TU Dresden
Center for Advanding Electronics Dresden
Chair for Compiler Construction

Detection and exploitation of data-parallelism in assignments of multi-dimensional tensors

Bachelor Thesis

Submitted by	Til Jasper Ullrich
Date of submission	2018-08-21
Degree program	Informatik (Bachelor)
Matriculation number	4514861
1st referee	Prof. Dr.-Ing. Jeronimo Castrillon
2nd referee	Prof. Dr.-Ing. habil. Heiko Vogler
Supervisor	Dr. Norman Rink

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema “Detection and exploitation of data-parallelism in assignments of multi-dimensional tensors” vollkommen selbstständig verfasst und keine anderen außer den angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 21.08.2018

Til Jasper Ullrich

Abstract

This thesis studies data-parallelism in tensor assignments. Building on the domain specific language described in [7], an extension of the formal language semantics is proposed to detect so called compatible statements, which describes when a statement is data-parallel. Using a type system, the correctness of the extension is shown and a conjecture about the optimality is proposed. Applying the extension, two optimizations for exploiting the data-parallelism are described. These optimizations reduce the memory usage for computation, therefore reducing cache misses and improving runtime. The speedup which can be gained mostly depends on the complexity of the kernel and the size of the tensors. For simple kernels like multiplication of a vector with a scalar or elementwise multiplication of two vectors, a speedup of up to 2x can be achieved. For more complex kernels like a kernel containing matrix-matrix multiplication, the speed difference is a few percent. Additionally, a kernel called interpolation consisting of incompatible statements is analysed to check whether a similar optimization can be applied. The result is that while the optimization does not result in a speedup, similar improvements might be possible. Finally, in order to gain a better understanding of what effect the optimizations might have, different kernels are analysed regarding the data size at which parallelism makes sense.

Contents

1 Introduction

Tensors are a way of storing multi-dimensional data and making calculations on it. They are the generalization of vectors and matrices to higher ranks. For example, a vector can be seen as a tensor of rank 1 and a matrix as a tensor of rank 2. A tensor of rank 3 would therefore be a cube of numbers, or a 3-dimensional array, with some operations defined.

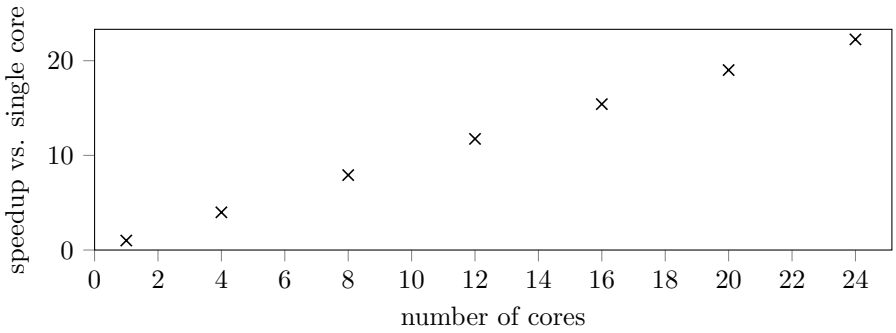
Tensors have a wide range of applications. In scientific computing and specifically physics, tensors are used for simulations (continuum mechanics), electromagnetism, general relativity and quantum mechanics (also quantum chemistry and quantum computing). In machine learning, neural networks are usually computed using tensor operations and in general, multi-dimensional data like videos are modelled as tensors of rank 3 or higher. Also, tensors are used in computer vision, image processing and, as vectors and matrices are a special case of tensors, everywhere these are used.

One problem with a lot of tensor applications is the large runtime of computations. Decreasing this runtime is done through a lot of optimizations which, as the programmer can't be reasonable expected to try to apply all of them, are usually included in the compiler. A specific method which can improve runtime significantly, but which is rarely included in the compiler, is parallelization.

1.1 Parallelization

The amount of runtime improvement gained by parallelization usually varies with how good the problem can be parallelized, the number of cores and the data sizes. As tensor computations consist of a large number of independent small computations (the elements), they are usually very good parallelizable and one can expect a linear speedup in the number of cores by distributing an equal amount of elements to every core. Figure 1 shows a simple scalar multiplication and the corresponding linear speedup gained from parallelization as measured by the author. N denotes the size of the vector and REP denotes how often the kernel has been measured in order to stabilize variance ($\sigma^2 < 0.0009$) and decrease standard error.

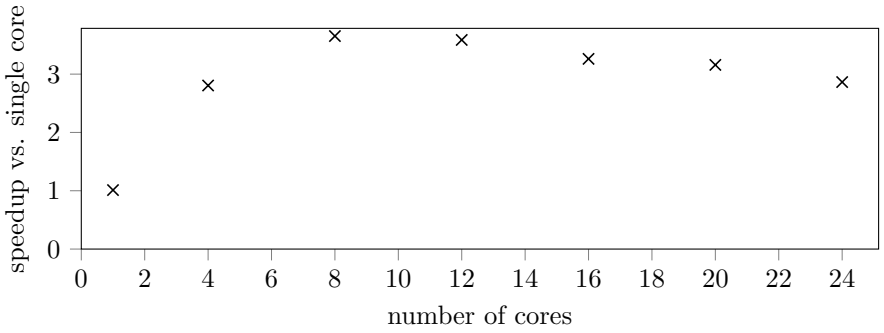
Figure 1: Scalar multiplication, N = 1 000 000, REP = 2^7



This linear speedup can be achieved for a lot of kernels. However, as the number of cores increase, so does the overhead of threads. This means that for smaller data sizes, the speedup can be limited by this overhead. Decreasing the

data size N from 1 000 000 to 10 000 shows this thread overhead as can be seen from Figure 2. In this case the speedup is not a linear function of the number of cores but gets saturated at 8 cores and decreases after that.

Figure 2: Scalar multiplication, N = 10 000, REP = 2^{14}



As parallelization in general can be difficult to apply, it is usually not used by the compiler but by the programmer. However tensors represent a particular exception and are often "embarrassingly parallel", which means that they can be parallelized with little to no effort. It is therefore possible for the compiler to assist the programmer in parallelizing the computation. While this can be done with general-purpose languages like C, detecting and optimizing the tensor computations is more difficult than with a domain specific language (DSL) specifically created for the problem, i.e. tensor computations.

This thesis builds upon an existing DSL defined in [7], which compiles to parallel C code. By extending an existing type system, we try to make more statements about programs in the language. We show the correctness of the extension formally and how it can be used for practical optimizations. Because the DSL is not the first of its kind, the next section presents alternative DSLs and the corresponding compilers. The DSL which will be used in this thesis will be presented in Section 2.2.

1.2 Existing DSLs and compilers

While these projects presented in this section mostly include all necessary tensor operations, they often have some primary use case. For example, while Tensorflow [5] could be used for a wide range of applications[1], it is mostly used for machine learning and is also marketed as such. The tools in this section are therefore divided by their primary use-case.

Scientific Computing: In [8], the authors designed a DSL called CFDlang similar to the DSL from [7] "for expressing and optimizing performance-critical operations in fluid dynamics simulations". The problem described mainly uses comparably small data structures, e.g. in which L1 cache size is not a limiting factor. The authors created a compiler for the language which generates C code. The compiler includes SIMD pragmas for parallelization and optimizes the code using algebraic transformations. The Tensor Contraction Engine [1] is another compiler from a Mathematica-style language to FORTRAN mainly designed for a class of quantum chemistry computations. The compiler also uses algebraic

[1] `https://www.tensorflow.org/tutorials/pdes`

transformations and optimizes for memory usage by fusing loops. ITensor [2] is a C++ library used in quantum physics and supports sparse tensors.

Machine Learning: Tensorflow [5] is a framework for machine learning which compiles graphs built in one of the languages providing an API (often python) to C++. While the system currently does not optimize across different operations, there exists an experimental compiler, XLA [11], which includes these optimiziations by using an IR. Besides AOT compilation (ahead of time, meaning before runtime), XLA can also use JIT compilation (just in time, meaning at runtime) and therefore enables specializing for actual runtime dimensions. Tensor Comprehensions [10] is a C++ library and mathematical language for generating efficient CUDA kernels. Notable is that, among many other optimization techniques, it uses evolutionary search to generate multiple alternative implementations and selects the best performing ones for the current hardware.

General Purpose: The Tensor Algebra Compiler [3] is a system for generating kernels from C++ and includes optimizations for both dense and sparse tensors. The High-Performance Tensor Transpose library [9] is a library for C++, C and python focusing specifically on tensor transpositions. As layout of data in memory can have a large impact on the runtime due to memory locality, these transpositions can be used to change the layout before applying more expensive operations, e.g. fast Fourier transform.

2 Background

2.1 Tensors and tensor operations

As tensors also exist for higher ranks than 1 (vectors) and 2 (matrices), it is also necessary to define operations for these higher ranks. These operations must generalize to all ranks (including 1 and 2) and it should be possible to build existing operations like matrix multiplication from these more general operations. Usually, the following operations, which are also used in the DSL from [7], are defined.

Scalar operations, which are defined for a scalar k and another tensor u. The scalar is a tensor of rank 0, i.e. a single element. For the language described here, only multiplication and division are used and notated as $k * u$ or u/k.

Elementwise operations, which are defined for two tensors u and v which have the same rank and dimensions. They are notated as $u \odot v$ where $\odot \in \{+, -, *, /\}$ and defined as $(u \odot v)_i = u_i \odot v_i$.

Tensor product, also called outer product and usually denoted as $u = v \otimes w$. It creates a new tensor with rank equal to the sum of two input tensors and is defined as $u_{ij} = v_i \cdot w_j$. In the DSL used here, it will be denoted as $u = v\#w$.

Tensor contraction $u = v$ **.** $[m\ n]$, which is the generalization of the trace of a matrix.[2] m and n denote two dimensions of the tensor v. The result of the contraction is a tensor which has its rank reduced by 2 and where every element is calculated as the trace (sum over diagonal) of the matrix spanned by the dimensions m and n.

[2]Normally, contraction is defined for two tensors instead of one and represents the generalization of matrix multiplication. However, defining it for one tensor represents no loss of generality as general tensor contraction can be built by applying the tensor product beforehand. While this would result in a large slowdown at runtime, this will not be important for the topic presented in this thesis.

Transposition $u = v \hat{} [m\ n]$, which switches two dimensions with each other. It is defined as $u_{i_1,\ldots,i_m,\ldots,i_n,\ldots,i_k} = v_{i_1,\ldots,i_n,\ldots,i_m,\ldots,i_k}$. In the case of a matrix $u = v^T$, the dimensions are implicit as there are only two, but in general they must be specified.

These operations can then be used to create other operations. For example matrix-vector multiplication can be calculated as $u = (v\#w)$. [2 3]. The same equation is true for matrix-matrix multiplication. Convolution can also be expressed as matrix-matrix multiplication which solves the problem of irregular memory access when doing convolution directly. This approach is used by multiple libraries [4].

2.2 A language for tensor manipulation

In [7], the described DSL contains all general tensor operations shown in the previous section with the same syntax. Besides other various useful properties for the language, [7] defines a type system for deriving the rank and dimensions of a tensor as well as a function for calculating tensor elements. As these two properties will be important for the topic presented here and in order to keep this self-contained, a short overview is given.

A program in the DSL contains a list of declarations followed by a list of statements. The declarations define all tensors which will be used including their dimensions. As an example, the following program computes a tensor product followed by a contraction, i.e. a matrix-vector multiplication.

```
var u : [300]
var v : [300 400]
var w : [400]

u = (v # w) . [2 3]
```

The first three lines are declarations and define `u`, `v`, `w` to be tensors of dimensions 300, 300 x 400 and 400 respectively. The last line is a statement and assigns the result of a tensor product of `v` and `w` followed by contraction to `u`.

In order to check the validity of statements, meaning that the types on the left and right side are equal, the type system defines rules including, but not limited to the ones in Figure 3. Using type derivations, often notated in form of a tree as shown in Figure 4, the rules can be used to find e.g. the type of `(v # w) . [2 3]` and validate that it is equal to the type of `u`.

First, `s-var` creates the static context Γ from the type declarations. Given some (possibly empty) sequence of declarations, this rule allows for deriving Γ for the sequence extended by one declaration. It does so by taking the declared type $[d_1, \ldots, d_k]$ for the variable and stores it in Γ through $\vdash_s$, e.g. $\Gamma(\mathrm{u}) = (300)$. If the variable is already stored in Γ, i.e. $\mathrm{x} \in \mathrm{dom}(\Gamma)$, then the statement sequence can't be validated. The rule `t-var` then starts type inference by using this context for all variables in an expression. For the previous example, this would result in $\Gamma \vdash \mathrm{u} : (300)$, $\Gamma \vdash \mathrm{v} : (300, 400)$ and $\Gamma \vdash \mathrm{w} : (400)$. The types are then propagated through rules like `t-prod` which are defined for every operation mentioned in Section 2.1. Using `t-prod` result in $\Gamma \vdash \mathrm{v} \# \mathrm{w} : (300, 400, 400)$, `t-paren` in $\Gamma \vdash (\mathrm{v} \# \mathrm{w}) : (300, 400, 400)$ and `t-contr` in $\Gamma \vdash (\mathrm{v} \# \mathrm{w})$. [2 3] : (300). Finally, the statement is validated

$$\frac{decl^* \vdash_s \Gamma \quad \mathrm{x} \notin \mathrm{dom}(\Gamma)}{decl^* \ \mathbf{var}\ \mathrm{x}\ \mathbf{:}\ [d_1, \ldots, d_k] \vdash_s \Gamma\{\mathrm{x} \mapsto (d_1, \ldots, d_k)\}}\ \texttt{s-var}$$

$$\frac{\mathrm{x} \in \mathrm{dom}(\Gamma) \quad \mathbf{t} = \Gamma(\mathrm{x})}{\Gamma \vdash \mathrm{x} : \mathbf{t}}\ \texttt{t-var}$$

$$\frac{\Gamma \vdash e0 : (d_{01}, \ldots, d_{0k}) \quad \Gamma \vdash e1 : (d_{11}, \ldots, d_{1l})}{\Gamma \vdash e0 \# e1 : (d_{01}, \ldots, d_{0k}, d_{11}, \ldots, d_{1l})}\ \texttt{t-prod}$$

$$\frac{\Gamma \vdash e : \mathbf{t}}{\Gamma \vdash (e) : \mathbf{t}}\ \texttt{t-paren}$$

$$\frac{\Gamma \vdash e : (d_1, \ldots, d_m, \ldots, d_n, \ldots, d_k) \quad d_m = d_n}{\Gamma \vdash e\ \mathbf{.}\ [m\ n] : (d_1, \ldots, \widehat{d_m}, \ldots, \widehat{d_n}, \ldots, d_k)}\ \texttt{t-contr}$$

$$\frac{\mathrm{x} \in \mathrm{dom}(\Gamma) \quad \mathbf{t} = \Gamma(\mathrm{x}) \quad \Gamma \vdash e : \mathbf{t}}{\Gamma \vdash \mathrm{x} = e\ \mathrm{ok}}\ \texttt{ok-stmt}$$

$$\frac{\begin{array}{c}\mathrm{x} \in \mathrm{dom}(\Gamma) \quad \mathbf{t} = \Gamma(\mathrm{x}) \\ \forall_{\mathbf{i} \leq \mathbf{t}}\ \mathrm{x}_{\mathbf{i}} \in \mathrm{dom}(\mu) \wedge r_{\mathbf{i}} = \mathrm{eval}_{\Gamma,\mu}(e_{\mathbf{i}})\end{array}}{\langle \mu, \mathrm{x} = e \rangle \rightarrow_\Gamma \langle \mu\{\forall_{\mathbf{i} \leq \mathbf{t}} \mathrm{x}_{\mathbf{i}} \mapsto r_{\mathbf{i}}\}, \emptyset \rangle}\ \texttt{ev-stmt}$$

Figure 3: Example rules from [7]

$$\dfrac{\mathrm{u} \in \mathrm{dom}(\Gamma) \quad (300) = \Gamma(\mathrm{u}) \quad \dfrac{\dfrac{\dfrac{\mathrm{v} \in \mathrm{dom}(\Gamma) \quad (300, 400) = \Gamma(\mathrm{v})}{\Gamma \vdash \mathrm{v} : (300, 400)}\ \texttt{t-var} \quad \dfrac{\mathrm{w} \in \mathrm{dom}(\Gamma) \quad (400) = \Gamma(\mathrm{w})}{\Gamma \vdash \mathrm{w} : (400)}\ \texttt{t-var}}{\Gamma \vdash \mathrm{v} \ \# \ \mathrm{w} : (300, 400, 400)}\ \texttt{t-prod}}{\Gamma \vdash (\mathrm{v} \ \# \ \mathrm{w}) : (300, 400, 400)}\ \texttt{t-paren}}{\Gamma \vdash (\mathrm{v} \ \# \ \mathrm{w})\ .\ [2\ 3] : (300)}\ \texttt{t-contr}}{\Gamma \vdash \mathrm{u} = (\mathrm{v} \ \# \ \mathrm{w})\ .\ [2\ 3]\ \mathrm{ok}}\ \texttt{ok-stmt}$$

Figure 4: Type derivation of `(v # w) . [2 3]`

through `ok-stmt` which only allow assignments $\mathrm{x} = e$ if the type of x is equal to the type of e. As in this case the type of `u` is equal to the type of the right side, this results in $\Gamma \vdash \mathrm{u} = (\mathrm{v} \ \# \ \mathrm{w}) \ . \ [2 \ 3]$ ok. These valid statements are then used further to validate statement sequences and the whole program using similar rules.

After the type checking has validated the program, the expressions are evaluated using an $\mathrm{eval}_{\Gamma,\mu}(.)$ function for calculating new values from already calculated ones. The $\mathrm{eval}_{\Gamma,\mu}(.)$ function uses a dynamic store μ to store its results and use them for following statements through $\mathrm{eval}_{\Gamma,\mu}(\mathrm{x}_{i_1,\ldots,i_k}) = \mu(\mathrm{x}_{i_1,\ldots,i_k})$. Rules like $\mathrm{eval}_{\Gamma,\mu}(e0\#e1_{i_{01},\ldots,i_{0k},i_{11},\ldots,i_{1l}}) = \mathrm{eval}_{\Gamma,\mu}(e0_{i_{01},\ldots,i_{0k}}) \cdot \mathrm{eval}_{\Gamma,\mu}(e1_{i_{11},\ldots,i_{1l}})$, which describe how a single element of a tensor is evaluated using its subexpressions, allow for calculation of new values. For tensor product, this means that the tensor resulting from $e0\#e1$ can be calculated at index $i_{01}, \ldots, i_{0k}, i_{11}, \ldots, i_{1l}$ by multiplying two elements of $e0$ and $e1$ at the corresponding indices. The result is stored in μ by using `ev-stmt` from Figure 3 which describes how the store changes after a single statements. Corresponding rules also exist for a statement sequence and the whole program.

An example of evaluating the statement $u = (v\#w) \ . \ [2 \ 3]$ is given. u, v and w are again tensors of rank 1, 2, 1 respectively and the expression on the right side is evaluated for the index (5), i.e. the 5th element of the vector u is calculated. Using the $\pi_i(\mathbf{t})$ function from [7], which returns the size of the dimension i of $\mathbf{t}$, let $\pi_1(\Gamma(u)) = \pi_1(\Gamma(v)) = \pi_2(\Gamma(v)) = \pi_1(\Gamma(w)) = 7$, i.e. every dimension of all tensors is 7. Then

$$
\begin{aligned}
&\mathrm{eval}_{\Gamma,\mu'}((v\#w) \ . \ [2 \ 3]_5) \\
&\quad = \textstyle\sum_{l=1}^{7} \mathrm{eval}_{\Gamma,\mu'}((v\#w)_{5,l,l}) \\
&\quad = \textstyle\sum_{l=1}^{7} \mathrm{eval}_{\Gamma,\mu'}(v\#w_{5,l,l}) \\
&\quad = \textstyle\sum_{l=1}^{7} (\mathrm{eval}_{\Gamma,\mu'}(v_{5,l}) \cdot \mathrm{eval}_{\Gamma,\mu'}(w_l)) \\
&\quad = \textstyle\sum_{l=1}^{7} (\mu(v_{5,l}) \cdot \mu(w_l))
\end{aligned}
$$

The μ's are then replaced with the stored values of v and w at the corresponding indices. The result from summing over all the values is stored in $\mu(u_5)$ and can be used for further calculations.

3 Compatible statements

The programs from the DSL are evaluated by executing every statement in a specific order (top to bottom). Every statement has the form $x = e$, meaning that e is evaluated and the result is assigned to x. For example scalar multiplication can be written as $x = 3 * x$ and matrix-vector multiplication as $x = uv$. These statements can be differentiated depending on whether the variable x on the left size also occurs on the right side, i.e. $x = uv$ vs. $x = ux$.

This difference has influence on how the statements are evaluated at runtime. In the case of $x = uv$, assuming that the three variables do not alias (guaranteed by the DSL) and memory has already been allocated, the assignment can be done directly by evaluating every element of x using u, v and writing it to the corresponding location in memory. However for $x = ux$, this is not possible. This is because for matrix-vector multiplication, in order to calculate one new

element of x, all old elements of x are needed. This means that using the same memory for both reading and writing would give the wrong result. In the case of sequential execution, the first old element of x would not be readable after the first new element has been calculated as it has been overwritten, which means that the second new element cannot be calculated. For parallel execution, the exact order of read- and write accesses for tensor elements from multiple cores is not deterministic. In both cases, in order for the calculation to be correct, all read accesses have to be done before the results are written. This can be achieved by writing the result to another memory location.

However, this approach has two main disadvantages. First, writing the result to another memory location increases the amount of memory which is used. While the memory can be freed after the statement has been executed, it is still larger during the calculation. As a result, it could happen that either the available memory is exceeded (e.g. hard drive) such that the calculation can't be completed or that slower memory must be used (e.g. L2 instead of L1 cache) which means that the calculation might be slower. Second, if the result is expected to be at the same memory location as before, it has to be copied back. This adds further runtime additionally to the first problem.

The main goal of this thesis is to detect statements where x occurs on the right side but where the same memory can be used for both reading and writing. These statements shall be called "compatible statements". Statements where x does not occur on the right side can be handled trivially, although it may be possible to rewrite and optimize them further, re-using memory of other variables.

Specifically, the statements of interest are those where, in order to calculate one new element of x, no old elements other than the element itself are needed. An example of a compatible statement is $x = 3 * x$. Calculating the first new element of x only requires the first old element of x. In other words, even though the old x is needed for calculating the new x, the same memory can be used because after overwriting an old element, it won't be needed anymore.

Definition 1 (Compatible statement, informal)**.** A statement $x = e$ is compatible, if for the calculation of one element of x, only the same element is needed for every occurence of x in e.

Once a statement has been found to be compatible, the same space can be used for reading and writing. Otherwise it can still be calculated using different memory. Note that while it would be possible to do the detection at runtime, doing it at compile-time improves performance and enables further optimizations, especially if the compiler generates parallel code.

Also note that in-place operations are already used and many libraries and frameworks feature corresponding operations, functions or arguments. However, this optimization is done manually by the programmer and not the compiler.

3.1 Detecting compatible statements

Compatible statements generally consist of variables different from x as well as scalar- and elementwise operations if x is used with them. An example of a compatible statement which combines all three of these components is $x = x + w * (uv * x)$ where u is a row vector, v is a column vector and w is a tensor of same rank and dimensions as x. Calculating uv results in a scalar

so $uv * x$ is a scalar multiplication which poses no problem. Also $x + w$ is an elementwise addition and therefore ok. It should then be intuitive that an elementwise multiplication of these two expressions and an assignment to x results in a compatible statement.

From this example, one might try to approach the problem of detecting compatible statements in a similar fashion, creating a tree of expressions and marking a node like $x + w * (uv * x)$ as ok if the children, in this example $x + w$ and $(uv * x)$ are ok and the operation itself, i.e. scalar multiplication is ok.

However, this approach fails if tensor product and contraction are involved in the expression. As an example, the statement for matrix-vector multiplication $x = (u\#x)$. [2 3] where u is a matrix and v a vector is not compatible. However, swapping u and x in the expression to $x = (x\#u)$. [2 3] makes the statement compatible. While it might not be immediately obvious, the second statement is a scalar multiplication of x with the trace of u. The two dimensions, 2 and 3 are the two dimensions of u so the contraction ignores x for calculation and contracts only over u, resulting in the trace. The tensor product and contraction can also be combined to more complex statements, requiring another approach for detection of compatible statements.

The intuition from Definition 1 that "only the same element is needed for every occurence of x in e" can also be formulated as "x may only be accessed at the same position currently assigned to". Using this formulation already gives an intuition on how compatible statements can be detected. In order to check if a statement is compatible, every occurence of x in the evaluated expression has to be checked with respect to the indices used for access. If x is only accessed at the same index currently assigned to, the statement it compatible and incompatible otherwise. While at runtime concrete numbers are used for the indices, at compile time the check can be done using the index variable itself.

4 Extension of the DSL

In order to implement this idea, the DSL described in [7] can be extended to account for compatible statements. This is done by augmenting the context Γ with the multi index (like `[i][j][k]` or $(\iota_1, \ldots, \iota_k)$) used to access the variable which is assigned to. Without loss of generality, let this variable be x and the index be $\bar{\iota} = (\iota_1, \ldots, \iota_k)$. The new context is then denoted as Γ;x;$\bar{\iota}$.

This augmented context determines whether an expression e can be accessed at a specific index in order for the statement to be compatible. For example $\Gamma;x;\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ can be read as "If x is the variable on the left side of the assignment and accessed at $\bar{\iota}$ and e is somewhere on the right side, e can be accessed at $\bar{\kappa}$ and the statement is compatible".

In order to proof later, that a type system using this context will only detect statements which are compatible, a formal definition of compatible statements is needed. The intuition that "x may only be accessed at the same index it is assigned to" can be expressed by using a function similar to $\text{eval}_{\Gamma,\mu}(.)$ with the difference of removing μ. The reason for this is that μ operates on concrete numbers like (3,4,5) instead of abstract indices like `[i][j][k]` or $(\iota_1, \iota_2, \iota_3)$ which are important here. To achieve this, $\text{eval}_{\Gamma,\mu}(.)$ can be split into two new functions, $\text{eval}_{\Gamma}(.)$ and $\text{eval}_{\mu}(.)$.

The definition of $\text{eval}_{\Gamma}(.)$ is shown in Figure 5 and very similar to $\text{eval}_{\Gamma,\mu}(.)$.

$$\text{eval}_\Gamma(\text{x}_{\iota_1,\ldots,\iota_k}) = \mu'(\text{x}_{\iota_1,\ldots,\iota_k})$$

$$\text{eval}_\Gamma((e)_{\iota_1,\ldots,\iota_k}) = \text{eval}_\Gamma(e_{\iota_1,\ldots,\iota_k})$$

$$\text{eval}_\Gamma(e0\#e1_{\iota_{01},\ldots,\iota_{0k},\iota_{11},\ldots,\iota_{1l}}) = \text{eval}_\Gamma(e0_{\iota_{01},\ldots,\iota_{0k}}) \cdot \text{eval}_\Gamma(e1_{\iota_{11},\ldots,\iota_{1l}})$$

$$\text{eval}_\Gamma(e \text{ ^ } [m\ n]_{\iota_1,\ldots,\iota_m,\ldots,\iota_n,\ldots,\iota_k}) = \text{eval}_\Gamma(e_{\iota_1,\ldots,\iota_n,\ldots,\iota_m,\ldots,\iota_k})$$

$$\text{eval}_\Gamma(e0 \mathbf{.} [m\ n]_{\iota_1,\ldots,\widehat{\iota_m},\ldots,\widehat{\iota_n},\ldots,\iota_k}) = \sum_{l=1}^{\pi_m(\mathbf{t})} \text{eval}_\Gamma(e_{\iota_1,\ldots,\iota_{m-1},l,\iota_{m+1},\ldots,\iota_{n-1},l,\iota_{n+1},\ldots,\iota_k}) \quad \text{where } \Gamma \vdash e : \mathbf{t}$$

$$\text{eval}_\Gamma(e0 \odot e1_{\iota_1,\ldots,\iota_k}) = \begin{cases} \text{eval}_\Gamma(e0) \cdot \text{eval}_\Gamma(e1_{\iota_1,\ldots,\iota_k}) \text{ if } \Gamma \vdash e0 : () \text{ and } \odot \equiv * \\ \text{eval}_\Gamma(e0_{\iota_1,\ldots,\iota_k}) \ / \ \text{eval}_\Gamma(e1) \text{ if } \Gamma \vdash e1 : () \text{ and } \odot \equiv / \\ \text{eval}_\Gamma(e0_{\iota_1,\ldots,\iota_k}) \odot \text{eval}_\Gamma(e1_{\iota_1,\ldots,\iota_k}) \text{ otherwise} \end{cases}$$

Figure 5: Definition of $\text{eval}_\Gamma(.)$

The difference is that while $\text{eval}_{\Gamma,\mu}(.)$ takes an expression and a concrete index like $\text{eval}_{\Gamma,\mu}((\text{u+v})_{3,4,5})$ and returns a number like 13, $\text{eval}_\Gamma(.)$ takes an expression and an abstract index like $\text{eval}_\Gamma((\text{u} + \text{v})_{\iota_1,\iota_2,\iota_3})$ and returns an expression like $\mu'(\text{u}_{\iota_1,\iota_2,\iota_3}) + \mu'(\text{v}_{\iota_1,\iota_2,\iota_3})$ where μ' is a variable. The expression can then be evaluated through $\text{eval}_\mu(.)$ for a concrete index like $(3, 4, 5)$ by setting $\mu' = \mu$ and $\iota_1 = 3, \iota_2 = 4, \iota_3 = 5$. Using this $\text{eval}_\Gamma(.)$ function, compatible statements can then be defined formally.

Definition 2 (Compatible statement)**.** Let $\text{x} = e$ be a statement and Γ be a context with $\text{x} \in \text{dom}(\Gamma)$. Also let $\bar{\iota} = (\iota_1, \ldots, \iota_k)$ be a new multi-index with $k = \text{rank}(\Gamma(\text{x}))$. The statement is called compatible, if for every $\mu'(\text{x}_{\bar{\kappa}})$ which results from $\text{eval}_\Gamma(e_{\bar{\iota}})$, the condition $\bar{\kappa} = \bar{\iota}$ is fulfilled.

This definition expresses the intuition that in a statement $\text{x} = e$, every occurence of x must only be accessed at the same index as on the left side.

Using the augmented context, the type system can then be extended using rules similar to the unaugmented context. The resulting rules are shown in Figure 6. After that, the rule `ok-c-stmt` from Figure 7 demands that, in order for a statement $\text{x} = e$ to be compatible according to the type system, it must be possible to access e at the same index as x according to the type system. In order to derive a statement of this form, one has to start at the rules `tc-var-xy` or `tc-var-xx` depending on the variable. These rules express, that every variable except x may be accessed at any index and x may only be accessed at the same index used on the left side of the assignment. The other rules then express how one can combine the statements to derive assumptions about the access of more complex expressions. The rule `tc-contr` introduces a new index for summing over the diagonal.

In order to give an example of how this detection would work, consider the statement $x = (x\#y)$. [2 3] where x is a vector and y a matrix. Showing that this statement is compatible is done by checking if $\Gamma \vdash \text{x} = (\text{x\#y})$. [2 3] ok has a derivation according to `ok-c-stmt` from Figure 7. The derivation tree for this is shown in Figure 8.

$$\frac{\mathrm{y} \in \mathrm{dom}(\Gamma) \quad \mathbf{t} = \Gamma(\mathrm{y}) \quad k = \mathrm{rank}(\mathbf{t}) \quad \mathrm{y} \neq \mathrm{x}}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{y} : (\mathbf{t}, \bar{\kappa}) \text{ for any } \bar{\iota}, \bar{\kappa} = (\kappa_1, \ldots, \kappa_k)} \texttt{ tc-var-xy}$$

$$\frac{\mathrm{x} \in \mathrm{dom}(\Gamma) \quad \mathbf{t} = \Gamma(\mathrm{x})}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{x} : (\mathbf{t}, \bar{\iota})} \texttt{ tc-var-xx}$$

$$\frac{\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c (e) : (\mathbf{t}, \bar{\kappa})} \texttt{ tc-paren}$$

$$\frac{\begin{array}{c}\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : ((d_{01}, \ldots, d_{0k}), (\kappa_{01}, \ldots, \kappa_{0k})) \\ \Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : ((d_{11}, \ldots, d_{1l}), (\kappa_{11}, \ldots, \kappa_{1l}))\end{array}}{\begin{array}{c}\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 \# e1 : (\mathbf{t}', \bar{\kappa}') \\ \mathbf{t}' = (d_{01}, \ldots, d_{0k}, d_{11}, \ldots, d_{1l}) \\ \bar{\kappa}' = (\kappa_{01}, \ldots, \kappa_{0k}, \kappa_{11}, \ldots, \kappa_{1l})\end{array}} \texttt{ tc-prod}$$

$$\frac{\begin{array}{c}\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa}) \\ \mathbf{t} = (d_1, \ldots, d_m, \ldots, d_n, \ldots, d_k) \\ \bar{\kappa}' = (\kappa_1, \ldots, \kappa_m, \ldots, \kappa_n, \ldots, \kappa_k)\end{array}}{\begin{array}{c}\Gamma;\mathrm{x};\bar{\iota} \vdash_c e \mathbin{\hat{}} [m\ n] : (\mathbf{t}', \bar{\kappa}') \\ \mathbf{t} = (d_1, \ldots, d_n, \ldots, d_m, \ldots, d_k) \\ \bar{\kappa}' = (\kappa_1, \ldots, \kappa_n, \ldots, \kappa_m, \ldots, \kappa_k)\end{array}} \texttt{ tc-trans}$$

$$\frac{\begin{array}{c}\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa}) \\ \mathbf{t} = (d_1, \ldots, d_m, \ldots, d_n, \ldots, d_k) \text{ with } d_m = d_n \\ \bar{\kappa} = (\kappa_1, \ldots, \kappa_m, \ldots, \kappa_n, \ldots, \kappa_k) \text{ with } \kappa_m = \kappa_n \\ \kappa_m \text{ a fresh index}\end{array}}{\begin{array}{c}\Gamma;\mathrm{x};\bar{\iota} \vdash_c e \mathbin{.} [m\ n] : (\mathbf{t}', \bar{\kappa}') \\ \mathbf{t} = (d_1, \ldots, \widehat{d_m}, \ldots, \widehat{d_n}, \ldots, d_k) \\ \bar{\kappa} = (\kappa_1, \ldots, \widehat{\kappa_m}, \ldots, \widehat{\kappa_n}, \ldots, \kappa_k)\end{array}} \texttt{ tc-contr}$$

$$\frac{\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}, \bar{\kappa}) \quad \Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}, \bar{\kappa}) \quad \odot \in \{+, -, *, /\}}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 \odot e1 : (\mathbf{t}, \bar{\kappa})} \texttt{ tc-elem}$$

$$\frac{\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : ((), ()) \quad \Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}_1, \bar{\kappa})}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 * e1 : (\mathbf{t}_1, \bar{\kappa})} \texttt{ tc-smul}$$

$$\frac{\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}) \quad \Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : ((), ())}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0/e1 : (\mathbf{t}_1, \bar{\kappa})} \texttt{ tc-sdiv}$$

Figure 6: Compatible expression typing

$$\frac{\begin{array}{c}\mathrm{x} \in \mathrm{dom}(\Gamma) \quad \mathbf{t} = \Gamma(\mathrm{x}) \\ k = \mathrm{rank}(\mathbf{t}) \quad \bar{\iota} = (\iota_1, \ldots, \iota_k) \\ \Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\iota})\end{array}}{\Gamma \vdash \mathrm{x} = e \text{ ok}}\ \texttt{ok-c-stmt}$$

Figure 7: Well-formedness of compatible statements.

$$\frac{\mathrm{x} \in \mathrm{dom}(\Gamma) \quad \mathbf{t}_1 = \Gamma(\mathrm{x})}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{x} : (\mathbf{t}_1, \bar{\iota})}\ \texttt{tc-var-xx} \qquad \frac{\mathrm{y} \in \mathrm{dom}(\Gamma) \quad \mathrm{y} \neq \mathrm{x} \quad \mathbf{t}_2 = (d_2, d_2) = \Gamma(\mathrm{y})}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{y} : (\mathbf{t}_2, \bar{\kappa})}\ \texttt{tc-var-xy}$$

$$\frac{\text{for any } \bar{\kappa} = (\kappa_2, \kappa_3)}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{x}\#\mathrm{y} : ((d_1, d_2, d_2), (\iota_1, \kappa_1, \kappa_1))}\ \texttt{tc-prod}$$

$$\frac{\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{x}\#\mathrm{y} : ((d_1, d_2, d_2), (\iota_1, \kappa_1, \kappa_1))}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c (\mathrm{x}\#\mathrm{y}) : ((d_1, d_2, d_2), (\iota_1, \kappa_1, \kappa_1))}\ \texttt{tc-paren}$$

$$\frac{\mathrm{x} \in \mathrm{dom}(\Gamma) \quad \mathbf{t}_1 = \Gamma(\mathrm{x}) \qquad \Gamma;\mathrm{x};\bar{\iota} \vdash_c (\mathrm{x}\#\mathrm{y}) : ((d_1, d_2, d_2), (\iota_1, \kappa_1, \kappa_1))}{\Gamma;\mathrm{x};\bar{\iota} \vdash_c (\mathrm{x}\#\mathrm{y}) \ . \ [2\ 3] : (\mathbf{t}_1, \bar{\iota})}\ \texttt{tc-contr}$$

$$\frac{\mathrm{x} \in \mathrm{dom}(\Gamma) \quad \mathbf{t}_1 = \Gamma(\mathrm{x}) \qquad \Gamma;\mathrm{x};\bar{\iota} \vdash_c (\mathrm{x}\#\mathrm{y}) \ . \ [2\ 3] : (\mathbf{t}_1, \bar{\iota})}{\Gamma \vdash \mathrm{x} = (\mathrm{x}\#\mathrm{y}) \ . \ [2\ 3] \text{ ok}}\ \texttt{ok-c-stmt}$$

Figure 8: Derivation tree for $x = (x\#y)\ .\ [2\ 3]$

While inference starts from the top of the tree, an algorithm for finding an derivation tree would start from the bottom. This is because of the quantification "for any $\bar{\kappa}$" in `tc-var-xy`, which allows variables other than x to be accessed at any index. Because of this, a statement like $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ does not always have a unique $(\mathbf{t}, \bar{\kappa})$, even though $\mathbf{t}$ is still unique as will be shown later. While it is possible to implement this universal quantification, it is easier to start from the bottom, as only a single index will be needed at a time.

Starting from $\Gamma \vdash \mathrm{x} = (\mathrm{x}\#\mathrm{y})\ .\ [2\ 3]$ ok, only `ok-c-stmt` can be applied to derive this statement, demanding that x is only accessed at the same index as $(x\#y)\ .\ [2\ 3]$. As the expression is a contraction, only `tc-contr` can be applied, introducing a new dimension and index. Contraction sums over a diagonal so the dimension and index are used two times each. `tc-paren` simply takes the inner expression and continues with the same type. `tc-prod` splits both the dimensions as well as the index. It therefore requires x to be accessed at $\bar{\iota} = (\iota_1)$ which is allowed according to `tc-var-xx` and y to be accessed at (κ_1, κ_1). y can be accessed for any (κ_2, κ_3), so this is especially true for $\kappa_2 = \kappa_3 = \kappa_1$.

Changing the order of x and y makes the statement incompatible. Before changing the order, the contraction has been done over the matrix y and x has only been accessed at the assigned index. After changing the order, the contraction is done over the second dimension of y as well as x. This means, that x is now accessed at a new index different from the index used for assignment. This fact also shows in the derivation tree, which is now impossible to construct as can be seen in Figure 9. The tree is approximately the same as for Figure 8, however when dividing the indices onto y and x, x must now be accessed at a new index $\kappa_1 \neq \bar{\iota}$. As statements about x can only be derived through `tc-var-xx` and this rule only allows $\bar{\iota}$ for access, it is not possible to find a precondition for this statement. It is however still possible to find a derivation in the unaugmented type system, as indices are not relevant there. Removing

the augmentation and changing the rules to the unaugmented versions yields a derivation tree similar to the one from Figure 4 on page 5.

$$\dfrac{\begin{array}{c}\text{x} \in \text{dom}(\Gamma) \\ \mathbf{t}_1 = \Gamma(\text{x})\end{array} \quad \dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\text{y} \in \text{dom}(\Gamma) \quad \text{y} \neq \text{x} \\ \mathbf{t}_1 = (d_1, d_2) = \Gamma(\text{y})\end{array}}{\begin{array}{c}\Gamma;\text{x};\bar{\iota} \vdash_c \text{y} : (\mathbf{t}_1, \bar{\kappa}) \\ \text{for any } \bar{\kappa} = (\kappa_2, \kappa_3)\end{array}}\ \mathtt{tc\text{-}var\text{-}xy} \qquad \dfrac{\text{no possible precondition}}{\Gamma;\text{x};\bar{\iota} \vdash_c \text{x} : ((d_2), (\kappa_1))}\ \mathtt{tc\text{-}var\text{-}xx}}{\Gamma;\text{x};\bar{\iota} \vdash_c \text{y\#x} : ((d_1, d_2, d_2), (\iota_1, \kappa_1, \kappa_1))}\ \mathtt{tc\text{-}prod}}{\Gamma;\text{x};\bar{\iota} \vdash_c (\text{y\#x}) : ((d_1, d_2, d_2), (\iota_1, \kappa_1, \kappa_1))}\ \mathtt{tc\text{-}paren}}{\Gamma;\text{x};\bar{\iota} \vdash_c (\text{y\#x})\ .\ [2\ 3] : (\mathbf{t}_1, \bar{\iota})}\ \mathtt{tc\text{-}contr}}{\Gamma \vdash \text{x} = (\text{y\#x})\ .\ [2\ 3]\ \text{ok}}\ \mathtt{ok\text{-}c\text{-}stmt}$$

Figure 9: Attempted derivation tree for $x = (y\#x)\ .\ [2\ 3]$

If the statement is determined to be compatible, the evaluation can proceed without using temporary variables. This is done by changing the rule `ev-stmt`, removing the temporary variable $r_{\mathbf{i}}$ and instead assigning the result from $\text{eval}_{\Gamma,\mu}(e_{\mathbf{i}})$ directly.

$$\dfrac{\text{x} \in \text{dom}(\Gamma) \quad \mathbf{t} = \Gamma(\text{x}) \quad \forall_{\mathbf{i} \leq \mathbf{t}} \text{x}_{\mathbf{i}} \in \text{dom}(\mu)}{\langle \mu, \text{x} = e \rangle \rightarrow_\Gamma \langle \mu\{\forall_{\mathbf{i} \leq \mathbf{t}} \text{x}_{\mathbf{i}} \mapsto \text{eval}_{\Gamma,\mu}(e_{\mathbf{i}})\}, \emptyset \rangle}\ \mathtt{ev\text{-}c\text{-}stmt}$$

The rule is not changed directly, instead a new rule with the changes is introduced. This is because if the statement is incompatible, it can still be evaluated using the unaugmented type system and the rule `ev-stmt`, however this assigns using temporary variables $r_{\mathbf{i}}$ for which space has to be allocated.

5 Correctness of the extension

Having established the type system, the next step is to show that it is correct, i.e. that compatibility under the type system does not differ from the one in Definition 2. In order to make this proof easier, it will be helpful if the derivation of a specific statement is unique, i.e. there are not multiple derivations of a statement like $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$. Because a statement about the multiplication or division of two scalars can both be derived from `tc-smul` and `tc-sdiv`, this is currently not the case. However, changing these rules will ensure uniqueness as expressed by the following lemma.

Lemma 1 (Uniqueness of type derivations)**.** Assume that the rules `tc-smul` and `tc-sdiv` from Figure 6 have been replaced with these:

$$\dfrac{\Gamma;\text{x};\bar{\iota} \vdash_c e0 : ((), ()) \quad \Gamma;\text{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}_1, \bar{\kappa}) \quad \mathbf{t}_1 \neq ()}{\Gamma;\text{x};\bar{\iota} \vdash_c e0 * e1 : (\mathbf{t}_1, \bar{\kappa})}\ \mathtt{tc\text{-}smul'}$$

$$\dfrac{\Gamma;\text{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}) \quad \Gamma;\text{x};\bar{\iota} \vdash_c e1 : ((), ()) \quad \mathbf{t}_0 \neq ()}{\Gamma;\text{x};\bar{\iota} \vdash_c e0/e1 : (\mathbf{t}_0, \bar{\kappa})}\ \mathtt{tc\text{-}sdiv'}$$

Then $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ has at most one derivation up to renaming of new indices.

Proof. By induction on the structure of e.

Case $e \equiv$ x: Only `tc-var-xx` can be applied so $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ has exactly one derivation for $\bar{\kappa} = \bar{\iota}$ and zero otherwise.

Case $e \equiv \mathrm{y} \not\equiv \mathrm{x}$: Only `tc-var-xy` can be applied so $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ has exactly one derivation for every $\bar{\kappa} = (\kappa_1, \ldots, \kappa_k)$ with $k = \mathrm{rank}(\mathbf{t})$ and zero otherwise.

Case $e \equiv e0 \# e1$: Given $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$, there is exactly one $(\mathbf{t}_0, \bar{\kappa}_0)$, $(\mathbf{t}_1, \bar{\kappa}_1)$ such that $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}_0)$, $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}_1, \bar{\kappa}_1)$ and `tc-prod` can be applied. As both of these statements have at most one derivation and `tc-prod` is the only appliable rule, $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ also has at most one derivation.

Case $e \equiv (e0)$ and $e \equiv e0 \ \hat{}\ [m\ n]$ are handled analogously.

Case $e \equiv e0\ \mathbf{.}\ [m\ n]$: While there are multiple $(\mathbf{t}_0, \bar{\kappa}_0)$ where $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ follows from $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}_0)$ by applying `tc-contr`, they only differ by the naming of the fresh index.

Case $e \equiv e0 \odot e1$ where $\odot \in \{+, -, *, /\}$: If $\odot \equiv *$, $\Gamma \vdash e0 : ()$ and $\Gamma(e1) \neq ()$, only `tc-smul'` can be applied as `tc-elem` would imply through de-augmentation that $\Gamma(e0) = \Gamma(e1)$. As both $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : ((), ())$ and $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}, \bar{\kappa})$ have a unique derivation, the same is true for $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$. The same schema can be applied for $\Gamma \vdash e1 : ()$ using `tc-elem`, $\Gamma(e0) \neq ()$ and $\odot \equiv /$. The last case with $\odot \in \{+, -\}$ is analogous to $e0 \# e1$. □

As shown in the last section, the rules in Figure 6 are similar to the the type system described in [7]. Given a derivation of a statement like $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$, removing the augmentation of the contexts and types yields a derivation for $\Gamma \vdash e : \mathbf{t}$ by the rules in [7]. This is also true after the rules have been changed in Lemma 1. Note however, that there can be multiple derivations of $\Gamma \vdash e : \mathbf{t}$, as this type system has not been changed similar to Lemma 1. This means, that the opposite direction is not always true. To make statements about this direction, the following definition is useful.

Definition 3 (Subexpressions)**.** Given two expressions e, e' such that $\Gamma \vdash e : \mathbf{t}$ and $\Gamma \vdash e' : \mathbf{t}'$. If there exists a derivation of $\Gamma \vdash e : \mathbf{t}$ in which $\Gamma \vdash e' : \mathbf{t}'$ occurs, e' is a subexpression of e. Additionally, as $\mathrm{eval}_\Gamma(.)$ defines an algebraic expression, subexpressions are already defined for it.

The next lemma about these subexpressions will be helpful.

Lemma 2. If $\Gamma \vdash e' : \mathbf{t}'$ occurs in one derivation of $\Gamma \vdash e : \mathbf{t}$, then it occurs in all derivations of $\Gamma \vdash e : \mathbf{t}$.

Proof. All derivations of $\Gamma \vdash e : \mathbf{t}$ only differ by whether `t-smul / t-sdiv` or `t-elem` have been used in the case of scalar-scalar multiplication or division and all three rules require the same precondition in this case. □

The following two lemmas then connect the two type systems.

Lemma 3 (De-augmentation of contexts and types)**.** Let e be an expression and let x be a variable (which is also an expression).

1. If $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$, then also $\Gamma \vdash e : \mathbf{t}$

2. If x is not a subexpression of e, then $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa}) \Leftrightarrow \Gamma \vdash e : \mathbf{t}$ for any $\bar{\iota}, \bar{\kappa} = (\kappa_1, \ldots, \kappa_k)$ where $k = \mathrm{rank}(\mathbf{t})$

Proof. For the $\Leftarrow$ direction of 2. by induction on the structure of e

Case $e \equiv \mathrm{y}$: Inverting `t-var` implies $\mathrm{y} \in \mathrm{dom}(\Gamma)$ and $\mathbf{t} = \Gamma(\mathrm{y})$. As x is not a subexpression of e, $\Gamma \vdash \mathrm{x} : \Gamma(\mathrm{x})$ does not occur in any derivation of $\Gamma \vdash e : \mathbf{t}$. Therefore $e \neq \mathrm{x}$ and $\mathrm{y} \neq \mathrm{x}$, so the statement follows from `tc-var-xy`.

Case $e \equiv e0 \# e1$: As x is not a subexpression of e, it is also not a subexpression of both $e0$, $e1$. Otherwise, in the case of $e0$, this would mean that $\Gamma \vdash \mathrm{x} : \Gamma(\mathrm{x})$ would occur in any derivation of $\Gamma \vdash e0 : \Gamma(e0)$ and as $\Gamma \vdash e0 : \Gamma(e0)$ is needed for $\Gamma \vdash e : \Gamma(e)$, $\Gamma \vdash \mathrm{x} : \Gamma(\mathrm{x})$ would occur in a derivation of $\Gamma \vdash e : \Gamma(e)$ which contradicts the fact that x is not a subexpression of e. The same is true for $e1$. Therefore the induction hypothesis can be applied. Starting from $\Gamma \vdash e : \mathbf{t}$ where $\mathbf{t} = (d_{01}, \ldots, d_{0k}, d_{11}, \ldots, d_{1l})$:

$$
\begin{aligned}
&\xRightarrow{\texttt{t-prod inverted}} \Gamma \vdash e0 : (d_{01}, \ldots, d_{0k}) \\
&\qquad\qquad\quad \Gamma \vdash e1 : (d_{11}, \ldots, d_{1l}) \\
&\xRightarrow{\texttt{induction}} \Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : ((d_{01}, \ldots, d_{0k}), \bar{\kappa}_0) \text{ for any } \bar{\iota}, \bar{\kappa}_0 = (\kappa_{01}, \ldots, \kappa_{0k}) \\
&\qquad\qquad\quad \Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : ((d_{11}, \ldots, d_{1l}), \bar{\kappa}_1) \text{ for any } \bar{\iota}, \bar{\kappa}_1 = (\kappa_{11}, \ldots, \kappa_{1l}) \\
&\xRightarrow{\texttt{tc-prod}} \Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa}) \text{ for any } \bar{\iota}, \bar{\kappa} = (\kappa_{01}, \ldots, \kappa_{0k}, \kappa_{11}, \ldots, \kappa_{1l})
\end{aligned}
$$

Case $e \equiv (e0)$ and $e \equiv e0 \ \hat{} \ [m\ n]$ are handled analogously.

Case $e \equiv e0\ .\ [m\ n]$: Rule `t-contr` can't be inverted for specific $d_m = d_n$, however there exists some $d_m = d_n$ such that $\Gamma \vdash e0 : (d_1, \ldots, d_m, \ldots, d_n, \ldots, d_k)$ and these $d_m = d_n$ are removed when applying `tc-contr` after the induction hypothesis.

Case $e \equiv e0 \odot e1$ where $\odot \in \{+, -, *, /\}$: When $\Gamma \vdash e : ()$ and $\odot \in \{*, /\}$, the exact rule used can't be determined but in all three rules `t-elem`, `t-smul` and `t-sdiv` the premise contains both $\Gamma \vdash e0 : ()$, $\Gamma \vdash e1 : ()$ so `tc-elem` can be applied after the induction hypothesis. Otherwise one has to differentiate on whether e is a scalar multiplication / division or an elementwise operation and use the corresponding rules analogously to $e0 \# e1$. □

Lemma 4 (Type derivations of subexpressions)**.** Let $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ and let e' be a subexpression of e. There exists $\mathbf{t}'$ and $\bar{\kappa}'$ such that $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ and $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$.

Proof. Take the derivation of $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$. Removing the augmentation from the context and the types yields a derivation of $\Gamma \vdash e : \mathbf{t}$ using Lemma 3. If $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ does not occur in the derivation of $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ for some $\mathbf{t}', \bar{\kappa}'$, $\Gamma \vdash e' : \mathbf{t}'$ also does not occur in the yielded derivation of $\Gamma \vdash e : \mathbf{t}$. According to Lemma 2, this means that $\Gamma \vdash e' : \mathbf{t}'$ does not occur in any derivation of $\Gamma \vdash e : \mathbf{t}$. However this violates the premisse that e' is a subexpression of e. □

Because the definition for compatible statements (Definition 2) is based on $\mathrm{eval}_\Gamma(.)$, establishing a connection between the type system and $\mathrm{eval}_\Gamma(.)$ makes it possible to proof the correctness of the type system. Having established a

connection between the augmented and unaugmented type system, the following theorem then connects the augmented type system and the $\text{eval}_\Gamma(.)$ function.

Theorem 1. Let $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ and let e' be a subexpression of e. Then for all $\bar{\kappa}'$, the following are equivalent up to renaming of fresh indices:

(a) $\text{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of $\text{eval}_\Gamma(e_{\bar{\kappa}})$.

(b) $\Gamma;\text{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$.

Proof. By induction on the structure of e.

The cases below assume that e' is a "real" subexpression of e, i.e. not $e \equiv e'$. If this is the case, both (a) and (b) are always true and therefore equivalent. For the following cases, it will therefore be assumed that $e \not\equiv e'$. The cases $e \equiv x$ and $e \equiv y$ cannot exist unless $e \equiv e'$, as otherwise e' would not be a subexpression of e. The theorem is therefore true for both of these cases.

Case $e \equiv e0 \# e1$, **(a) ⇒ (b):** As $\text{eval}_\Gamma(e_{\bar{\kappa}}) = \text{eval}_\Gamma(e0_{\bar{\kappa}_0}) \cdot \text{eval}_\Gamma(e1_{\bar{\kappa}_1})$, with $\bar{\kappa} = (\kappa_{01}, \ldots, \kappa_{0k}, \kappa_{11}, \ldots, \kappa_{1l})$, $\bar{\kappa}_0 = (\kappa_{01}, \ldots, \kappa_{0k})$ and $\bar{\kappa}_1 = (\kappa_{11}, \ldots, \kappa_{1l})$, $\text{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of at least one of $\text{eval}_\Gamma(e0_{\bar{\kappa}_0})$, $\text{eval}_\Gamma(e1_{\bar{\kappa}_1})$. Using $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ with $\mathbf{t} = (d_{01}, \ldots, d_{0k}, d_{11}, \ldots, d_{1l})$ and inverting `tc-prod` gives $\Gamma;\text{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}_0)$ and $\Gamma;\text{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}_1, \bar{\kappa}_1)$ where $\mathbf{t}_0 = (d_{01}, \ldots, d_{0k})$ and $\mathbf{t}_1 = (d_{11}, \ldots, d_{1l})$. The induction hypothesis implies that $\Gamma;\text{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of at least $\Gamma;\text{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}_0)$ or $\Gamma;\text{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}_1, \bar{\kappa}_1)$. As both of these statements are needed to conclude that $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$, $\Gamma;\text{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$.
(b) ⇒ (a): $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ can only be concluded using `tc-prod` and both $\Gamma;\text{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}_0)$, $\Gamma;\text{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}_1, \bar{\kappa}_1)$. Therefore $\Gamma;\text{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of at least one of these two statements. Applying the induction hypothesis implies that $\text{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of at least one of $\text{eval}_\Gamma(e0_{\bar{\kappa}_0})$, $\text{eval}_\Gamma(e1_{\bar{\kappa}_1})$. As both of these expressions are subexpressions of $\text{eval}_\Gamma(e_{\bar{\kappa}})$, $\text{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of $\text{eval}_\Gamma(e_{\bar{\kappa}})$.

Case $e \equiv (e0)$ and $e \equiv e0 \;\hat{}\; [m\ n]$ are handled analogously.

Case $e \equiv e0 \;.\; [m\ n]$, **(a) ⇒ (b):** Using the prerequisite $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ where $\mathbf{t} = (d_1, \ldots, \widehat{d_m}, \ldots, \widehat{d_n}, \ldots, d_k)$, $\bar{\kappa} = (\kappa_1, \ldots, \widehat{\kappa_m}, \ldots, \widehat{\kappa_n}, \ldots, \kappa_k)$ and inverting `tc-contr` implies $\Gamma;\text{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}_0)$ with $\mathbf{t}_0 = (d_1, \ldots, d_m, \ldots, d_n, \ldots, d_k)$, $\bar{\kappa}_0 = (\kappa_1, \ldots, \kappa_m, \ldots, \kappa_n, \ldots, \kappa_k)$ where $\kappa_m = \kappa_n$ is a fresh index. Because $\text{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of $\text{eval}_\Gamma(e0 \;.\; [m\ n]_{\bar{\kappa}})$, it is also a subexpression of $\text{eval}_\Gamma(e0_{\bar{\kappa}_0})$ where the sum index $l = \kappa_m = \kappa_n$ ist a fresh index ranging from 1 to $\pi_m(\mathbf{t})$. Using the induction hypothesis implies that $\Gamma;\text{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of $\Gamma;\text{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}_0)$. As this statement is a necessary condition to conclude that $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$, $\Gamma;\text{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$.
(b) ⇒ (a): Only `tc-contr` can be applied to conclude $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$. Therefore $\Gamma;\text{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of $\Gamma;\text{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}_0, \bar{\kappa}_0)$. By induction, $\text{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of $\text{eval}_\Gamma(e0_{\bar{\kappa}_0})$. As this expression is always a subexpression of $\text{eval}_\Gamma(e_{\bar{\kappa}})$ up to renaming of the fresh index, $\text{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of $\text{eval}_\Gamma(e_{\bar{\kappa}})$.

Case $e \equiv e0 \odot e1$ where $\odot \in \{+, -, *, /\}$, **(a) ⇒ (b):** If $\odot \equiv *$ and $\Gamma \vdash e0 : ()$, $\text{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of at least one of $\text{eval}_\Gamma(e0)$, $\text{eval}_\Gamma(e1_{\bar{\kappa}})$. If

additionally $\Gamma \vdash e1 : ()$ then the only rule which can be applied is `tc-elem` as `tc-smul'` needs $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}_1, \bar{\kappa})$ with $\mathbf{t}_1 \neq ()$ as a precondition which by Lemma 3 would imply $\Gamma \vdash e1 : \mathbf{t}_1$. Therefore `tc-elem` can be inverted which gives $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : (\mathbf{t}, \bar{\kappa})$, $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : (\mathbf{t}, \bar{\kappa})$. Lemma 3 implies $\Gamma \vdash e0 : \mathbf{t}$. As $\Gamma(e0)$ is unique, $\mathbf{t} = ()$ and therefore $\bar{\kappa} = ()$. [3] Using the induction hypothesis implies that $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of at least one of the two statements. As both of the two statements are needed in `tc-elem`, $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$. The proof for $\Gamma(e1) \neq ()$ is analogous as it is for $\Gamma(e0) \neq ()$. The same is true for $\odot \equiv /$. The last case for $\odot \in \{+, -\}$ is analogous to $e0 \# e1$.
(b) $\Rightarrow$ (a): If $\odot \equiv *$, $\Gamma \vdash e0 : ()$ and $\Gamma \vdash e1 : ()$ then only `tc-elem` can be applied to conclude $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$. Therefore $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e' : (\mathbf{t}', \bar{\kappa}')$ occurs in the derivation of at least one of $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e0 : ((), ())$, $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e1 : ((), ())$. By induction $\mathrm{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of at least one of $\mathrm{eval}_\Gamma(e0)$, $\mathrm{eval}_\Gamma(e1)$. As both of these expressions are subexpressions of $\mathrm{eval}_\Gamma(e_{\bar{\kappa}})$, $\mathrm{eval}_\Gamma(e'_{\bar{\kappa}'})$ is a subexpression of $\mathrm{eval}_\Gamma(e_{\bar{\kappa}})$. The remaining cases are analogous. □

Using this relationship, one can then proof the correctness of the type system. This is done by showing that, given a statement $\mathrm{x} = e$ where x is also a subexpression of e, if the type system implies that the statement is compatible, then x is only accessed at the same index currently assigned to.

Corollary 1 (Correctness of the type system)**.** Let $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\iota})$. Assume that the expression x (variable) is a subexpression of e. Then the following hold:

(a) $\mathrm{eval}_\Gamma(\mathrm{x}_{\bar{\iota}})$ is a subexpression of $\mathrm{eval}_\Gamma(e_{\bar{\iota}})$.

(b) If $\mathrm{eval}_\Gamma(\mathrm{x}_{\bar{\kappa}})$ is a subexpression of $\mathrm{eval}_\Gamma(e_{\bar{\iota}})$, then $\bar{\kappa} = \bar{\iota}$.

Proof. (a) Let $\mathbf{t}' = \Gamma(\mathrm{x})$, $\mathbf{t} = \Gamma(e)$. As x is a subexpression of e, by definition $\Gamma \vdash \mathrm{x} : \mathbf{t}'$ occurs in the derivation of $\Gamma \vdash e : \mathbf{t}$. Rule `tc-var-xx` implies $\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{x} : (\mathbf{t}', \bar{\iota})$. Also $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\iota})$ is given. Using Lemma 4 implies that $\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{x} : (\mathbf{t}', \bar{\iota})$ occurs in the derivation of $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\iota})$. Applying Theorem 1 implies that $\mathrm{eval}_\Gamma(\mathrm{x}_{\bar{\iota}})$ is a subexpression of $\mathrm{eval}_\Gamma(e_{\bar{\iota}})$.

(b) Theorem 1 and (a) implies that $\Gamma;\mathrm{x};\bar{\iota} \vdash_c \mathrm{x} : (\mathbf{t}', \bar{\kappa})$ occurs in the derivation of $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\iota})$. Rule `tc-var-xx` then implies that $\bar{\kappa} = \bar{\iota}$. □

Note that $\mu'(\mathrm{x}_{\bar{\kappa}})$ can only result from $\mathrm{eval}_\Gamma(\mathrm{x}_{\bar{\kappa}})$ and (b) implies that for every $\mathrm{eval}_\Gamma(\mathrm{x}_{\bar{\kappa}})$ which results from $\mathrm{eval}_\Gamma(e_{\bar{\iota}})$, the condition $\bar{\kappa} = \bar{\iota}$ is fulfilled. This means, that $\bar{\iota} = \bar{\kappa}$ is also true for every $\mu'(\mathrm{x}_{\bar{\kappa}})$ resulting from $\mathrm{eval}_\Gamma(e_{\bar{\iota}})$. In other words, the statement is compatible.

The corollary therefore proofs, that if the statement is compatible under the type system i.e. $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\iota})$, it is also compatible by Definition 2. In other words, the type system is correct. It does not show however, that if the statement is compatible according to Definition 2, then it is also compatible under the type system, i.e. the type system is optimal. This can be expressed by the following conjecture, which is however left for further work. The case if x is not a subexpression of e has already be shown by Lemma 3.

[3] It is not possible to derive a statement of the form $\Gamma;\mathrm{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\kappa})$ with $\mathbf{t} = (d_1, \ldots, d_{k1})$, $\bar{\kappa} = (\kappa_1, \ldots, \kappa_{k2})$ where $k1 \neq k2$. This is because there is no rule which has no statements or only statements where $k1 = k2$ as conditions to derive an statement where $k1 \neq k2$

Conjecture 1 (Optimality of the type system)**.** Let x be a subexpression of e and let $\bar{\iota}$ be an arbitary index. Then if for every $\text{eval}_\Gamma(\text{x}_{\bar{\kappa}})$ which is a subexpression of $\text{eval}_\Gamma(e_{\bar{\iota}})$ the condition $\bar{\kappa} = \bar{\iota}$ is fulfilled, the statement $\Gamma;\text{x};\bar{\iota} \vdash_c e : (\mathbf{t}, \bar{\iota})$ has a derivation.

6 Performance evaluation

In order to evaluate the impact, optimizations based on compatible statements could have, different kernels have been measured. The two most important kernels are scalar operations and elementwise operations as these can be seen as the basic building blocks for compatible statements.

As parallelization is an often used tool, it is also interesting to evaluate these kernels in a parallel setting. This has been achieved by adding OpenMP [6] pragmas to the code in order to evaluate it using multiple CPU cores.

Normally, to compute a statement in the DSL, temporary memory is neccessary to store the result of the calculation. Additionally, if the result is expected to be in a certain memory location, it must be copied back. However, if it is known that a statement is compatible, one can instead use the same memory for reading and writing, resulting in a possible speedup. In Section 6.1, we therefore compare two versions of different kernels, one writing inplace and the other writing to another location and then copying back.

If the memory location of the result can be changed, the copy operation is not needed. However, different memory locations are still used for input and output tensors. A possible speedup of this version is discussed in Section 6.2.

6.1 Copy vs. in-place (avoid-copy)

The first kernel of interest is multiplication of a vector with a scalar. The code for the version using a temporary variable is depicted in Figure 10a while the version without in Figure 10b. In the single core version, the OpenMP pragmas are removed.

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
    xtmp[i] = s * x[i];

#pragma omp parallel for
for (int i = 0; i < N; i++)
    x[i] = xtmp[i];
```

(a) with a temporary variable

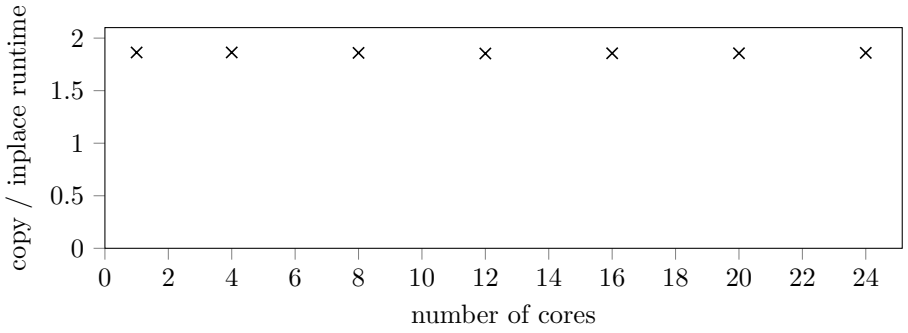
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
    x[i] = s * x[i];
```

(b) without a temporary variable

Figure 10: Code for scalar multiplication

Comparing the runtime of these versions and plotting the speedup for different amount of cores then yields the graph from Figure 11. As can be seen, the speedup is practically independent of the amount of cores used for computation and is around 1.859x. As the standard deviation is up to 3% and the amount of repetitions is 2^7, the standard error of the mean is around 0.2%. The last decimal might therefore not have much significance.

Figure 11: Scalar multiplication, N = 1 000 000, REP = 2^7



The second kernel is elementwise multiplication, with the code shown in Figure 12. Measuring this kernel yields a plot very similar to Figure 11. The only difference is that the speedup is approximately constant around 2.007x.

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
    xtmp[i] = x[i] * v[i];

#pragma omp parallel for
for (int i = 0; i < N; i++)
    x[i] = xtmp[i];
```

(a) with a temporary variable

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
    x[i] = x[i] * v[i];
```

(b) without a temporary variable

Figure 12: Code for elementwise multiplication

The third measured kernel is a linear combination. It can be expressed as $x = s1 * x + s2 * u + s3 * v$ where $s1, s2, s3$ are scalars and x, u, v vectors of the same size. The measured speedup for this kernel is again almost independent of the number of cores and is around 1.697x.

As can be seen, the speedup gained by re-using the memory decreases with an increasing kernel complexity. This result is not suprising as the time taken for the copy-operation is approximately constant while the computation time for the kernel itself only increases. Notably however, the kernel for elementwise multiplication results in a higher speedup than for scalar multiplication.

This result is also visible when analysing kernels like $x = uv * x$ where uv is a matrix-vector multiplication resulting in a vector which is then elementwise mutliplied with x. The code for this kernel is shown in Figure 13. Note that the use of a single temporary variable is allowed. The goal is to prevent copying the whole tensor (vector).

In this case, the speedup of re-using memory is not independent of N. For the previous kernels, the runtime complexity of both the copy operation as well as the kernel itself was $\mathcal{O}(\mathrm{N})$. The constant factor between these two runtimes was equal to the gained speedup minus one. However as the runtime complexity of this kernel is $\mathcal{O}(\mathrm{N}^2)$ while the copy operation is still $\mathcal{O}(\mathrm{N})$, it is necessary to plot the speedup for different N.

As can be seen from Figure 14, the speedup gained by re-using memory

```
for (int i = 0; i < N; i++) {
    double tmp = 0.0;
    for (int j = 0; j < N; j++)
        tmp += u[i][j] * v[j];
    xtmp[i] = tmp * x[i];
}

for (int i = 0; i < N; i++)
    x[i] = xtmp[i];
```

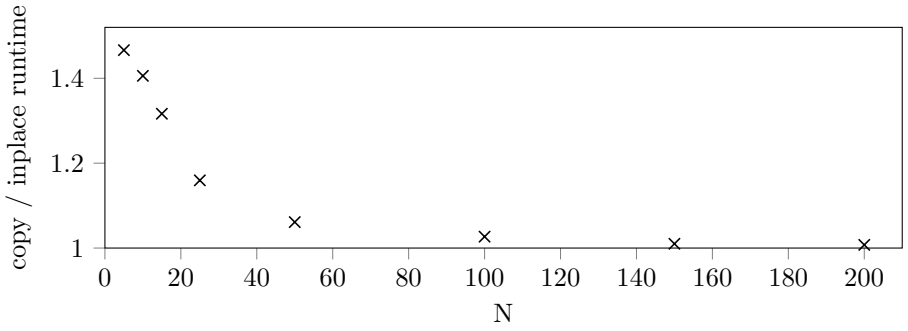
(a) with a temporary variable

```
for (int i = 0; i < N; i++) {
    double tmp = 0.0;
    for (int j = 0; j < N; j++)
        tmp += u[i][j] * v[j];
    x[i] *= tmp;
}
```

(b) without a temporary variable

Figure 13: Code for matrix-vector elementwise multiplication ($x = uv * x$)

Figure 14: Matrix-vector elementwise multiplication ($x = uv * x$)



gets smaller as N increases. The reason for this is that while the runtime of the copy operation increases lineary with N, the runtime of the kernel increases quadratically and therfore the percentage of the runtime which is due to the copy operation gets smaller with increasing N. However even for large N like 200, there is still a speedup of approximately 1.0075x (REP $= 2^{15} \Rightarrow \sigma^{-} \approx 0.00016$x) which might be relevant for long running computations.

Generally, it can be said that the higher the fraction of data size to kernel runtime is, the more data must be copied which results in a higher speedup when re-using memory. In the case of scalar multiplication, both the data size and runtime are in $\mathcal{O}(\mathrm{N})$ which results in a high speedup. Note that the data size does not always equal N, e.g. for matrix-matrix multiplication, the data size is in $\mathcal{O}(\mathrm{N}^2)$ and the runtime in $\mathcal{O}(\mathrm{N}^3)$.

6.2 Other variable vs. in-place (reduce-cache-miss)

6.2.1 Explanation of the optimization

The kernels of interest are the same as in the last section. However the compared versions include no copy operation. The only difference between the two versions is the variable they write to. For example, the code for scalar multiplication is shown in Figure 15.

The expectation might be that the code from Figure 15b is faster than the code from Figure 15a for some N. For example, if the L1 data cache size is 32K

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
    y[i] = s * x[i];
```

(a) write to different variable

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
    x[i] = s * x[i];
```

(b) write to same variable

Figure 15: Code for scalar multiplication

and `x` has a size of 25K, it will not fit into cache twice. This means that for Figure 15a, only one of `x, y` can fit inside the cache at one time.

When the code from Figure 15a gets executed, after a number of iterations parts of both `x` and `y` will be in the cache. While it might not be necessary for `y` to be cached, the exact behaviour depends on a variety of factors like the type of computing unit (CPU, GPU[4]) or the important memory for the data sizes (cache, DRAM, etc.) and might not be changeable by the programmer. Also, while it might be possible that writing y overwrites only cachelines which belong to the part of `x` which has already been used, this is not guaranteed and it is not realistic to expect it to happen every time. The main reason why the code from Figure 15b might not be noticable faster is prefetching, as only the next few iterations are important for low-latency access.

In order to explain why the comparison is useful in the context of this thesis, consider the two code examples from Figure 16 using the defined DSL.

```
y = s * x
v = y # w
```

(a) write to different variable

```
x = s * x
v = x # w
```

(b) write to same variable

Figure 16: Example code from the DSL

Given Figure 16a, if the compiler knows, e.g. through liveness analysis, that `x` will not be needed in the future, it can transform the code to the faster running Figure 16b, replacing every occurence of `y` after the first line with `x`. Note that this is different from re-using the same memory of `x` in general, the difference being that `x` is re-used in the same statement where it is also used for the last time, possibly resulting in better performance. Also, in general, the memory of `x` can only be re-used after the first line. This specific optimization is only possible if the transformation of the first statement would be compatible.

6.2.2 Measuring the impact

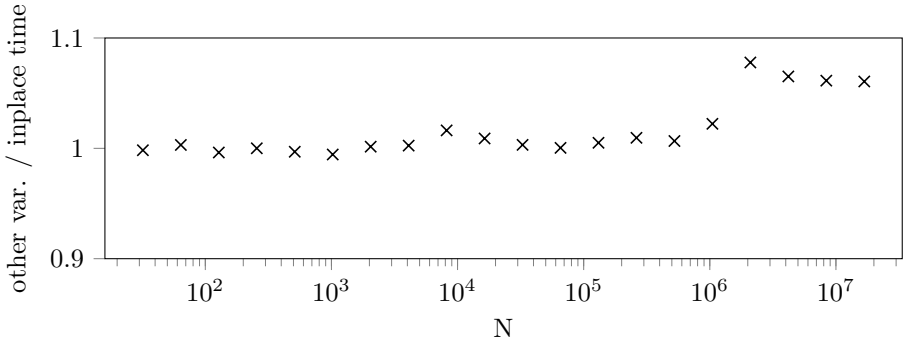
As the cache sizes are important in order to understand the results, they are specified beforehand. All of the measurements have been done using an Intel Xeon CPU E5-2680 v3 with cache sizes L1d/32K, L2/256K, L3/30M.

From these cache sizes one can already calculate the N's for which interesting results might happen. For the L1d cache, a vector with size N = 4096 would fill the whole cache. In the case of elementwise multiplication, N = 1024 and N = 2048 are therefore relevant. As the runtime for these small N's on multiple

[4]although only CPUs are tested here, in general the behaviour depends on the type of computing unit which is why GPUs are mentioned here

cores would mostly be dominated by the thread overhead resulting in very high variance, only a single core is used.

Figure 17: Elementwise multiplication



Measuring elementwise multiplication does not result in any measurable difference for the sizes relevant for L1d and L2 cache as shown in Figure 17. This is probably due to data prefetching as the access pattern is very predictable. However $N = 2^{21}$ does result in a speed difference of 7.8%. In this case, the size of a single vector is 16M. The version which writes to a different variable (`u = v * x`) uses 3 vectors and does not fit into L3 cache while the other version (`x = v * x`) uses 2 vectors which almost fit into L3 cache. The difference is therefore only visible once the last cache is exceeded and DRAM must be used. While the speed difference for larger N's becomes lower as both versions have to use more DRAM, it is still 6.1% for $N = 2^{24}$.

The same is true for scalar multiplication. The difference becomes measureable at $N = 2^{21}$ with 3.3%, peaks at $N = 2^{23}$ with 8.2% and then falls again to 6.8% with $N = 2^{24}$. These numbers again correspond to the L3 cache size.

It can be estimated, that the speedup gained from the optimization is mostly due to avoiding cache misses. As for kernels with a simple access pattern, the number of avoided cache misses is correlated to the size of the "saved variable", this size has influence on the speedup. For elementwise multiplication, 3 vectors of size N are reduced to 2 vectors of size N (33% reduction), resulting in a speed difference of up to 7.8%. For scalar multiplication, 2 vectors of are reduced to a single vector (50% reduction), resulting in a speed difference of up to 8.2%. The same result also shows for the linear combination reducing 4 to 3 vectors (25% reduction) with a resulting difference of 4.3% for $N = 2^{21}$.

However, this means that for simple kernels, if the time complexity of the kernel and the space complexity of the "saved variable" are different, no measurable speedup will be achieved for larger N. That this is the case can be seen using the matrix-vector elementwise multiplication kernel from Figure 13. The resulting plot (not shown here) is simply an approximately straight line with speedup equal to 1x. Therefore, with a time complexity of $\mathcal{O}(N^2)$ and a space complexity of $\mathcal{O}(N)$, no significant speedup (larger than 1x) is measurable.

6.3 Memory reusing for incompatible statements

The avoid-copy and reduce-cache-miss optimization presented in Sections 6.1 and 6.2 are based on the concept of compatible statements introduced in this

thesis. This means, that they can only be applied if statements are compatible or can be made compatible. The goal of this section is to look at a kernel consisting of incompatible statements, to see whether similar optimizations can be applied. The kernel which will be analysed is called the "Interpolation operator" and is motivated by its usage in computational fluid dynamics and a project related to this thesis, CFDlang [8] which uses a similar DSL.

Changing the DSL notation slightly, the interpolation kernel can be written as `v = A # B # C # u . [[2 9] [4 8] [6 7]]`. `A`, `B`, `C` are matrices and `v`, `u` are tensors of rank 3. The double brackets denote a group of contractions which is applied without changing the dimension numbers as it would normally be the case if the contractions would be applied in succession. As calculating the tensor product of the 4 tensors would result in a tensor of rank 9, this calculation is normally done in multiple steps to reduce runtime complexity. Applying the contraction directly after each tensor product results in the code from Figure 18.

```
tmp1 = C # u . [2 5]
tmp2 = B # tmp1 . [2 5]
v = A # tmp2 . [2 5]
```

Figure 18: Interpolation with immediate contractions

Because this code does not contain any variables on both sides in a single statement, the avoid-copy optimization does not apply. The reduce-cache-miss optimization would try to transform each statement in order to reuse memory. Taking the first statement as an example, the transformed code is shown in Figure 19. However, as the first transformed statement is not compatible, this optimization is not possible for the first statement. Transforming the other statements also gives a negative result.

```
u = C # u . [2 5]
tmp2 = B # u . [2 5]
v = A # tmp2 . [2 5]
```

Figure 19: Wrongly transformed interpolation

Assuming that `u` will not be needed anymore, it is however possible to reuse a variable in another statement than where it is currently needed. The resulting code is shown in Figure 20.

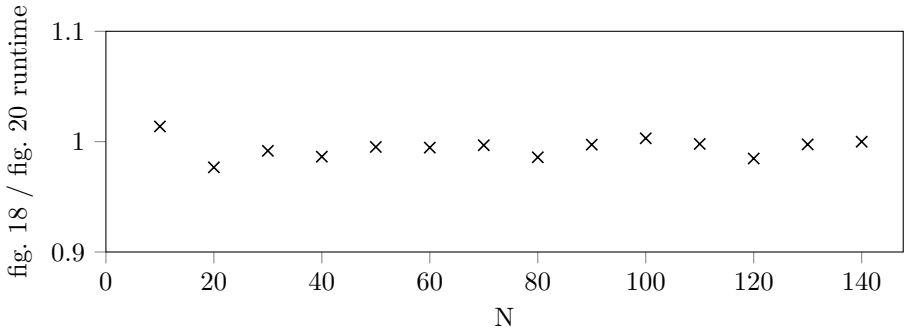
```
v = C # u . [2 5]
u = B # v . [2 5]
v = A # u . [2 5]
```

Figure 20: Correct transformed interpolation

The basic idea behind this optimization is the same as for the reduce-cache-miss-optimization. By reducing the amount of memory used at a time, it may be possible for the data to be in a lower level memory. Comparing the implementation of the code from Figure 20 (shown in appendices, page 29, Figure 26b) with Figure 18 (shown in Figure 26a) yields Figure 21.

As the L3 cache size is 30M and a single rank 3 tensor for N = 140 is around 21M, a possible speedup for Figure 20 should have appeared before this N.

Figure 21: Interpolation



However, this is not the case. While the specific cause is not known to the author, two possible reasons are given.

The first reason could be that the space complexity of the "saved variables", in this case `tmp1` and `tmp2` with $\mathcal{O}(N^3)$ is again smaller than the time complexity of the kernel with $\mathcal{O}(N^4)$ meaning that for $N > 100$, the difference is too small to be measurable. However, due to the sum from contraction, the access pattern is not as simple as before and this might not be a valid reason.

The second possible reason is that amount of memory used in a single statement is still the same. While the memory is reduced for the whole program, the cache misses might only occur the first time the data is needed and might be negligible.

7 Evaluation of data sizes for parallelization

While the optimizations based on compatible statements presented in this thesis can be applied in both sequential and parallel computation, the speedup might depend on the method which is used. For example, the reduce-cache-miss optimization has been evaluated using a single core as for small data sizes (e.g. a bit larger than the size of the L1 cache) the variance would be too high due to thread overhead. As multiple cores often share the same L3 cache, the measurements might be different for multiple cores. It is therefore useful to find data sizes (N) where one might switch between sequential and parallel computation as thread benefit outweigh thread overhead in order to understand the results from the previous sections better.

Doing this not only makes sense for the kernels analysed previously or even only kernels using compatible statements. While trying to optimize the interpolation operator in Section 6.3 gave a negative result, this does not mean that similar optimizations are not possible. Finding the N between sequential and parallel computations for different kernels makes sense in general and can help to optimize and exploit parallelism better.

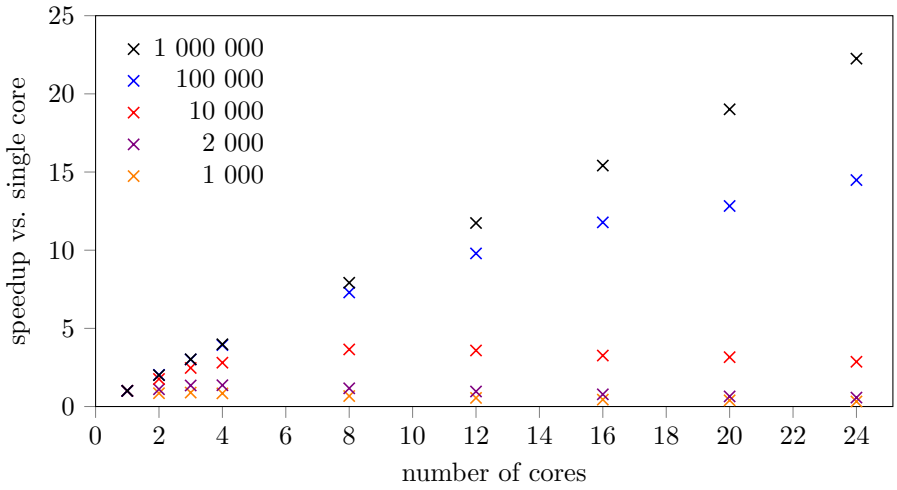
This section analyses a number of different kernels. The first two are scalar- and elementwise operations due to their role in compatible statements. The 3rd kernel is the interpolation operator due to its usage in computational fluid dynamics and CFDlang [8]. For similar reasons, a stencil kernel is chosen, generally also used in computational fluid dynamics and convolution. The last kernel is matrix-matrix multiplication because of its general significance in a

wide range of problems.

The N at which one might apply parallelism is not always the same and depends on the problem. For some problems, it might be sufficient to achieve a speedup of e.g. 1.3x using two cores while for other problems one might only apply parallelism if it gives a linear speedup, i.e. 2x. To accommodate these different needs, one method is to fix different N's for a kernel and plot the speedup with the number of cores as a variable.

The plots for scalar multiplication with N = 1 000 000 and N = 10 000 were already shown in Figure 1 and Figure 2 but are shown again in Figure 22. Even with N = 10 000 one can still achieve a speedup of 2.8x for 4 cores which might be enough for some use cases. It is therefore useful to look at even smaller N to see where one might switch from sequential to parallel computation. For N = 1 000, it is not possible to achieve a speedup when using 2 cores, so one would not apply parallelism in this case. However for N = 2 000, a speedup of 1.3x is achieved using 3 cores. The decision on whether to apply parallel or sequential computation in this case depends on the use case. Measuring elementwise multiplication yields almost the same results as scalar multiplication. For N = 1 000, no speedup can be achieved and N = 2 000 allows for a speedup of 1.3x using 3 cores.

Figure 22: Scalar multiplication, N in legend

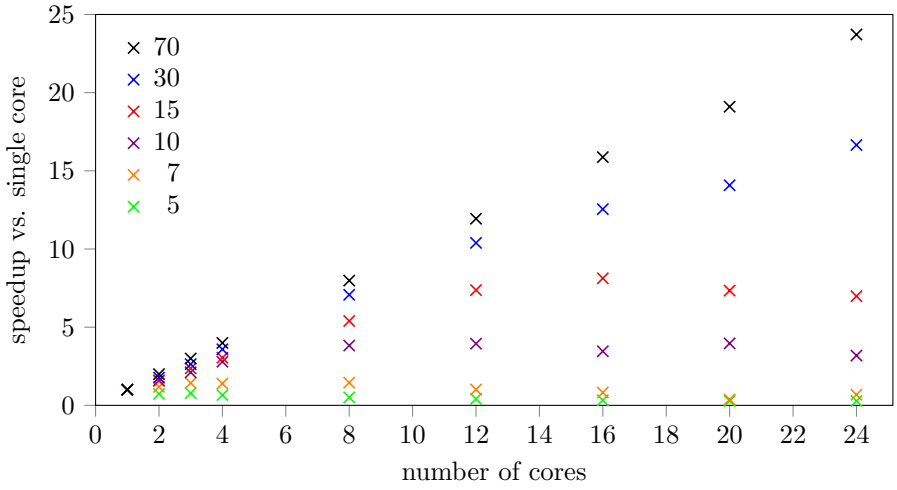


Scalar and elementwise operations have a very low runtime complexity of $\mathcal{O}(N)$. As the thread overhead is mostly constant for some fixed number of cores and the thread benefit increases approximately linear with runtime, an increasing runtime complexity corresponds to lower N needed for thread benefit to outweigh overhead. Therefore the following kernels, which have a higher runtime complexity, need lower N to a achieve linear speedup.

The interpolation operator has $\mathcal{O}(N^4)$ runtime complexity and very small N's are needed. As shown in Figure 23, for N = 10, it is already possible to achive a speedup similar to N = 10 000 for scalar multiplication, i.e. 2.8x for 4 cores. Decreasing N further to N = 7, it is still possible to achieve 1.2x with 2 and 1.4x with 3 cores. For N = 6, the result is a speedup of 1.2x with 3 cores but no speedup with any other number of cores. This is also the lowest N where

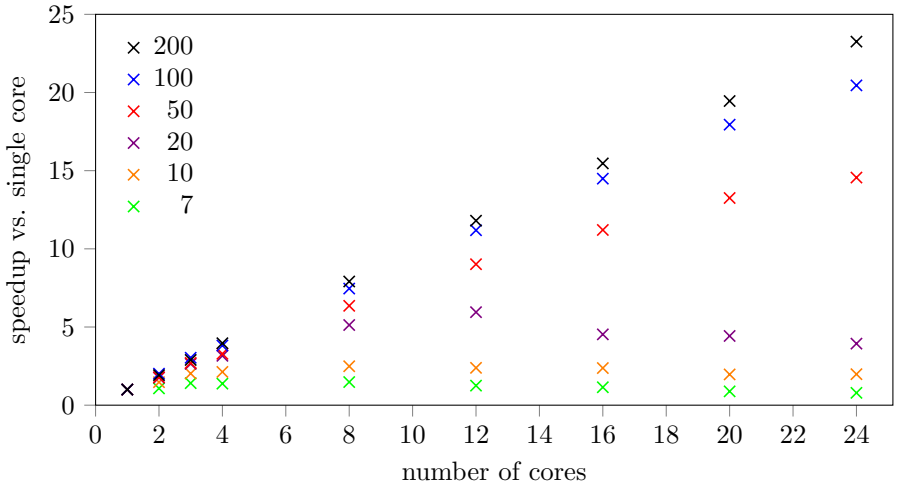
one might apply parallelism, as for N = 5 using any number of cores does not give a speedup.

Figure 23: Interpolation, N in legend



As can be seen from Figure 24, approximately the same holds true for the stencil operator (code in appendices, Figure 27, page 30). For N = 10, it is possible to achieve a speedup of 2.5x with 8 and 2.2x with 4 cores. Decreasing N to 7 still gives a speedup of 1.4x with 3 cores, N = 6 results in 1.2x for 3 cores and parallelizing N = 5 with multiple cores does not give any speedup.

Figure 24: Stencil, N in legend

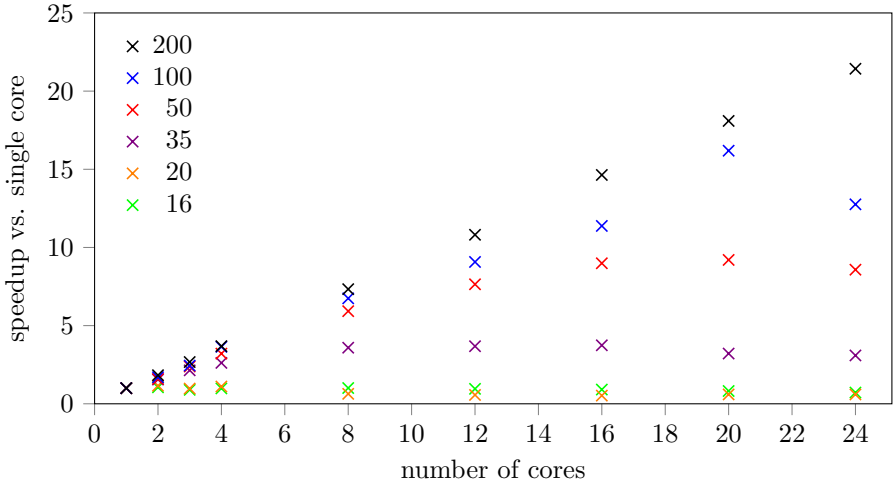


The last kernel, matrix-matrix multiplication shown in Figure 25 (code in appendices, Figure 28, page 30) is a bit more difficult to analyse. This is due to an additional parameter, the tile size, which has to be changed depending on both N and the number of cores. While one could analyse a matrix-matrix multiplication without tiling, in some cases tiling is a decisive factor on whether parallelization makes sense. As an example, consider the kernel with N = 20 and

2 cores. Tiling does not make a noticable difference when using a single core, however using 2 cores with tiling results in a speedup of 1.15x while no tiling gives a slowdown. This is also true for even smaller N like N = 16 where the matrix can be split into tiles of 8x8 elements. Doing so gives 4 tiles which can be distributed across 2 cores, resulting in a speedup of 1.05x while any other N like 15 or 17 only results in a slowdown when split across cores. Therefore there is no N which separates usage of sequential and parallel computation. Instead, the usage depends on the specific N.

As finding the optimal tiling size is a problem on its own, the size has been adjusted manually and the best performing tiling size has been selected. Note that this has not been done for every data point but instead for a group of points with similar N and number of cores, e.g. N = 200 and #cores = 8 shares the same tiling size with N = 200 and #cores = 12 even though it may not be the optimal size. The real values might therefore be slightly different, however the error should be reasonable.

Figure 25: Matrix-matrix multiply, N in legend



8 Summary

This thesis studied the detection and exploitation of data-parallelism in tensor assignments. By proposing a mechanism for detecting data-parallelism and using this to optimize statements in a domain specific language, it has been shown that these optimizations can lead to a reasonable speedup. While the speedup of up to 2x can only be achieved for a few simple kernels, a speedup of a few percent can still be achieved for a larger number of kernels. Also, even though optimizing the interpolation kernel using a similar optimization does not result in a measurable speedup, this does not imply that similar optimizations are not possible.

9 Outlook

While the speedup which can be obtained for specific kernels is analyzed, this does not represent the speedup which would be obtained for more complex problems which often use multiple different kernels. It would therefore be useful to take programs using different kernels, analyse what number of them could be optimized and what the actual resulting speedup would be. Furthermore, while no speedup has been achieved for the interpolation kernel, similar optimizations might be possible and could be analysed further.

Concerning the DSL and the corresponding type system, further extensions might be possible. An example would be nonlinear operations like activation functions often used in neural networks. These elementwise activation functions behave essentially the same as parenthesis from an index perspective, i.e. a rule would simply propagate the index down. Another possible concept which could be introduced is convolution.

Going even further, one could consider allowing more than one element to be accessed at a time. For example, it is possible to calculate convolution of a $N * N$ matrix with kernel size $2K + 1$ using only $K * N$ temporary memory for storing the previous rows. While this would not improve copy performance as the same amount of data is still copied, the amount of data used at one time is smaller which might improve cache performance. This is especially the case if the filter size is small and the matrix is large as often the case with simple image filters, e.g. filter size = 3 so $K = 1$, and $N = 1024$. It might also be possible, to generalize the concept of accessing multiple elements to other kernels.

Lastly, Conjecture 1 about the optimality of the type system still needs to be proved.

References

[1] G. Baumgartner et al. "Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models". In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 276–292. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840311.

[2] *ITensor: Intelligent Tensor Library*. http://itensor.org/. Accessed: 2018-07-01.

[3] F. Kjolstad et al. "The Tensor Algebra Compiler". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 77:1–77:29. ISSN: 2475-1421. DOI: 10.1145/3133901. URL: http://doi.acm.org/10.1145/3133901.

[4] X. Li et al. "Performance Analysis of GPU-Based Convolutional Neural Networks". In: *2016 45th International Conference on Parallel Processing (ICPP)*. Aug. 2016, pp. 67–76. DOI: 10.1109/ICPP.2016.15.

[5] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[6] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.1*. https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf. 2011.

[7] N. A. Rink. *Modeling of languages for tensor manipulation*. 2018. arXiv: 1801.08771.

[8] N. A. Rink et al. "CFDlang: High-level Code Generation for High-order Methods in Fluid Dynamics". In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*. RWDSL2018. Vienna, Austria: ACM, 2018, 5:1–5:10. ISBN: 978-1-4503-6355-6. DOI: 10.1145/3183895.3183900. URL: http://doi.acm.org/10.1145/3183895.3183900.

[9] P. Springer, T. Su, and P. Bientinesi. "HPTT: A High-Performance Tensor Transposition C++ Library". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY 2017. Barcelona, Spain: ACM, 2017, pp. 56–62. ISBN: 978-1-4503-5069-3. DOI: 10.1145/3091966.3091968. URL: http://doi.acm.org/10.1145/3091966.3091968.

[10] N. Vasilache et al. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. 2018. arXiv: 1802.04730.

[11] *XLA: Accelerated Linear Algebra*. Accessed: 2018-06-30. URL: https://www.tensorflow.org/performance/xla/.

Appendices

```
for (int i1 = 0; i1 < N; i1++) {
  for (int j1 = 0; j1 < N; j1++) {
    for (int k1 = 0; k1 < N; k1++) {
      tmp1[i1][j1][k1] = 0.0;
      for (int l = 0; l < N; l++) {
        tmp1[i1][j1][k1] +=
          C[i1][l] * u[j1][k1][l];
      }
    }
  }
}

for (int i2 = 0; i2 < N; i2++) {
  for (int j2 = 0; j2 < N; j2++) {
    for (int k2 = 0; k2 < N; k2++) {
      tmp2[i2][j2][k2] = 0.0;
      for (int l = 0; l < N; l++) {
        tmp2[i2][j2][k2] +=
          B[i2][l] * tmp1[j2][k2][l];
      }
    }
  }
}

for (int i3 = 0; i3 < N; i3++) {
  for (int j3 = 0; j3 < N; j3++) {
    for (int k3 = 0; k3 < N; k3++) {
      v[i3][j3][k3] = 0.0;
      for (int l = 0; l < N; l++) {
        v[i3][j3][k3] +=
          A[i3][l] * tmp2[j3][k3][l];
      }
    }
  }
}
```

(a) implementation of Figure 18 (untransformed)

```
for (int i1 = 0; i1 < N; i1++) {
  for (int j1 = 0; j1 < N; j1++) {
    for (int k1 = 0; k1 < N; k1++) {
      v[i1][j1][k1] = 0.0;
      for (int l = 0; l < N; l++) {
        v[i1][j1][k1] +=
          C[i1][l] * u[j1][k1][l];
      }
    }
  }
}

for (int i2 = 0; i2 < N; i2++) {
  for (int j2 = 0; j2 < N; j2++) {
    for (int k2 = 0; k2 < N; k2++) {
      u[i2][j2][k2] = 0.0;
      for (int l = 0; l < N; l++) {
        u[i2][j2][k2] +=
          B[i2][l] * v[j2][k2][l];
      }
    }
  }
}

for (int i3 = 0; i3 < N; i3++) {
  for (int j3 = 0; j3 < N; j3++) {
    for (int k3 = 0; k3 < N; k3++) {
      v[i3][j3][k3] = 0.0;
      for (int l = 0; l < N; l++) {
        v[i3][j3][k3] +=
          A[i3][l] * u[j3][k3][l];
      }
    }
  }
}
```

(b) implementation of Figure 20 (transformed)

Figure 26: Code for the interpolation kernel

```
#pragma omp parallel for collapse(2)
for (int k = 0; k < STENCIL_CHANNELS; k++) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            atmp[k][i][j] = 0.0;
            for (int ip = 0; ip < STENCIL_SIZE; ip++) {
                for (int jp = 0; jp < STENCIL_SIZE; jp++) {
                    if (i + ip < N && j + jp < N) {
                        atmp[k][i][j] += a[k][i + ip][j + jp] * S[ip][jp];
                    }
                }
            }
        }
    }
}

#pragma omp parallel for collapse(2)
for (int k = 0; k < STENCIL_CHANNELS; k++)
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            a[k][i][j] = atmp[k][i][j];
```

Figure 27: Code for the stencil kernel

```
#pragma omp parallel for collapse(2)
for (int it = 0; it < N; it += TILE_SIZE) {
    for (int jt = 0; jt < N; jt += TILE_SIZE) {
        for (int i = it; i < it + TILE_SIZE; i++) {
            for (int j = jt; j < jt + TILE_SIZE; j++) {
                Mtmp[i][j] = 0.0;
                for (int k = 0; k < N; k++) {
                    Mtmp[i][j] += M1[i][k] * M2[k][j];
                }
            }
        }
    }
}

#pragma omp parallel for
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        M1[i][j] = Mtmp[i][j];
```

Figure 28: Code for matrix-matrix multiplication