forall-to-parallel ¶



Convert SCF forall loops to SCF parallel loops

-scf-parallel-for-to-nested-fors ¶

Convert SCF parallel for loops to nested SCF for loops

This pass transforms SCF::ParallelOp operations into a nest of SCF::ForOp operations. The transformation is useful for cases where the parallel loop can be expressed as a series of sequential iterations, allowing for more fine-grained control over the loop execution.

-scf-parallel-loop-fusion ¶

Fuse adjacent parallel loops

-scf-parallel-loop-specialization ¶

Specialize parallel loops for vectorization

-scf-parallel-loop-tiling ¶

Tile parallel loops

Options ¶

```
-parallel-loop-tiling-sizes : Factors to tile parallel loops by
-no-min-max-bounds        : Perform tiling with fixed upper bound with inbound check inside
                           the internal loops
```

-test-scf-parallel-loop-collapsing ¶

Test parallel loops collapsing transformation

This pass is purely for testing the scf::collapseParallelLoops transformation. The transformation does not have opinions on how a parallel loop should be collapsed, so this pass is structured for the common case on GPUs of collapsing to a 3d parallel loop. 3 lists can be provided to collapsed-indices-{0,1,2} to represent how the loop should be collapsed and must reference every iterator in the original parallel loop.

```
scf.parallel (%arg0, %arg1)
```

```

        = (%c0, %c0) to (%c2, %c2) step (%c1, %c1) {
    "test.sink"(%5, %3) : (index, index) -> ()
    scf.yield
}

# After:
scf.parallel (%arg0) = (%c0) to (%c4) step (%c1) {
    %0 = arith.remsi %arg0, %c2 : index
    %1 = arith.divsi %arg0, %c2 : index
    %2 = arith.muli %0, %c7 : index
    %3 = arith.addi %2, %c3 : index
    %4 = arith.muli %1, %c7 : index
    %5 = arith.addi %4, %c3 : index
    "test.sink"(%5, %3) : (index, index) -> ()
}

```

Options ¶

```

-collapsed-indices-0 : Which loop indices to combine 0th loop index
-collapsed-indices-1 : Which loop indices to combine into the position 1 loop index
-collapsed-indices-2 : Which loop indices to combine into the position 2 loop index

```

‘shape’ Dialect Passes ¶

-outline-shape-computation ¶

Using shape.func to preserve shape computation

This pass outlines the shape computation part in high level IR by adding shape.func and populate corresponding mapping information into ShapeMappingAnalysis. The shape computation part is usually introduced by shape reification, and each single dynamic shape is denoted by shape.with_shape.

There're two main reasons this shape-outline pass is needed:

1. Many passes don't take shape reification part into consideration. Therefore we need to “remove” the shape reification part temporarily for these passes.



2. Sometimes we cannot redo shape reification after converting from dialect A to dialect B. Because op-level shape reification is only implemented on A.

Input:

```
func.func @main(%arg0: tensor<?x4x?xf32>, %arg1: tensor<2x4x?xf32>) ->
  tensor<?x4x?xf32> {
    %c2 = arith.constant 2 : index
    %c0 = arith.constant 0 : index
    %c4 = arith.constant 4 : index
    %0 = shape.shape_of %arg0 : tensor<?x4x?xf32> -> tensor<3xindex>
    %1 = shape.get_extent %0, %c2 : tensor<3xindex>, index -> index
    %2 = "test.abs"(%arg0) : (tensor<?x4x?xf32>) -> tensor<?x4x?xf32>
    %3 = shape.with_shape %2, %0 : tensor<?x4x?xf32>, tensor<3xindex>
    %4 = shape.value_of %3 : tensor<?x4x?xf32>
    %5 = "test.concat"(%4, %arg1) {axis = 0 : i64} : (tensor<?x4x?xf32>,
      tensor<2x4x?xf32>) -> tensor<?x4x?xf32>
    %6 = shape.get_extent %0, %c0 : tensor<3xindex>, index -> index
    %7 = arith.addi %6, %c2 : index
    %8 = shape.from_extents %7, %c4, %1 : index, index, index
    %9 = shape.with_shape %5, %8 : tensor<?x4x?xf32>, !shape.shape
    %10 = shape.value_of %9 : tensor<?x4x?xf32>
    return %10 : tensor<?x4x?xf32>
  }
```

Output

```
func.func @main(%arg0: tensor<?x4x?xf32>, %arg1: tensor<2x4x?xf32>) ->
  tensor<?x4x?xf32> {
    %0 = "test.abs"(%arg0) : (tensor<?x4x?xf32>) -> tensor<?x4x?xf32>
    %1 = "test.concat"(%0, %arg1) {axis = 0 : i64} : (tensor<?x4x?xf32>,
      tensor<2x4x?xf32>) -> tensor<?x4x?xf32>
    return %1 : tensor<?x4x?xf32>
  }
shape.func private @shape_cal_1(%arg0: tensor<?x4x?xf32>) -> !shape.shape {
  %c2 = arith.constant 2 : index
  %c0 = arith.constant 0 : index
```

```

%c4 = arith.constant 4 : index
%0 = shape_of %arg0 : tensor<?x4x?xf32> -> tensor<3xindex>
%1 = get_extent %0, %c2 : tensor<3xindex>, index -> index
%2 = get_extent %0, %c0 : tensor<3xindex>, index -> index
%3 = arith.addi %2, %c2 : index
%4 = from_extents %3, %c4, %1 : index, index, index
return %4 : !shape.shape
}
shape.func private @shape_cal_0(%arg0: tensor<?x4x?xf32>) -> tensor<3xindex> {
  %0 = shape_of %arg0 : tensor<?x4x?xf32> -> tensor<3xindex>
  return %0 : tensor<3xindex>
}

```

For the above example, the shape computation is inlined in the input IR, which is used for two values' (test.abs and test.concat) shape. And the shape computation part is outlined in the output IR.

And the shape mapping information will be:

```

// ---- Shape Mapping Information ----
// - Shape for: %0 = "test.abs"(%arg0) : (tensor<?x4x?xf32>) -> tensor<?x4x?xf32> ::
@shape_cal_0(<block argument> of type 'tensor<?x4x?xf32>' at index: 0)
// - Shape for: %1 = "test.concat"(%0, %arg1) {axis = 0 : i64} : (tensor<?x4x?xf32>,
tensor<2x4x?xf32>) -> tensor<?x4x?xf32> :: @shape_cal_1(<block argument> of type 'tensor<?
x4x?xf32>' at index: 0)

```

-remove-shape-constraints ¶

Replace all cstr ops with a true witness_

-shape-to-shape-lowering ¶

Legalize Shape dialect to be convertible to Arith

‘sparse_tensor’ Dialect Passes ¶

-lower-sparse-foreach-to-scf ¶



Decompose a complex sparse operation into multiple stages

A pass that lowers `sparse_tensor.foreach` operation to `scf` dialect.

-lower-sparse-iteration-to-scf ¶

Lower `sparse_tensor.iterate/coiterate` into `scf` loops

This pass lowers `sparse_tensor.iterate` operations into `scf.for/while` operations. The pass is not yet stabilized.

-lower-sparse-ops-to-foreach ¶

Applies sparse tensor rewriting rules after sparsification

A pass that lowers high-level sparse operations to `sparse_tensor.foreach`.

Options ¶

```
-enable-runtime-library : Enable runtime library for manipulating sparse tensors
-enable-convert         : Enable rewriting rules for the convert operator
```

-pre-sparsification-rewrite ¶

Applies sparse tensor rewriting rules prior to sparsification

A pass that applies rewriting rules to sparse tensor operations prior to running the actual sparsification pass.

-sparse-assembler ¶

Add `[dis]assemble` operations on external sparse tensors

Unlike dense tensors, MLIR does **not** provide a direct `_mlir_ciface_` ABI for passing sparse tensors as arguments from and to external methods (within MLIR-generated methods, sparse tensors can be freely passed around, but this eventually uses a bespoke parameter passing format that is subject to change; like opaque pointers when the sparse runtime support library is used or the constituent arrays and structs for direct IR codegen). The sparse assembler pass, however, can be used to obtain a stable `_mlir_ciface_` API for passing sparse tensors from and to an external environment, such as Python, PyTorch, or JAX.



The pass converts public entry methods that use sparse tensors as input parameters and/or output return values into wrapper methods that [dis]assemble the individual tensors that constitute the actual storage used externally into MLIR sparse tensors. This pass can be used to prepare the public entry methods of a program that is compiled by the MLIR sparsifier to interface with an external runtime, e.g., when passing sparse tensors as numpy arrays from and to Python. Note that eventual bufferization decisions (e.g. who [de]allocates the underlying memory) should be resolved in agreement with the external runtime.

By default, the pass uses the [dis]assemble operations to input and output sparse tensors. When the direct-out option is set, however, the output directly returns the MLIR allocated buffers to the external runtime.

The pass should always run before the actual sparsification passes.

Options ¶

```
-direct-out : Directly returns buffers externally
```

-sparse-buffer-rewrite ¶

Rewrite sparse primitives on buffers to actual code

A pass that rewrites sparse primitives on buffers to the MLIR implementation of the primitives. For example, `sparse_tensor.sort` operator is implemented in this pass.

Options ¶

```
-enable-buffer-initialization : Enable zero-initialization of the memory buffers
```

-sparse-gpu-codegen ¶

Generates GPU code during sparsification

Enables the sparsifier to use GPU acceleration. When the number of GPU threads is set to zero, the pass tries to enable GPU acceleration by means of direct library calls (like cuSPARSE).

Options ¶



```
-num-threads          : Sets the number of GPU threads
-enable-runtime-library : Enable runtime library for manipulating sparse tensors
```

-sparse-reinterpret-map ¶

Reinterprets sparse tensor type mappings

A pass that reinterprets the mappings in all sparse tensor types in a way that enables subsequent sparsification. This involves expressing all `linalg.generic` operations in terms of level coordinates (rather than the dimension coordinates of the input tensors) to align the iteration space with the potentially remapped level space as well as resolving cycles in the resulting iteration graphs with explicit sparse tensor conversions where needed.

Options ¶

```
-scope                : Set the reiteration scope
-loop-ordering-strategy : Set the loop ordering strategy for sparse code generation
```

-sparse-space-collapse ¶

Sparse space collapsing pass

This pass collapses consecutive sparse spaces (extracted from the same tensor) into one multi-dimensional space. The pass is not yet stabilized.

-sparse-storage-specifier-to-llvm ¶

Lower sparse storage specifier to llvm structure

This pass rewrites sparse tensor storage specifier-related operations into LLVM Dialect, and converts sparse tensor storage specifier into an `llvm.struct`.

Example of the conversion:

Before:

```
%0 = sparse_tensor.storage_specifier.get %arg0 dim_sz at 0
: !sparse_tensor.storage_specifier<#CSR> to i64
```

After:

```
%0 = llvm.extractvalue %arg0[0, 0] : !llvm.struct<(array<2 x i64>, array<3 x i64>>
```

-sparse-tensor-codegen ¶

Convert sparse tensors and primitives to actual code

A pass that converts sparse tensor types and primitives to actual compiler visible buffers and compiler IR that implements these primitives on the selected sparse tensor storage schemes.

This pass provides an alternative to the SparseTensorConversion pass, eliminating the dependence on a runtime support library, and providing much more opportunities for subsequent compiler optimization of the generated code.

Example of the conversion:

Before:

```
func.func @foo(%arg0: tensor<8x8xf32, #CSR>) -> memref<?xindex> {
  %0 = sparse_tensor.positions %arg0 {level = 1 : index}
    : tensor<8x8xf32, #CSR> to memref<?xindex>
  return %0 : memref<?xindex>
}
```

After:

```
func.func @foo(%arg0: memref<?xindex>,
               %arg1: memref<?xindex>,
               %arg2: memref<?xf32>,
               %arg3: !sparse_tensor.storage_specifier<#sparse>) -> memref<?xindex> {
  %0 = sparse_tensor.storage_specifier.get %arg3 pos_mem_sz at 1
    : !sparse_tensor.storage_specifier<#sparse>
  %subview = memref.subview %arg0[0] [%0] [1]
    : memref<?xindex> to memref<?xindex>
  return %subview : memref<?xindex>
}
```

Options ¶



-enable-buffer-initialization : Enable zero-initialization of the memory buffers

-create-sparse-deallocs : Specify if the temporary buffers created by the sparse compiler should be deallocated. For compatibility with core bufferization passes. This option is only used when enable-runtime-library=false. See also create-deallocs for BufferizationOption.

-sparse-tensor-conversion ¶

Convert sparse tensors and primitives to library calls

A pass that converts sparse tensor primitives into calls into a runtime support library. Sparse tensor types are converted into opaque pointers to the underlying sparse storage schemes.

The use of opaque pointers together with runtime support library keeps the conversion relatively simple, but at the expense of IR opacity, which obscures opportunities for subsequent optimization of the IR. An alternative is provided by the SparseTensorCodegen pass.

Example of the conversion:

Before:

```
func.func @foo(%arg0: tensor<8x8xf32, #CSR>) -> memref<?xindex> {
  %0 = sparse_tensor.positions %arg0 {level = 1 : index}
    : tensor<8x8xf32, #CSR> to memref<?xindex>
  return %0 : memref<?xindex>
}
```

After:

```
func.func @foo(%arg0: !llvm.ptr) -> memref<?xindex> {
  %c1 = arith.constant 1 : index
  %0 = call @sparsePositions0(%arg0, %c1)
    : (!llvm.ptr, index) -> memref<?xindex>
  return %0 : memref<?xindex>
}
```

-sparse-vectorization ¶



Vectorizes loops after sparsification

A pass that converts loops after sparsification into vector loops. The vector dialect is used as target to provide an architectural neutral way of exploiting any platform that supports SIMD instructions.

The vector length (viz. `v1`) describes the number of packed data elements (e.g. both `vector<16xf32>` and `vector<16xf64>` have a vector length of 16 even though the actual bitwidths differ). A small multiple of the actual lengths supported in hardware typically results in efficient SIMD code, since the backend will map longer vectors to multiple vector registers, thereby effectively unrolling an addition level within the generated for-loop.

Example of the conversion:

Before:

```
%3 = memref.load %2[] : memref<f32>
%4 = scf.for %arg3 = %c0 to %c1024 step %c1 iter_args(%arg4 = %3) -> (f32) {
  %6 = memref.load %0[%arg3] : memref<?xf32>
  %7 = memref.load %1[%arg3] : memref<1024xf32>
  %8 = arith.mulf %6, %7 : f32
  %9 = arith.addf %arg4, %8 : f32
  scf.yield %9 : f32
}
memref.store %4, %2[] : memref<f32>
```

After:

```
%3 = memref.load %2[] : memref<f32>
%4 = vector.insert %3, %cst [0] : f32 into vector<32xf32>
%5 = scf.for %arg3 = %c0 to %c1024 step %c32 iter_args(%arg4 = %4) -> (vector<32xf32>)
{
  %8 = vector.load %0[%arg3] : memref<?xf32>, vector<32xf32>
  %9 = vector.load %1[%arg3] : memref<1024xf32>, vector<32xf32>
  %10 = arith.mulf %8, %9 : vector<32xf32>
  %11 = arith.addf %arg4, %10 : vector<32xf32>
  scf.yield %11 : vector<32xf32>
}
%6 = vector.reduction <add>, %5 : vector<32xf32> into f32
memref.store %6, %2[] : memref<f32>
```



Options ¶

```
-v1                : Set the vector length (use 0 to disable vectorization)
-enable-vla-vectorization : Enable vector length agnostic vectorization
-enable-simd-index32    : Enable i32 indexing into vectors (for efficient gather/scatter)
```

-sparsification ¶

Automatically generate sparse tensor code from sparse tensor types

A pass that implements the core functionality of a **sparsifier**. Each Linalg operation (MLIR's tensor index notation) that operates on sparse tensor types is converted into code in which the sparsity is explicit both in terms of co-iterating looping logic as well as selected sparse storage schemes.

See the `SparseTensor` dialect documentation for more background.

Example input:

```
#matvec = {
  indexing_maps = [
    affine_map<(i,j) -> (i,j)>, // A
    affine_map<(i,j) -> (j)>,   // b
    affine_map<(i,j) -> (i)>    // x (out)
  ],
  iterator_types = ["parallel", "reduction"],
  doc = "X(i) += A(i,j) * B(j)"
}

// Multiply a sparse matrix A with a dense vector b into a dense vector x.
func.func @kernel_matvec(%arga: tensor<?x?xf64, #SparseMatrix>,
                        %argb: tensor<?xf64>,
                        %argx: tensor<?xf64>) -> tensor<?xf64> {
  %0 = linalg.generic #matvec
  ins(%arga, %argb: tensor<?x?xf64, #SparseMatrix>, tensor<?xf64>)
  outs(%argx: tensor<?xf64>) {
    ^bb(%a: f64, %b: f64, %x: f64):
      %0 = arith.mulf %a, %b : f64
```



```

    %1 = arith.addf %x, %0 : f64
    linalg.yield %1 : f64
  } -> tensor<?xf64>
  return %0 : tensor<?xf64>
}

```

Options ¶

```

-parallelization-strategy : Set the parallelization strategy
-sparse-emit-strategy     : Emit functional code or interfaces (to debug) for sparse loops
-enable-runtime-library   : Enable runtime library for manipulating sparse tensors

```

-sparsification-and-bufferization ¶

Mini-pipeline that combines bufferization and sparsification

This pass forms a mini-pipeline that combines bufferization and sparsification.

Options ¶

```

-vl                               : Set the vector length (use 0 to disable vectorization)
-enable-vla-vectorization         : Enable vector length agnostic vectorization
-enable-simd-index32             : Enable i32 indexing into vectors (for efficient gather/scatter)
-enable-gpu-libgen               : Enable GPU acceleration by means of direct library calls
-sparse-emit-strategy            : Emit functional code or interfaces (to debug) for sparse loops
-parallelization-strategy        : Set the parallelization strategy

```

-stage-sparse-ops ¶

Decompose a complex sparse operation into multiple stages

A pass that decomposes a complex sparse operation into multiple stages. E.g., CSR -> CSC is staged into CSR -> COO (unordered) -> sort -> CSC.

‘spv’ Dialect Passes ¶



`-decorate-spirv-composite-type-layout` ¶

Decorate SPIR-V composite type with layout info

Module pass that converts composite types used by objects in the `StorageBuffer`, `PhysicalStorageBuffer`, `Uniform`, and `PushConstant` storage classes to attach layout information. Right now this pass only supports Vulkan layout rules.

`-spirv-canonicalize-gl` ¶

Canonicalize GLSL ops

Pass to run canonicalization patterns that involve GL ops. These patterns cannot be run in default canonicalization because GL ops aren't always available. So they should be invoked specifically when needed.

`-spirv-lower-abi-attrs` ¶

Lower SPIR-V ABI attributes to global variables and entry points

Operation pass that lowers the ABI attributes specified during SPIR-V Lowering. Specifically:

1. Creates the global variables for arguments of entry point function using the specification in the `spirv.interface_var_abi` attribute for each argument.
2. Inserts the `EntryPointOp` and the `ExecutionModeOp` for entry point functions using the specification in the `spirv.entry_point_abi` attribute.

`-spirv-promote-to-replicated-constants` ¶

Convert splat composite constants and spec constants to corresponding replicated constant composite ops defined by `SPV_EXT_replicated_composites`

`-spirv-rewrite-inserts` ¶

Rewrite sequential chains of `spirv.CompositeInsert` operations into `spirv.CompositeConstruct` operations

`-spirv-unify-aliased-resource` ¶

Unify access of multiple aliased resources into access of one single resource



`-spirv-update-vce` ¶

Deduce and attach minimal (version, capabilities, extensions) requirements to spirv.module ops

Operation pass that deduces and attaches the minimal version/ capabilities/extensions requirements for spirv.module ops. For each spirv.module op, this pass requires a `spirv.target_env` attribute on it or an enclosing module-like op to drive the deduction. The reason is that an op can be enabled by multiple extensions/capabilities. So we need to know which one to pick. `spirv.target_env` gives the hard limit as for what the target environment can support; this pass deduces what are actually needed for a specific spirv.module op.

`-spirv-webgpu-prepare` ¶

Prepare SPIR-V to target WebGPU by expanding unsupported ops and replacing with supported ones

'tensor' Dialect Passes ¶

`-fold-tensor-subset-ops` ¶

Fold tensor subset ops into producer/consumer ops

The pass folds tensor subset ops into producer/consumer ops.

At the moment, the following foldings occur when possible:

- `tensor.extract_slice` into `vector.transfer_read`
- `vector.transfer_write` into `tensor.insert_slice`

`-scalarize-single-element-tensor-return` ¶

Scalarize private functions that return single-element tensor

Rewrites private functions returning exactly one statically shaped ranked tensor of exactly one element into functions returning the tensor element type, provided all transitive users can be updated safely.

'transform' Dialect Passes ¶

`-transform-dialect-check-uses` ¶

Warn about potential use-after-free in the transform dialect



This pass analyzes operations from the transform dialect and its extensions and warns if a transform IR value may be used by an operation after it was “freed” by some other operation, as described by side effects on the `TransformMappingResource`. This statically detects situations that lead to errors when interpreting the Transform IR.

The pass is capable of handling branching control flow and reports all *potential* use-after-free situations, e.g., a may-use-after-free is reported if at least one of the control flow paths between the definition of a value and its use contains an operation with a “free” effect on the `TransformMappingResource`. It does not currently perform an SCCP-style data flow analysis to prove that some branches are not taken, however, SCCP and other control flow simplifications can be performed on the transform IR prior to this pass provided that transform ops implement the relevant control flow interfaces.

-transform-infer-effects ¶

Infer transform side effects for symbols

This pass analyzes the definitions of transform dialect callable symbol operations, such as `transform.named_sequence`, and annotates the symbol arguments with attributes indicating the side effects that the nested operations have on them.

-transform-interpreter ¶

Transform dialect interpreter

This pass runs the transform dialect interpreter and applies the named sequence transformation specified by the provided name (defaults to `TransformDialect::kTransformEntryPointSymbolName`, i.e. `__transform_main`).

Additional options can be used to narrow down the pass applicability for debugging purposes:

- `debugPayloadRootTag` makes the transform script apply to the payload operation that has a `transform.target_tag` string attribute with the given value, rather than to the anchor operation of the pass.
- `debugBindTrailingArgs` allows one to bind values to trailing arguments of the transform entry point as follows:
 - arguments of `TransformHandleTypeInterface` type can be bound to all payload operations with the name provided as a simple string;
 - arguments of `TransformValueHandleTypeInterface` type can be bound to a flattened list of results of all operations with the name provided as a string prefixed with `^`;



- arguments of `TransformParamTypeInterface` type can be bound to integer constants provided as ; - separated list prefixed with # .
- `entryPoint` specifies the name of the transform symbol to serve as the entry point.

Options ¶

`-debug-payload-root-tag` : Select the operation with 'transform.target_tag' attribute having the given value as payload IR root. If empty select the pass anchor operation as the payload IR root.

`-debug-bind-trailing-args` : Binds trailing arguments of the entry point to the payload operations with specified names.

`-disable-expensive-checks` : Disable expensive checks in the interpreter for a faster run.

`-entry-point` : Entry point of the pass pipeline.

`-transform-preload-library` ¶

Preload transform dialect library

This pass preloads a transform library and makes it available to subsequent transform interpreter passes. The preloading occurs into the Transform dialect and thus provides very limited functionality that does not scale.

Warning: Only a single such pass should exist for a given MLIR context. This is a temporary solution until a resource-based solution is available.

Options ¶

`-transform-library-paths` : Optional paths to files with modules that should be merged into the transform module to provide the definitions of external named sequences.

‘vector’ Dialect Passes ¶

`-lower-vector-mask` ¶

Lower ‘vector.mask’ operations

`-lower-vector-multi-reduction` ¶



Lower 'vector.multi_reduction' operations

Options ¶

`-lowering-strategy` : Select the strategy to control how multi_reduction is lowered.

`-lower-vector-to-from-elements-to-shuffle-tree` ¶

Lower `vector.to_elements` and `vector.from_elements` to a tree of `vector.shuffle` operations

TOSA Dialect Passes ¶

`-tosa-arith-const-to-tosa-const` ¶

Convert `tensor.arith.constant` operations into `tosa.const`

Normalizes tensor-valued `arith.constant` operations into `tosa.const` so that subsequent TOSA passes operate on a consistent representation of constants.

`-tosa-attach-target` ¶

Attach `tosa.target_env` information to the given module.

This pass allows the user to specify a TOSA target environment consisting of the following components: level, profiles and extensions.

The target environment is attached to the module as an attribute, allowing other transformations to query the selected target and adapt their behaviour based on this information.

Options ¶

`-specification_version` : The specification version that TOSA operators should conform to.
`-level` : The TOSA level that operators should conform to. A TOSA level defines operator argument ranges that an implementation shall support.
`-profiles` : The TOSA profile(s) that operators should conform to. TOSA profiles enable efficient implementation on different classes of device. Each profile is an independent set of operations and data type combinations.



`-extensions` : The TOSA extension(s) that operators should conform to. TOSA profile extensions define optional operation and data type combinations.

`-tosa-convert-integer-type-to-signless` ¶

Convert integer types to signless

This pass converts signed or unsigned integer types to signless. It currently does this greedily for all operators and can also change the signature of the function. Should the signature of the entrypoint function change, it will be the responsibility of the user to carry signedness information of the inputs and outputs independently.

This can be a useful transformation for conversion to other formats that require strict adherence to the TOSA specification.

`-tosa-downgrade-1-1-to-1-0` ¶

Downgrade TOSA 1.1 specification constructs to TOSA 1.0

Rewrites constructs which are only compatible in TOSA specification 1.1 and above to their TOSA 1.0 counterparts where possible. Downgrading is best-effort and validation should be performed afterwards to ensure compatibility with the TOSA 1.0 specification.

`-tosa-experimental-input-shape` ¶

Override dynamic function arguments to specified static shapes.

Pass that overrides the dynamic input shapes of function arguments to specified static shapes. If a specified static shape conflicts with the static dimensions in an original input shape, an error is reported.

Options ¶

`-args` : Comma-separated list of shape descriptions. Each description contains the argument name, a colon, and a shape with dimensions separated by x

`-tosa-infer-shapes` ¶

Propagate shapes across TOSA operations



Pass that uses operand types and propagates shapes to TOSA operations. This includes legalizing rankless and dynamic shapes towards static.

Options ¶

```
-fold-shape-expressions      : Fold TOSA shape operations when they have known input values
-convert-function-boundaries : If enabled, the pass will convert function I/O types as
                             well. Otherwise casts will be inserted at the I/O boundaries.
```

-tosa-layerwise-constant-fold ¶

Fold layerwise operations on constant tensors

Pass that enables folding of full-layer operations on constant tensors.

Options ¶

```
-aggressive-reduce-constant : Always perform the reduce constant optimization. May add more
                             tosa.const but would reduce runtime calculations
```

-tosa-make-broadcastable ¶

TOSA rank Reshape to enable Broadcasting

Pass that enables broadcast by making all input arrays have the same number of dimensions. Insert RESHAPE operations to prepend dimensions of size one until the number of dimensions is equal. Implements approach similar to step 1 of Numpy 4-step broadcasting: <https://numpy.org/doc/stable/reference/ufuncs.html#broadcasting>

-tosa-narrow-f64-to-f32 ¶

Narrow F64 TOSA operations to F32

This pass narrows TOSA operations with 64-bit floating-point tensor types to 32-bit floating-point tensor types. While TOSA itself has no double precision support, upstream conversions or frontends may still materialize F64 tensors temporarily, so this pass removes them before handing off to a TOSA backend.

Options ¶



```
-aggressive-rewrite      : If enabled, all TOSA operations are rewritten, regardless of
whether the narrowing is safe. This option may lead to data loss if not used carefully.
-convert-function-boundaries : If enabled, the pass will convert function I/O types as
well. Otherwise casts will be inserted at the I/O boundaries.
```

-tosa-narrow-i64-to-i32 ¶

Narrow 164 TOSA operations to I32

This pass narrows TOSA operations with 64-bit integer tensor types to 32-bit integer tensor types. This can be useful for backends that do not support the EXT-INT64 extension of TOSA.

Options ¶

```
-aggressive-rewrite      : If enabled, all TOSA operations are rewritten, regardless of
whether the narrowing is safe. This option may lead to data loss if not used carefully.
-convert-function-boundaries : If enabled, the pass will convert function I/O types as
well. Otherwise casts will be inserted at the I/O boundaries.
```

-tosa-optional-decompositions ¶

Applies Tosa operations optional decompositions

Pass to apply the Tosa operations decompositions exposed as populate functions in
include/mlir/Dialect/Tosa/Transforms/Passes.h

-tosa-reduce-transposes ¶

Reduce transposes through other operators

Pass that identifies and reduces tosa.TRANSPOSE operations through chains of operators.

The pass traverses dependencies of tosa.TRANSPOSE operations until they terminate in either a tosa.RESHAPE that we can fold the hoisted tosa.TRANSPOSE into, a tosa.TRANSPOSE that forms the identity with the hoisted one, or a tosa.CONST with a dense elements attribute. It then propagates the hoisted transform upward through the intervening operators if the support is implemented. Finally, it observes that no duplication



will occur of both the chain that was hoisted through and the new chain that results, and if so, it replaces the hoisted tosa.TRANSPOSE.

The pass has an important use-case in cleaning up the results of frameworks that introduce a lot of data-layout transformations when legalizing to TOSA, a common one being transformations between NHWC and NCHW layouts.

-tosa-validate ¶

Validates TOSA dialect

This pass validates if input TOSA operations match the specification for given criteria, e.g. TOSA profile.

Options ¶

```
-strict-op-spec-alignment          : Verify if the properties of certain operations
align the spec requirement
-allow-invalid-op-datatype-combinations : Disable checks for operations that are determined
to be invalid due to their operand/result datatypes not aligning with the 'Supported Data
Types' sections of the specifciation
-validate-function-signature       : Verify if the function signature is valid for
TOSA dialect, i.e. all function arguments and return types are tosa-conformant tensor types
```

XeGPU Dialect Passes ¶

-xegpu-blocking ¶

Block XeGPU ops into smaller size.

This pass partitions operations that process large shapes into multiple operations on smaller shapes, as specified by the `inst_data` in the layout attribute. This enables each resulting operation to be efficiently mapped to a hardware instruction.

-xegpu-optimize-peephole ¶

Optimize XeGPU block load operations

This pass rewrites XeGPU loadNd operations into more optimal forms to improve performance. This includes,



- Rewriting transpose B loads into more optimal forms to use HW block transpose instructions for better performance.

-xegpu-propagate-layout ¶

Propagate and assign XeGPU layout information

This pass propagates the XeGPU layout information accross ops. Starting from a set of anchor operations (e.g. `dpas`, `store_nd`), this will propagate the layouts required for their operands to the producers. With this propagated layout information, pass will then update op result type with the layout information.

layout-kind option values:

- `inst` Propagate the `inst_data` field of the layout attribute. The default is chosen to maximize instruction-level granularity so that the user shape can be processed with the fewest instructions. For N-D operations, this granularity depends on W (width) and H (height) of the instruction shape. The B (block) dimension (or array length) is not included in the default configuration and must be enabled via a separate optimization pass.
- `lane` Propagate the `lane_layout` and `lane_data` fields of the layout attribute. Default values are selected to align with hardware.
- `subgroup` Propagate the `sg_layout` and `sg_data` fields of the layout attribute. Default values are selected to align with hardware.

Options ¶

```
-print-analysis-only : Print the result of layout propagation analysis and exit.
-layout-kind         : Propagate `subgroup` / `inst` / `lane` level of xegpu layouts.
-index-bitwidth      : Vectors of `index` type should also be distributable, inst-data and
lower levels need to know the index size.
```

-xegpu-sg-to-lane-distribute ¶

Distribute XeGPU ops to lanes

The pass distributes subgroup level XeGPU ops to lane level XeGPU ops.

-xegpu-vector-linearize ¶



Linearize n-D vectors to 1-D vectors

This pass linearizes n-D vectors to 1-D vectors for lowering to XeVM.

`-xegpu-wg-to-sg-distribute` ¶

Transform WorkGroup level XeGPU code to SubGroup level

This transform pass distributes the workgroup level computation to multiple subgroups based on the `sg_layout` and `sg_data` attributes.

[← Prev - Pass Infrastructure](#)

[Next - Pattern Rewriting : Generic DAG-to-DAG Rewriting →](#)

Powered by [Hugo](#). Theme by [TechDoc](#). Designed by [Thingsym](#).