CompTN

A Compiler Infrastructure for High-Performance Tensor Network Computing

Francisco Kusch Domínguez
Advisors: Nathaniel Tornow, Oğuzcan Kırmemiş
Chair of Computer Systems
https://dse.in.tum.de/

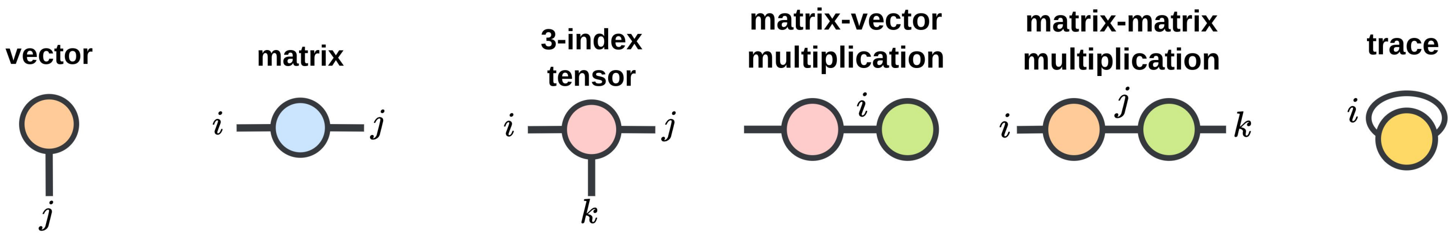
28.05.2024 - 28.09.2024

Tensor Network

- **Definition:** Tensor networks are a set of tensors connected by contractions, in a graph-like structure.
- **Einstein Notation**
  - Leave out the summation

$$c_j = \sum_i A_{ji} \cdot b_i \longrightarrow c_j = A_{ji} \cdot b_i$$

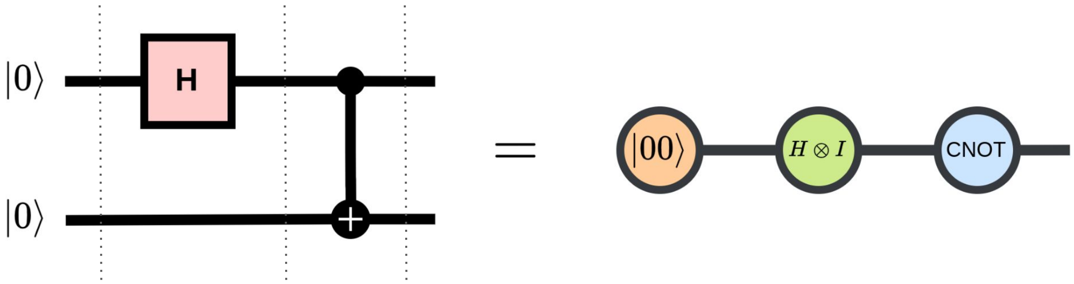
- **Graphical Notation**
  - Rank of the tensor is represented by number of outgoing edges
  - Connecting two tensors with an edge, represents the contraction along that index



Tensor Network

Applications

- Quantum computing: Efficient simulation of quantum systems

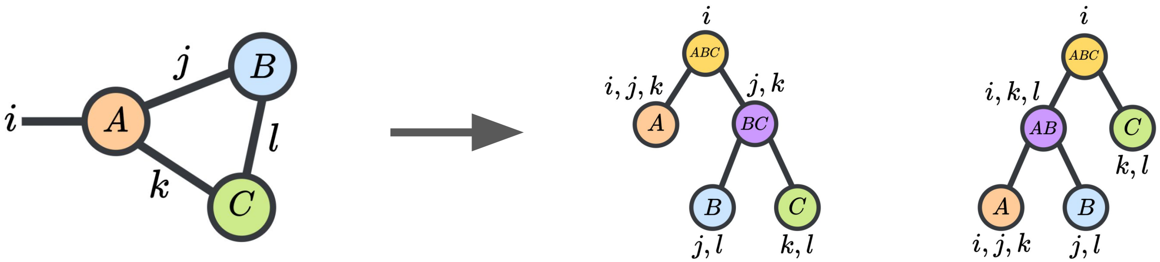


- Machine learning: Tensor networks used for dimensionality reduction and efficiency
- Chemistry and physics: Modeling complex systems and interactions

Challenge

Finding the optimal **contraction path** for tensors is **NP-hard**

- Requires **approximations and heuristics** to manage computational complexity and resources
- Contraction **path makes difference,** because of the different sizes of the **intermediate tensors**
- **Indices and shapes** can be determined **statically**



Research gap

State-of-the-art (cotengra, NumPy, opt_einsum) are **libraries**

- Library-based methods **lack whole-program optimizations**
- Use generic routines
- Do **not** target **specific hardware**
- Inefficient to use in JIT-compiled programs

Limitations of compiled approaches

- Compiler-based methods are **difficult to develop** and maintain, often being too rigid and **complex**

⇒ Opportunity for **Compiled Solution** with MLIR: There is a gap in combining the **flexibility of libraries** with the efficiency and **optimization power of compilers**

System and design goals

Extensibility

- Should accommodate **new optimizations**, target different **hardware**, and extend to **other fields**

Programmability

- Control what optimizations are applied
- Take away the need to fine-tune configurations for specific computation

Performance

- Improve performance through optimizations like tensor slicing, rank simplification, and contraction pathfinding

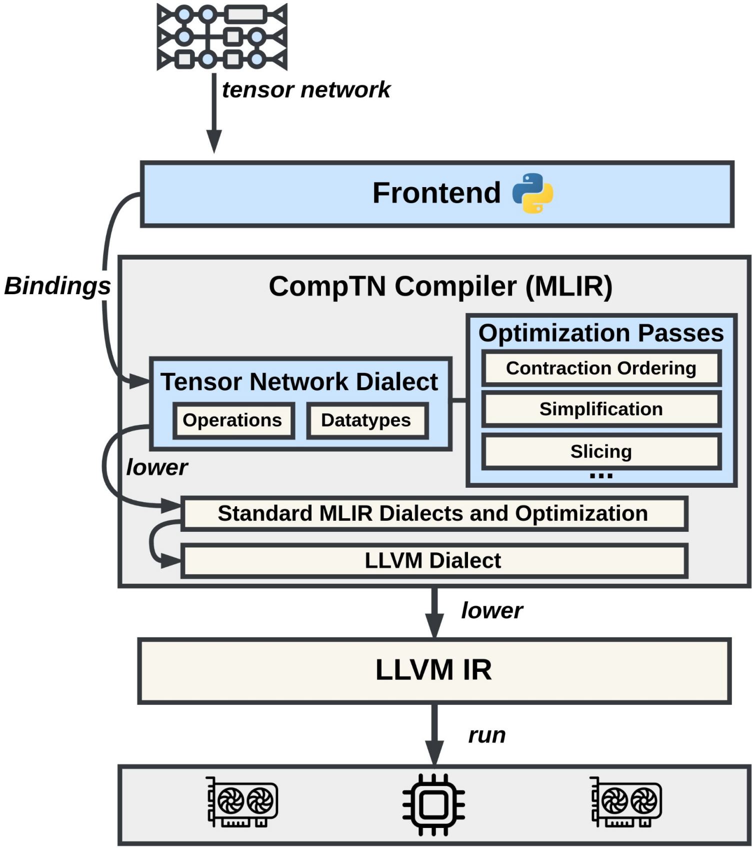
System Overview

Contributions

- Custom Python bindings allowing JIT-compilation
- Tensor Network dialect
- Optimization and lowering passes for TN dialect

We use

- Optimization and lowering passes for standard dialects
- Execution engine to retrieve computation result



Tensor Network Dialect

Operations

- *index*
- *tensor* (→arith.constant)
- *contract* (→linalg.generic)
- *contract_multiple*
- *add* (→linalg.generic)

Types

- !tensor_with_indices (→!builtin.tensor)
  - Tensor shape
  - Index list
- !index_label
  - Dimension size

```python
import tensor_network_ext as tn
import numpy as np

# Initialize the module
mm = tn.ModuleManager()

# Define the indices
indexI = mm.Index(6)
indexJ = mm.Index(6)
indexK = mm.Index(6)
indexL = mm.Index(6)

# Define the tensor values
array1 = np.random.rand(6, 6, 6)
array2 = np.random.rand(6, 6)
array3 = np.random.rand(6, 6)

# Define the tensors, with the corresponding indices
tensorA = mm.Tensor(array1, indexI, indexJ, indexK)
tensorB = mm.Tensor(array2, indexJ, indexL)
tensorC = mm.Tensor(array3, indexK, indexL)

# Contract the tensors
mm.contract_multiple(tensorA, tensorB, tensorC)

# Compile and run the contraction
result = mm.run()

# Manipulate the result
print("CompTN Result: " + str(result))
```

MLIR Module

```
module {
  func.func @main() attributes {llvm.emit_c_interface} {
    %0 = "tensor_network.index"() <{name = "index_0", size = 6 : i64}> : () -> !tensor_network.indexlabel<6 : i64, "index_0">
    %1 = "tensor_network.index"() <{name = "index_1", size = 6 : i64}> : () -> !tensor_network.indexlabel<6 : i64, "index_1">
    %2 = "tensor_network.index"() <{name = "index_2", size = 6 : i64}> : () -> !tensor_network.indexlabel<6 : i64, "index_2">
    %3 = "tensor_network.index"() <{name = "index_3", size = 6 : i64}> : () -> !tensor_network.indexlabel<6 : i64, "index_3">
    %4 = "tensor_network.tensor"(%0, %1, %2) <{value = dense< %% Values %% > : tensor<6x6x6xf64>}> : (!tensor_network.indexlabel<6 :
i64, "index_0">, !tensor_network.indexlabel<6 : i64, "index_1">, !tensor_network.indexlabel<6 : i64, "index_2">) ->
!tensor_network.tensor_with_indices<tensor<6x6x6xf64>, [!tensor_network.indexlabel<6 : i64, "index_0">, !tensor_network.indexlabel<6 :
i64, "index_1">, !tensor_network.indexlabel<6 : i64, "index_2">]>
    %5 = "tensor_network.tensor"(%1, %3) <{value = dense< %% Values %% > : tensor<6x6xf64>}> : (!tensor_network.indexlabel<6 : i64,
"index_1">, !tensor_network.indexlabel<6 : i64, "index_3">) -> !tensor_network.tensor_with_indices<tensor<6x6xf64>,
[!tensor_network.indexlabel<6 : i64, "index_1">, !tensor_network.indexlabel<6 : i64, "index_3">]>
    %6 = "tensor_network.tensor"(%2, %3) <{value = dense< %% Values %% > : tensor<6x6xf64>}> : (!tensor_network.indexlabel<6 : i64,
"index_2">, !tensor_network.indexlabel<6 : i64, "index_3">) -> !tensor_network.tensor_with_indices<tensor<6x6xf64>,
[!tensor_network.indexlabel<6 : i64, "index_2">, !tensor_network.indexlabel<6 : i64, "index_3">]>
    %7 = "tensor_network.contract_multiple"(%4, %5, %6) : (!tensor_network.tensor_with_indices<tensor<6x6x6xf64>,
[!tensor_network.indexlabel<6 : i64, "">, !tensor_network.indexlabel<6 : i64, "index_1">, !tensor_network.indexlabel<6 : i64,
"index_2">]>, !tensor_network.tensor_with_indices<tensor<6x6xf64>, [!tensor_network.indexlabel<6 : i64, "index_1">,
!tensor_network.indexlabel<6 : i64, "index_3">]>, !tensor_network.tensor_with_indices<tensor<6x6xf64>, [!tensor_network.indexlabel<6 :
i64, "index_2">, !tensor_network.indexlabel<6 : i64, "index_3">]>) -> !tensor_network.tensor_with_indices<tensor<6xf64>,
[!tensor_network.indexlabel<6 : i64, "index_0">]
    return
  }
}
```

Passes

Optimization

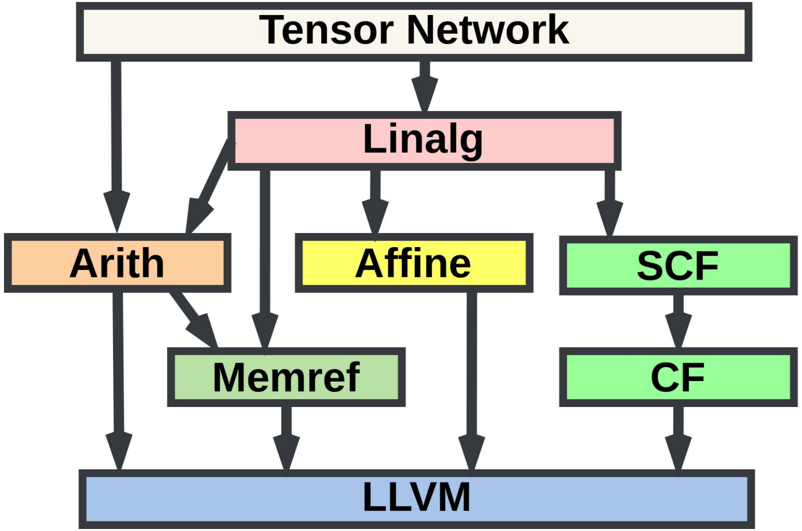
- Rank simplification
- Tensor Network Slicing

Path search

- Greedy with FLOP heuristic
- Greedy with Gray-Kourtis heuristic

Lowering

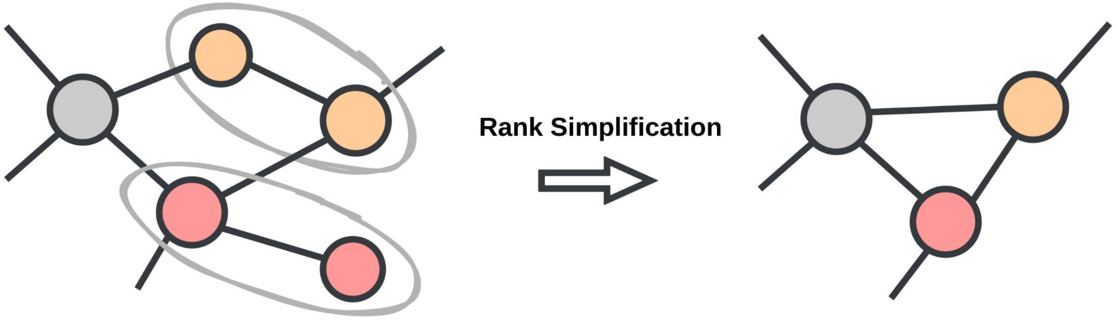
- Eliminate indices
- Lower to *Linalg* and *Arith* dialects



Rank-Simplification

Idea: Contract trivial tensors with their neighbors, before path-search

- **Reduce** number of tensors in the search
- Tensors of rank 1 and 2 do **not increase** rank of intermediate tensors

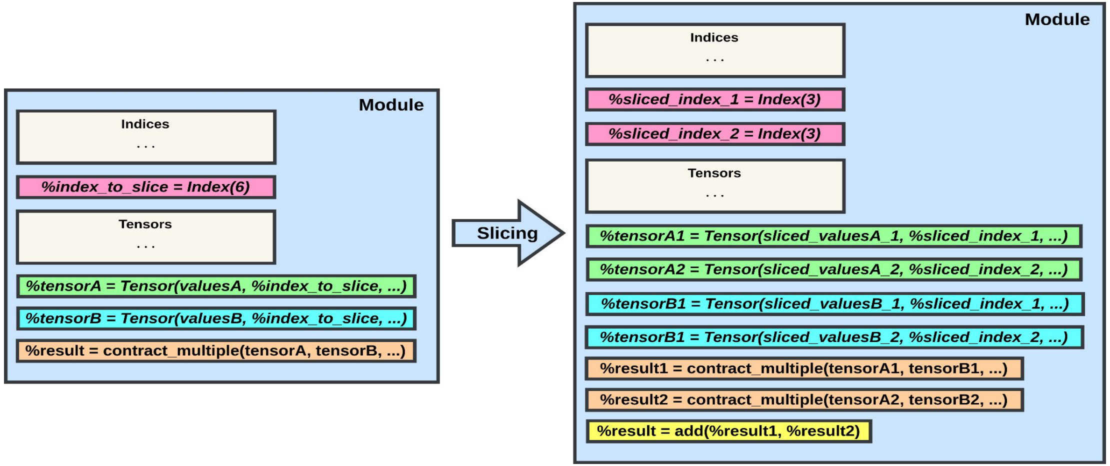


Tensor Network Slicing

Idea: Divide the tensor network in **multiple pieces,** by separating the summation

- Pieces can be computed as a **separate tensor networks,** and added afterwards
- **Reduces the size** of the intermediate tensors

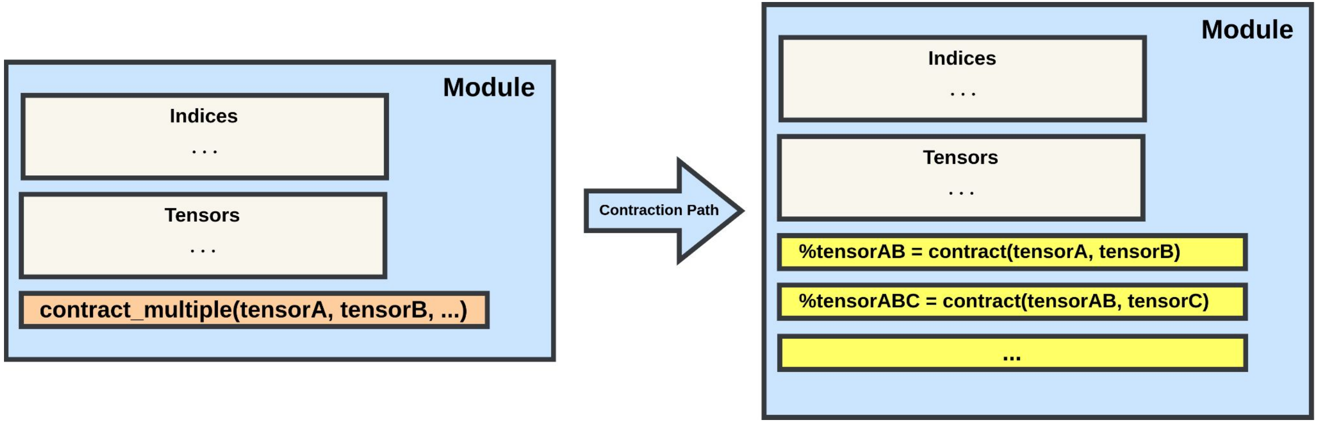
$$\sum_{j=1}^{6}\sum_{i,k,l} A_{i,j,k} \cdot B_{j,l} \cdot C_{k,l} = [\sum_{j=1}^{3}\sum_{i,k,l} A_{i,j,k} \cdot B_{j,l} \cdot C_{k,l}] + [\sum_{j=4}^{6}\sum_{i,k,l} A_{i,j,k} \cdot B_{j,l} \cdot C_{k,l}]$$



Path-finding

Idea: Abstract the actual path away through *contract_multiple*

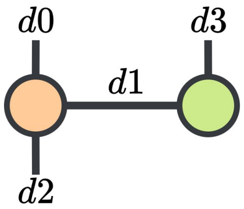
- Path is determined after optimizations by **rewriting** *contract_multiple* into a series of *contract* operations
- Result **shape and indices** can be determined **statically**



Lowering the contraction

Idea: Use the **index information** to determine **access patterns** for the contraction

- Loops and operations created during the lowering of *linalg.generic*
  - Only need to determine access patterns and operations
- Contraction has 2 input and 1 output tensors
  - Input: Access on all their dimensions
  - Output: Access the indices that are not shared




```
// Input tensors
affine_map<(d0, d1, d2, d3) -> (d0, d1, d2)>
affine_map<(d0, d1, d2, d3) -> (d1, d3)>

//Output tensor
affine_map<(d0, d1, d2, d3) -> (d0, d2, d3)>
```

Lowering the contraction

Idea: Use the **index information** to determine **access patterns** for the contraction

- Loops and operations created during the lowering of *linalg.generic*
  - Only need to determine access patterns and operations
- Contraction has 2 input and 1 output tensors
  - Input: Access on all their dimensions
  - Output: Access the indices that are not shared

```
%0 = linalg.generic {indexing_maps = [affine_map<(d0, d1, d2, d3) -> (d0, d1, d2)>, affine_map<(d0, d1, d2, d3) -> (d1, d3)>,
affine_map<(d0, d1, d2, d3) -> (d0, d2, d3)>], iterator_types = ["parallel", "reduction", "parallel", "parallel"]} ins(%cst,
%cst_3 : tensor<6x6x6xf64>, tensor<6x6xf64>) outs(%cst_5 : tensor<6x6x6xf64>) {
    ^bb0(%in: f64, %in_7: f64, %out: f64):
      %2 = arith.mulf %in, %in_7 : f64
      %3 = arith.addf %2, %out : f64
      linalg.yield %3 : f64
    } -> tensor<6x6x6xf64>
```

Evaluation

Questions

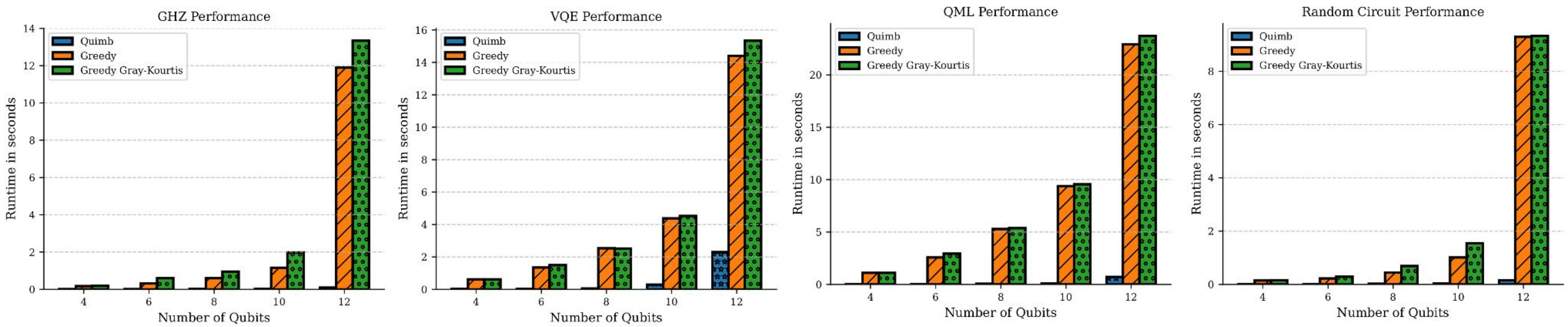
- How does the **end-to-end runtime** of CompTN **compare** to state-of-the-art?
- How is the **end-to-end runtime** divided?

Benchmarks

- Greenberger–Horne–Zeilinger State (GHZ)
- Variational Quantum Eigensolver (VQE)
- Quantum Machine Learning (QML)
- Random circuits

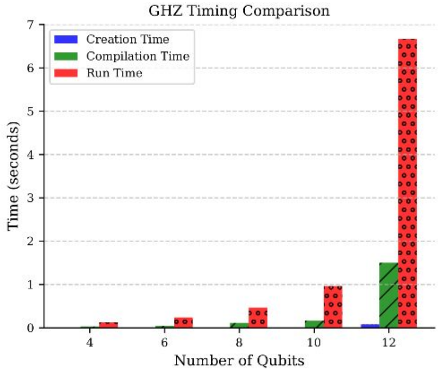
End-to-end runtime

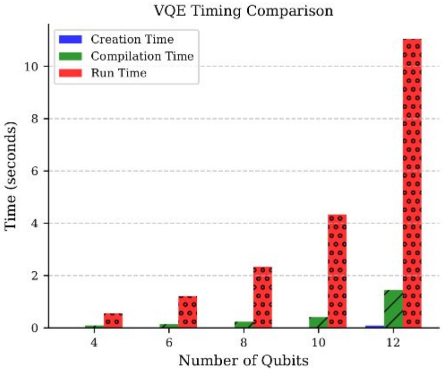
- *Quimb*: state-of-the-art python library for quantum simulation using tensor networks
- *CompTN* still "just" proof-of-concept:
  - More optimizations needed to compete with state-of-the-art

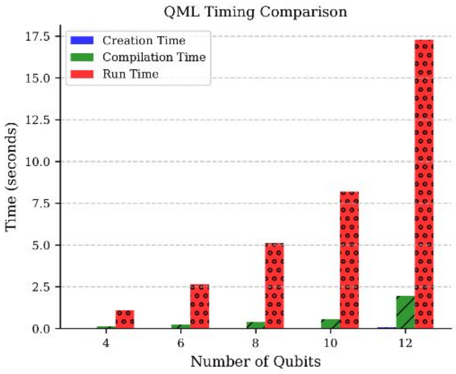


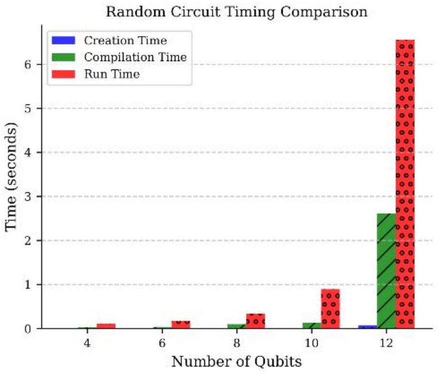
Time distribution

- **No considerable impact** of using **Python** to create the module
  - Profit from using **Python control flow structures** to built up complex tensor networks
  - Make use of ***CompTN*** to optimize the actual **tensor network contraction**
- Still room for **more optimizations** during **compile-time**









Future work

- **Multithreading** through slicing
- Target **GPU** through GPU dialect
- **Sparse tensor** optimizations
- **Transposition (GEMM)** to align index accesses
- **Sampling** of the tensor network to tune **hyperparameters**

Conclusion

Motivation

- State-of-the-art library approaches lack whole-program optimizations
- Inefficient to use in JIT-compiled program

Conclusion

- Designed **compiler-based approach** while keeping **flexibility** and **modularity**
  - Python Bindings for JIT-compilation
  - Tensor Network dialect
  - Lowering and optimization passes
- **Extensibility** of MLIR allows to easily build on top of ***CompTN***
  - *Future focus on optimizations to compete with state-of-the-art*

Try it out!

github.com/PanchoK50/tensor-network-compiler