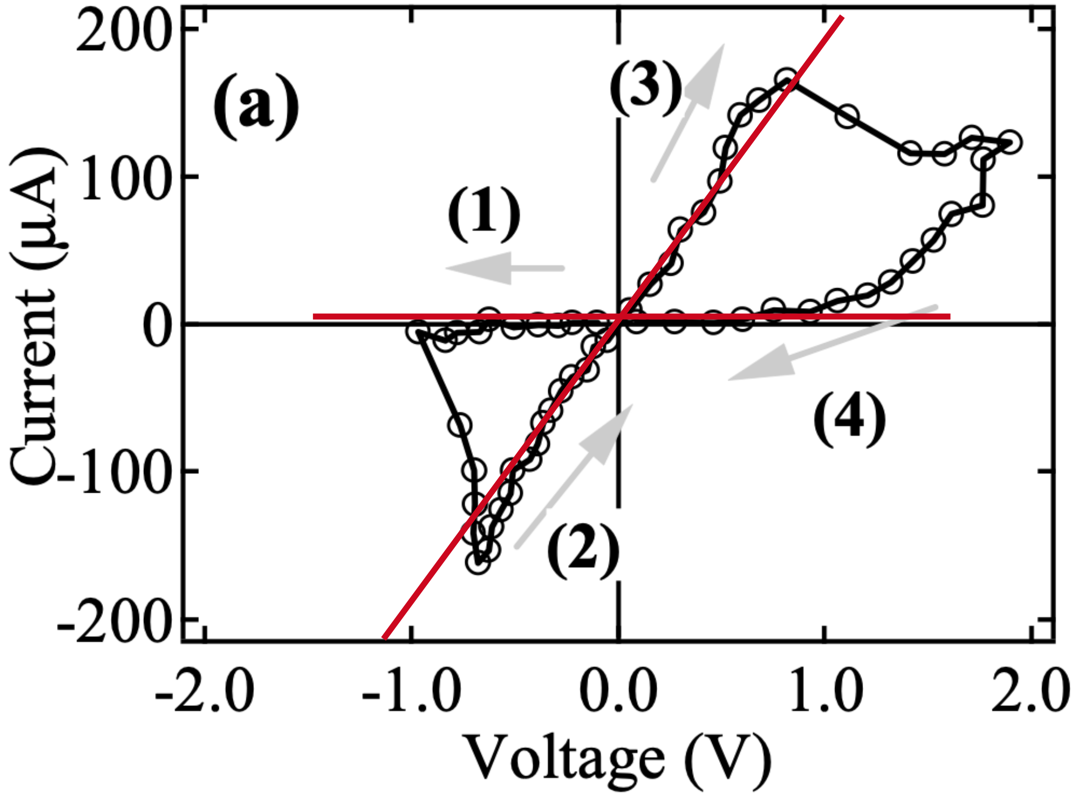
Using MLIR to test ReRAM cells

Maximilian Bartel
EuroLLVM 2024

Institute for Communication Technologies and Embedded Systems
RWTH AACHEN UNIVERSITY

ReRam Cells – What are they?

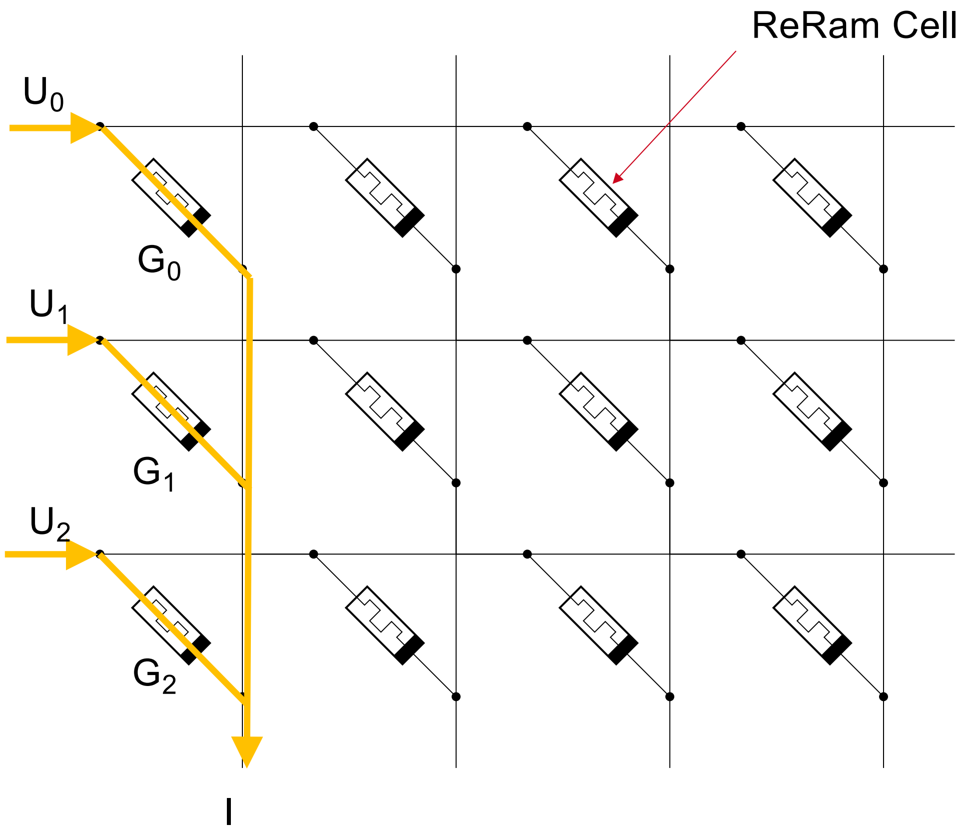


Wei, Zhiqiang, et al. "Highly reliable TaOx ReRAM and direct evidence of redox reaction mechanism.“ 2008 IEEE international electron devices meeting. IEEE, 2008.

- **Re**sistive **Ram** Cells (ReRam - often called memristors) are a new type of electronic device
- Can switch between different levels of resistance
- Switch is non-volatile
- Read speeds comparable to SRAM
- **They are not reliable yet** 🙁

ReRAM – Why is it exciting?

Circuit

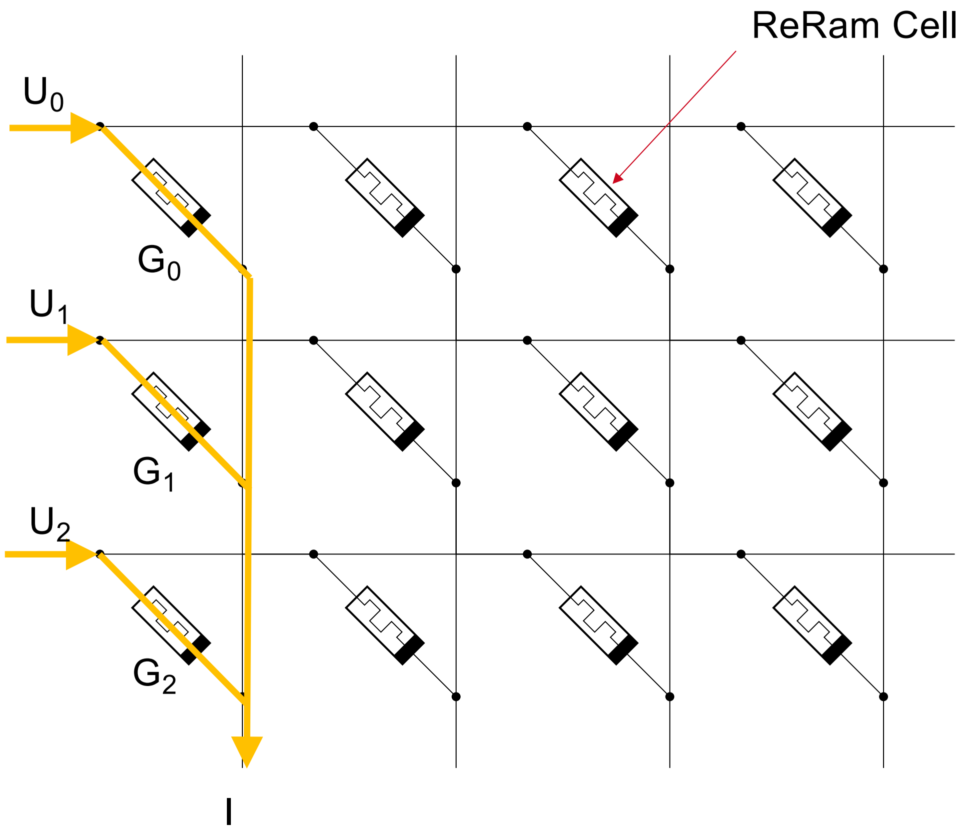


Characteristics

- Typically arranged in *crossbars*
- You get I with: $I = \sum_i U_i G_i$
- This is a multiply-accumulate (MAC) operation!
- Can apply multiple voltages at the same time: Operation is highly parallel and latency independent of number of voltages

ReRAM – Why is it exciting?

Circuit

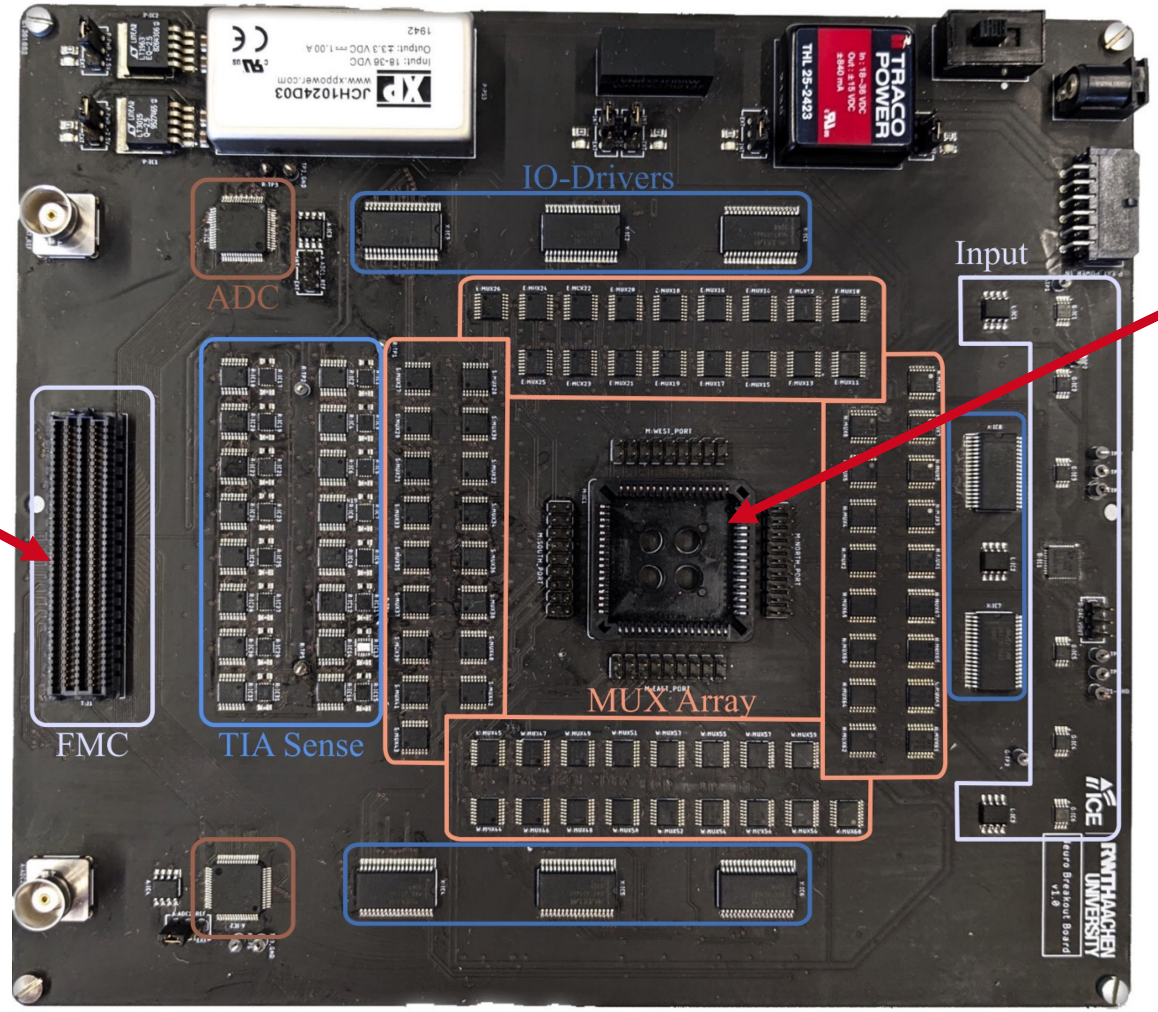


We can use ReRam cells for highly efficient machine learning inference

… If they would work

We developed the NeuroBoad to test single devices and crossbars with real ML workloads

Space for a MCU



Space for the crossbar

Everything can be controlled with a custom python library

Institute for Communication Technologies and Embedded Systems

Linalg Matmul – The base operation we looked at

```
func.func @matmul_tensors(%arg0: tensor<?x?xf32>, %arg1: tensor<?x?xf32>, %arg2:
tensor<?x?xf32>) -> tensor<?x?xf32> {

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

  return %0 : tensor<?x?xf32>
}
```

Using the transform dialect for custom op insertion and tiling to crossbar sizes

Resource: Transform Dialect Tutorial - Docs

```
transform.sequence failures(propagate) {
^bb0(%arg0: !transform.any_op):
  %0 = transform.structured.match ops{["linalg.matmul"]} in %arg0
  %1 = transform.get_consumers_of_result %0[0]
  %2 = transform.get_producer_of_operand %1[1]
  %3 = transform.neuro.create.alloc %2
  %tiled_linalg_op, %loops:3 = transform.structured.tile_using_for %1[1, 0, 4, 16]
  %4 = transform.get_producer_of_operand %tiled_linalg_op[1]
  %5 = transform.neuro.create.write_matrix %3, %4
  %tiled_linalg_op_0, %loops_1 = transform.structured.tile_using_for %tiled_linalg_op[0, 1, 0]
  %6 = transform.get_producer_of_operand %tiled_linalg_op_0[0]
  %7 = transform.get_producer_of_operand %tiled_linalg_op_0[2]
  %8 = transform.neuro.create.matvec %3, %6, %7
  transform.neuro.replace_cast_reshape %tiled_linalg_op_0, %8
}
```

Using the transform dialect for custom op insertion and tiling to crossbar sizes

Resource: Transform Dialect Tutorial - Docs

```
transform.sequence failures(propagate) {
^bb0(%arg0: !transform.any_op):
  %0 = transform.structured.match ops{["linalg.matmul"]} in %arg0
  %1 = transform.get_consumers_of_result %0[0]
  %2 = transform.get_producer_of_operand %1[1]
  %3 = transform.neuro.create.alloc %2
  %tiled_linalg_op, %loops:3 = transform.structured.tile_using_for %1[1, 0, 4, 16]
  %4 = transform.get_producer_of_operand %tiled_linalg_op[1]
  %5 = transform.neuro.create.write_matrix %3, %4
  %tiled_linalg_op_0, %loops_1 = transform.structured.tile_using_for %tiled_linalg_op[0, 1, 0]
  %6 = transform.get_producer_of_operand %tiled_linalg_op_0[0]
  %7 = transform.get_producer_of_operand %tiled_linalg_op_0[2]
  %8 = transform.neuro.create.matvec %3, %6, %7
  transform.neuro.replace_cast_reshape %tiled_linalg_op_0, %8
}
```

Using the transform dialect for custom op insertion and tiling to crossbar sizes

Resource: Transform Dialect Tutorial - Docs

Crossbar size

```
transform.sequence failures(propagate) {
^bb0(%arg0: !transform.any_op):
  %0 = transform.structured.match ops{["linalg.matmul"]} in %arg0
  %1 = transform.get_consumers_of_result %0[0]
  %2 = transform.get_producer_of_operand %1[1]
  %3 = transform.neuro.create.alloc %2
  %tiled_linalg_op, %loops:3 = transform.structured.tile_using_for %1[1, 0, 4, 16]
  %4 = transform.get_producer_of_operand %tiled_linalg_op[1]
  %5 = transform.neuro.create.write_matrix %3, %4
  %tiled_linalg_op_0, %loops_1 = transform.structured.tile_using_for %tiled_linalg_op[0, 1, 0]
  %6 = transform.get_producer_of_operand %tiled_linalg_op_0[0]
  %7 = transform.get_producer_of_operand %tiled_linalg_op_0[2]
  %8 = transform.neuro.create.matvec %3, %6, %7
  transform.neuro.replace_cast_reshape %tiled_linalg_op_0, %8
}
```

Convert to function call to match API of the NeuroBoard python package

```
func.func @matvec(%arg0: index, %arg1: memref<12xf32>, %arg2: memref<2x3xf32>) {
  neuro.matvec %arg0, %arg0, %arg1 ,%arg2: (index, index, memref<12xf32>, memref<2x3xf32>)
  return
}
```

Casts to dynamic shapes, but can also collapse or expand shapes if necessary

```
func.func @matvec(%arg0: index, %arg1: memref<12xf32>, %arg2: memref<2x3xf32>) {
  %0 = memref.cast %arg1 : memref<12xf32> to memref<?xf32>
  %1 = memref.cast %arg2 : memref<2x3xf32> to memref<?x?xf32>
  call @neuro_matvec(%arg0, %arg0, %0, %1) : (index, index, memref<?xf32>, memref<?x?xf32>)
  return
}
func.func private @neuro_matvec(index, index, memref<?xf32>, memref<?x?xf32>)
```

Resource: LLVM IR Target - Docs

Python Execution Engine to simulate and to run on the NeuroBoard

Resource: mlir/test/python/execution_engine.py - Tests

```
weight_dict = {}

@ctypes.CFUNCTYPE(
None, ...,
ctypes.POINTER(
make_nd_memref_descriptor(3, np.ctypeslib.as_ctypes_type(np.float32)),
), ...
)
def neuro_matvec(a, b, c, d):
  input = ranked_memref_to_numpy(c).astype(np.float32, copy=False)
  output = ranked_memref_to_numpy(d).astype(np.float32, copy=False)
  output[:] += np.dot(
    weight_dict[a][b : b + (input.size * output.size)].reshape(
      (-1, input.size),
    ),
    input.flatten(),
  )
  return
```

Python Execution Engine to simulate and to run on the NeuroBoard

```
weight_dict = {}
```

“simulation”

Resource: mlir/test/python/execution_engine.py - Tests

```
@ctypes.CFUNCTYPE(
None, ...,
ctypes.POINTER(
make_nd_memref_descriptor(3, np.ctypeslib.as_ctypes_type(np.float32)),
), ...
)
def neuro_matvec(a, b, c, d):
  input = ranked_memref_to_numpy(c).astype(np.float32, copy=False)
  output = ranked_memref_to_numpy(d).astype(np.float32, copy=False)
  output[:] += np.dot(
    weight_dict[a][b : b + (input.size * output.size)].reshape(
      (-1, input.size),
    ),
    input.flatten(),
  )
  return
```

Python Execution Engine to simulate and to run on the NeuroBoard

```python
weight_dict = {}
```

“simulation”

Resource: mlir/test/python/execution_engine.py - Tests

```python
@ctypes.CFUNCTYPE(
None, ...,
ctypes.POINTER(
make_nd_memref_descriptor(3, np.ctypeslib.as_ctypes_type(np.float32)),
), ...
)
def neuro_matvec(a, b, c, d):
  input = ranked_memref_to_numpy(c).astype(np.float32, copy=False)
  output = ranked_memref_to_numpy(d).astype(np.float32, copy=False)
  output[:] += np.dot(
    weight_dict[a][b : b + (input.size * output.size)].reshape(
      (-1, input.size),
    ),
    input.flatten(),
  )
  return
```

Thank you for your attention!