ANATOMY OF LINALG.PACK AND LINALG.UNPACK: EFFICIENT TILING AND VECTORIZATION

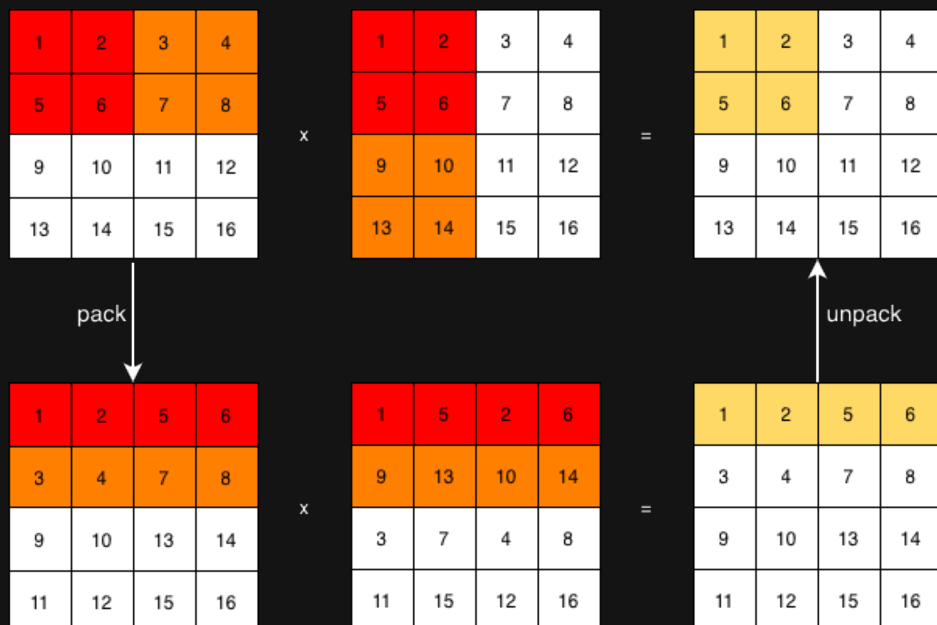
Ege Beysel
beysel@roofline.ai

roofline

LINALG.PACK AND LINALG.UNPACK ARE USED FOR MEMORY LAYOUT TRANSFORMATIONS

Matmul in MLIR

```
1 %matmul = linalg.matmul
2   ins(%lhs, %rhs : tensor<4x4xf32>, tensor<4x4xf32>)
3   outs(%acc : tensor<4x4xf32>) -> tensor<4x4xf32>
```



Data tiling

Data-tiled matmul in MLIR

```
1 %packed_lhs = linalg.pack %lhs ... inner_tiles = [2, 2]
2     into %lhs_dest : tensor<4x4xf32> -> tensor<2x2x2x2xf32>
3 %packed_rhs = linalg.pack %rhs ... inner_tiles = [2, 2]
4     into %rhs_dest : tensor<4x4xf32> -> tensor<2x2x2x2xf32>
5 %packed_acc = linalg.pack %acc ... inner_tiles = [2, 2]
6     into %acc_dest : tensor<4x4xf32> -> tensor<2x2x2x2xf32>
7 %mmt4d = linalg.mmt4d
8     ins(%packed_lhs, %packed_rhs : tensor<2x2x2x2xf32>, ...)
9     outs(%acc_dt : tensor<2x2x2x2xf32>)
10 %unpack = linalg.unpack %mmt4d inner_tiles = [2, 2]
11     into %acc : tensor<2x2x2x2xf32> -> tensor<4x4xf32>
```

- Pack operands in a SIMD, cache-friendly layout
- Layouts dependent on target hardware
- Pack and unpack operations can be fused into their producers and consumers

WE FUSE LINALG.PACK AND LINALG.UNPACK WITH THEIR PRODUCERS AND CONSUMERS TO HIDE THEIR LATENCIES

Entire computation graph

```
1  %elemwise0 = linalg.elementwise ... ins(%in0, %in1 : tensor<128x128xf32>, ...)
2      outs(%elemwise_init0 : tensor<128x128xf32>) -> tensor<128x128xf32>
3  %elemwise1 = linalg.elementwise ... ins(%in2, %in3 : tensor<128x128xf32>, ...)
4      outs(%elemwise_init1 : tensor<128x128xf32>) -> tensor<128x128xf32>
5  %packed_lhs = linalg.pack %elemwise0 ... inner_tiles = [8, 1]
6      into %lhs_dest : tensor<128x128xf32> -> tensor<16x128x8x1xf32>
7  %packed_rhs = linalg.pack %elemwise1 ... inner_tiles = [8, 1]
8      into %rhs_dest : tensor<128x128xf32> -> tensor<16x128x8x1xf32>
9  %mmt4d = linalg.mmt4d
10     ins(%packed_lhs, %packed_rhs : tensor<16x128x8x1xf32>, ...)
11     outs(%acc_dt : tensor<16x16x8x8xf32>)
12 %unpack = linalg.unpack %mmt4d inner_tiles = [8, 8]
13     into %acc : tensor<16x16x8x8xf32> -> tensor<128x128xf32>
14 %bias_add = linalg.elementwise ... ins(%unpack, %in4 : tensor<128x128xf32>, ...)
15     outs(%elemwise_init1 : tensor<128x128xf32>) -> tensor<128x128xf32>
```

Partition

Partitioned subgraphs

```
1  %elemwise* = linalg.elementwise ... ins(%in0, %in1 : tensor<128x128xf32>, ...)
2      outs(%elemwise_init0 : tensor<128x128xf32>) -> tensor<128x128xf32>
3  %packed_*hs = linalg.pack %elemwise0 ... inner_tiles = [8, 1]
4      into %lhs_dest : tensor<128x128xf32> -> tensor<16x128x8x1xf32>
```

```
1  %mmt4d = linalg.mmt4d
2      ins(%packed_lhs, %packed_rhs : tensor<16x128x8x1xf32>, ...)
3      outs(%acc_dt : tensor<16x16x8x8xf32>)
```

```
1  %unpack = linalg.unpack %mmt4d inner_tiles = [8, 8]
2      into %acc : tensor<16x16x8x8xf32> -> tensor<128x128xf32>
3  %bias_add = linalg.elementwise ... ins(%unpack, %in4 : tensor<128x128xf32>, ...)
4      outs(%elemwise_init1 : tensor<128x128xf32>) -> tensor<128x128xf32>
```

PACKING AND UNPACKING OPERATIONS CAN BE PROPAGATED THROUGH OTHER OPERATIONS

Before layout propagation

```
1 %elemwise* = linalg.elementwise ... ins(%in0, %in1 : tensor<128x128xf32>, ...)
2     outs(%elemwise_init0 : tensor<128x128xf32>) -> tensor<128x128xf32>
3 %packed_*hs = linalg.pack %elemwise0 ... inner_tiles = [8, 1]
4     into %lhs_dest : tensor<128x128xf32> -> tensor<16x128x8x1xf32>
```

After layout propagation

Extra pack operations!

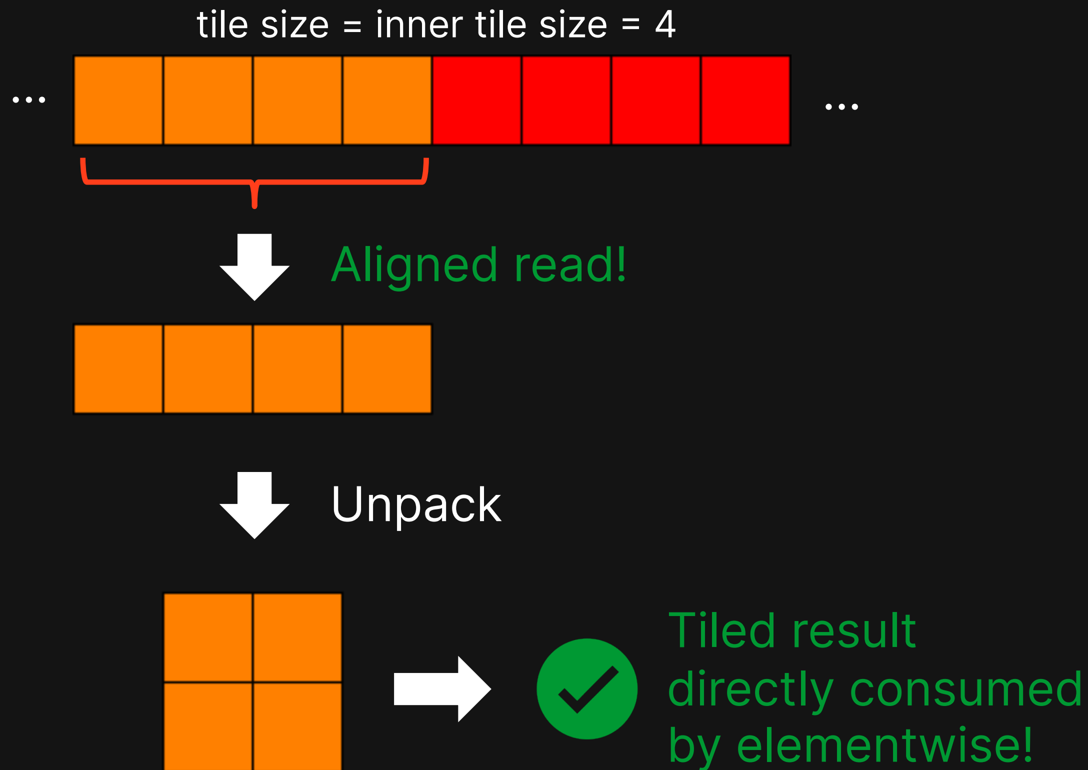
```
1 %packed_in0 = linalg.pack %in0 ... inner_tiles = [8, 1]
2     into %in0_dest : tensor<128x128xf32> -> tensor<16x128x8x1xf32>
3 %packed_in1 = linalg.pack %in1 ... inner_tiles = [8, 1]
4     into %in1_dest : tensor<128x128xf32> -> tensor<16x128x8x1xf32>
5 %elemwise* = linalg.elementwise ...
6     ins(%packed_in0, %packed_in1 : tensor<16x128x8x1xf32>, ...)
7     outs(%elemwise_init0 : tensor<16x128x8x1xf32>) -> tensor<16x128x8x1xf32>
```

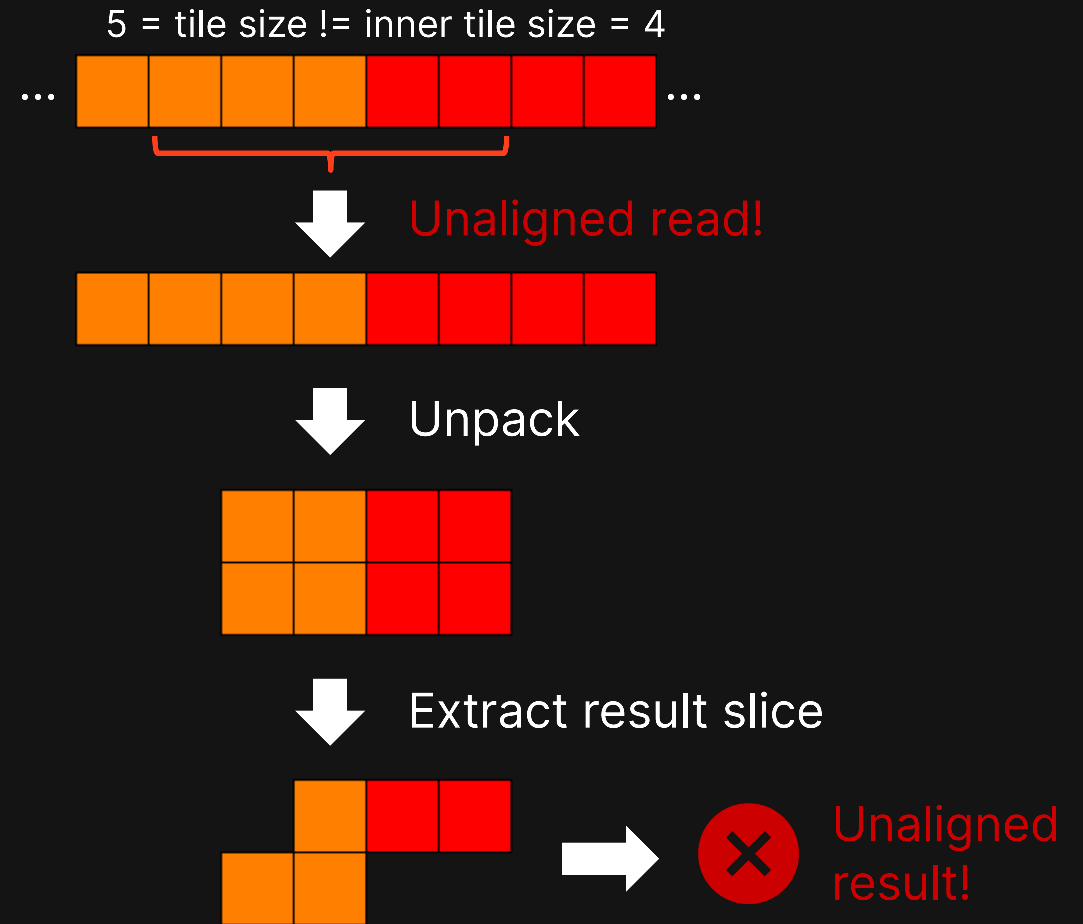
```
1 %unpack = linalg.unpack %mmt4d inner_tiles = [8, 8]
2     into %acc : tensor<16x16x8x8xf32> -> tensor<128x128xf32>
3 %bias_add = linalg.elementwise ... ins(%unpack, %in4 : tensor<128x128xf32>, ...)
4     outs(%elemwise_init1 : tensor<128x128xf32>) -> tensor<128x128xf32>
```

Extra pack operations!

```
1 %packed_in4 = linalg.pack %in4 inner_tiles = [8, 8]
2     into %acc4 : tensor<128x128xf32> -> tensor<16x16x8x8xf32>
3 %bias_add = linalg.elementwise ...
4     ins(%mmt4d, %packed_in4 : tensor<16x16x8x8xf32>, ...)
5     outs(%elemwise_init1 : tensor<16x16x8x8xf32>) -> tensor<16x16x8x8xf32>
6 %unpack = linalg.unpack %bias_add inner_tiles = [8, 8]
7     into %acc : tensor<16x16x8x8xf32> -> tensor<128x128xf32>
```

LINALG.PACK AND LINALG.UNPACK INNER TILE SIZES IMPOSE LOOP TILE SIZE ALIGNMENT OF THEIR PRODUCERS AND CONSUMERS





PACKING HAS PADDING SEMANTICS, UNPACKING IS ABLE TO REVERT THIS PADDING

Padding semantics

- Packing has padding semantics
- Unpacking has “unpadding” semantics.
- Masks can be avoided for *operands* in the packed layouts (e.g., mmt4d)!
- Packing and unpacking transform operands between packed and unpacked layouts:
  - Output & input operands of packing & unpacking can be unmasked!

```
1 // Packed dimension 127 is padded to 16 * 8 = 128!
2 %packed_*hs = linalg.pack %elemwise0 ...
3     padding_value(%c0 : f32) inner_tiles = [8, 1]
4     into %lhs_dest : tensor<127x127xf32> -> tensor<16x127x8x1xf32>
5 %unpack = linalg.unpack %mmt4d inner_tiles = [8, 8]
6     into %acc : tensor<16x16x8x8xf32> -> tensor<127x127xf32>
```

OPERANDS IN PACKED LAYOUTS CAN AVOID EXPENSIVE MASKING

Masked Vectorization

- For unaligned problem sizes, masks will be necessary on boundaries of packed layouts.
- Propagation helps bring operations into packed layouts.
- Peeling can also be applied for remainder loops.

```
1 %mmt4dv = vector.transfer_read %mmt4d_slice ... :
2     tensor<2x1x8x8xf32>, vector<2x1x8x8xf32>
3 %transpose = vector.transpose %mmt4dv, [1, 2, 0, 3] :
4     vector<2x1x8x8xf32> to vector<1x8x2x8xf32>
5 %cast = vector.shape_cast %transpose : vector<1x8x2x8xf32> to vector<8x16xf32>
6 %maskedwrite = vector.mask %maskw {
7     vector.transfer_write %cast, ... tensor<?x?xf32>, vector<8x16xf32>
8     } : vector<8x16xi1> -> tensor<?x?xf32>
9 // Vectorized elementwise ... No store-to-load forwarding due to mask!
10 // Results have to be written to and read from an intermediate buffer!
11 %unpackv = vector.mask %maskr {
12     vector.transfer_read %maskedwrite ... : tensor<?x?xf32>, vector<8x16xf32>
13     } : vector<8x16xi1> -> vector<8x16xf32>
```

LINALG.PACK AND LINALG.UNPACK CANNOT REPRESENT ARBITRARY LAYOUT TRANSFORMATIONS

Current packing scheme does not support 2×2 block of 2×2 tiles

```
1  %packed_lhs = linalg.pack %lhs ... inner_tiles = [4, 4]
2      into %lhs_dest : tensor<128x128xbf16> -> tensor<32x32x4x4xbf16>
3  %packed_rhs = linalg.pack %rhs ... inner_tiles = [4, 4]
4      into %rhs_dest : tensor<128x128xbf16> -> tensor<32x32x8x4xbf16>
5  %packed_acc = linalg.pack %acc ... inner_tiles = [4, 4]
6      into %acc_dest : tensor<128x128xf32> -> tensor<32x32x4x4xf32>
7  %mmt4d = linalg.mmt4d
8      ins(%packed_lhs, %packed_rhs : tensor<32x32x4x4xbf16>, ...)
9      outs(%packed_acc : tensor<32x32x4x4xf32>)
10 %unpack = linalg.unpack %mmt4d inner_tiles = [4, 4]
11     into %acc : tensor<32x32x4x4xf32> -> tensor<128x128xf32>
```

```
1 %lhs0 = vector.extract %lhs ... vector<2x4xbf16> from vector<8x4xbf16>
2 %lhs1 = vector.extract %lhs ... vector<2x4xbf16> from vector<8x4xbf16>
3 %rhs0 = vector.extract %rhs ... vector<2x4xbf16> from vector<8x4xbf16>
4 %rhs1 = vector.extract %rhs ... vector<2x4xbf16> from vector<8x4xbf16>
5 // Shape casts from 2x4xbf16 -> 8xbf16 omitted!
6 %res00 = arm_neon.intr.bfmmla %acc00, %lhs0, %rhs0 : vector<8xbf16> to vector<4xf32>
7 %res01 = arm_neon.intr.bfmmla %acc01, %lhs0, %rhs1 : vector<8xbf16> to vector<4xf32>
8 %res10 = arm_neon.intr.bfmmla %acc10, %lhs1, %rhs0 : vector<8xbf16> to vector<4xf32>
9 %res11 = arm_neon.intr.bfmmla %acc11, %lhs1, %rhs1 : vector<8xbf16> to vector<4xf32>
```

BFMMLA kernel unrolled twice on each parallel dimension

unrolling factor = 2

vector<4x2xbf16>

0	1	8	9
2	3	10	11
4	5	12	13
6	7	14	15

unrolling factor = 2

vector<2x4xbf16>

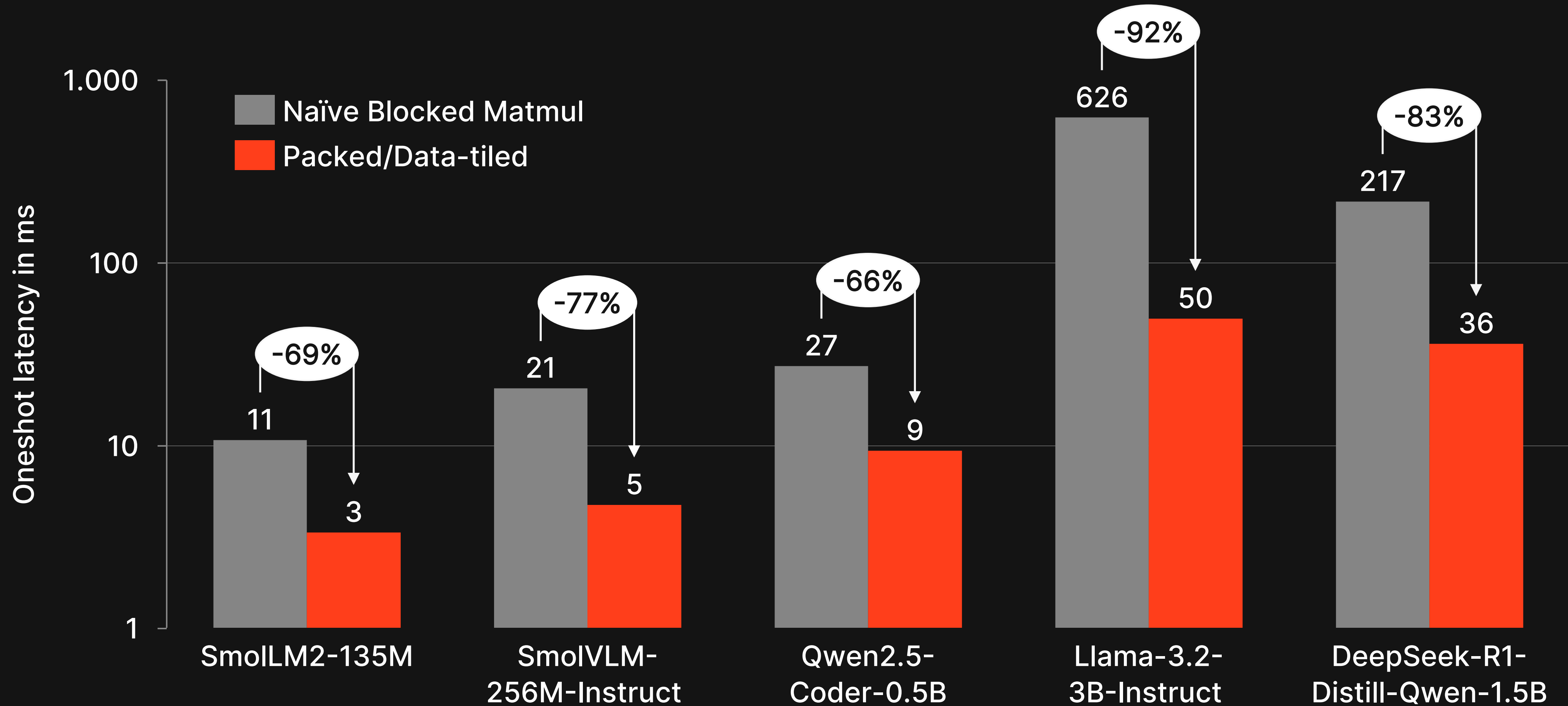
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

vector<2x2xf32>

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

RESULTS

WE SEE LATENCY DECREASES UP TO 92% WHEN RUNNING FULL MODELS



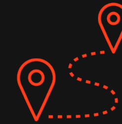
roofline

CONCLUSION

PACK AND UNPACK MAKE MEMORY ACCESSES VERY EFFICIENT, BUT THEY COME AT A COST

How to use pack effectively

- Fuse packing and unpacking operators with their producers and consumers to hide their latencies
- Constant-evaluate them when possible
- Align loop tile sizes of producers and consumers to the packed layouts
- Propagate packed layouts when profitable



What are unexplored paths?

- Explore data tiling for other kernels like convolutions
- Representing arbitrary layouts and code generation with them
    - IREE already supports this through special operations!
- Proper cost model for packed layout propagation profitability

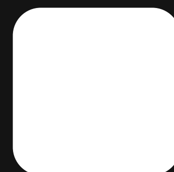
ABOUT US

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.



Ege Beysel

beysel@roofline.ai

www.roofline.ai

Trusted and supported by:

Co-funded by the European Union

Federal Ministry for Economic Affairs and Climate Action

liftoff