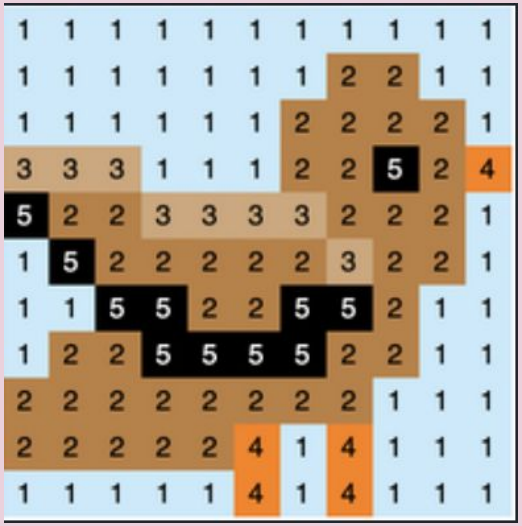
My Internship Journey



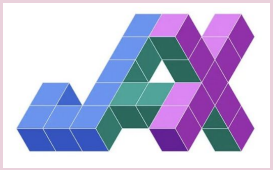
A huge thank you to @Mateusz for his incredible guidance and support!

Agenda

- What is Finch-tensor?
  - Reasons to use Finch
- Project Overview
  - Benefits of MLIR backend
  - Code Generation Pipeline (Dense Matmul kernel)
- Benchmarking Results
- Future Goals and Directions enabled by the MLIR Backend

What is Finch-tensor?

PyTorch

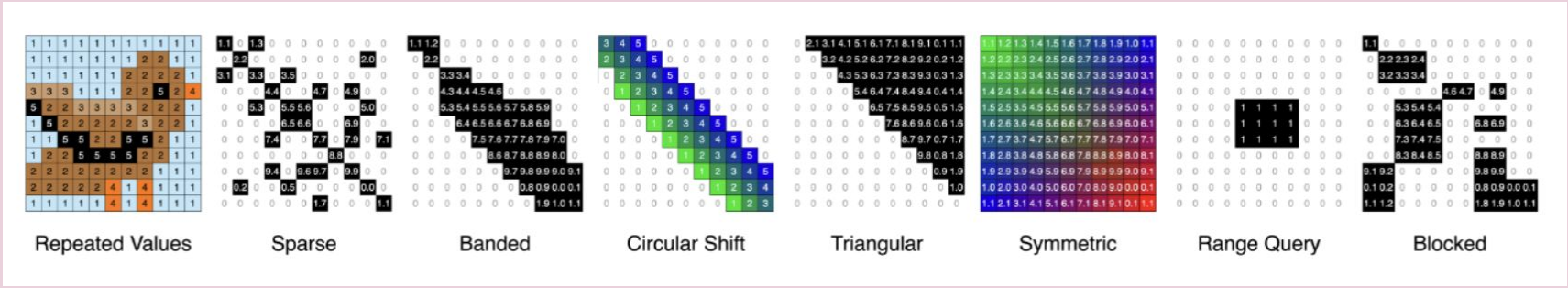


TensorFlow

Ecosystem offers different tensor libraries.

Finch is a compiler for sparse and structured tensors.

Real world tensors often contain underlying structure, such as sparsity, runs of repeated values, or symmetry.



Implementation Burden Grows Exponentially to support all kernels…

Operation

$$C_{ij} = \sum_k A_{ik} * B_{kj}$$

$$y_i = \sum_j A_{ij} x_j$$

$$C_{ij} \mathrel{+}= M_{ij} \sum_k A_{ik} * B_{kj}$$

X

Structure

Sparse
Banded
Symmetric
Blocked

X

Data Format

CSR
COO
Hash
DIA

X

Architecture

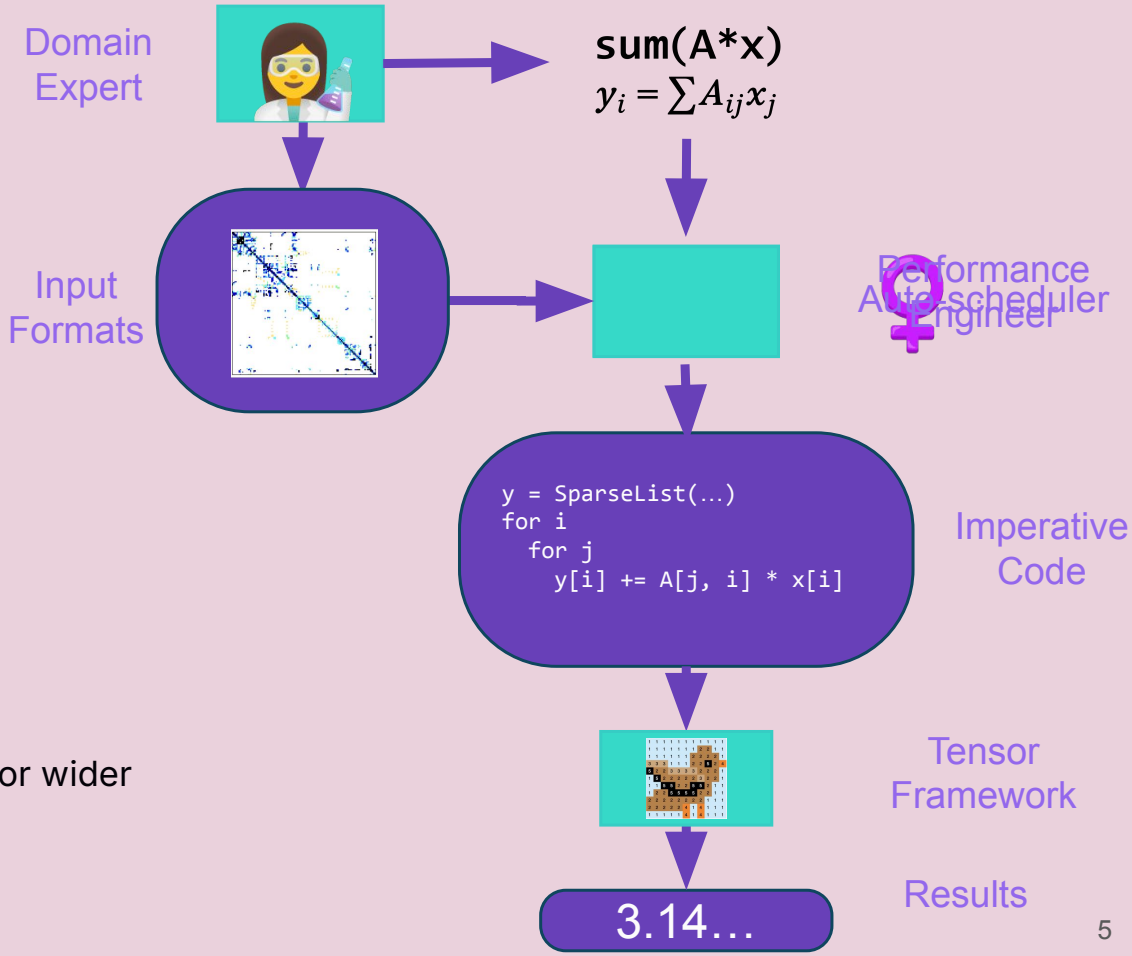
CPU
GPU
TPU
ASIC

$$a = Bc$$
$$A = B \quad a = B^T c$$
$$a = Bc + a \quad a = Bc + b \quad a = b + c$$
$$A_{ij} = \sum_k B_{ijk} * c_k \quad a = B^T c + d \quad a = \alpha Bc + \beta a$$
$$a = BCd \quad A = BC \quad A = 0$$
$$A = (B + C)(d + e) \quad A = B + C \quad A = (B + C)d$$
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A_{ij} = \sum_k B_{ijk} * c_k \quad A = \alpha I \quad A = B + C + D$$
$$A_{ilk} = \sum_j B_{ijk} * C_{lj} \quad A_{il} = \sum_{j,k} B_{ijk} * C_{jl} * D_{kl}$$
$$A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad A_{ik} = \sum_i B_{ijk} * c_j \quad A_{ijl} = \sum_k B_{ijk} * C_{lk}$$
$$A_{ij} = B_{ij}(C_{ik} D_{kj}) \quad A_{ij} = B_{ij} \sum_k (C_{ik} D_{kj}) \quad A_{ij} = \sum_k B_{ijk} C_{ijk} + D_{ij} \quad A_{ij} = \sum_k C_{ijk} D_{ijk} \quad A_{ij} = \sum_{i,k} B_{ilk} * C_{lj} * D_{kj} \quad A_{ij} = \sum_{k,l} B_{ikjl} * C_{kl}$$

Compilers can help!

High-Level Sparse Array Programming in Python

- Domain expert:
  - Writes Numpy Code
  - Writes Array Formats
- Optimization used to be manual
  - Loop Order Choices
  - Format Choices
  - Fusion Choices
- Python/Numpy interface is necessary for wider applicability



Why Finch?

Array API Adoption.

- Allows for faster adoption of new backends in downstream libraries, such as SciPy and scikit-learn.
- Reduces inconsistencies in library APIs.
- Test suite for measuring compliance.

```python
import numpy as np

a, b = np.array(in1), np.array(in2)
res = np.dot(a, b.T) + const
a = np.reshape(a, newshape=shape)
c = np.unique(a, return_counts=True)
```

```python
import sparse

a, b = sparse.as_coo(in1), np.array(in2)
res = a @ b.transpose() + const
a = sparse.reshape(a, shape=shape)
c = sparse.unique_counts(a)
```

Array API

```python
import numpy as np
import jax.numpy as jnp
xp = np

a, b = xp.asarray(in1), xp.asarray(in2)
res = a @ b.mT + xp.asarray(const)
a = xp.reshape(a, shape=shape)
c = xp.unique_counts(a)
```

Why Finch?

Fused Operation

$$C_{ij} = \sum_k A_{ik} * B_{kj}$$

General Sparse Matrix-Matrix Multiplication (SpGEMM)

Without Fusion:

D[i, j, k] = A[i, k] * B[k, j]
C[i, j] = Σ_k D[i, j, k]

Finch uses Tracing JIT

Interface

```
def spgemm(a, b):
    return finch.compute(finch.matmul(finch.defer(a), finch.defer(b)))
```

Trace computation lazily

Traced Computation

Query(Y, Aggregate(add, 0, MapJoin(mul, A[i,k], B[k,j]), (k,)))

Fuse operations

Fused Execution

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

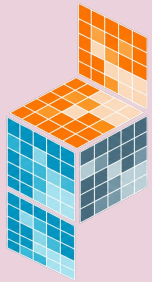
Finch versus sparse libraries

Matricized tensor times Khatri-Rao product (MTTKRP)

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

	Finch	Scipy.sparse	Pydata/sparse
Internals	N-dim	2-dim *	N-dim
Array API Adoption	Yes	No	Yes
Fused Operation	Yes	No	No
Einsum	Yes	No	Yes

* only COO is N-dim in SciPy



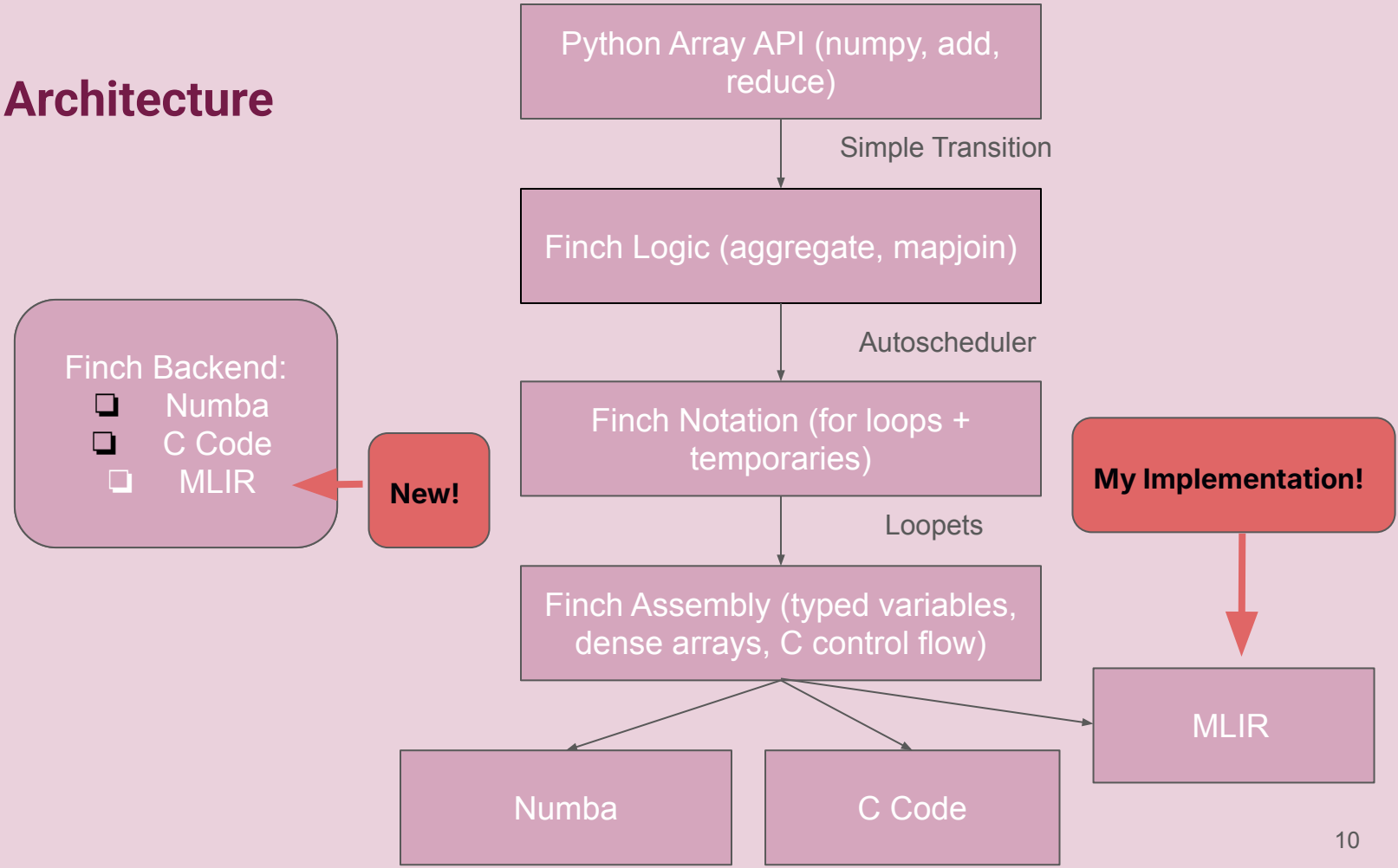
PyData/ sparse

python™

What is MLIR?

- **MLIR (Multi-Level Intermediate Representation) is a compiler infrastructure within the LLVM ecosystem designed to optimize programs at multiple levels of abstraction before lowering them to machine code.**

Finch Architecture



Benefits of MLIR

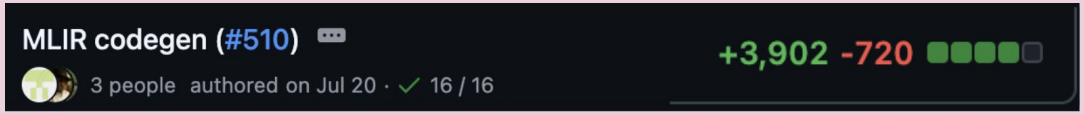
Backend	Possible Hardware Targets
C Code	CPU*
Numba	CPU
MLIR (via python bindings	CPU, GPUs, SIMDs (linalg, vector, gpu)

* C has a CUDA backend that supports GPUs, but it requires a new backend.

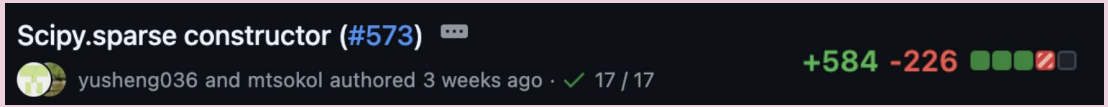
Moving away from Numba

- More novel compiler infrastructure, designed for modern compiler workloads, such as machine learning and high performance computing.
- Dedicated dialects for vectorization to optimise dense operations.
- Dedicated dialects for multi-threading to support parallelism in Finch.
- Better memory ownership - allows zero-copy resizing in MLIR (Numba performs full copy resizing).

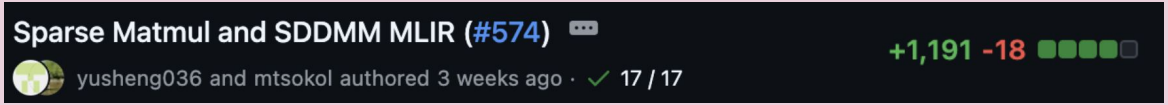
Code Delivery



- Framework of MLIR codegen
- Package mlir-python-bindings via eudsl



- Implement from_scipy to convert Scipy CSR and COO to Finch tensors
- Implement to_scipy with pattern matching detection to scipy arrays



- Adding sparse features

Code Generation Pipeline of Finch Assembly IR

MLIR Operation

Type Handling

Mul on two index-type

```
%v_27 = arith.muli %v_26, %v_24 : index
%v_28 = arith.addi %v_19, %v_27 : index
```

Unique SSA

```python
def mlir_function_name(op, arg: FType) -> str:
    match op:
        case ffuncs.add:
            if MLIROperator.is_float(arg):
                return "arith.addf"
            return "arith.addi"
        case ffuncs.sub:
            if MLIROperator.is_float(arg):
                return "arith.subf"
            return "arith.subi"
        case ffuncs.mul:
            if MLIROperator.is_float(arg):
                return "arith.mulf"
            return "arith.muli"
        case ffuncs.truediv:
            if MLIROperator.is_float(arg):
                return "arith.divf"
            if MLIROperator.is_unsigned(arg):
                return "arith.divui"
            return "arith.divsi"
        case ffuncs.floordiv:
            if MLIROperator.is_float(arg):
                raise NotImplementedError(
                    f"MLIR backend does not yet support floordiv on {arg}"
                )
            if MLIROperator.is_unsigned(arg):
                return "arith.divui"
            return "arith.floordivsi"
        case ffuncs.mod:
            if MLIROperator.is_unsigned(arg):
                return "arith.remui"
            raise NotImplementedError(f"MLIR backend does not yet support mod on {arg}")
        case ffuncs.min:
            if MLIROperator.is_float(arg):
                return "arith.minimumf"
            if MLIROperator.is_unsigned(arg):
                return "arith.minui"
            return "arith.minsi"
```

Unique Operations for Float and Unsigned

Buffers Unpacking

Finch Assembly IR

Level Format

```
#_A_2#34_buf: np_buf_t(float64) = #_A_2#34.lvl.lvl.lvl.val
```

MLIR

SSA Assignment of Buffer

Level Format Indexing

Converting LLVM struct to MLIR memref

```
%v = llvm.extractvalue %_A_15[0, 0, 0, 0] : !llvm.struct<(!llvm.struct<(!llvm.struct<(!llvm.struct<(ptr, p
%v_2 = builtin.unrealized_conversion_cast %v : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)> to memref<?xf64>
```

Finch Assembly IR

Unpack dimension from #A_2

```
#A_2_dim_0#35: int64 = #_A_2#34.lvl.dimension
```

MLIR

Converting i64 value to MLIR index type

```
%v_11 = llvm.extractvalue %_A_16[0, 0, 1] : !ll
%v_12 = arith.index_cast %v_11 : i64 to index
```

Loading Buffers

Finch Assembly IR

Index Position

```
load(slot(#_A_3#38_buf_slot, np_buf_t(float64)), #i_3#43__pos)
```

Buffer Slot

Buffer Type

Finch Assembly IR

Index Position

```
store(slot(#_A_3#38_buf_slot, np_buf_t(float64)), #i_3#43__pos,
```

Buffer Slot

Buffer Type

Buffer index stored in SSA

Buffer type accessed

MLIR

```
%v_51 = memref.load %v_14[%v_46] : memref<?xf64>
```

Storing Buffers

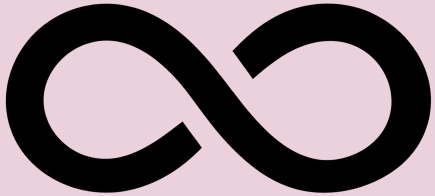
Buffer type stored

MLIR

```
memref.store %v_55, %v_14[%v_46] : memref<?xf64>
```

Value stored at buffer index

For Loops Handling



MLIR code

SSA for start

SSA for length of buffer slot

SSA for step

```
%v_19 = arith.constant 0 : index
%v_20 = memref.dim %v_14, %v_19 : memref<?xf64>
%v_21 = arith.constant 1 : index
scf.for %v_22 = %v_19 to %v_20 step %v_21 {
```

All SSA needs to be similar type (i.e index type)

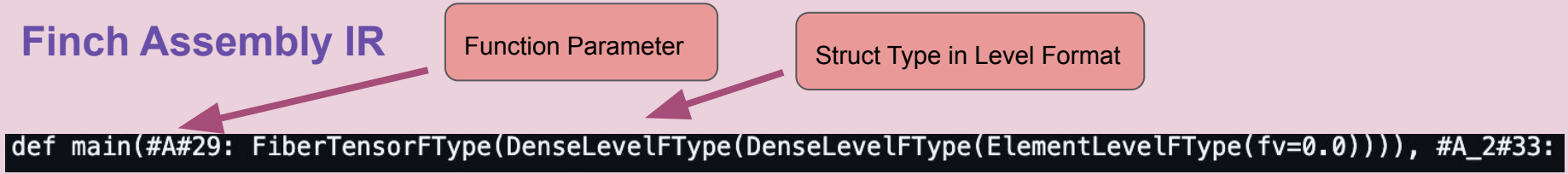
Finch Assembly IR

```
for i in range(0, length(slot(#_A_3#38_buf_slot, np_buf_t(float64)))):
```

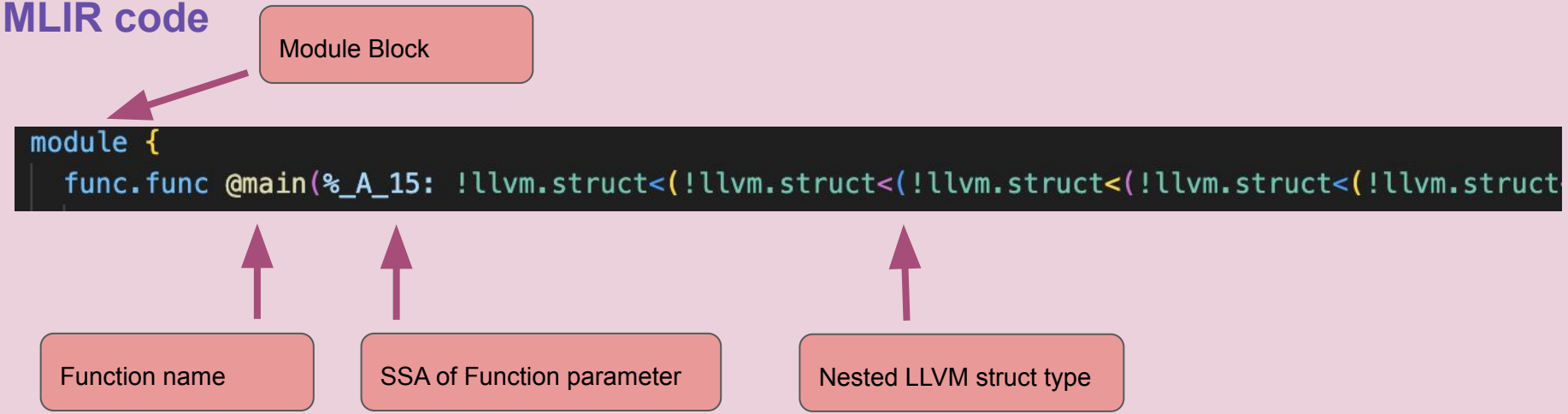
Length of the Buffer Slot

Function Definition and Module

Finch Assembly IR



MLIR code



Return Statement

Creates a tuple containing the BufferizedNDArray descriptor.

Finch Assembly IR

```
main_return: tuple(BufferizedNDArray_f64_shape_i64_i64_strides_i64_i64) = make_tuple(#A_3#37)
return main_return
```

Return the tuple

SSA of struct with fields populated

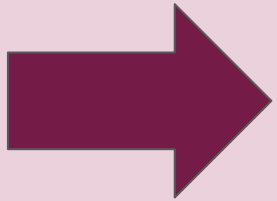
MLIR code

Creates an undefined LLVM struct type

```
%v_56 = llvm.mlir.undef : !llvm.struct<(!llvm.struct<(!llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>,
%v_57 = llvm.insertvalue %_A_7, %v_56[0] : !llvm.struct<(!llvm.struct<(!llvm.struct<(ptr, ptr, i64, array<1 x i64>,
llvm.store %v_57, %_ret : !llvm.struct<(!llvm.struct<(!llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>,
func.return
```

Store struct type in %_ret pointer from function parameter

Code showcase



If Else Handling

If Condition

```
if lt(load(slot(#_A#15_idx_slot, np_buf_t(int64)), q), 0):
    q: int64 = scansearch(slot(#_A#15_idx_slot, np_buf_t(int64)), 0, q, sub(q_stop, 1))
```

Finch Assembly IR

SSA for if Condition

Return SSA

MLIR code

Else statement

```
%v_82 = memref.load %v_16[%v_73] : memref<?xindex>
%v_83 = arith.cmpi slt, %v_82, %v_27 : index
%v_86 = scf.if %v_83 -> (index) {
  %v_84 = arith.subi %v_75, %v_29 : index
  %v_85 = func.call @scansearch(%v_16, %v_27, %v_73, %v_84) : (memref<?xindex>, index, index, index) -> index
  scf.yield %v_85 : index
} else {
  scf.yield %v_73 : index
}
```

Return type

Return the SSA containing the final value

Return its original value

While Loop Handling

Finch Assembly IR

While Condition

```
while lt(min(i_stop, i_stop_2), min(min(#A_2_dim_1#21, add(i_last, 1)), add(i_last_2, 1))):
```

MLIR code

SSA Reassignment

Initial types

Return types

SSA Return Variables

```
%v_130:3 = scf.while (%v_55 = %v_27, %v_56 = %v_54, %v_57 = %v_49) : (index, index, index) -> (index, index, index) {
  %v_58 = arith.addi %v_48, %v_29 : index
  %v_59 = arith.minsi %v_10, %v_58 : index
  %v_60 = arith.cmpi slt, %v_57, %v_59 : index
  scf.condition(%v_60) %v_55, %v_56, %v_57 : index, index, index
} do {
  ^bb_2(%v_55: index, %v_56: index, %v_57: index):
```

While Condition

```
  scf.yield %v_125, %v_126, %v_129 : index, index, index
}
```

Do Block

Return the SSA containing the final value

```
%v_130#0
%v_130#1
%v_130#2
```

SSA for return values after while loop

Condition Region

Finch Assembly string	(SSA, mlir_type)
0	(%v_27, index)

```
%v_27 = arith.constant 0 : index
```

```
%v_130:3 = scf.while (%v_55 = %v_27, %v_56 = %v_54, %v_57 = %v_49) : (index, index, index) -> (index, index, index) {
  %v_58 = arith.addi %v_48, %v_29 : index
  %v_59 = arith.minsi %v_10, %v_58 : index
  %v_60 = arith.cmpi slt, %v_57, %v_59 : index
  scf.condition(%v_60) %v_55, %v_56, %v_57 : index, index, index
} do {
  ^bb_2(%v_55: index, %v_56: index, %v_57: index):
```

```
  scf.yield %v_125, %v_126, %v_129 : index, index, index
}
```

Dataflow Analysis

Condition Region

Finch Assembly string	(SSA, mlir_type)
0	(%v_27, index)
0	(%v_55, index)

```
%v_27 = arith.constant 0 : index

%v_130:3 = scf.while (%v_55 = %v_27, %v_56 = %v_54, %v_57 = %v_49) : (index, index, index) -> (index, index, index) {
  %v_58 = arith.addi %v_48, %v_29 : index
  %v_59 = arith.minsi %v_10, %v_58 : index
  %v_60 = arith.cmpi slt, %v_57, %v_59 : index
  scf.condition(%v_60) %v_55, %v_56, %v_57 : index, index, index
} do {
  ^bb_2(%v_55: index, %v_56: index, %v_57: index):

  scf.yield %v_125, %v_126, %v_129 : index, index, index
}
```

Dataflow Analysis

Body Region

Finch Assembly string	(SSA, mlir_type)
0	(%v_55, index)

Condition Region

Finch Assembly string	(SSA, mlir_type)
0	(%v_27, index)
0	(%v_55, index)

```
%v_27 = arith.constant 0 : index
```

```
%v_130:3 = scf.while (%v_55 = %v_27, %v_56 = %v_54, %v_57 = %v_49) : (index, index, index) -> (index, index, index) {
  %v_58 = arith.addi %v_48, %v_29 : index
  %v_59 = arith.minsi %v_10, %v_58 : index
  %v_60 = arith.cmpi slt, %v_57, %v_59 : index
  scf.condition(%v_60) %v_55, %v_56, %v_57 : index, index, index
} do {
  ^bb_2(%v_55: index, %v_56: index, %v_57: index):
```

```
  scf.yield %v_125, %v_126, %v_129 : index, index, index
}
```

Dataflow Analysis

Body Region

Finch Assembly string	(SSA, mlir_type)
0	(%v_55, index)
new_val	(%v_125, index)

Condition Region

Finch Assembly string	(SSA, mlir_type)
0	(%v_27, index)
0	(%v_55, index)

```
%v_27 = arith.constant 0 : index
```

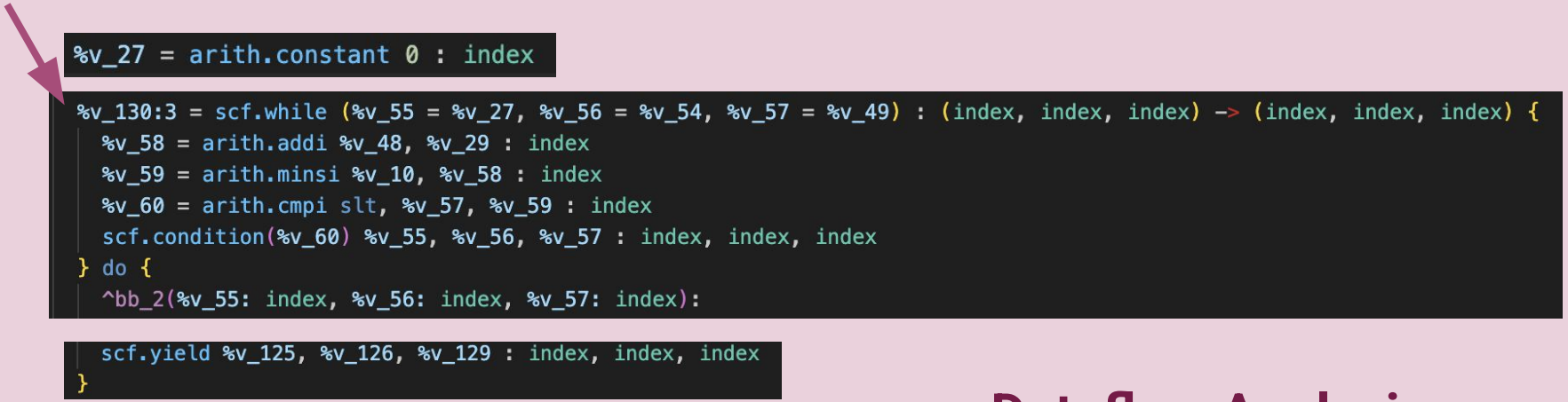
```
%v_130:3 = scf.while (%v_55 = %v_27, %v_56 = %v_54, %v_57 = %v_49) : (index, index, index) -> (index, index, index) {
  %v_58 = arith.addi %v_48, %v_29 : index
  %v_59 = arith.minsi %v_10, %v_58 : index
  %v_60 = arith.cmpi slt, %v_57, %v_59 : index
  scf.condition(%v_60) %v_55, %v_56, %v_57 : index, index, index
} do {
  ^bb_2(%v_55: index, %v_56: index, %v_57: index):
```

```
  scf.yield %v_125, %v_126, %v_129 : index, index, index
}
```

Dataflow Analysis

Condition Region

Finch Assembly string	(SSA, mlir_type)
0	(%v_27, index)
0	(%v_55, index)
new_val	(%v_130#0, index)



Dataflow Analysis

Function Calling Convention

Finch Assembly IR

Scansearch Parameter

```
q_2: int64 = scansearch(slot(#_A_2#19_idx_slot, np_buf_t(int64)), 0, q_2, sub(q_stop_2, 1))
```

Return Type

MLIR code

SSA called by the function

```
%v_85 = func.call @scansearch(%v_16, %v_27, %v_73, %v_84) : (memref<?xindex>, index, index, index) -> index
```

SSA for the result

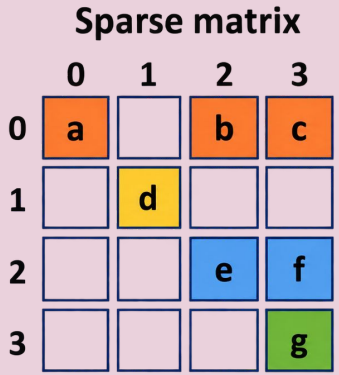
Parameter Type

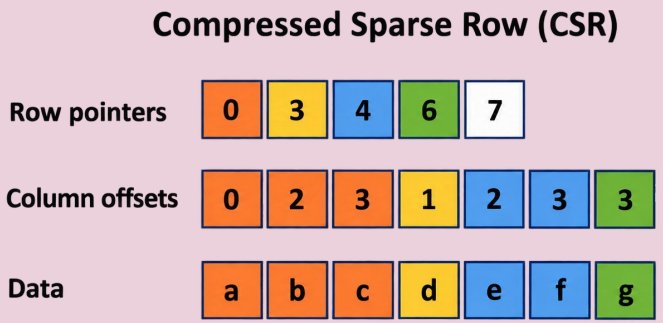
Return Type

Benchmark

Hardware

- Apple M4 chip

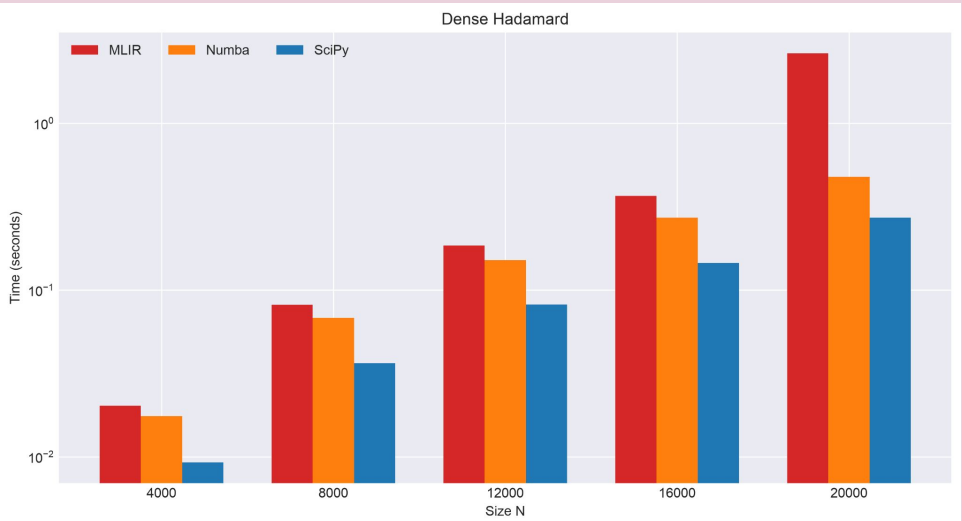




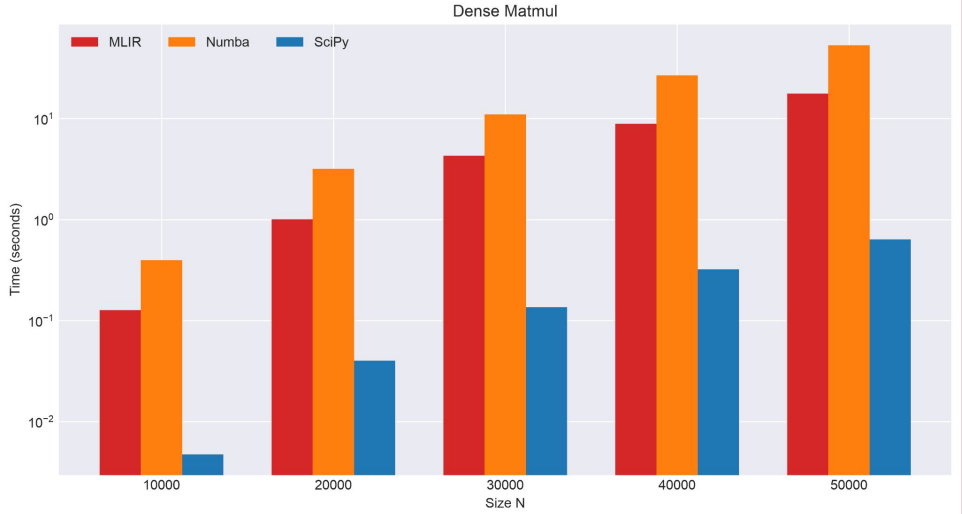
Benchmark Setup

- Finch has precompile run
- Sparse formats are CSR

Dense Hadamard



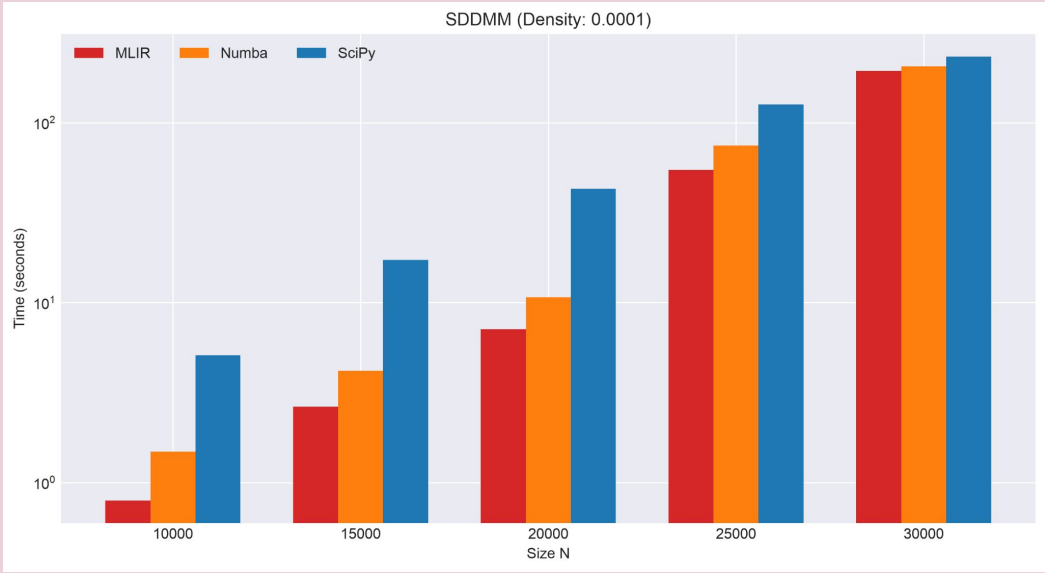
Dense Matmul



Machine Learning: Sampled Dense-Dense Matrix Product

```
@fuse
def sddmm(mask, a, b):
    return finch.compute(
        finch.multiply(
            finch.defer(mask),
            finch.matmul(finch.defer(a), finch.defer(b)),
        )
    )
```

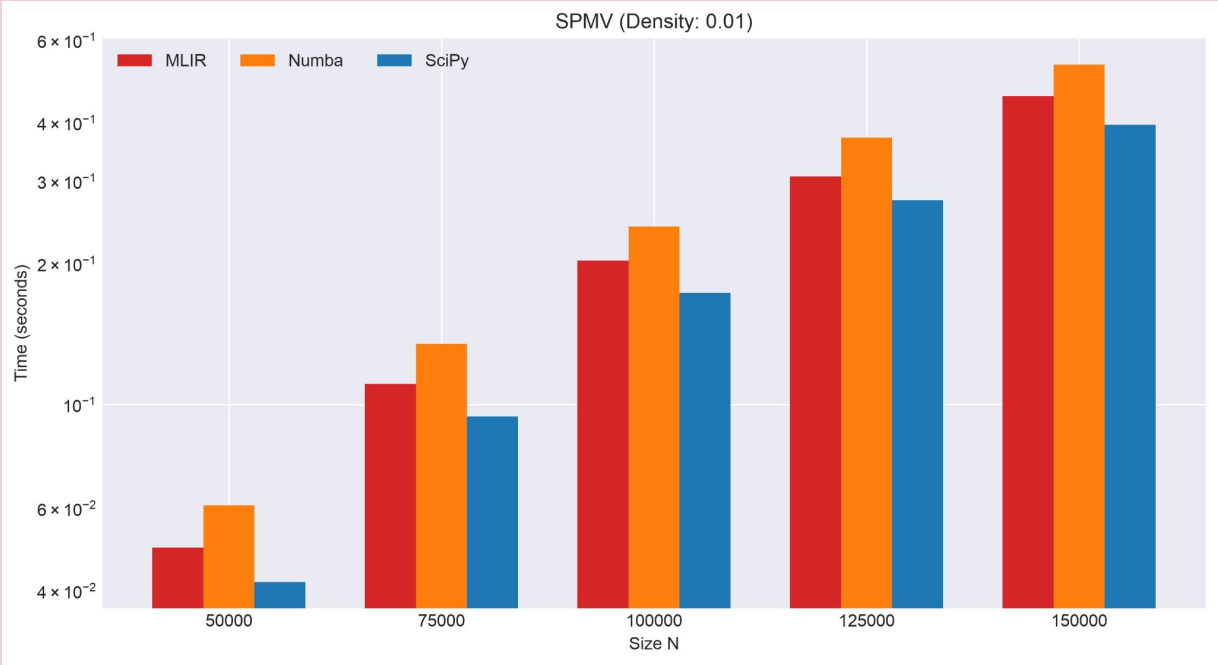
```
def scipy_sddmm(mask, a, b):
    return mask.multiply(a @ b)
```

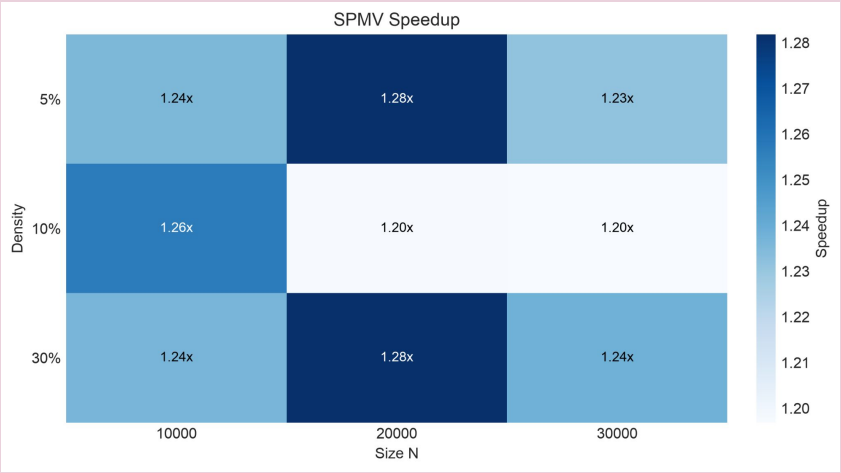


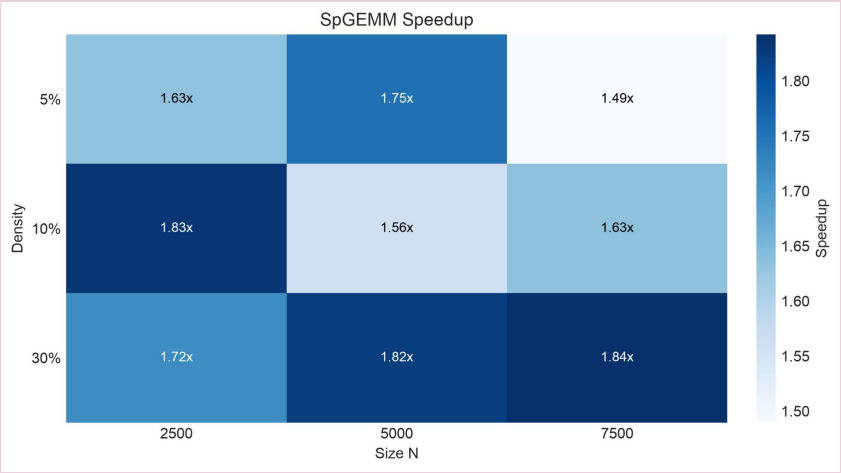
Scientific Computing: Sparse Matrix-Vector Multiplication

```
@fuse
def spmv(a, b):
    return finch.compute(
        finch.matmul(
            finch.defer(a),
            finch.defer(b)
        )
    )
```

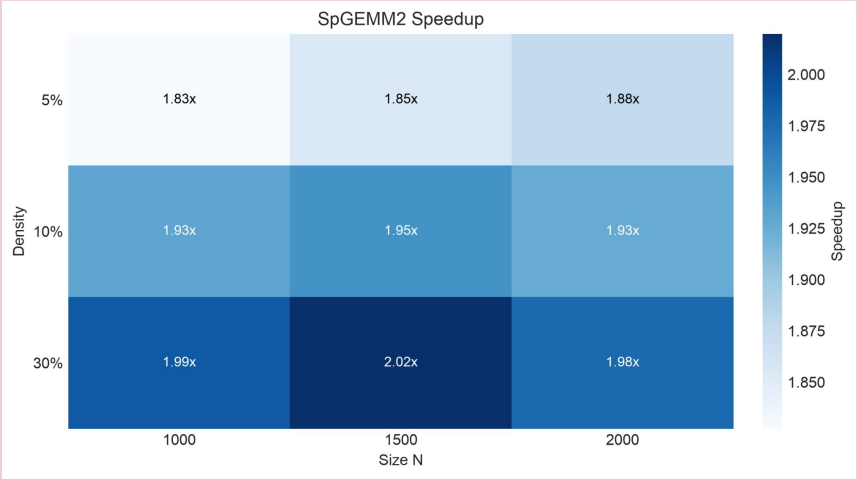
```
def scipy_spmv(a, b):
    return a @ b
```







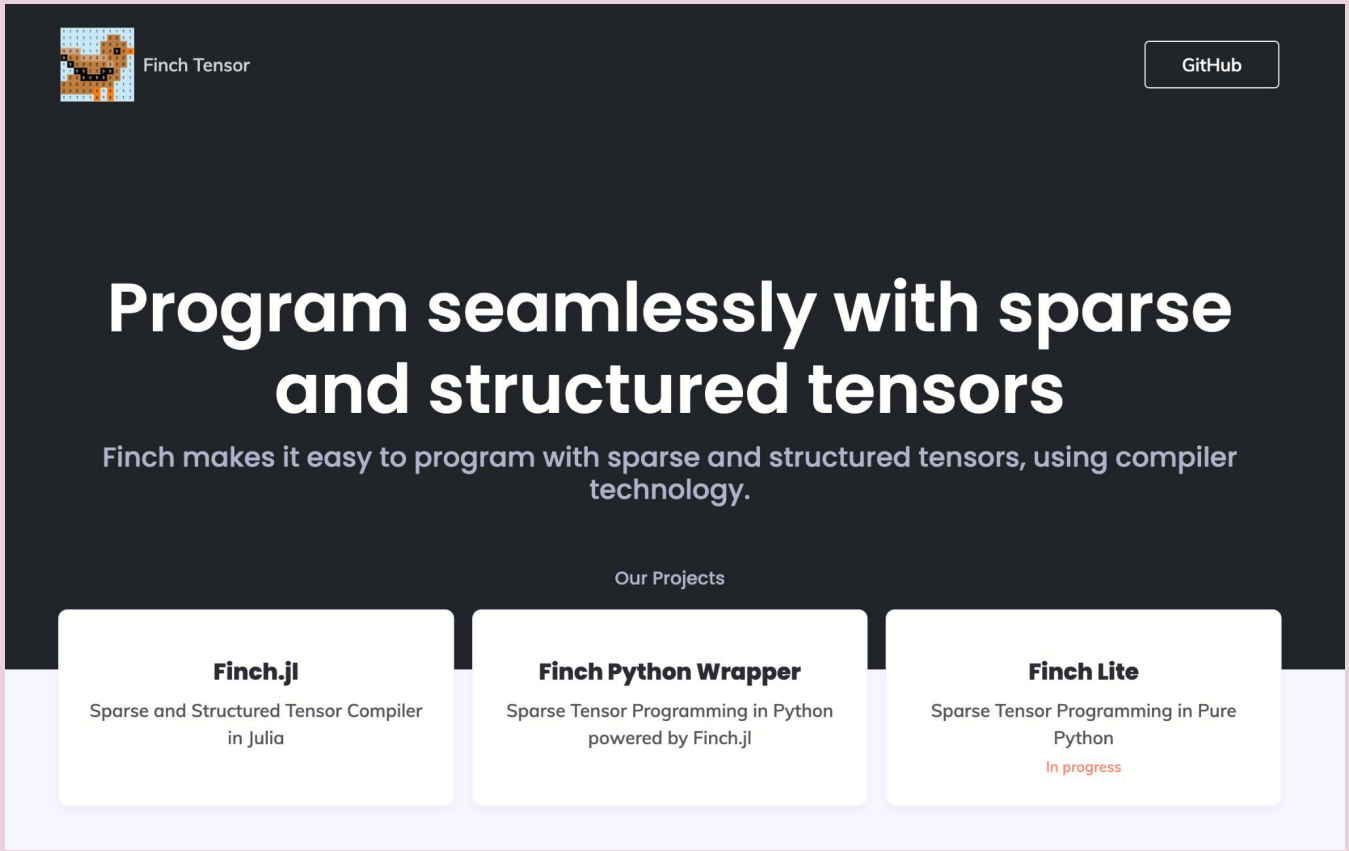
Benchmark against Numba backend



SPGEMM: (a @ b)

SPGEMM2: (a @ b) @ c

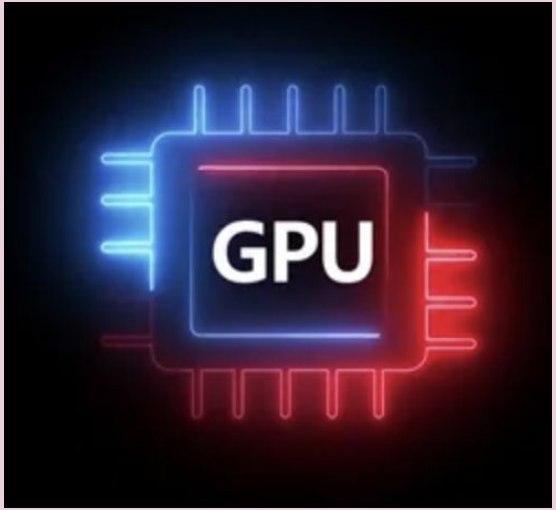
Check out the website



https://finch-tensor.org/

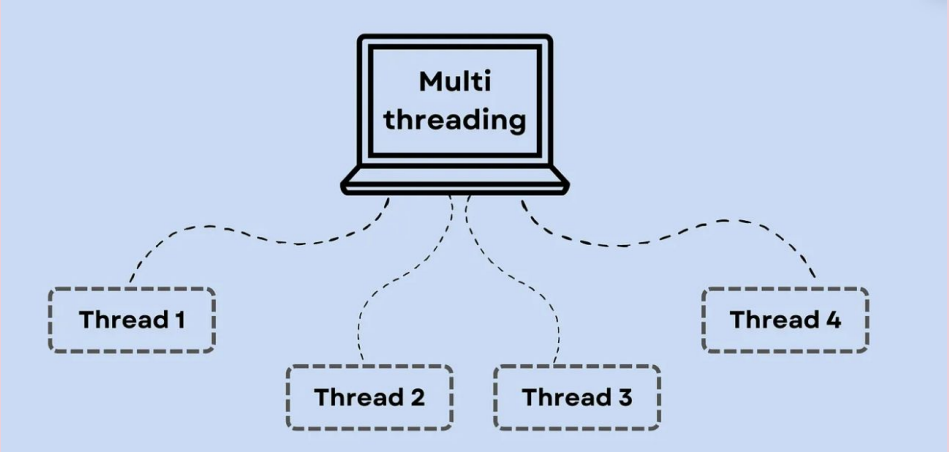
Direction enabled by MLIR

- Extend MLIR support to GPUs.



Feature Completeness Needed

- Multi-threading and vectorization
- Scalar support for MLIR
- Linalg dialect
- Memory management with Resize



Thank you!