SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

CompTN: A Compiler Infrastructure for High-Performance Tensor Network Computing

Francisco Kusch Domínguez

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

CompTN: A Compiler Infrastructure for High-Performance Tensor Network Computing

CompTN: Eine Compiler-Infrastruktur für Hochleistungs-Tensor-Netzwerk Berechnungen

Author:	Francisco Kusch Domínguez
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Nathaniel Tornow, Oğuzcan Kırmemiş
Submission Date:	28. September 2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 28. September 2024 Francisco Kusch Domínguez

Acknowledgments

I would like to thank Prof. Pramod Bhatotia for giving me the opportunity to work on this exciting and challenging project. I would also like to thank both my advisors Nathaniel Tornow and Oğuzcan Kırmemiş, who have kindly guided me through it and given me their valuable time during our weekly meetings.

Abstract

Tensor Networks are mathematical structures representing the contraction of multiple tensors in a graph-like form. They are crucial in various fields, including machine learning, physics, chemistry, and quantum computing. However, finding optimal contraction paths is NP-hard, and contractions can require exponential time and memory, necessitating approximations.

State-of-the-art solutions like NumPy, opt_einsum, and Cotengra use library approaches, offering flexibility but limiting whole-program optimizations. Compiler-based approaches enable such optimizations and low-level targeting, but are typically more complex to develop.

We present CompTN, a compiler infrastructure for tensor network contractions using MLIR (Multi-Level Intermediate Representation), a compiler framework by Google focused on extensibility and programmability. CompTN aims to bridge the gap between library and compiler approaches, providing compiler optimizations while maintaining programmability and flexibility.

The contributions of this work are a new MLIR Tensor Network dialect, a set of optimization and lowering passes, and Python bindings to MLIR, allowing the JIT-compilation of tensor network contractions. Due to the extensibility of CompTN on the backend (MLIR) and frontend (Python), it can easily be expanded with new optimizations and specific use cases like quantum computing.

While our current implementation doesn't yet outperform state-of-the-art libraries, it demonstrates a proof-of-concept for an extensible compilation infrastructure for tensor network contractions. This approach holds promise for future performance gains through compiler optimizations, potentially offering a powerful alternative to existing library-based solutions.

Kurzfassung

Tensornetzwerke sind mathematische Strukturen, die die Kontraktion mehrerer Tensoren in einer graphähnlichen Form darstellen. Sie sind in verschiedenen Bereichen von entscheidender Bedeutung, darunter maschinelles Lernen, Physik, Chemie und Quantencomputer. Das Finden optimaler Kontraktionspfade ist jedoch NP-schwer, und Kontraktionen können exponentielle Zeit und Speicherplatz erfordern, so dass Näherungen notwendig sind.

Modernste Lösungen wie NumPy, opt_einsum und Cotengra verwenden Bibliotheksansätze, die zwar Flexibilität bieten, aber Optimierungen des gesamten Programms einschränken. Compiler-basierte Ansätze ermöglichen solche Optimierungen und Low-Level-Targeting, sind aber in der Regel komplexer zu entwickeln.

Wir stellen CompTN vor, eine Compiler-Infrastruktur für Tensornetz-Kontraktionen unter Verwendung von MLIR (Multi-Level Intermediate Representation), einem Compiler-Framework von Google mit Schwerpunkt auf Erweiterbarkeit und Programmierbarkeit. CompTN zielt darauf ab, die Lücke zwischen Bibliotheks- und Compiler-Ansätzen zu überbrücken, indem es Compiler-Optimierungen unter Beibehaltung der Programmierbarkeit bietet.

Die Beiträge dieser Arbeit sind ein neuer MLIR-Tensornetzwerk-Dialekt, eine Reihe von Optimierungs- und Senkungspässen und Python-Bindungen an MLIR, die die JIT-Kompilierung von Tensornetzwerk-Kontraktionen ermöglichen. Aufgrund der Erweiterbarkeit von CompTN auf dem Backend (MLIR) und Frontend (Python) kann es leicht mit neuen Optimierungen und für spezielle Anwendungsfälle wie Quantencomputing erweitert werden.

Auch wenn unsere derzeitige Implementierung noch nicht die Leistung von State-of-the-Art-Bibliotheken übertrifft, so zeigt sie doch ein Proof-of-Concept für eine erweiterbare Kompilierungsinfrastruktur für Tensornetzkontraktionen. Dieser Ansatz verspricht zukünftige Leistungssteigerungen durch Compiler-Optimierungen und könnte eine leistungsstarke Alternative zu bestehenden bibliotheksbasierten Lösungen darstellen.

Contents

1 Introduction

A tensor network is a collection of tensors connected by contractions in a graph-like structure [BB17]. Its applications range in a variety of fields, like quantum computing, chemistry, physics, and machine learning. The computation of these tensor networks is resource-intensive, which is why efficient solutions are needed in this field.

An efficient infrastructure offers multiple advantages: it can reduce the **computational complexity** of calculations, enabling the simulation of larger and more complex systems, while also minimizing **memory requirements**. This approach not only allows for the handling of larger tensor networks but also enhances computational **accuracy** by reducing the need for approximations due to hardware limitations.

Two primary approaches exist for implementing these systems: compilation and library-based methods. Designing a library for the computation of tensor networks is usually simpler and allows faster prototyping. It allows defining a set of generic routines and the choice of where to optimize and how to make the computation is taken during runtime. Because of this, library approaches often are not optimized for the specific use case and do not necessarily target specific hardware. A compiled approach, on the other hand, is generally much more complex and less flexible but it allows for optimizations that make use of the whole program, as well as properties of the specific tensor network that is being computed.

To illustrate the library approach, we can consider Quimb, a state-of-the-art Python library for quantum circuit simulations. Quimb exemplifies the advantages and limitations of library-based methods by using cotengra for tensor network contractions [GK21]. It incorporates methods such as tree sampling to tune the meta-parameters for the specific tensor network and dynamic slicing for memory efficiency and parallelism. These optimizations are executed during runtime and could potentially be shifted into compile time, which could also allow a series of other optimizations like path-reuse and target-specific low-level optimizations.

The main idea of this thesis is to present a design for a Tensor Network Compiler using MLIR, a compiler framework developed at Google, which could potentially address some of the downsides of compiled approaches. Leveraging MLIR's pre-implemented compiler components can significantly reduce the complexity of designing and implementing a compiler. MLIR's structure allows CompTN to be easily extensible, accommodating new optimization passes, operations, and types, while facilitating targeting of multiple hardware platforms.

The main contributions of this work are:

- **Tensor Network Dialect:** A specialized dialect including **operations** and custom **types** to easily represent Tensor Networks, based on the usual literature notation.
- **Optimization and Lowering Passes:** A set of passes operating at the Tensor Network Dialect level, designed to **enhance performance** and allow the usage of optimization passes at lower abstraction levels.
- **Python Bindings:** A set of Python interfaces to easily compute tensor network contractions using **Just-In-Time (JIT) compilation**, bridging high-level ease of use with low-level performance.

The evaluation done in Chapter 6, shows that there is still space for optimizations for CompTN to compete with the state-of-the-art libraries. Nonetheless, with this work, we present a first proof-of-concept for an extensible and programmable tensor network compiler.

2 Background and Motivation

The optimizations and design choices made for the Tensor Network Compiler are rooted, in an understanding of tensor network structure and some of their mathematical properties, and the operational details of MLIR.

2.1 Tensors and Tensor Networks

In this section, we are going to clarify some properties and concepts about tensor networks, the mathematical structure at the core of CompTN. We will approach this topic from the bottom up. First, we will explain what tensors are and how they are contracted. Then, we will examine tensor networks and their mathematical and graphical notations. Finally, we will explore some specific properties that can be leveraged to optimize tensor network computations.

2.1.1 Tensors

Essentially, tensors are a generalization for scalars, vectors, and matrices. They can be thought of as a collection of numbers that can be accessed via indices. The minimal amount of indices needed to fully determine a number is called the order of the tensor. For instance, a vector is a tensor of first order, since you would need exactly 1 index to determine an entry. For a Matrix you would need 2 indices, therefore it is generalized to a second-order tensor. [Ran+20]

2.1.2 Reshaping tensors

Depending on the need, a tensor can be reshaped to have a different order. For instance, reshaping a tensor into a vector involves defining an index ordering and creating a new single index from their combinations. This ordering preserves the relationship between the tensor's original indices and the vector's entries, allowing bidirectional mapping between them.

This concept is also useful for storing tensors in memory, which can only be accessed among one dimension. For this, the tensor must be reshaped into a vector.

Reshaping tensors also allows us to use common vector and matrix operations on them. For example, we can apply Singular Value Decomposition (SVD) to a reshaped tensor.

The following third-order tensor serves to illustrate the concept of reshaping further. It has dimensions $2 \times 2 \times 2$ and entries C_{ijk}:

$$C_{000} = 1,\ C_{001} = 2,\ C_{010} = 3,\ C_{011} = 4,$$

$$C_{100} = 5,\ C_{101} = 6,\ C_{110} = 7,\ C_{111} = 8$$

In this example, by grouping the first $n - 1 = 2$ indices of the $n = 3$ order tensor, the tensor is reshaped into a $2^{n-1} \times 2$ matrix. The dimensions of the matrix, are determined by the number of states that each index can have. With the creation of a new index $i' = i * 2 + j$, the resulting matrix with entries $M_{i'k}$ is:

$$M_{00} = C_{000} = 1,\ M_{01} = C_{001} = 2,$$

$$M_{10} = C_{010} = 3,\ M_{11} = C_{011} = 4,$$

$$M_{20} = C_{100} = 5,\ M_{21} = C_{101} = 6,$$

$$M_{30} = C_{110} = 7,\ M_{31} = C_{111} = 8$$

In theory, reshaping does not necessarily affect the tensor in memory, but can be viewed as just a reinterpretation of the information. However, in practice, when doing a reshaping for a specific algorithm, data locality is very important. The tensor needs to be transposed in such a way, that the access to the tensor leads to a minimum of cache misses.

2.1.3 Tensor Contractions

Tensor contractions represent the higher-order equivalent of matrix multiplications, in which you contract over a given set of indices [Tia+21].

In classical matrix multiplication, the dimension, or index, that is summed over is already implicitly defined in the mathematical conventions. Namely, the second index of the first tensor and the first index of the second tensor, which is why those dimensions must match to be able to multiply them. With explicit indices, a matrix multiplication would look like this:

$$C_{i,k} = \sum_j A_{i,j} \cdot B_{j,k}$$

For tensor contractions, you have to explicitly specify the indices over which you are contracting, which can also be more than one, given that their dimension sizes match. An example of a tensor contraction in mathematical notation is given below:

$$F_{i,l,m} = \sum_j \sum_k D_{i,j,k} \cdot E_{j,k,l,m}$$

The rank of the tensors may also be different. In this case, the tensor D has rank 3 and E has rank 4. In the end, the resulting tensor contains all the dimensions, that the contraction did not sum over, represented by their respective indices. There is even the possibility of contracting more than 2 tensors at a time, which just follows the same principle.

GEMM

When contracting two tensors computationally one must remember, the layout in which the tensor is stored in memory. As discussed earlier, to store a tensor in memory it must be reshaped into a vector (rank 1), since memory only allows access along one index. For this, an ordering of the indices must be determined. If the summation index is on the higher side of the ordering, it can happen that values that are along that specific index are not close to one another in memory, causing lots of cache misses when iterating over it.

A good way to avoid this is to use a General Matrix Multiplication (GEMM) for the contraction. For this, the two tensors are first transposed to matrices and then a matrix multiplication is performed. This transposition changes the ordering of the dimensions, such that the specified index is now the lowest in the ordering, meaning their values are next to each other in memory. However, this leads to overhead, due to the transposing of the two input tensors to matrices and the output matrix, back to a tensor. When implemented correctly, this overhead can be much less than executing a naive nested loop. [Tia+21]

2.1.4 Tensor Networks

The contraction of multiple tensors is called a tensor network [Ran+20]. They have gained importance in the fields of quantum information science, condensed matter physics, mathematics, and computer science. For instance, quantum circuits are a special class of tensor networks, with restricted tensor arrangement and types. [BB17] The ability to manipulate such tensor networks in a way that allows computing results in reasonable time and space is very valuable. For this reason, it makes sense to observe and analyze some of the mathematical properties of tensor networks, to enable computational optimizations.

2.1.5 Notation

Writing out tensor contractions can be repetitive, which is why there is a simpler notation, which makes use of the fact, that when summing over a specific index, we write this index in both tensors.

Einstein Notation

The Einstein notation for tensor contractions is identical to the previously introduced formula, but leaving out the summation symbols, since they are already implicit. The previous example in Einstein Notation would look like this:

$$F_{i,l,m} = D_{i,j,k} \cdot E_{j,k,l,m}$$

This idea, that the common indices imply a summation over this index when contracting the tensors, is the basis of the Tensor Network dialect, which will be discussed further in this work.

The Einstein Notation can also be used to represent tensor networks and not only single tensor contractions. For this one simply writes out the tensors, and uses a common index name for all the dimensions, that two tensors are contracted over.

$$D_i = A_{i,j,k} \cdot B_{j,l} \cdot C_{k,l}$$

In the above example, one can see that the tensor A shares the index j with tensor B, B and C share index l. Index k is shared by A and C. The resulting Tensor D simply has the remaining dimension, that was not shared by any tensor, and therefore not contracted over.

To compute the result, one would first compute the contraction of one pair of tensors and then contract the intermediate result with the remaining tensor. The choice of which pair of tensors to first contract, is arbitrary because of the associative property of tensor contractions.

$$D_i = [(A_{i,j,k} \cdot B_{j,l}) \cdot C_{k,l}] = [A_{i,j,k} \cdot (B_{j,l} \cdot C_{k,l})]$$

Nevertheless, this choice can have a considerable impact on how much space and time is needed to run the computation. This is one of the main challenges that CompTN aims to solve.

Graphical Notation

Another tool for representing tensors and tensor networks is the tensor diagram notation, proposed by Roger Penrose [Pen71]. A tensor is represented by a shape, and the number of lines going out of the shape represents its order. A generic tensor is usually drawn as a circle, but the shape chosen often also serves to clarify some specific properties of the tensor. For example, isometric tensors as triangles, unitary tensors as rectangles, and diagonal tensors are shapes with a diagonal line through them.

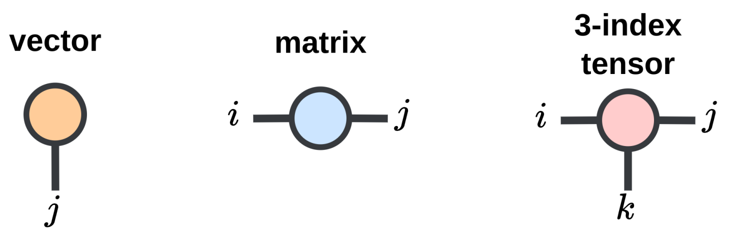


Figure 2.1: Examples of graphical notation for single tensors [Sto24]

In Figure 2.1 you can see, for example, that a vector, which has only 1 dimension is drawn as a circle with a single line, whilst a matrix which has 2 dimensions is drawn as a circle with 2 lines. The same can be extended to tensors with 3 indices and more.

This notation is not only good for easily describing the rank of tensors but also allows for the representation of arbitrary operations between tensors. Since each line corresponds to a specific index of the tensors, connecting two tensors with a line, represents a shared index that is summed over, just like in the Einstein Notation.

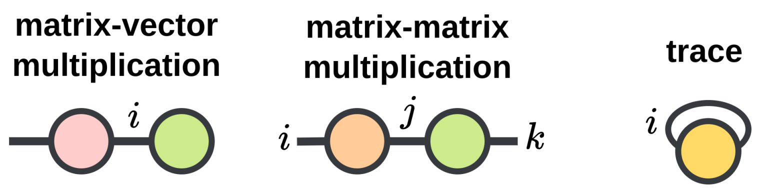


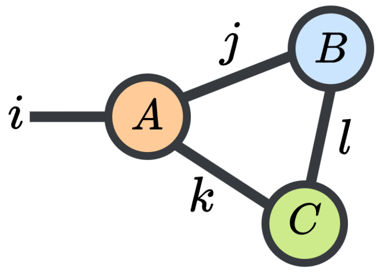
Figure 2.2: Examples of graphical notation for common tensor operations [Pen71]

As you can see in Figure 2.2 a simple matrix-vector multiplication sums over the index i, which is why the edge representing i is connecting the two tensors. The same can be said for matrix-matrix multiplication. For visualization, here is the formula of a matrix-vector multiplication:

$$\sum_i A_{j,i} \cdot v_i$$

The trace is a good example to show how the graphical notation is also useful for determining the rank of the output tensor of the operation. As we saw before, in single tensors the number of outgoing indices indicates the rank of the tensor. For operations, one simply observes the number of free indices of all the tensors involved, which means indices that do not connect to the other tensors in the operation. For the matrix-matrix multiplication, one can intuitively see that this is correct since the indices i and k are free, meaning the result is also a matrix, which only makes sense. For the trace, one can see that the input is a matrix since it has two outgoing edges, but the number of free indices is 0 which means that the result must be a scalar.

When multiple tensors are involved, this notation allows us to quickly get an overview of the topology of a tensor network, as well as the order of the tensor resulting from the contraction. This can be determined by just looking at the number of overall free indices.



$$= A_{i,j,k} \cdot B_{j,l} \cdot C_{k,l}$$

Figure 2.3: Tensor Network Graphical Notation

In Figure 2.3 you can see the same example, that was used to explain the Einstein Notation one more time in graphical notation. From this graph, we can extract, that the result must be a tensor of rank 1, because there is only one free index, namely i.

2.1.6 Quantum Circuits

Quantum circuits are a perfect example of the application of the graphical notation for tensor networks. As noted before, quantum circuits are nothing more than tensor networks with special arrangement and type restrictions [BB17]. In Figure 2.4 you can see an example of how the tensor network representation looks for a simple circuit containing one Hadamard gate on the first qubit followed by a CNOT gate with the first qubit as control and the second qubit as target.

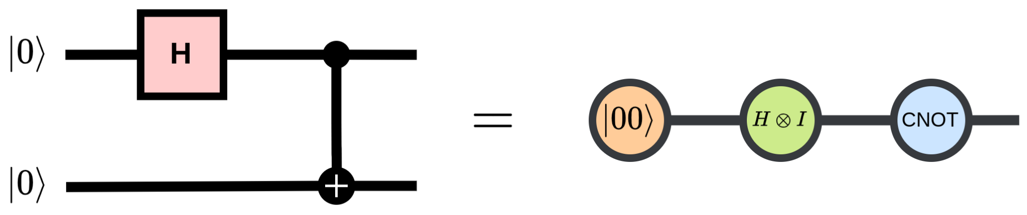


Figure 2.4: Quantum circuits can be generalized to tensor networks

2.1.7 Contraction Trees

First, the contraction of an arbitrary tensor network is not always possible without approximations. This is because some contractions require exponential memory, which becomes a problem even at a low number of Tensors. [Ran+17]

Contraction Trees provide a visual way of specifying the ordering in which tensors in a tensor network are contracted and which indices are still available after each contraction. This also allows for visualization of the intermediate tensors that arise when computing the tensor network. There can be multiple contraction trees for the same tensor network, and the choice of which one to execute is a key challenge of CompTN.

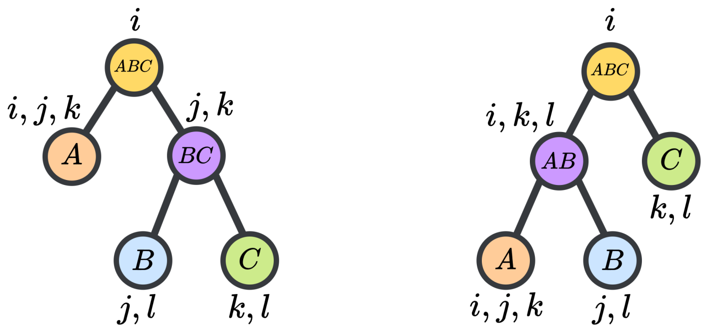


Figure 2.5: Two possible contraction trees for the example tensor network

In Figure 2.5 you can see two possible contraction trees for the tensor network we have been using as an example. On the left, you can see that we first contract the two tensors of rank 2, and then the resulting tensor with A, which is of rank 3. On the right, we first contract A and B into another rank 3 tensor and then perform another contraction with C. Both contraction trees deliver the same result, because of the associative property of tensor networks, but they have different intermediate tensors, which affects the necessary memory and total FLOP count.

To visualize this, you can see how the FLOP count scales with the index size in Figure 2.6. In this case, the left contraction tree performs much better than the right one, reaching a factor of 50 times less FLOPs at index size 100.

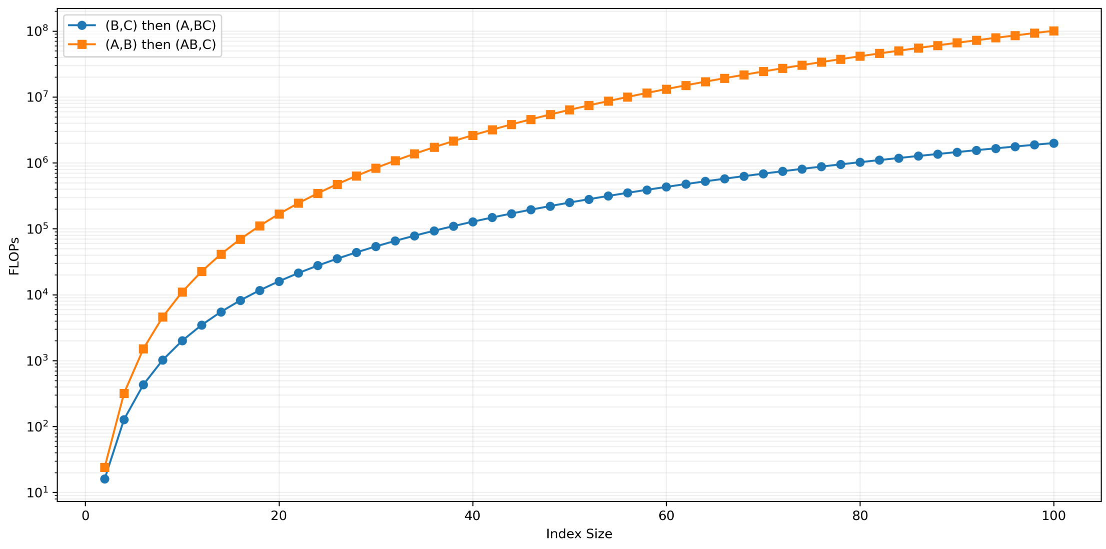


Figure 2.6: FLOPs vs index size for both contraction trees

Intuitively, one can explain this in the following way: The number of operations that a tensor contraction takes is proportional to the number of unique indices from both input tensors. This is because performing a tensor contraction requires iterating over all the available indices, and the operations performed are always a multiplication of the two values at the respective indices and an addition to the correct value in the result tensor.

On one hand, contracting A and B requires iterating over 4 indices (i, i, k, l), and the contraction of AB with C another 3 indices (i, k, l). On the other hand, first contracting B and C requires iterating over 3 indices (j, k, l) and contracting BC with A another 3 indices (i, j, k). In other words, one is "carrying" more indices along during the computation in the left, whilst the other option first contracts the smaller two tensors and in the end performs the contraction with A.

2.1.8 Tensor Network Slicing

As we have seen, Tensor Networks are essentially just a series of tensor contractions, which under the associative principle can be contracted in any arbitrary order. Because of this, one can divide the computation of a Tensor Network into two separate networks and then add their results at the end to obtain the original result. This procedure is called "Slicing".

Taking the example from above, one could for instance slice the tensor network through the index j. Assuming the size of that dimension is 6, one could slice the index by half by doing the following:

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

Of course, one can slice the index in as many parts as the size allows. If one decides to rename the index, one can consider the two parts, as completely different tensor networks, which could be computed in parallel.

Another advantage that slicing has, is that it can reduce the size of the intermediate tensors of the computation.

2.2 MLIR - Multi-Level Intermediate Representation

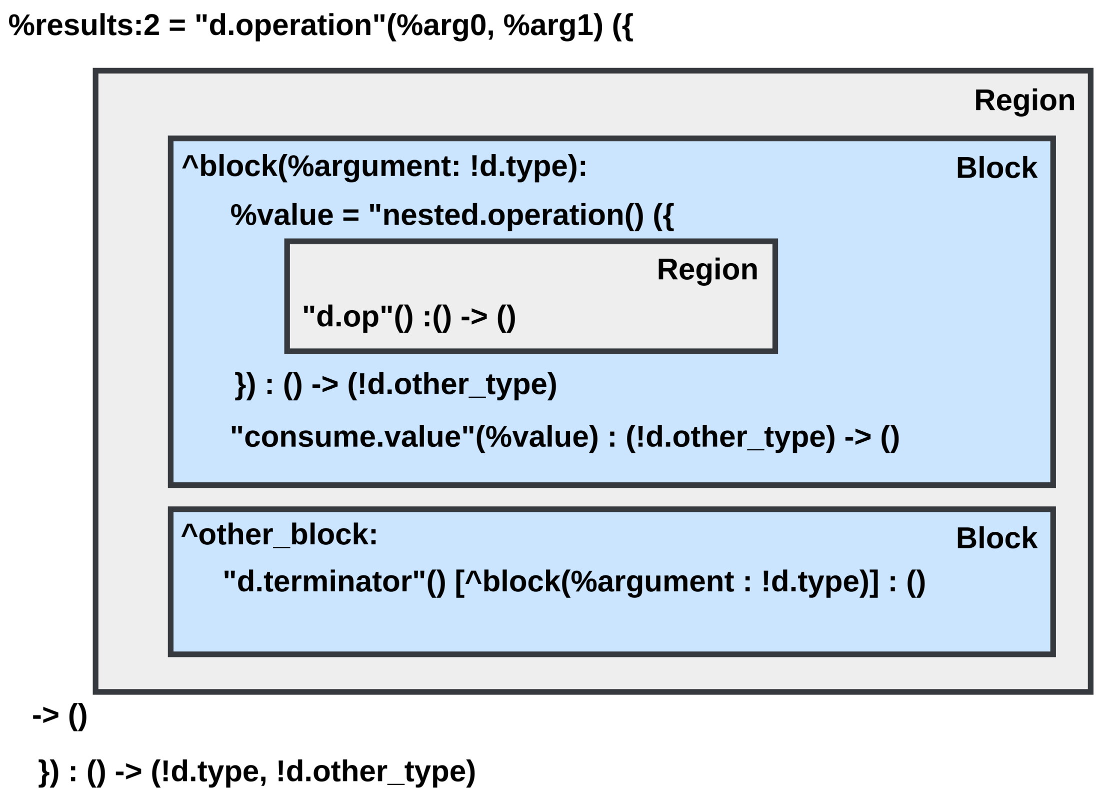
The MLIR project is, as expressed on their website, a "novel approach to building reusable and extensible compiler infrastructure" [MLIb]. It was developed at Google under the guidance of Chris Lattner, who has also worked on projects like LLVM, Clang, LLDB, Swift, TensorFlow, and more recently Mojo.

Previously, domain-specific compilers had to implement multiple components from scratch, that have already been built in some other form for other compilers. This includes pass management, diagnostics, multi-threading, serialization and deserialization, and many others.

As a solution, MLIR generalizes some of those components, allowing their reuse and extension to fit the needs of the users. At the center of MLIR are dialects, which correspond to a set of operations, types, and passes. The user can define their dialect or make use of already existing ones. This is the aspect that makes MLIR extensible, since one can lower a dialect into any pre-existing dialect, and subsequently make use of their lowering pipelines, without having to worry about lower abstraction levels. Even LLVM IR is implemented as its own dialect in MLIR. [Lat+20]

2.2.1 MLIR IR

The MLIR IR is structured in a graph-like data structure, where operations are the nodes and values are the edges. Values can be interpreted as the result of an operation, and each Value can be assigned a specific type. The overall structure of the MLIR IR



```
%results:2 = "d.operation"(%arg0, %arg1) ({
        ^block(%argument: !d.type):
            %value = "nested.operation() ({
                "d.op"() :() -> ()
            }) : () -> (!d.other_type)
            "consume.value"(%value) : (!d.other_type) -> ()
        ^other_block:
            "d.terminator"() [^block(%argument : !d.type)] : ()
  -> ()
  }) : () -> (!d.type, !d.other_type)
```

Figure 2.7: MLIR IR Structure

is recursive. Each operation can have multiple nested Regions, where each Region is a list of Blocks and each Block holds itself a list of Operations. SSA Values do not escape to the outside of their respective blocks, but inside each block, one can reference SSA values from the outside.

Another important aspect is that operations are not restricted to a specific use case, and can vary from function definitions to buffer allocations or even target-specific instructions or logic gates. Even the module itself is just an operation of the Builtin dialect. [MLIa]

2.2.2 Dialects

Dialects use a prefix, serving as a namespace for the defined types and operations. They can be thought of as similar to C++ libraries and are loaded into the MLIR context, where they provide various hooks. The IR verifier, for example, enforces invariants on the IR. It is also possible to define semantics for operations in a dialect.

For instance, one could define that an operation has no side effects, which would allow certain optimizations to run on that operation. [Lat+20] Overall, a dialect is composed of a set of operations, types, and optimization and lowering passes, which aim to represent an area at a specific level of abstraction.

2.2.3 Operations

One difference, to LLVM, is that MLIR uses operations instead of instructions. Operations can have distinct levels of complexity, due to their recursive structure. They can contain a nested set of other operations, or just be a straightforward addition or matrix-multiplication. In MLIR operations also have a location, which is used and manipulated by the API. This is important to know because after replacing an operation, the new location should also be updated.

2.2.4 Tablegen

Tablegen is a declarative language specific to LLVM [AR]. It serves to reduce boilerplate and make the code easier to understand and extend. Tablegen also provides an easy way to add documentation to operations. In Tablegen one can access 3 components: Directives, literals, and variables. Directives are built-in functions, like attr-dict or type. Literals are marked by a special character (`) and can contain keywords and punctuation. Lastly, variables can access entities that have been registered by the operation itself. The input and output of an operation are also defined declaratively using these components. All operations and types of the Tensor Network dialect presented in this work are declared using Tablegen.

2.2.5 Traits

Traits are a way to define additional functionality, properties, and verification of attributes, operations, and types. They are binary properties, that are checked opaquely by transformations and analysis [AR]. Examples of traits could be the commutativity of an operation or the number of operands. Another useful trait is the pure trait. It marks that the operation has no side effects, and allows pre-defined transformations to replace the operation.

2.2.6 Interfaces

Interfaces are similar to abstract classes, as they allow transformations and analyses to gain specific information on the operations implementing them.

2.2.7 Rewrite Patterns

MLIR provides the possibility to transform operations through Rewrite Patterns. By specifying an operation through the visitor pattern, one can access information about the specific operation type, like its operands and return type, and many others. The Rewrite Pattern is then applied to every one of those operations available in the module. The benefit of handling the rewrite passes through patterns, is that it allows for modular and composable transformations, which can then be applied by different algorithms already provided in MLIR itself. For instance, a simple greedy algorithm that applies the patterns until no more match is found.

A Rewrite Pattern in MLIR is composed of the Matcher and the Rewriter. The Matcher analyses the operation for a specific pattern one wants to rewrite. One example to illustrate this is a pattern that eliminates double transposes [AR]. By analyzing every transpose operation in the module, one would first check if the operand of the transpose operation was created through another transpose operation. If that is the case, then there is a match. But if the operand was the result of any other operation, we have no match and don't need to rewrite anything. In the second step, one would need to specify how to transform the operation, in this case, it would be a simple replacement of the transpose operation with the argument of the most inner transpose.

2.2.8 Dialect Conversion Framework

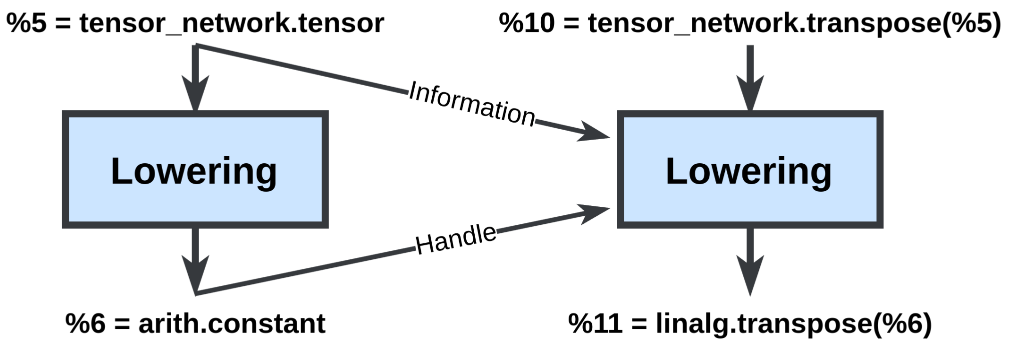

%5 = tensor_network.tensor
%10 = tensor_network.transpose(%5)
Information
Lowering
Lowering
Handle
%6 = arith.constant
%11 = linalg.transpose(%6)

Figure 2.8: Dialect Conversion

Lowering in MLIR is the process of converting operations and types from one dialect to another, usually of lower abstraction level. For this purpose, MLIR provides the Dialect Conversion Framework. This framework implements a series of mechanisms that facilitate the translation between dialects.

One of the problems that arise when trying to lower a dialect using a modular approach like Pattern Matching is that you don't have control over the order in which the transformations are applied. This creates a series of problems:

1. You don't know if the operands were already lowered
2. If the operands were already lowered, you cannot access the higher-type information
3. If the operands were not lowered yet, you cannot access the lowered operands to pass them as operands to operations of lower dialects.

To solve these problems, the Dialect Conversion Framework extends the existing Rewrite Patterns, which makes the translation between incompatible dialects easier. Every Lowering Pattern includes a handle to the original operands, which still have the higher type information, as well as a handle to the already lowered operands. This allows you to use the higher-level information to create the lowered operations, as well as use the already lowered values in those operations.

The Conversion Patterns also need access to a Type Converter, which allows registering different functions that specify how types are to be converted. This is important because the types for the handles of the lowered operands are determined this way, since the order of the lowering is not yet determined.

Another feature of the Type Converter is that it allows for registering materializations. The difference to normal type conversions is that materializations can create new IR.

Using the conversion framework, one can mark different dialects and operations as legal or illegal. The Patterns are then applied until no illegal operation is left in the module. If that is not possible, then the conversion fails. Another feature is dynamic legality, which allows setting conditions for the legality of operations, which can be useful if one wants to mark operations as illegal, that take operands of certain dialects.

2.2.9 Lowering Pipeline

Every dialect included in MLIR has its own set of conversion patterns and type converters, which can be used to lower to other standard dialects. This means, that when creating an own dialect in MLIR, one simply needs to lower it to any of the standard dialects, and then use the available lowering passes. A set of consecutive lowering passes is considered a lowering pipeline in MLIR. Of course, the order of the lowering passes is important, since there is not a conversion between every two dialects. Because of this, one needs to find a path to lower all dialects into the LLVM dialect, which can be mapped 1:1 to LLVM IR and consequently into an executable binary. MLIR provides the infrastructure to lower to other targets as well.

2.2.10 JIT Execution Engine

MLIR also includes an Execution Engine, which allows a JIT compilation and execution of an MLIR Module, containing MLIR IR. This way, one can execute and retrieve the results of the function, which abstracts away the need to serialize and deserialize the resulting data structures.

2.2.11 Important Dialects

MLIR provides a set of pre-defined dialects, meaning a set of operations and types, together with existing optimization and lowering passes for them. In this project, we make use of some of those dialects during the lowering process. In the following, some of those dialects will be presented.

Builtin Dialect

The Builtin dialect of MLIR includes a fundamental set of operations, attributes, and types with broad applicability across numerous domains and abstractions. The dialect is essential to the core MLIR IR implementation and is automatically loaded into every MLIR Context. It includes a wide range of different types like Float16Type, IntegerType, or MemRefType and attributes like StringAttr and DenseArrayAttr. The MLIR module itself is also just the *builtin.module* operation from this dialect.

Func Dialect

As we've discussed earlier in MLIR even function definitions are operations. The Func dialect provides the operation to define (func.func) and call (func.call) a function, as well as an operation to specify what SSA Value to return (func.return).

Arith Dialect

The Arith dialect operates at the level of integer and floating-point arithmetic. It provides operations like addition (arith.addf, arith.addi), subtraction (arith.subf, arith.subi), multiplication (arith.mulf, arith.muli) and division (arith.divf, arith.divui, arith.divsi), amongst many others. Arith also includes the arith.constant operation, which allows creating a tensor with constant values, which can be used as the initial value for subsequent tensor operations.

Tensor Dialect

The Tensor dialect, seeks to be a collection of operations and types, as small as possible, that are too specific to be associated with any other dialect. The tensor type itself does not live in this dialect, but rather in the Builtin dialect. Interesting operations in the Tensor dialect include the extraction of slices, concatenation of tensors and the creation of an empty tensor with shape information.

Linalg Dialect

The Linalg dialect operates at the abstraction level of tensors, and defines operations like addition (linalg.add), absolute value (linalg.abs), substraction (linalg.sub) or even operations like matrix-vector (linalg.matvec) and matrix-matrix (linalg.matmul) multiplication. There are also operations like slicing (linalg.slice) and reshaping (linalg.reshape) that modify the tensor itself or create a new view of the tensor. One of the most important operations that the Linalg dialect includes is the Generic operation, which allows specifying a custom operation on tensors, which is very useful when contracting a tensor on a specific index and maybe over multiple indices at the same time.

Affine Dialect

This dialect provides an abstraction for affine operations. The most important component of the dialect for this work is the affine map, which helps specify the access patterns for specific tensors. We use these maps to create the *linalg.generic* operations for each lowering of a specific tensor contraction.

Memref Dialect

The Memref dialect, allows the manipulation of memory segments and includes operations such as memref.load and memref.alloc, memref.dealloc and memref.copy. Many of the other dialects are first lowered into the Memref dialect during the lowering pipeline.

Structured Control Flow and Control Flow Dialects

The SCF (Structured Control Flow) dialect was first introduced as a common lowering stage between the Affine and Linalg dialects. This dialect contains structures such as "if" and "for", which are then lowered into the CF (Control Flow) dialect. The CF dialect does only include 4 operations, namely *Assert*, *Branch*, *Conditional Branch* and *Switch*. In the end, CF is directly converted into the target, like LLVM or SPIR-V.

2.3 Motivation

The graphical notation we've explored for tensor networks provides a powerful visual language for representing complex tensor operations and networks. This notation's ability to model arbitrary tensor operations makes it an ideal foundation for designing the Tensor Network dialect in MLIR.

Through this abstraction, we aim to facilitate implementing ideas from literature, as well as allow the lowering from even higher abstraction levels, like quantum circuits or some areas of chemistry, for instance.

MLIR, with its focus on improved programmability and extensibility, offers significant advantages over traditional compiler construction approaches. By building our tensor network compiler on this framework, we aim to bridge the gap between the flexibility and ease of use typically associated with library solutions, and the powerful low-level and whole-program optimizations characteristic of compiler-based approaches.

Furthermore, the extensibility of MLIR aligns perfectly with the evolving nature of tensor network research. As new algorithms and optimization techniques are developed, they can be readily incorporated into our compiler infrastructure, ensuring that it remains at the cutting edge of the field.

For this work, we aim to create a first design and proof-of-concept for this goal, with the long-term goal of surpassing the state-of-the-art library approaches for tensor network contractions.

3 Overview

The following overview outlines our tensor network compiler's system architecture, design goals, workflow, and key components with their functions.

3.1 System Overview

Figure 3.1 shows the structure of the CompTN infrastructure. In the following, we outline the steps involved in compiling and optimizing tensor network contractions, from the initial network representation to the final executable code.

1. **Python Frontend**: The tensor network is first defined through a Python-based frontend, which interacts with the MLIR implementation through custom bindings. In the background, a module is created and filled with the necessary operations.
2. **Tensor Network Abstraction**:
   a) **Tensor Network Dialect**: The module is created using the operations and types defined in our own Tensor Network dialect. The dialect is designed around the graphical and Einstein notations for tensor networks, often used in literature.
   b) **Optimization Passes**: We include a series of optimization passes, including two contraction path search algorithms, rank-simplification and slicing.
3. **Lowering Process**: We implemented a series of lowering patterns, that lower the operations and types from the tensor network dialect to the MLIR standard dialects. By doing this we can make use of existing lowering and optimization passes from other dialects to reach the LLVM Dialect.
4. **Execution**: The final LLVM IR representing the optimized tensor network contraction is transformed to a target binary and executed. By using the MLIR execution engine, we can easily deserialize the results, so they can be further manipulated from within the python program.

The exact details of how the Tensor Network dialect is structured will be discussed in Section 4 and the optimization and lowering passes in Sections 4 and 5.

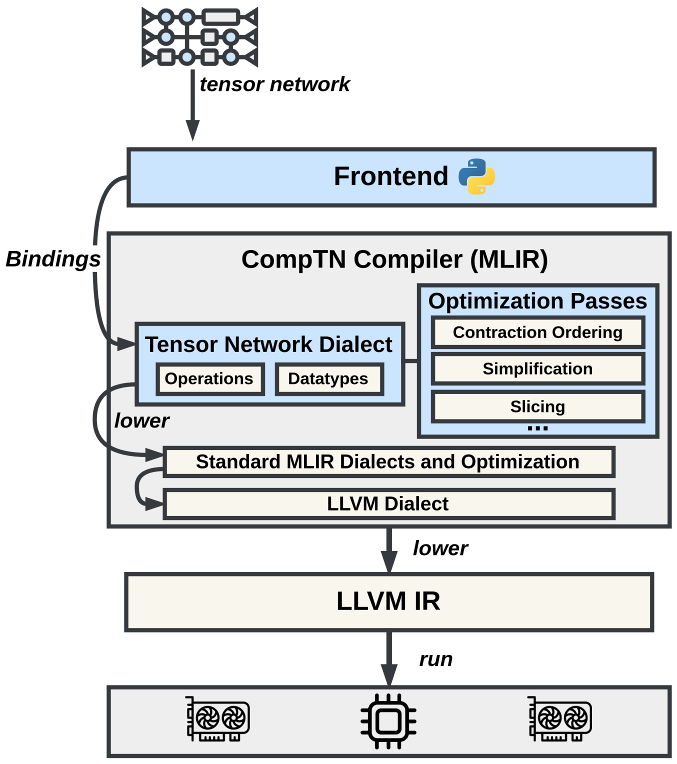


Figure 3.1: Overview of the CompTN Infrastructure

3.2 Design Goals

3.2.1 Extensibility

CompTN is the first step in shifting the problem of efficient tensor network contractions from library approaches to compilation. We aim to design CompTN in such a way that the extension of the project by researchers should be easy.

1. **MLIR**: We use MLIR as the underlying compiler framework, which is designed in a modular and extensible way, such that adding new operations, types, and optimizations is easy.

2. **Dialect structure**: By structuring our dialect around the graphical and Einstein notations of tensor networks, we aim to provide a good base abstraction, usable by researchers from many fields. Thus simplifying building upon CompTN through either another higher-level dialect or frontend extensions.

3.2.2 Programmability

CompTN also aims to be programmable, to specifically address the user's hardware requirements and optimization wishes. As we determined during the evaluation in Chapter 6, depending on the specific network that is being computed, different optimizations reach different outcomes. For this, we aim to provide the user as much liberty as possible to decide which optimizations and what hyperparameter values to use.

3.2.3 Performance

CompTN seeks to optimize the contraction of tensor networks. The Tensor Network dialect includes a contraction-path finder, as well as rank-simplification and slicing optimizations, which seek to make the computation of tensor network contraction much more efficient.

3.3 CompTN Python Program

The tensor network contraction is performed in a JIT compilation, where the module is built, lowered, compiled, and run during the execution of the Python program. The following code segment shows an example of how the program would look like, for the example tensor network used in Figure 2.3 of Section 2.

1. **Create a module:** First, one starts by creating a *ModuleManager*, which is a wrapper around the MLIR module where the operations are to be inserted.
2. **Insert indices:** Then one can insert all the indices available in the tensor network containing the dimension sizes.
3. **Define tensors:** The tensors take the starting arrays (as NumPy arrays), as well as the set of indices they have.
4. **Contract tensors:** By calling *contract_multiple* the compiler determines the best path to contract the tensors provided to the call.
5. **Run the computation**: Either compile and run the computation directly by using *run* or specify the compilation configurations and run the program in two steps.
6. **Post-processing:** Subsequently, one can continue manipulating the result, in this example, the code simply prints the result values.

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

mm = tn.ModuleManager()
indexI = mm.Index(6)
indexJ = mm.Index(6)
indexK = mm.Index(6)
indexL = mm.Index(6)

array1 = np.random.rand(6, 6, 6)
array2 = np.random.rand(6, 6)
array3 = np.random.rand(6, 6)

tensorA = mm.Tensor(array1, indexI, indexJ, indexK)
tensorB = mm.Tensor(array2, indexJ, indexL)
tensorC = mm.Tensor(array3, indexK, indexL)

mm.contract_multiple(tensorA, tensorB, tensorC)
result = mm.run()
print("CompTN Result: " + str(result))
```

Figure 3.2: Code to determine the contraction for the example tensor network

```python
mm.compile(use_greedy_gray_kourtis=True,
 ↪  enable_rank_simplification=True, enable_slicing=False,
 ↪  gray_kourtis_alpha=0.6)
result = mm.run_compiled()
```

Figure 3.3: Optionally: Compile the module with custom configurations before execution

4 Design

4.1 Tensor Network Dialect

The Tensor Network dialect is the basis of the CompTN compiler. At the core of this dialect are two concepts: smart indices and tensors with associated indices. Smart indices represent the edges of the tensor network graph and contain the size information about a specific dimension. Tensors, on the other hand, contain the tensor values, as well as a set of smart indices, that must match the rank and dimension sizes of the tensor.

This idea emerges from both the graphical and Einstein notations for tensor networks. So when two tensors share the same index, it is assumed that they are to be contracted over that dimension and that they are connected by an edge in the graphical notation.

The following section will present the operations, types, and passes available to the Tensor Network dialect.

4.1.1 Types

The Tensor Network Dialect defines two types. These custom types allow the lowering passes for CompTN operations to access the necessary information about the tensors more easily, such as the tensor shape and their respective indices.

The purpose of assigning the SSA Value resulting from any given operation to a type containing the shape and index information is that you only have to compute the index set once for each tensor. Each operation taking the tensor values as operands can then access the information without having to traverse the operation tree again.

- **IndexLabelType**: The $IndexLabelType$ holds the information about the size of the dimension that the tensor represents, as well as a name that serves to identify the indices.

- **TensorWithIndicesType**: This type holds an attribute containing the tensor shape and a variadic list of $IndexLabel$s, one for each dimension of the tensor, where the index size and the dimension size must match.

4.1.2 Operations

The Tensor Network Dialect specifies a number of Operations that allow for the representation of a Tensor Network and its computation.

- **IndexOp**: This operation takes the size of the index as an argument and allows for the creation of an SSA Value of *IndexLabelType*. One can optionally name the index, in which case the Index type will also hold a custom name, which is not strictly necessary but can make the tracking of a specific index easier.
- **TensorOp**: The TensorOp operation creates an SSA Value of *TensorWithIndicesType*, given the values, shape, and indices that the tensor should contain.
- **ContractMultipleOp**: This operation takes a variable number of Tensors and returns an SSA Value of *TensorWithIndicesType*, containing the correct shape and indices of the result tensor. The goal of this operation is to provide an intermediate step before determining the contraction path. In the end, this operation should be rewritten into a sequence of ContractOp operations.
- **ContractOp**: This operation takes two input tensors and produces an SSA Value of *TensorWithIndicesType* holding the resulting tensor's shape and indices. ContractOp abstracts the actual contraction computation. During lowering, it is transformed into a series of loops that perform the tensor contraction.
- **AddOp**: This operation adds two tensors. Tensor addition does not change the shape or indices of both tensors. It is necessary to correctly add two tensors without losing the index information, after computing multiple separate tensor networks, for instance through slicing.

4.2 Contraction Path finding

Finding the optimal contraction path for an arbitrary tensor network is an NP-hard problem. The chosen contraction path can substantially influence both the runtime and the space needed for the computation. Determining the optimal path is often not possible, which is why certain heuristics like memory and time are used to find solutions. The contraction path for the given tensor network in the Tensor Network dialect is determined during the transformation of a *ContractMultipleOp* operation into a series of 2-tensor *ContractOp* operations.

Because of the modularity of MLIR patterns, adding new path-finding algorithms is easy and only requires creating a new structure with the function and adding it to the pattern set.

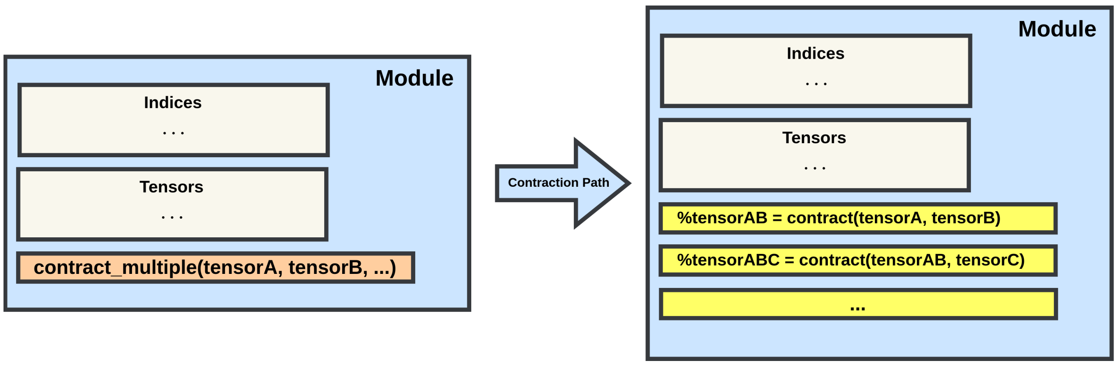

Module
Indices
...
Tensors
...
contract_multiple(tensorA, tensorB, ...)
Contraction Path
Module
Indices
...
Tensors
...
%tensorAB = contract(tensorA, tensorB)
%tensorABC = contract(tensorAB, tensorC)
...

Figure 4.1: The contraction path is determined when transforming the *contract_multiple* operation into a sequence of *contract* operations

4.2.1 Greedy Path Finding

One of the simplest ways to determine a good contraction path, is to use a greedy approach. The algorithm for a greedy contraction path search is as follows:

1. **Initialization:** Initialize a list with all the tensors to be contracted.
2. **Cost calculation:** Determine the contraction cost for each tensor pair.
3. **Optimal pair selection:** Contract the tensor pair with the minimum contraction cost.
4. **List update:** Replace both contracted tensors in the list with the result.
5. **Iteration:** Repeat steps 2-4 until only one tensor remains.

The contraction cost can be a simple metric proportional to the number of floating-point operations needed, which is one of the heuristics used for this work. This can be computed by calculating all the indices that one needs to iterate over for the contraction:

$$cost(T_i, T_j) = |T_i.indices| + |T_j.indices| - |T_i.indices \cap T_j.indices|$$

Another possibility would be the heuristic introduced in [GK21], which includes a tunable constant α, where if $\alpha = 1$, it would model the change in memory, were one to contract the two tensors T_i and T_j to obtain the new tensor T_k.

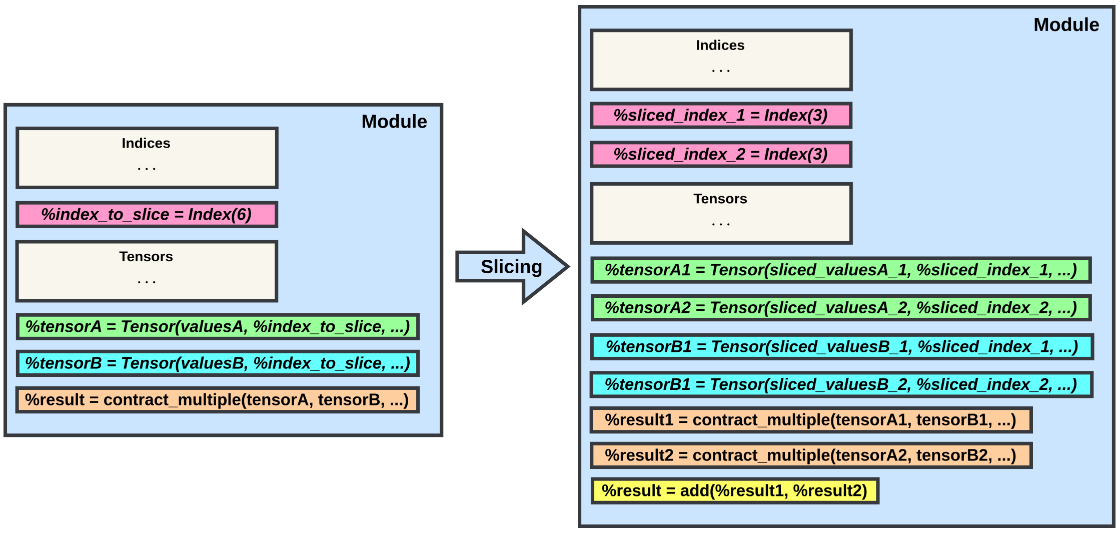
$$cost(T_i, T_j) = size(T_k) - \alpha(size(T_i) + size(T_j))$$

This approach usually delivers a good contraction path, because minimizing contraction costs keeps the intermediate tensors considerably small. One can think of it as avoiding having to iterate over large dimensions of non-shared indices too many times.

4.3 Optimization Passes

4.3.1 Tensor Network Slicing

One of the main problems when computing large tensor networks is that the tensors can become too large to fit in memory. This can be true for the initial tensors, as well as intermediate results. One way of combating that, whilst also allowing for the parallelization of the overall contraction, is *slicing*. In Figure 4.2 you can find a visualization of this process.



```
%index_to_slice = Index(6)
%tensorA = Tensor(valuesA, %index_to_slice, ...)
%tensorB = Tensor(valuesB, %index_to_slice, ...)
%result = contract_multiple(tensorA, tensorB, ...)
```

```
%sliced_index_1 = Index(3)
%sliced_index_2 = Index(3)
%tensorA1 = Tensor(sliced_valuesA_1, %sliced_index_1, ...)
%tensorA2 = Tensor(sliced_valuesA_2, %sliced_index_2, ...)
%tensorB1 = Tensor(sliced_valuesB_1, %sliced_index_1, ...)
%tensorB1 = Tensor(sliced_valuesB_2, %sliced_index_2, ...)
%result1 = contract_multiple(tensorA1, tensorB1, ...)
%result2 = contract_multiple(tensorA2, tensorB2, ...)
%result = add(%result1, %result2)
```

Figure 4.2: Visualization of the module during the slicing process

The slicing of a tensor network is done by fixing the values for the iteration over one of the indices of the network. By doing this, one can divide the tensor network into multiple tensor networks, and then add the separate results to obtain the original value for the tensor network contraction. To achieve this using the Tensor Network dialect, one needs to:

1. **Split the index:** Replace the index one wants to slice with multiple indices of smaller sizes.

2. **Create smaller tensors:** For each tensor that uses the original large index, create new, smaller tensors. These smaller tensors contain only a portion of the original data. The dimension corresponding to the split index is reduced in size.
3. **Replace the original operation:** Replace the original *contract_multiple* operation with multiple *contract_multiple* operations, one for each sliced index, with their respective new tensor slices. These operations represent the new smaller tensor networks.
4. **Combine results:** In the end, add the individual results to obtain the original tensor network contraction.

4.3.2 Rank Simplification

Rank Simplification is essentially just a pre-processing step. The more tensors are available in the tensor network, the more time it takes for the path-finding algorithm to find a good contraction path. For this reason, rank-simplification analyses the tensor network before determining a contraction path and contracts every two tensors that, when contracted, do not increase the rank of the resulting tensor. By doing this, one eliminates the tensors from the tensor network, that are trivial to contract and do not affect the performance negatively. [Hai22; GK21]

4.4 Lowering of the Tensor Network Dialect

The following section will focus on the lowering of the Tensor Network dialect into the LLVM dialect. The main challenge during this process is to bring the operations and types of the dialect into any of the dialects already included in MLIR. If one manages to get to those dialects, one can continue the lowering by using the already pre-implemented lowering passes. In Figure 4.3 you can see a summary of the lowering process from the Tensor Network dialect into the LLVM dialect.

With proper implementation, the lowering process can be simplified to focus on just three key operations: *TensorOp*, *ContractOp*, and *AddOp*.

- **2-Tensor Contractions** The *contract* operation represents a single contraction of two tensors. This operation gets lowered to the Linalg dialect, in the form of a *linalg.generic* operation, which allows specifying a computation with a series of access patterns and a given set of nested operations for the execution body. The advantage of using *linalg.generic* is that it abstracts away the need to create and correctly order the loops for the contraction. By just specifying the overall available indices of the operation and the access patterns for every input tensor,

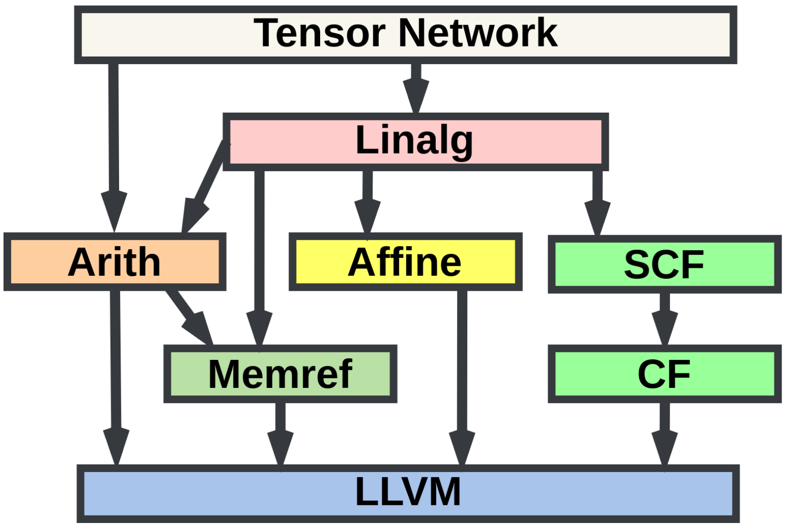


Figure 4.3: Lowerings from the Tensor Network Dialect into the LLVM Dialect

the *linalg.generic* operation is correctly lowered into a correct sequence of $scf.for$ loops, memory accesses from the Memref dialect, and arithmetic operations from the Arith dialects. Because the Tensor Network dialect revolves around the concept of indices, determining the access patterns is very easy:

$$T_k[(T_i.indices \cup T_j.indices) - (T_i.indices \cap T_j.indices)] = T_i[T_i.indices] \cdot T_j[T_j.indices]$$

For the result tensor, we need to iterate over the indices that do not appear in both input tensors, since those are the dimensions that are contracted over, and would therefore disappear. For the input tensors, we simply iterate over their complete list of indices. The creation of the *linalg.generic* operation is further explained in Chapter 5.

- **Tensors** The TensorOp operation gets lowered into the *arith.constant* operation, where the values of the Tensor get stored as an attribute of the operation. The Arith operation then gets bufferized, and the values can then be accessed using operations from the Memref dialect.
- **Addition** The lowering process for this operation is straightforward, as the access patterns are identical for both input tensors and the output tensor, given that their indices should match. We lower this operation to a *linalg.generic* operation.

The other Operations are the IndexOp and ContractMultipleOp, which do not need to be lowered for the following reasons:

- **IndexOp**: For the IndexOp, the necessary information is stored in its SSA Value of *IndexLabelType*, which is passed to the Tensor on construction, and then further stored in the *TensorWithIndicesType*. Therefore, the SSA Value of the IndexOp is only necessary when building up and optimizing the module, and can be deleted after that.

- **ContractMultipleOp**: This operation should already be rewritten into a series of single 2-tensor contractions, which is why it is no longer available in the module at the start of the lowerings.

4.5 Frontend

The frontend for CompTN is implemented using Python bindings, which provide a user-friendly interface to the underlying MLIR-based system. The bindings are structured in such a way, that defining a tensor network is essentially a 1:1 mapping to the Tensor Network dialect, making use of smart indices. The usage of NumPy also allows for the seamless creation and manipulation of the initial tensor values or the computed contraction results. Another advantage of this approach, is the ability to use control flow structures and I/O operations already defined in the language, without having to re-implement them. The choice of Python specifically, is only rooted in the fact, that many existing tensor network and quantum computing tools use this language already.

4.5.1 Python Bindings to CompTN

The bindings available in the Python program are the following, where all operations, except *run* return a wrapper around the actual MLIR SSA Value, that can be recursively used as arguments for the other operations:

```
ModuleManager()
Index(size: int)
Tensor(indices: Index[])
contract(lhsTensor: Tensor, rhsTensor: Tensor)
contract_multiple(tensors: Tensor[])
run() -> result: NumPy.array
```

5 Implementation

5.1 MLIR IR

To further illustrate how the module gets built and lowered, we will take the example tensor network and program from Sections 2. Executing the program will generate an MLIR module in the background, filled with the respective operations of the Tensor Network dialect.

As you can see, the structure of the IR is very similar to how the Python program is written. And this example correctly illustrates how each SSA value carries the tensor and index information in their respective type. Below you will find the IR for some of the operations in the dialect.

```
%0 = "tensor_network.index"() <{name = "index_0", size = 6 : i64}> : () ->
↪  !tensor_network.indexlabel<6 : i64, "index_0">
```

```
%4 = "tensor_network.tensor"(%0, %1, %2) <{value = dense<[ %% Values for the tensors %%
↪  ]> : 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">]>
```

```
%7 = "tensor_network.contract_multiple"(%4, %5, %6) :
↪  (!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">]>,
↪  !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">]>
```

5.1.1 Lowering

After executing the first set of lowering passes, further explained in Chapter 4, the module in Figure 5.1 only contains the Arith, Linalg, Affine and Func dialects. The workflow starts by creating a set of constants, corresponding to the values and shape of the tensors, which are then passed down to a chain of *linalg.generic* operations. Lastly, the final computation result is returned by the function, which later on can be invoked to retrieve the return value.

Affine maps

Each *linalg.generic* operation has its unique three affine maps, that correspond to the accessing patterns of the specific tensor contraction. The first two correspond to the access patterns of the input tensors, and the last map to the output tensors. For the *linalg.generic* operation, these maps must follow a specific structure. The preimages of the maps must always be the set of all indices available, for each of the indices specified here, the linalg dialect later creates a set of nested for-loops. The shape of the input and output tensors determine the ranges of the iterations. The images of the maps correspond to the access to the specific tensors during the loops, which is identical to the Einstein notation for tensor contractions. Therefore, the affine maps below, would correspond to the computation of:

$$C_{d0,d2,d3} = \sum_{d1} A_{d0,d1,d2} \cdot B_{d1,d3}$$

```
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)>
```

5.2 Pass Execution

Depending on the configurations, that the user can specify during compilation through the Python bindings, the passes are executed in the following order:

1. **Rank-Simplification**: First we apply rank simplification, by doing this we avoid having to carry trivial tensors of rank 1 and 2 into the other steps.
2. **Slicing**: Then we perform slicing, this is because we can slice the network without having information about the contraction path. Therefore, doing this before the path-finding simplifies the process.
3. **Contraction-Path Search**: Here the user can choose between using Greedy path-search with either a FLOP or the Gray-Kourtis heuristic.
4. **Lowering Passes**: Lastly, the lowering pass is executed, which lowers every operation in the way described in Chapter 4.

```
module {
  func.func @main() -> tensor<6xf64> attributes {llvm.emit_c_interface, rewritten} {
    %cst = arith.constant dense<[ %% Values for the tensor %% ]> : tensor<6x6x6xf64>
    %cst_3 = arith.constant dense<[ %% Values for the tensor %% ]> : tensor<6x6xf64>
    %cst_4 = arith.constant dense<[ %% Values for the tensor %% ]> : tensor<6x6xf64>
    %cst_5 = arith.constant dense<0.000000e+00> : tensor<6x6x6xf64>
    %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>
    %cst_6 = arith.constant dense<0.000000e+00> : tensor<6xf64>
    %1 = linalg.generic {indexing_maps = [affine_map<(d0, d1, d2) -> (d0, d1, d2)>,
↪   affine_map<(d0, d1, d2) -> (d1, d2)>, affine_map<(d0, d1, d2) -> (d0)>],
↪   iterator_types = ["parallel", "reduction", "reduction"]} ins(%0, %cst_4 :
↪   tensor<6x6x6xf64>, tensor<6x6xf64>) outs(%cst_6 : tensor<6xf64>) {
    ^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<6xf64>
    return %1 : tensor<6xf64>
  }
}
```

Figure 5.1: The MLIR module after applying the first set of lowering passes

6 Evalutation

The main idea of this project is to analyze the viability of an MLIR compiler-based infrastructure for tensor network contractions, which could enable just-in-time (JIT) compilation for full tensor network contraction, instead of doing the optimization during runtime using a TN-library. To investigate the possible advantages of our approach, we aim to answer the following research questions:

1. How does the end-to-end runtime of CompTN compare to state-of-the-art Python frameworks?
2. What is the impact of our compiler optimization on the execution time of contractions?

In the following, we describe our experimental methodology and our evaluation results.

6.1 Methodology

6.1.1 Experimental Testbed

We conducted our benchmarks on a Linux machine running Ubuntu 22.04.4, equipped with an Intel-Core i7-1165G7 CPU (4 physical cores, 8 threads) and 16GB of DDR4 RAM.

6.1.2 Benchmarks

We benchmark by using workloads that simulate various types of quantum circuits, which are particularly suitable for tensor network simulation. Because CompTN does not yet implement imaginary numbers, the circuits are initialized only with real values.

Greenberger-Horne-Zeilinger (GHZ) State

The GHZ circuit generates a maximally entangled state among multiple qubits, where measuring one qubit instantly affects all others. This unique property makes it valuable for quantum cryptography applications. Additionally, its high sensitivity and simulation complexity make the GHZ state an excellent benchmark for testing efficient computations.

Variational Quantum Eigensolver (VQE)

The VQE algorithm computes the eigenvalues of a given matrix. Its tensor network representation is noteworthy due to its relatively sparse structure. However, the contraction process requires repeated evaluation of numerous similar circuits, making it an ideal benchmark.

Quantum Machine Learning (QML)

QML circuits seek to exponentially accelerate classical machine learning tasks such as classification, clustering, and optimization. These circuits exhibit a high level of complexity, making them excellent benchmarks for tensor network computation techniques.

Random Quantum Circuits

Lastly, we evaluate our implementation using random circuits generated by Qiskit. This approach tests the overall performance on non-standard circuits, providing insights into the implementation's versatility and robustness.

6.1.3 Metrics

We measure the execution time of both compilation (with optimizations) of the tensor network and the runtime of the contractions. Additionally, for CompTN we measure the construction time of the module and compilation time.

6.1.4 Baseline

We compare against quimb [Gra18], which uses cotengra [Gra24a] to drive the contraction. Note that, as CompTN is the first framework to use compilation at the TN level, all baselines are optimizing and contracting tensor networks as a library during their execution in Python.

6.2 End-to-end Runtime Analysis

For this analysis, the version of CompTN running the Gray-Kourtis Greedy path-search algorithm runs rank-simplification, and $\alpha = 0.6$. The other implementation is a heuristic only optimizing for the number of FLOPs for each contraction, for which we do not run rank-simplification.

We chose those options because they provide the best runtime for their respective heuristics. Further analysis of the impact of rank-simplification and the value for constant α can be found in the following sections.

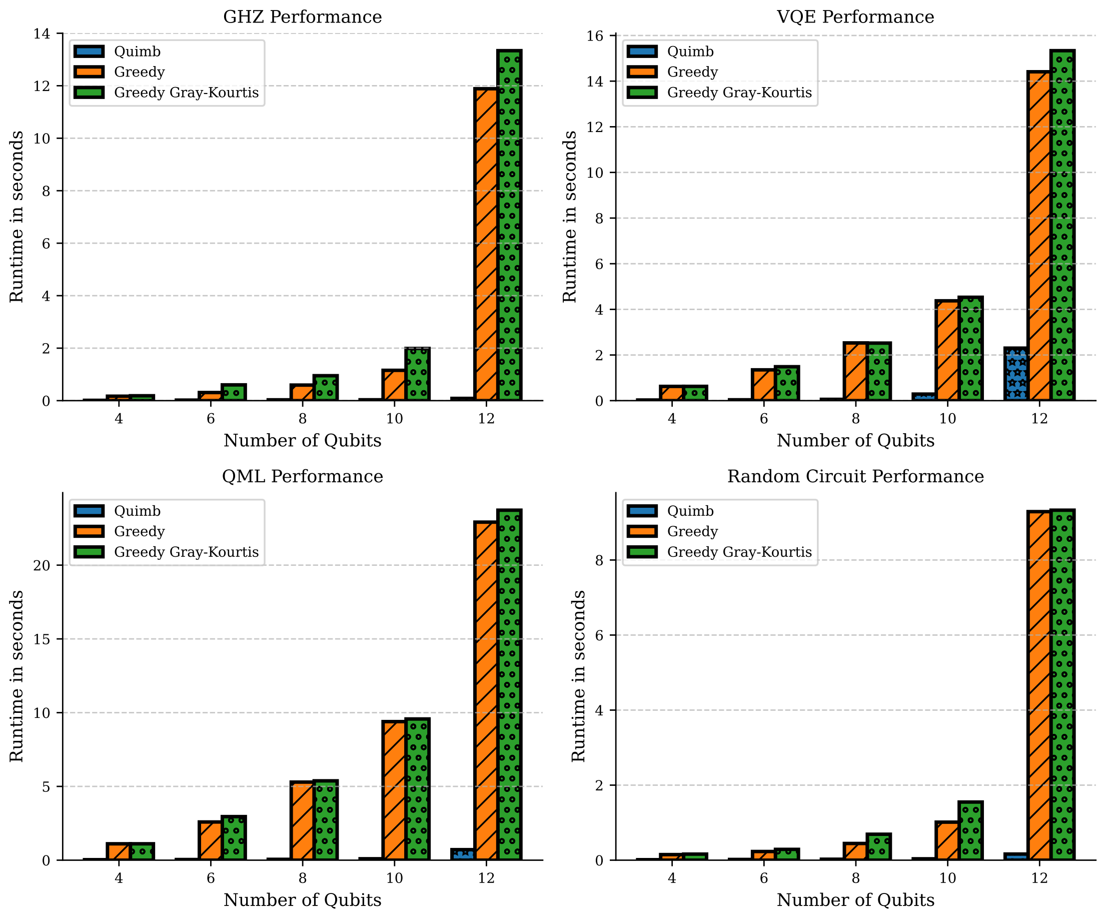


Figure 6.1: A comparison of the end-to-end runtime for the benchmark quantum circuits

In Figure 6.1 you can see the analysis for the end-to-end runtime for CompTN, running two different heuristics with their respective optimization sets. Our results indicate that our implementation does not yet outperform Quimb. This could be attributed to the fact that CompTN still does not implement optimizations such as GEMM, and sparse tensor optimizations, as well as multithreading. Generally quantum circuits include many sparse tensors, which is why CompTN is executing operations, that are known to result in 0-entries.

For our own implementations, our evaluation found that if tuned correctly, they can achieve a similar runtime for the same quantum circuits.

6.3 Runtime Breakdown

In Figure 6.2 you can observe the division of the end-to-end execution time into construction time, compilation time, and run-time. The construction time describes the time needed to process the Python program and insert the operations into the MLIR module for further lowering. The compilation time starts at the beginning of the lowering and finishes when the module is ready to be executed, and the runtime corresponds to the time needed to run the compiled program.

The time needed to build up the module is low, since the insertion of the operations only scales linearly with the number of indices and tensors, but does not depend on the overall number of contractions. The compilation increases with the number of indices, but as the number of qubits gets larger, the compilation time takes up proportionally less time in the overall end-to-end execution time. However, we can observe that for 12 qubits, the compilation time takes up about 28% of the end-to-end execution time for the random circuit and just 18% for the GHZ circuit, indicating that the compilation time increases relatively to the overall execution time for more structure-less networks. Nonetheless, our implementation still shows a relatively low compilation time, suggesting that there is still space for more optimizations, to shift some of the run-time into the compilation time, and potentially make use of more sophisticated analysis. An interesting future approach, would be to link high-performing tensor contraction libraries into the compiler and execute the trivial contractions during compilation time, reducing the runtime.

6.4 Impact of Compiler Optimizations

6.4.1 Greedy FLOP Heuristic

In Figure 6.3 you can see the impact of the rank-simplification pass on the program's runtime for the 4 benchmark circuits. The results show that running the rank simplification on the tensor networks increases the runtime of the program by up to 7% for the 12 qubit QML circuit.

This could be explained by the fact that we are using a greedy path-search approach, which by definition first contracts all the cheaper tensors. Applying the rank simpli-

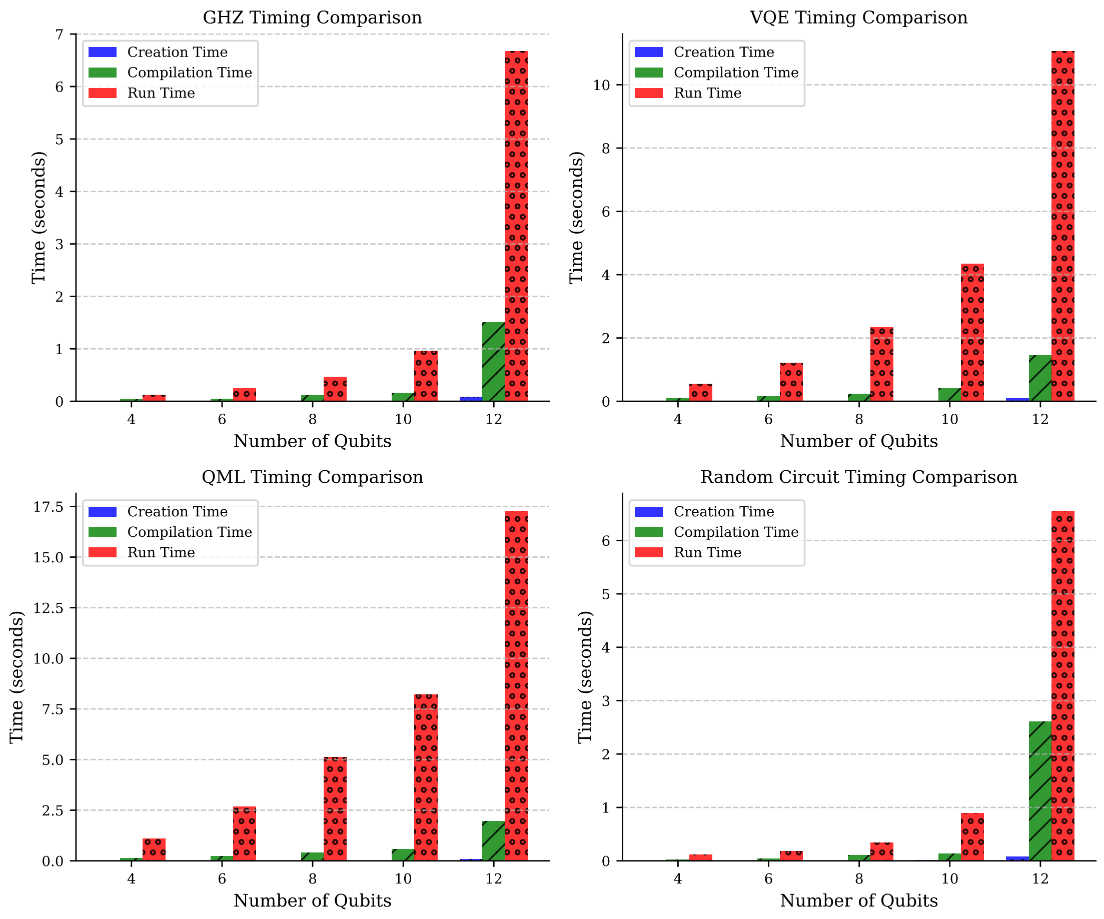


Figure 6.2: The time distribution for the computation of different benchmarks, using Greedy path-finding

fication contracts all the tensors in the network of rank 1 and 2 with neighboring tensors, which interferes with the greedy path search, especially for the tensors early in the path. This could also explain why the difference in runtime is relatively small, since the path is only modified for the tensors in the network with the smallest rank.

An interesting analysis would be to test the rank simplification with other contraction-path search algorithms, that do not necessarily interfere with each other.

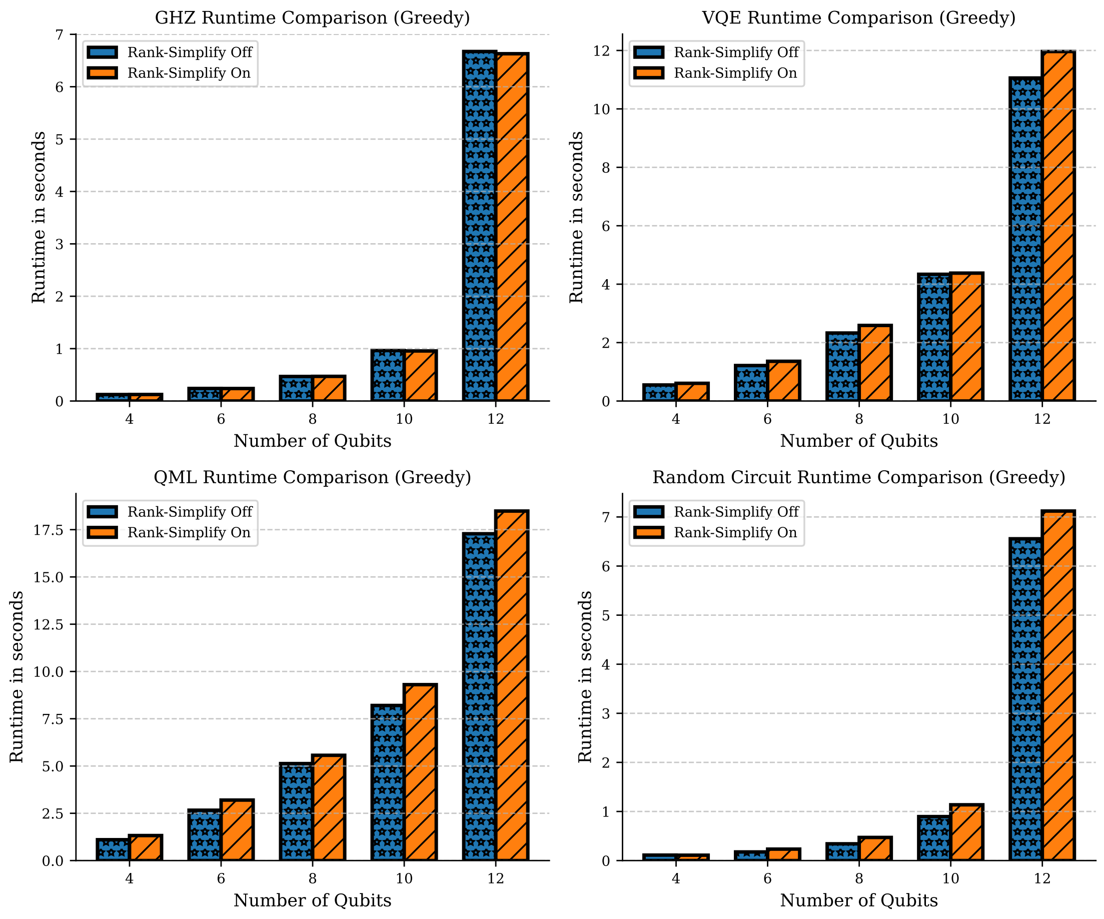


Figure 6.3: Impact on run-time for the rank simplification for the greedy path-search using heuristic for FLOPs

6.4.2 Gray-Kourtis Heuristic

Tunable Constant

The heuristic presented in [GK21] is the following, being T_i and T_j the input tensors and T_k the potential output tensor.

$$cost(T_i, T_j) = size(T_k) - \alpha(size(T_i) + size(T_j))$$

First, we analyze the impact of the hyperparameter α on the runtime of the program. For this, we executed 20 iterations on both the QML and a Random circuit using 8 qubits and no rank-simplifications. The results can be found in Figures 6.4 and 6.5.

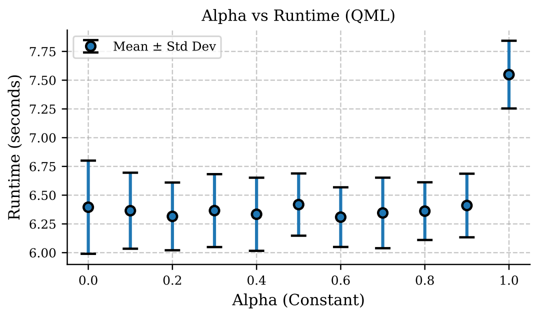


Figure 6.4: QML with 8 qubits

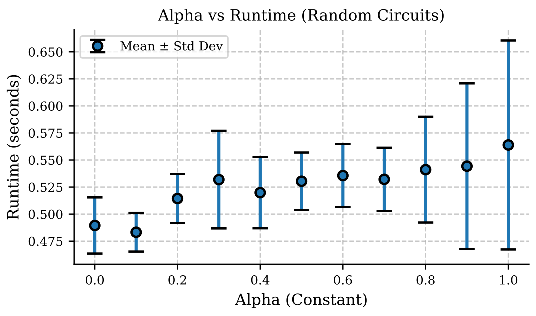


Figure 6.5: Random Circuit with 8 qubits

Taking $\alpha = 1$ would mean that the cost is directly proportional to the change in memory if one were to perform the contraction [GK21]. The results show that using $\alpha = 1$, performs worse than any other value for the QML circuit and is rather unpredictable for random circuits. Therefore, using a moderate value for the general case is advised. In the future one could even think of sampling the tensor network, to fine-tune hyperparameters like this to the specific tensor network, due to the variability of the results for each network.

Rank Simplification

Figure 6.6 shows the impact of applying the rank-simplification pass together with the greedy path-search using the Gray-Kourtis heuristic and $\alpha = 1$. This means that the greedy path search optimizes for the change in memory. The results show that rank simplifications an ambiguous impact on the runtime, depending on what circuit it is being run on. The rank simplification speeds up the contraction by 35% and 29% for the 12 qubit random circuit and GHZ, respectively. For the other two circuits, applying the rank simplification has no considerable impact on the runtime.

Having looked at the impact of the hyperparameter on the runtime, it makes sense to also observe the impact of rank simplification with $\alpha = 0.6$, an overall good value for both circuits in Figures 6.4 and 6.5. The results of this analysis can be found in Figure 6.7.

One can observe that the rank simplification still slightly decreases the runtime for the GHZ and random circuits and slows down the VQE and QML. Nevertheless, the difference between both runtimes is only about 5% for the GHZ circuit on 12 qubits. This could be attributed to the fact that the overall runtime of the contraction, also significantly decreases when choosing $\alpha = 0.6$.

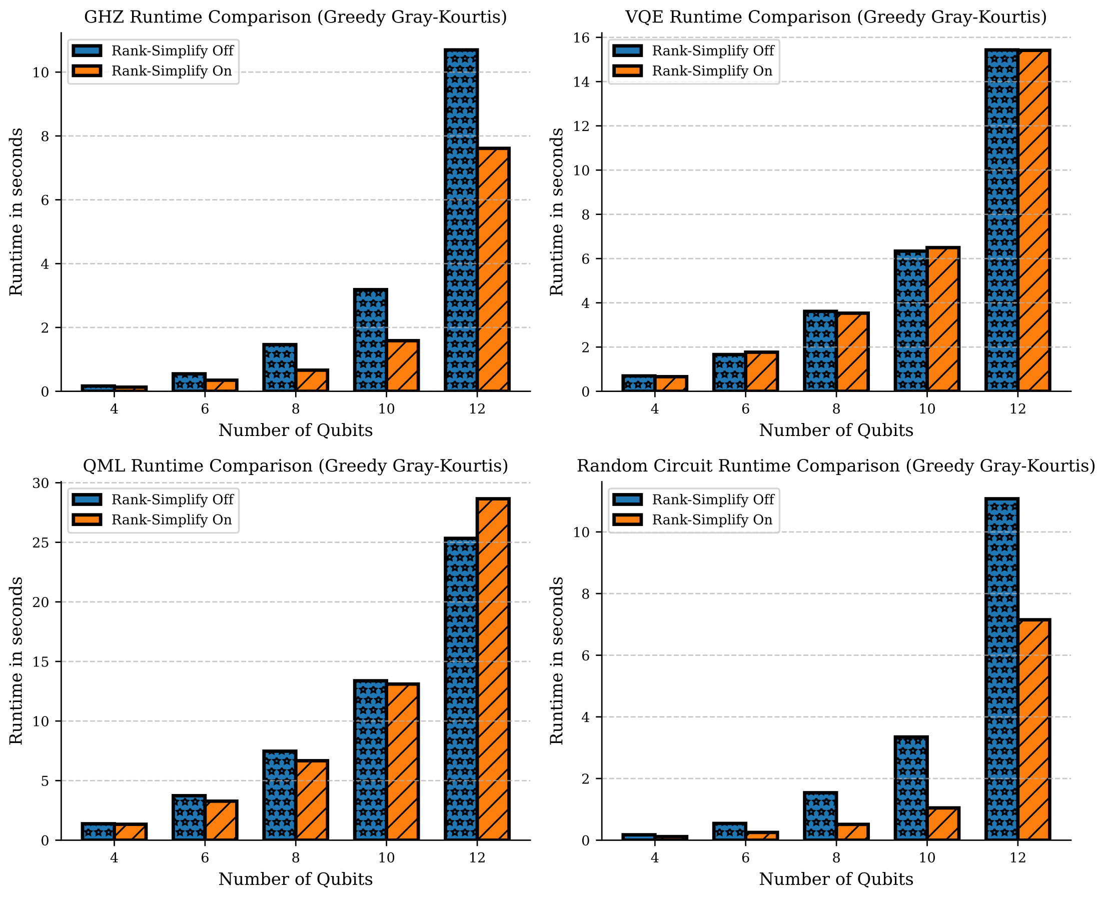


Figure 6.6: Impact on run-time for the rank simplification for Gray-Kourtis Greedy path-search with $alpha = 1.0$

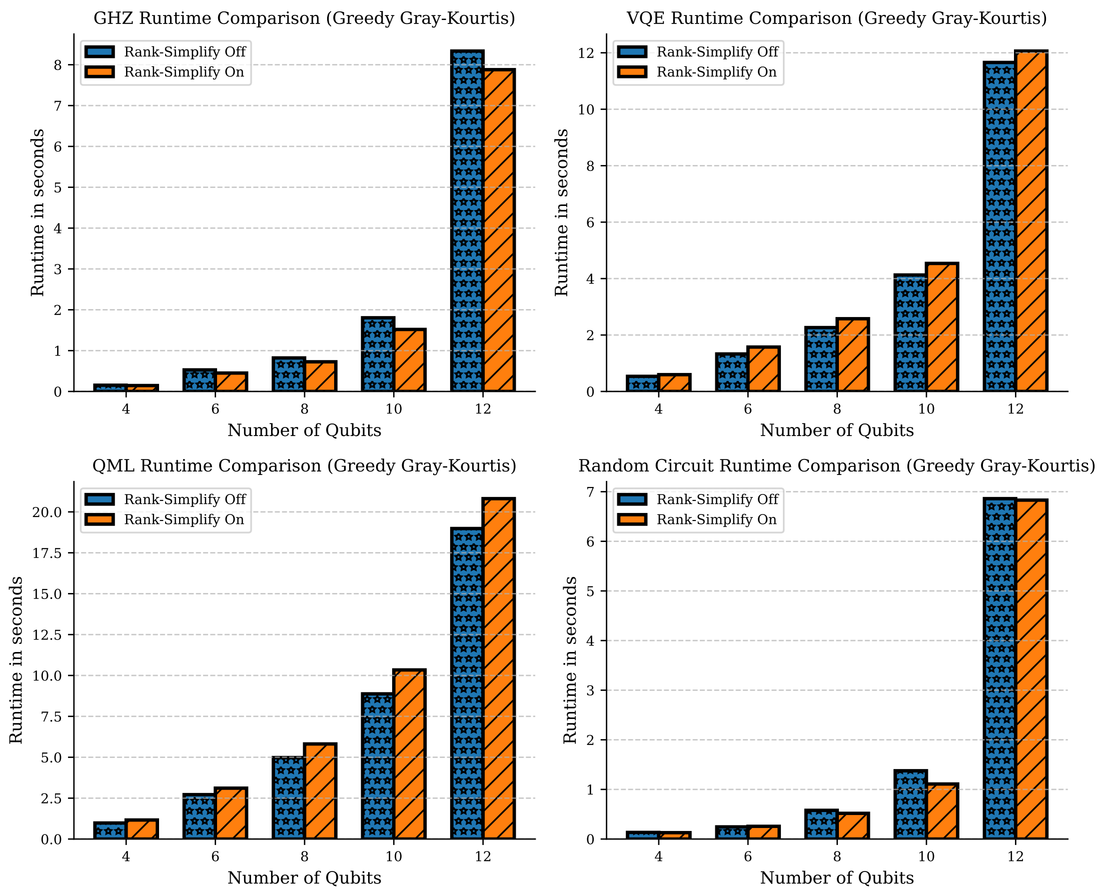


Figure 6.7: Impact on run-time for the rank simplification for Gray-Kourtis Greedy path-search with $alpha = 0.6$

7 Related Work

7.1 Optimizations

There is a lot of research focusing on efficient tensor network contractions. In the following, we present some of the optimizations present tensor network optimization present in recent literature.

1. **Simplifications**: Gray and Kourtis present a series of simplifications that can be done as a pre-processing step, before contraction path search. *Diagonal reduction* reduces the rank of a tensor, by replacing simple indices with hyper-indices, also enabling hypergraph optimizations. *Rank simplification* simplifies the network by contracting tensors of rank 1 and 2 with their neighbors without increasing their rank. *Split simplification* allows dividing a tensor in the network through SVD, for example, which could be useful to create certain tensor network states that are easier to contract. [GK21]

2. **Path-search Algorithms**: Gray and Kourtis [GK21] present a series of possible path-search algorithms. Exhaustive search, for example, uses a dynamic programming approach to build the contraction tree based on connected sub-graphs, which is used for the opt-einsum library. [GG18] They also present the option of using hyperparameters in heuristics for greedy path-search which could be tuned based on sampling many paths of a specific network, and subsequently choosing from a set of paths, given a probability determined by introducing a 'Boltzmann Factor', that configures the 'adventourosity' of the search. [GK21]

   Ibrahim et al. also present a novel approach of reducing the path-finding to a graph linear-ordering problem and a set of heuristics that could outperform the existing state-of-the-art libraries. [Ibr+22]

3. **Slicing**: Slicing is presented as an optimization in [GK21; Hua+21; Lee+24]. It allows dividing the computation of a tensor network into separate problems, which can considerably reduce the memory needed for the computation, as well as allow parallel execution. Lee et al. even introduce a design for executing the contraction in a distributed way [Lee+24].

4. **Tiling and Padding**: Katel et al. present an MLIR infrastructure for matrix-matrix multiplications optimized by using tiling and padding targeting GPUs specifically [KKB21]. Springer et al. introduce a design for high-performance tensor contractions using transposition and tiling. [SB17].

5. **Topology Manipulation**: Handschuh et al. present a way of manipulating the topology of the network by adding or removing edges to the graph. By doing this, one could create better conditions for path-finding algorithms or slicings. [Han12]

This literature inspired some of the optimizations included in CompTN like, rank simplification as a pre-processing step, tensor network slicing and two path-search algorithms. Other optimizations can be included in a future iteration, due to the extensibility of the infrastructure.

7.2 Library Solutions

The optimizations presented in Gray et al. [GK21] are implemented in cotengra and opt_einsum, python libraries specialized in tensor network contractions. These libraries are also used in Quimb, which is a python library for state-of-the-art quantum circuit computations. [Gra18; Gra24b; Gra24a]

7.3 Compiler Solutions

An approach to Tensor Algebra is shown in COMET [Tia+21]. They present an MLIR dialect and a DSL, focused on optimizing sparse tensor operations. The difference to our work is that CompTN works on an abstraction level higher, by optimizing contractions using the topology information of the network. Springer et al. [SHB16; SB17] present an compiler approach to tensor transposition. Efficient transposition of tensors, is beneficial to increase spacial locality for the dimensions being contracted over. Some of these optimizations could be included into CompTN in future iterations.

8 Conclusion and Outlook

8.1 Summary and Conclusion

This work is a first step into the direction of compiler based tensor network contractions. We present an MLIR Tensor Network dialect, to optimize the contractions through slicing, rank simplifications and different path search algorithms at the highest abstraction level. Subsequently, we can make use of the dialects included in MLIR to lower and optimize the program, and finally execute it. We also presented a set of Python bindings, which allow for the JIT-compilation of the contraction, and further manipulation of the resulting values through the python programming language.

The presented Tensor Network dialect is structured around the graphical and Einstein notations for tensor networks, where you define the tensors by giving them indices. This allows to quickly translate literature and domain specific computations into the Tensor Network Dialect. By Lowering into the Linalg and Arith dialects, we make use of the existing lowering passes for these dialects to reach the LLVM dialect and execute the program using MLIRs execution engine.

Our findings show that CompTN still requires more optimizations to be able to compete with the state-of-the-art tensor network libraries, but with this work we provide a proof-of-concept for creating a compiler based approach, as well as an extensible infrastructure to build upon.

8.2 Future Work

8.2.1 Path Reuse

One of the possible advantages of using a compiled solution for tensor network contractions, is path reuse. In future work, one could implement the reading of different values directly from memory instead of including them into the MLIR module, which could possibilitate to use the same program for multiple values. Another advantage of this, is that computing the optimal contraction path, which is normally the most expensive, could be computed with an amortized cost, if the contraction is executed multiple times.

8.2.2 Multithreading

During this work we presented a slicing pass, that divides the tensor network into multiple separate computations, that are added at the end to obtain the final result. The code repository also contains the necessary lowering pass to construct a module using the Async Dialect. In a future iteration of this project, one could set up the JIT execution engine to enable the parallel execution of the sliced computations.

8.2.3 GPU Execution

MLIR also provides the infrastructure to lower to the GPU dialect, which could target a series of GPUs to compute the tensor network contractions. The Graphical Processing Units are specialized in tensor computations, which could decrease the overall runtime of the contractions significantly. Using a compiled approach, for this, has the advantage that one could target a specific GPU architecture and uncover a series of low-level optimizations.

8.2.4 Extension to other fields

The CompTN infrastructure allows for the extension of the dialect to other fields where tensor networks are used. Some of the fields that could profit from a compiled approach to tensor networks include quantum computing, machine learning, chemistry or phyiscs.

One approach is to extend CompTN directly through the python frontend. An example can be found in the code used for executing the benchmarks for Chapter 6. There through python one could create quantum circuits using qiskit, and process them into CompTN through a short function. In the future, one could think of creating python libraries with the most common tensor networks for a domain, since we saw that constructing the module takes up negligible time.

Another approach, that makes use of the extensibility of the Tensor Network Compiler, is to build another domain specific dialect, for instance a quantum dialect [MN21], that could lower to the Tensor Network dialect and make use of its optimization passes. Creating another dialect on top, would also allow for more optimizations, than just doing it directly in the python program. This is because one can make use of information at a higher level, like the knowledge that a sequence of quantum gates can be simplified, which is not known at the tensor network level.

List of Figures

Bibliography

[AR] M. Amini and R. Riddle. *Building a Compiler with MLIR*. https://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf [Accessed 2024-06-24].

[BB17] J. Biamonte and V. Bergholm. *Tensor Networks in a Nutshell*. arXiv:1708.00006 [cond-mat, physics:gr-qc, physics:hep-th, physics:math-ph, physics:quant-ph]. July 2017.

[GG18] D. G. A. Smith and J. Gray. "opt_einsum - A Python package for optimizing contraction order for einsum-like expressions." In: *Journal of Open Source Software* 3.26 (June 2018), p. 753. ISSN: 2475-9066. DOI: 10.21105/joss.00753.

[GK21] J. Gray and S. Kourtis. "Hyper-optimized tensor network contraction." en. In: *Quantum* 5 (Mar. 2021). arXiv:2002.01935 [cond-mat, physics:physics, physics:quant-ph], p. 410. ISSN: 2521-327X. DOI: 10.22331/q-2021-03-15-410.

[Gra18] J. Gray. "quimb: A python package for quantum information and many-body calculations." In: *Journal of Open Source Software* 3.29 (Sept. 2018), p. 819. ISSN: 2475-9066. DOI: 10.21105/joss.00819.

[Gra24a] J. Gray. *jcmgray/cotengra*. original-date: 2019-01-22T16:22:26Z. Sept. 2024.

[Gra24b] J. Gray. *jcmgray/quimb*. original-date: 2015-12-09T14:02:41Z. Sept. 2024.

[Hai22] F. Haidar. *Scaling Quantum Circuit Simulation with NVIDIA cuTensorNet*. en-US. Mar. 2022.

[Han12] S. Handschuh. *Changing the topology of Tensor Networks*. arXiv:1203.1503 [math]. Mar. 2012.

[Hua+21] C. Huang, F. Zhang, M. Newman, X. Ni, D. Ding, J. Cai, X. Gao, T. Wang, F. Wu, G. Zhang, H.-S. Ku, Z. Tian, J. Wu, H. Xu, H. Yu, B. Yuan, M. Szegedy, Y. Shi, H.-H. Zhao, C. Deng, and J. Chen. "Efficient parallelization of tensor network contraction for simulating quantum computation." en. In: *Nature Computational Science* 1.9 (Sept. 2021). Publisher: Nature Publishing Group, pp. 578–587. ISSN: 2662-8457. DOI: 10.1038/s43588-021-00119-7.

[Ibr+22] C. Ibrahim, D. Lykov, Z. He, Y. Alexeev, and I. Safro. *Constructing Optimal Contraction Trees for Tensor Network Quantum Circuit Simulation*. en. arXiv:2209.02895 [quant-ph]. Sept. 2022.

[KKB21] N. Katel, V. Khandelwal, and U. Bondhugula. *High Performance GPU Code Generation for Matrix-Matrix Multiplication using MLIR: Some Early Results*. en. arXiv:2108.13191 [cs]. Aug. 2021.

[Lat+20] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. en. Feb. 2020.

[Lee+24] J. Lee, S. Gonzalez-Garcia, Z. Zhang, and H. Jeong. *Coded Computing Meets Quantum Circuit Simulation: Coded Parallel Tensor Network Contraction Algorithm*. en. arXiv:2405.13946 [cs, math]. May 2024.

[MLIa] MLIR. *MLIR Language Reference - MLIR*. `https://mlir.llvm.org/docs/LangRef` [Accessed: 2024-09-19].

[MLIb] MLIR. *Multi-Level IR Compiler Framework*. `https://mlir.llvm.org/` [Accessed: 2024-09-10].

[MN21] A. McCaskey and T. Nguyen. "A MLIR Dialect for Quantum Assembly Languages." In: *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Oct. 2021, pp. 255–264. DOI: `10.1109/QCE52317.2021.00043`.

[Pen71] R. Penrose. *Applications of negative dimensional tensors*. 1971.

[Ran+17] S.-J. Ran, E. Tirrito, C. Peng, X. Chen, G. Su, and M. Lewenstein. "Review of Tensor Network Contraction Approaches." In: (Aug. 2017).

[Ran+20] S.-J. Ran, E. Tirrito, C. Peng, X. Chen, L. Tagliacozzo, G. Su, and M. Lewenstein. *Tensor Network Contractions: Methods and Applications to Quantum Many-Body Systems*. en. Vol. 964. Lecture Notes in Physics. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-34488-7 978-3-030-34489-4. DOI: `10.1007/978-3-030-34489-4`.

[SB17] P. Springer and P. Bientinesi. *Design of a high-performance GEMM-like Tensor-Tensor Multiplication*. arXiv:1607.00145 [cs]. Nov. 2017.

[SHB16] P. Springer, J. R. Hammond, and P. Bientinesi. *TTC: A high-performance Compiler for Tensor Transpositions*. arXiv:1603.02297 [cs]. Mar. 2016.

[Sto24] Stoudenmire. *TensorNetwork/tensornetwork.org*. original-date: 2018-07-08T15:56:35Z. Sept. 2024.

[Tia+21] R. Tian, L. Guo, J. Li, B. Ren, and G. Kestor. *A High-Performance Sparse Tensor Algebra Compiler in Multi-Level IR*. en. arXiv:2102.05187 [cs]. Feb. 2021.