LINALG AND THE TRANSFORM DIALECT

Maximilian Bartel

roofline

ALL EXERCISES AND SOLUTIONS ARE IN THE REPOSITORY!

Exercises	git checkout exercises/day-4-session-4
Solutions	git checkout solutions/day-4-session-4

AFTER THIS SESSION YOU CAN WRITE AND SCHEDULE LINEAR ALGEBRA ALGORITHMS USING LINALG AND TRANSFORM

1. Understand Linalg dialect and why this is a very helpful abstraction in a tensor compiler
2. Apply transformations to Linalg to make a tensor program run fast
3. Define your own transformations with the transform dialect
4. Get to know projects that use these concepts on real workloads in production

The running example for this lecture is going to be a matmul followed by an elementwise add

CLASSICAL KERNELS ARE WRITTEN TO GET THE BEST PERFORMANCE FOR A GIVEN TASK AND COMMIT TO A SPECIFIC SCHEDULE

```c
#include <stddef.h>

void mm(int m, int n, int k, const float *restrict A,
        const float *restrict B,
        float *restrict C) {
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
      float s = 0;
      for (int p = 0; p < k; p++)
        s += A[(size_t)i * k + p] * B[(size_t)p * n + j];
      C[(size_t)i * n + j] = s;
    }
}
```

Algorithm

C[i,j] += A[i,k] * B[k,j]

The mathematical update is independent of the loop nesting.

Schedule

loop order	i -> j -> k
work granularity	scalar
locality	bad
parallelism	single threaded

KERNELS ARE MADE OF THE ALGORITHM AND THE SCHEDULE OF THE ALGORITHM

Algorithm

```
C[i,j] += A[i,k] * B[k,j]
```

Schedule A

- plain loops
- row-major traversal
- scalar update

Schedule B

- tile M,N to 64
- parallelize outer tiles
- tile again for registers
- vectorize inner work

While the semantics of the computation stays the same, how the computation is executed to reach your optimization target vastly differs based on the hardware.

The idea is to split the what from the how

LINALG DESCRIBES LINEAR ALGEBRA DECLARATIVELY WITHOUT DECIDING HOW IT WILL BE EXECUTED

```
#map = affine_map<(d0, d1) -> (d0, d1)>
%0 = linalg.generic {
        indexing_maps = [#map, #map, #map],
        iterator_types = ["parallel", "parallel"]
    }
    ins(%arg0, %arg1 : tensor<?x?xf32>, tensor<?x?xf32>)
    outs(%arg2 : tensor<?x?xf32>) {
    ^bb0(%in: f32, %in_0: f32, %out: f32):
        %1 = arith.addf %in, %in_0 : f32
        linalg.yield %1 : f32
    } -> tensor<?x?xf32>

```

linalg.generic

- The most important operation in the Linalg Dialect is linalg.generic.
- It describes linear algebra operations consisting of perfectly nested loops when executed naively.
- It is designed to make reasoning about the loop structure easy.

LINALG DESCRIBES LINEAR ALGEBRA DECLARATIVELY WITHOUT DECIDING HOW IT WILL BE EXECUTED

```
#map = affine_map<(d0, d1) -> (d0, d1)>
%0 = linalg.generic {
        indexing_maps = [#map, #map, #map],
        iterator_types = ["parallel", "parallel"]
    }
    ins(%arg0, %arg1 : tensor<?x?xf32>, tensor<?x?xf32>)
    outs(%arg2 : tensor<?x?xf32>) {
    ^bb0(%in: f32, %in_0: f32, %out: f32):
        %1 = arith.addf %in, %in_0 : f32
        linalg.yield %1 : f32
    } -> tensor<?x?xf32>

```

Iteration space

- The iteration space for a loop is fully defined by the dimension sizes of inputs and outputs, even when they are dynamic.
- The indexing_maps then define which element is picked in which iteration of the innermost loop.

LINALG DESCRIBES LINEAR ALGEBRA DECLARATIVELY WITHOUT DECIDING HOW IT WILL BE EXECUTED

```
1   #map = affine_map<(d0, d1) -> (d0, d1)>
2   %0 = linalg.generic {
3           indexing_maps = [#map, #map, #map],
4           iterator_types = ["parallel", "parallel"]
5       }
6       ins(%arg0, %arg1 : tensor<?x?xf32>, tensor<?x?xf32>)
7       outs(%arg2 : tensor<?x?xf32>) {
8       ^bb0(%in: f32, %in_0: f32, %out: f32):
9           %1 = arith.addf %in, %in_0 : f32
10          linalg.yield %1 : f32
11      } -> tensor<?x?xf32>
12
```

Iterator_types

- The iterator_types tell you easily how many loops a naïve lowering would produce and what properties the loops has.
- Upstream MLIR implements parallel and reduction loops.

LINALG DESCRIBES LINEAR ALGEBRA DECLARATIVELY WITHOUT DECIDING HOW IT WILL BE EXECUTED

```
#map = affine_map<(d0, d1) -> (d0, d1)>
%0 = linalg.generic {
        indexing_maps = [#map, #map, #map],
        iterator_types = ["parallel", "parallel"]
    }
    ins(%arg0, %arg1 : tensor<?x?xf32>, tensor<?x?xf32>)
    outs(%arg2 : tensor<?x?xf32>) {
    ^bb0(%in: f32, %in_0: f32, %out: f32):
        %1 = arith.addf %in, %in_0 : f32
        linalg.yield %1 : f32
    } -> tensor<?x?xf32>

```

Linalg operation body

- The body of a linalg operation then operates on scalar values and tell you what the operation does semantically.

LINALG DESCRIBES LINEAR ALGEBRA DECLARATIVELY WITHOUT DECIDING HOW IT WILL BE EXECUTED

```
#map = affine_map<(d0, d1) -> (d0, d1)>
%0 = linalg.generic {
        indexing_maps = [#map, #map, #map],
        iterator_types = ["parallel", "parallel"]
    }
    ins(%arg0, %arg1 : tensor<?x?xf32>, tensor<?x?xf32>)
    outs(%arg2 : tensor<?x?xf32>) {
    ^bb0(%in: f32, %in_0: f32, %out: f32):
        %1 = arith.addf %in, %in_0 : f32
        linalg.yield %1 : f32
    } -> tensor<?x?xf32>

```

Destination passing style

- The linalg.generic operation is what we call a ”destination passing style” operation.
- This is because we are passing in a tensor as an argument where the result should be written to.
- This breaks the tensor abstraction but makes bufferization later a lot easier.

EXERCISE 1

MAPPING LINEAR ALGEBRA TO A LINALG GENERIC NEEDS A BIT OF PRACTICE

```
%0 = linalg.generic {
        indexing_maps = [???],
        iterator_types = [???]
    }
    ins(???)
    outs(???) {
    ^bb0(???):
        ???
    } -> ???

```

1. C = A * B Matmul
2. D = A * B + C Matmul and elementwise addition
3. NHWC convolution
4. Bonus: NCHW convolution

This exercise should hopefully give you a bit of intuition on how to map real workloads onto linalg.generic operations.

EXERCISE 2

UNDERSTANDING THE ALGORITHM FROM THE LINALG OPERATION ALSO TAKES PRACTICE

```
%result = linalg.generic {
    indexing_maps = [
      affine_map<(b, m, n, k) -> (b, m, k)>,
      affine_map<(b, m, n, k) -> (n, k)>,
      affine_map<(b, m, n, k) -> (b, m, n)>
    ],
    iterator_types = [
      "parallel", "parallel", "parallel", "reduction"
    ]
  }
  ins(%A, %B : tensor<?x?x?xf32>, tensor<?x?xf32>)
  outs(%C : tensor<?x?x?xf32>) {
  ^bb0(%a: f32, %b: f32, %acc: f32):
    %product = arith.mulf %a, %b : f32
    %sum = arith.addf %acc, %product : f32
    linalg.yield %sum : f32
  } -> tensor<?x?x?xf32>
```

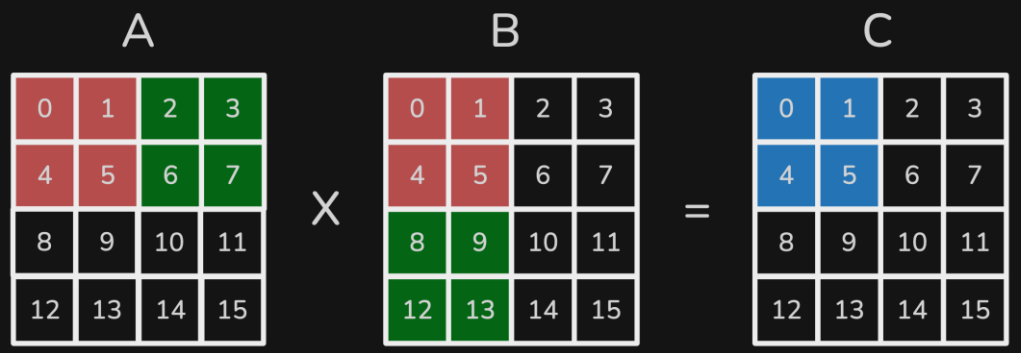
Interpreting linalg.generic

- Interpreting linalg.generic operations is also not always trivial.
- Let's practice this a few times! The output should be a pseudo loop nest like

```
for b in 0..B:              // parallel
  for i in 0..M:            // parallel
    for j in 0..N:          // parallel
      for k in 0..K:        // reduction
        C[b,i,j] += A[b,i,k] * B[b,k,j]
```

TILING IS A LOOP TRANSFORMATION THAT SPLITS LOOPS TO GET BETTER MEMORY ACCESS PATTERNS

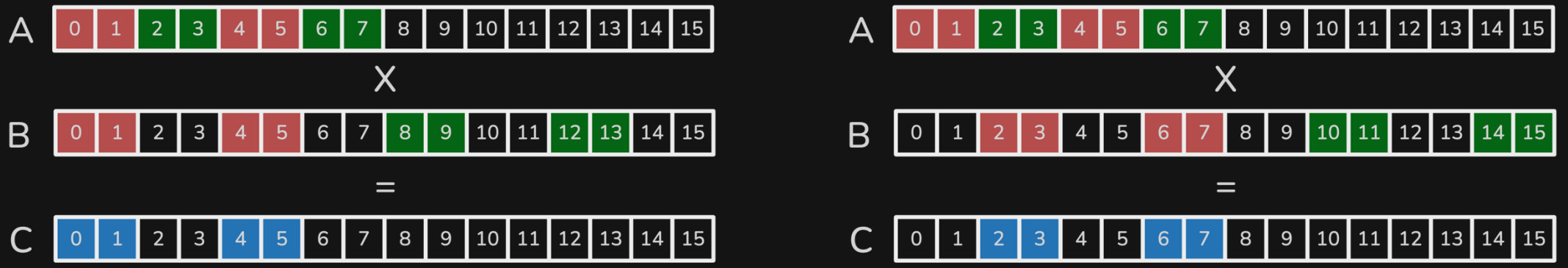
Logical layout



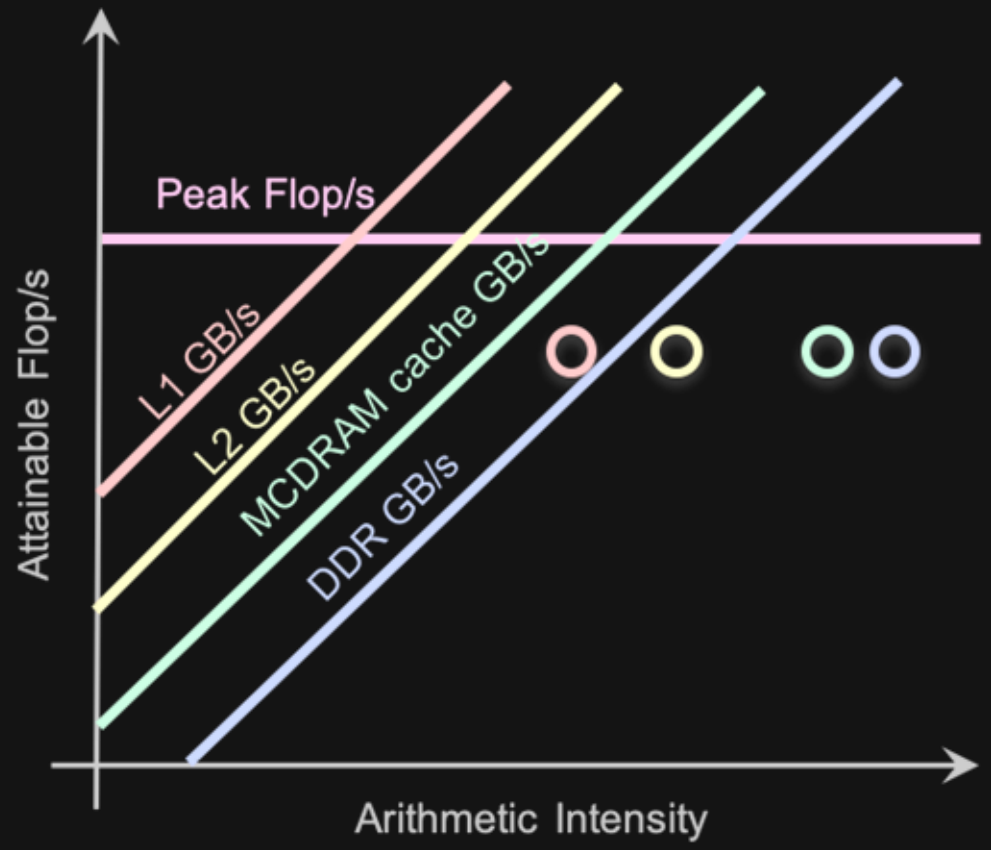
Fast algorithm, but it generates cache misses:

- ✓ Reads from A are good
- ✗ Reads from B and C are not

In-memory layout



TILING INCREASES THE PERFORMANCE BECAUSE WE ARE REUSING DATA THAT IS AVAILABLE IN FASTER MEMORIES



TILING A LINALG OPERATION GIVES YOU A NEW LINALG OPERATION TO CREATE YOUR SCHEDULE STEP BY STEP

```
%0 = linalg.matmul
    ins(%arg0, %arg1 : tensor<?x?xf32>, tensor<?x?xf32>)
    outs(%arg2 : tensor<?x?xf32>) -> tensor<?x?xf32>
```

- Tiling a linalg operation will create loops and a new linalg operation.
- You can tile in multiple steps or stages and create your schedule with that.
- While this transformation is very mechanical, composing them in a meaningful way is very complex.

Tiling

```
%0 = scf.for %arg6 = %c0 to %dim step %c8
        iter_args(%arg7 = %arg2) -> (tensor<?x?xf32>) {
    %1 = scf.for %arg8 = %c0 to %dim_0 step %c16
            iter_args(%arg9 = %arg7) -> (tensor<?x?xf32>) {
        %2 = affine.min #map(%arg6)[%dim]
        %3 = affine.min #map1(%arg8)[%dim_0]
        %extracted_slice = tensor.extract_slice ...
        %extracted_slice_2 = tensor.extract_slice ...
        %extracted_slice_3 = tensor.extract_slice ...
        %4 = linalg.matmul
            ins(%extracted_slice, %extracted_slice_2 :
                tensor<?x?xf32>, tensor<?x?xf32>)
            outs(%extracted_slice_3 : tensor<?x?xf32>)
                -> tensor<?x?xf32>
        %inserted_slice = tensor.insert_slice %4 into ...
        scf.yield %inserted_slice : tensor<?x?xf32>
    }
    scf.yield %1 : tensor<?x?xf32>
}
```

TRANSFORM DIALECT CAN REPRESENT TRANSFORMATIONS AS IR WITHOUT THE NEED TO CHANGE THE COMPILER

```
func.func @foo(
    %arg0: tensor<?x?xf32>, %arg1: tensor<?x?xf32>, %arg2: tensor<?x?xf32>)
        -> tensor<?x?xf32> {
    %0 = linalg.matmul {__producer__}
        ins(%arg0, %arg1 : tensor<?x?xf32>, tensor<?x?xf32>)
        outs(%arg2 : tensor<?x?xf32>) -> tensor<?x?xf32>
    return %0 : tensor<?x?xf32>
}
module attributes {transform.with_named_sequence} {
    transform.named_sequence @__transform_main(
        %arg0: !transform.any_op {transform.readonly}) {
        %0 = transform.structured.match
            attributes {__producer__} in %arg0 : (!transform.any_op)
            -> !transform.any_op
        %tiled_linalg_op, %loops:2 = transform.structured.tile_using_for %0
            tile_sizes [8, 16] : (!transform.any_op)
            -> (!transform.any_op, !transform.any_op, !transform.any_op)
      transform.yield
    }
}
```

Transform dialect

- The transform dialect lets you describe transformations of IR as IR.
- The IR that you are going to transform is called payload IR.
- The transformation instructions transform IR.

THE TRANSFORMS ARE DRIVEN BY AN INTERPRETER ON OPERATIONS THAT IMPLEMENT SPECIFIC INTERFACES

Writing transform ops

- To write your own transform operation you need to define the operation and then write implementations for a few interfaces.
- Calling --transform-interpreter runs the code then.

```
// Define the new operation. By convention, prefix its name with the name of the dialect
// extension, "my.". The full operation name will be further prefixed with "transform.".
def ChangeCallTargetOp : Op<Transform_Dialect, "my.change_call_target",
    // Indicate that the operation implements the required TransformOpInterface and
    // MemoryEffectsOpInterface.
    [DeclareOpInterfaceMethods<TransformOpInterface>,
     DeclareOpInterfaceMethods<MemoryEffectsOpInterface>]> {
  // Provide a brief and a full description. It is recommended that the latter describes
  // the effects on the operands and how the operation processes various failure modes.
  let summary = "Changes the callee of a call operation to the specified one";
  let description = [{ ...
  }];
```

```

  // The arguments include the handle to the payload operations and the attribute that
  // specifies the new callee. The handle must implement TransformHandleTypeInterface.
  // We use a string attribute as the symbol may not exist in the transform IR so the
  // verification may fail.
  let arguments = (ins
    TransformHandleTypeInterface:$call,
    StrAttr:$new_target);

  // The results are empty as the transformation does not produce any new payload.
  let results = (outs);

  // Provide nice syntax.
  let assemblyFormat = "$call `,` $new_target attr-dict `:` type($call)";
}
```

EXERCISE 3

WHEN WRITING COMPLEX TRANSFORMATION CHAINS, YOU WANT TO INSPECT THE CODE SOMETIMES

Everyone makes mistakes. Let’s implement a transform operation that lets us printf debug the IR during the execution of a transform script!

TRANSFORM DIALECT CAN BE USED VERY SIMILAR TO HALIDE TO SCHEDULE LINALG OPERATIONS

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

Halide, transform & schedules

- Halide and transform dialect follow the same mental model of separation of concerns.
- Schedules are explicit, reviewable, testable, and can be change independently of the payload.
- Transform IR goes even a step beyond that because to change the schedule there is no need to recompile the compiler itself. It is an input that can be written in its own tool.

WHILE EXECUTING A TRANSFORM SCRIPT YOU ARE CHANGING THE IR SIMULTANEOUSLY

```
%mm = transform.structured.match ...

%tiled, %loops:2 =
    transform.structured.tile_using_for %mm
      tile_sizes [8, 16, 0]

// Error: %mm was consumed.
transform.debug.emit_remark_at %mm, "old matmul"

// Correct: %tiled is the new tiled matmul operation.
transform.debug.emit_remark_at %tiled, "new matmul"
```

Handle invalidation

- The transform interpreter changes the payload IR on the fly.
- This means that if a transformation is destructive, e.g., erasing an operation, then handle is also invalid.
- There is --transform-dialect-check-uses to make sure the script is valid. With dynamic behavior this can get quite complicated.

Rule of thumb: After a consuming transform, use the new handles it returns.

YOU CAN USE MULTIPLE TRANSFORM OPERATIONS AFTER ONE ANOTHER TO EXECUTE MULTIPLE TRANSFORMATIONS

```
transform.named_sequence @__transform_main(%arg0: !transform.any_op {transform.readonly}) {
    %0 = transform.structured.match attributes {__root__} in %arg0
        : (!transform.any_op) -> !transform.any_op
    %1 = transform.structured.match attributes {__producer__} in %arg0
        : (!transform.any_op) -> !transform.any_op
    %tiled_linalg_op, %loops:2 = transform.structured.tile_using_for %0 tile_sizes [8, 16]
        : (!transform.any_op) -> (!transform.any_op, !transform.any_op, !transform.any_op)
    %fused_op, %new_containing_op = transform.structured.fuse_into_containing_op %1 into %loops#0
        : (!transform.any_op, !transform.any_op) -> (!transform.any_op, !transform.any_op)
    transform.yield
}
```

EXERCISE 4

WRITING SCHEDULES FOR LINALG WITH THE TRANSFORM DIALECT IS NOT HARD

1. Part: Let’s get a bit of practice writing longer transform scripts! On the right you see the transformations we want to apply.

2. Part: After you finished the script, try to lower the IR to a lower-level abstraction to see if your mental model of the computations are still correct.

```
mlir-opt exercise.mlir \
 --transform-interpreter \
 --linalg-generalize-named-ops \
 --one-shot-bufferize="bufferize-function-boundaries" \
 --convert-linalg-to-loops
```

Reference for a 2048×2048 float32 Matmul on Apple M3 MAX:

- Naïve loops: 8.082 s
- Accelerate Blas: 10.747 ms

Start from the matmul + add payload. Complete the transform sequence.

1. Match add with __root__
2. Match fill + matmul with __producer__
3. Tile the root with [8, 16]
4. Fuse producers into one generated loop
5. Tile the root again with [4, 4]
6. print or diff the resulting payload

UPSTREAM MLIR USES TRANSFORM AND LINALG FOR END-TO-END TESTING OF SOME HW FEATURES

Upstream MLIR SME tests

https://github.com/llvm/llvm-project/blob/main/mlir/test/Integration/Dialect/Linalg/CPU/ArmSME/multi-tile-matmul.mlir

SPOTLIGHT

PROJECT LIGHTHOUSE AND IREE ARE PROMINENT USERS OF THE TRANSFORM DIALECT

MLIR Lighthouse Project

https://github.com/llvm/lighthouse/blob/5d6431fe66f80469589b0f987727d4d80baf1f9e/examples/llama/test_llama3.py#L61

IREE

https://github.com/iree-org/iree/tree/13d9ad9501d878ff61ca1a0f753419f6d78ce85c/samples/transform_dialect

ABOUT US

WE BRING THE EXPERTISE TO ENABLE YOUR AI DEPLOYMENT

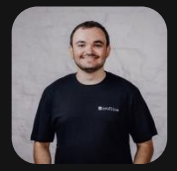
Founded by AI compiler and AI system engineers from RWTH Aachen University.

Built around a core team of ex-AMD engineers, researchers and strategists.

Multi-million funding secured from public institutions and private investors.

CEO
Dr.-Ing. Moritz Joseph
joseph@roofline.ai

www.roofline.ai



CTO
Max Bartel
bartel@roofline.ai

CFO & COO
Dr. Thomas Zimmermann
zimmermann@roofline.ai

Trusted and supported by:

NXP arm intrinSic Co-funded by the European Union Federal Ministry for Economic Affairs and Climate Action

NVIDIA INCEPTION PROGRAM EDGEAI FOUNDATION PARTNER intel liftoff eXIST SPRIN-D