Fachhochschule Westküste
Hochschule für Wirtschaft & Technik

HAW HAMBURG

Master's thesis

Enrico Uhlenberg

Computational optimization of an integration method for explicit time-discrete multilinear models

Fachhochschule Westküste
Fachbereich Technik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Information and Electrical Engineering

Hamburg University of Applied Sciences
Faculty of Engineering and Computer Science
Department Information and Electrical Engineering

Enrico Uhlenberg

Computational optimization of an integration method for explicit time-discrete multilinear models

Master's thesis submitted as part of the Master's examination
in the joint Master's degree program Microelectronic Systems
at the Department of Engineering
of the Fachhochschule Westküste
and
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
at the Hamburg University of Applied Sciences

Supervising examiner: Prof. Dr. Klaus Jünemann
Second reviewer: Prof. Dr. Kristina Schädler

Submitted on: August 7, 2025

Enrico Uhlenberg

Thema der Arbeit

Computergestützte Optimierung einer Integrationsmethode für explizite zeitdiskrete multilineare Modelle

Stichworte
Multilineare Systeme, rechnergestützte Optimierung, SIMD-Vektorisierung, Algorithmen für dünnbesetzte Matrizen, Hochleistungsrechnen, Regelungstechnik

Kurzzusammenfassung
Diese Arbeit befasst sich mit der rechnerischen Optimierung von Simulationsmethoden für explizite zeitdiskrete multilineare Modelle. Aktuelle Implementierungen basieren auf proprietärer Allzwecksoftware, was die Ausführungszeiten und damit den praktischen Einsatz multilinearer Modelle für größere Anwendungen potenziell einschränkt. Durch systematische algorithmische Analyse und gezielte Low-Level-Optimierungstechniken entwickelt diese Arbeit eine neuartige Hochleistungsimplementierung, die im Vergleich zur Referenzimplementierung eine bis zu 72-fache Beschleunigung erzielt. Der Optimierungsansatz kombiniert die Darstellung spärlicher Matrizen im Compressed Sparse Column (CSC)-Format, eine benutzerdefinierte SIMD-Vektorisierung unter Verwendung von AVX2-Befehlen, Loop-Fusion-Techniken und Fused Multiply-Add (FMA)-Umformulierungen. Die Arbeit liefert sowohl theoretische Erkenntnisse zu Optimierungsstrategien für multilineare Systeme als auch eine frei verfügbare Open-Source-Softwarelösung, die die rechnerische Machbarkeit multilinearer Steuerungssysteme erheblich verbessert.

Enrico Uhlenberg

Title of Thesis

Computational optimization of an integration method for explicit time-discrete multilinear models

Keywords
multilinear systems, computational optimization, SIMD vectorization, sparse matrix algorithms, high-performance computing, control systems simulation

Abstract
This thesis addresses the computational optimization of simulation methods for explicit time-discrete multilinear models. Current implementations rely on proprietary general purpose software, potentially limiting execution times and thereby the practical deployment of multilinear models for larger applications. Through systematic algorithmic analysis and targeted low-level optimization techniques this work develops a novel high-performance implementation that achieves up to 72× speedup compared to the reference implementation. The optimization approach combines sparse matrix representation in Compressed Sparse Column (CSC) format, custom SIMD vectorization using AVX2 instructions, loop fusion techniques, and fused multiply-add (FMA) reformulations. The work contributes both theoretical insights into multilinear system optimization strategies and a freely available, open-source software solution that significantly advances the computational feasibility of multilinear control systems.

Contents

List of Figures vii

List of Tables viii

1 Motivation 1

1.1 Limitations of linear models . . . 1

1.2 Multilinear Systems: Structured Nonlinearity . . . 2

1.3 Computational Opportunity . . . 2

2 Theoretical Background 4

2.1 Linear Systems . . . 5

2.2 Multilinear Systems . . . 5

2.2.1 Matrix representation . . . 6

2.2.2 Tensor representation . . . 6

2.3 Application of tensor decomposition . . . 7

2.4 HPC Considerations . . . 9

2.4.1 Compressed Sparse Column Matrix Format . . . 9

2.4.2 Single Instruction Multiple Data . . . 10

2.5 State of research . . . 11

2.5.1 Current Implementation . . . 11

3 Requirements 13

3.1 Performance . . . 13

3.2 Compatibility . . . 13

3.3 Maintainability . . . 13

4 Methodology 15

4.1 Data preparation . . . 15

4.1.1 Model selection . . . 15

4.2 Benchmarking . . . 16

4.3 Profiling . . . 16
4.3.1 Hotspot Analysis . . . 17
4.3.2 HPC Performance Characterization . . . 17
4.4 Algorithmic analysis . . . 17
4.4.1 Sparse Matrix Logistics . . . 18
4.4.2 Precomputation of constants . . . 18
4.4.3 Reformulation towards FMA operations . . . 18
4.4.4 Loop Fusion . . . 20
4.4.5 Parallelism . . . 21

5 Implementation **22**
5.1 Sparse linear algebra library . . . 22
5.2 Vectorized linear algebra library . . . 24
5.3 Custom SIMD Implementation . . . 26
5.3.1 Preprocessor Directives . . . 28
5.3.2 Aligned data structures . . . 28
5.3.3 SIMD Scalar Add . . . 29
5.3.4 Gather . . . 30
5.3.5 Accumulate . . . 31
5.4 Custom SIMD Implementation with Loop fusion . . . 32
5.4.1 Combined Gather and Accumulate . . . 33

6 Results **35**
6.1 Overall Performance Comparison . . . 35
6.2 Profiling Analysis Results . . . 36
6.3 Scalability . . . 37
6.4 Precision Analysis . . . 39

7 Discussion and further research **40**
7.1 Algorithmic and Computational Insights . . . 40
7.2 Portable and performant simulation solution . . . 42
7.3 Conclusion . . . 42

Bibliography **43**

A Appendix **45**

Declaration of Authorship **73**

List of Figures

1.1 Sample System Two Rooms 2

2.1 CP Decomposed Sample System with monomial vector 8
2.2 CPN-1 Decomposed Sample System with monomial vector 9
2.3 CSC Representation in Memory 10

5.1 Hotspot analysis for oneMKL implementation 23
5.2 Hotspot analysis for oneMKL vectorized implementation 26
5.3 Hotspot analysis for custom SIMD implementation 27
5.4 Hotspot analysis for custom SIMD implementation with loop fusion 32

6.1 Performance progression through implementation stages 36
6.2 CPU time distribution comparison: (left) oneMKL implementation, (right) custom SIMD implementation with loop fusion 37
6.3 Performance comparison for CSTR model implementations demonstrating scalability characteristics across varying timestep ranges. 38
6.4 Model size scaling comparison between Custom (loop fusion) implementation and MATLAB baseline, showing execution time vs. number of states on logarithmic scales with power-law fit lines. 38

List of Tables

4.1 Comparison of Model Parameters 16
4.2 Simulation parameters for Benchmarking 16

5.1 Simulation time of MATLAB and oneMKL implementations 23
5.2 Simulation time of oneMKL vector and sparse implementations 25
5.3 Simulation time of custom SIMD and oneMKL implementation 27
5.4 Simulation time of custom SIMD implementations with and without loop fusion 33

6.1 Overall Performance Comparison of All Implementations 35
6.2 Precision vs Performance Trade-off Analysis 39

1 Motivation

The simulation of multilinear systems—mathematical models that capture structured nonlinear interactions between system variables represents a promising field of research modern control engineering, with computational optimization being less explored than in comparable disciplines due to its novelty. Current implementations, like the MTI-TOOLBOX developed in the high-level environment MATLAB, are held back by the proprietary and closed-source back-ends used for critical computations. This computational bottleneck not only limits the scale of problems that can be addressed but also prevents the broader adoption of multilinear modeling in time-critical control systems where their enhanced modeling capabilities could be utilized for a wider range of applications. This thesis aims to explore this opportunity utilizing only open-source tools, suitable for high performance computing.

1.1 Limitations of linear models

Linear models are ubiquitous in control engineering due to their simplicity and well-established tools. However, their fundamental assumption — that system variables interact additively — falls short under real-world complexity.

Consider a simple thermal system: Two adjacent rooms separated by a wall and a door (Figure 1.1). When the door is closed, heat transfer through the wall can be approximated linearly. Yet, opening the door introduces heat flux dependent on both the temperature difference ($\Delta T = T_1 - T_2$) and the door's status ($u_1 \in \{0, 1\}$). This seemingly trivial interaction already surpasses the realm of purely linear systems.

Note: For clarity of exposition, this simplified example omits detailed thermodynamic considerations such as thermal capacity, convection effects, and radiation. The parameters U_{Wall} and U_{Door} represent composite heat transfer coefficients that encompass thermal conductivity, convective heat transfer coefficients, and geometric factors.

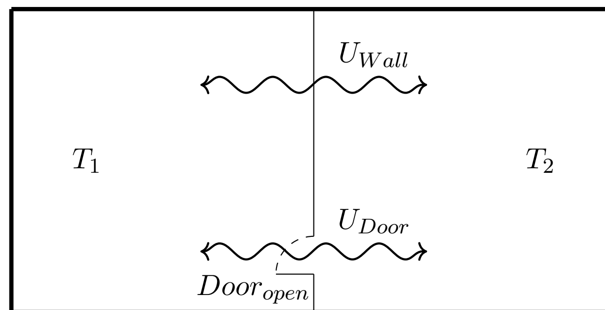


Figure 1.1: Sample System Two Rooms

In order to circumvent these limitations, workarounds such as piece wise linearization or conservative control strategies are often deployed.[1] These mitigations, while solving the problem at hand, introduce drawbacks such as loss of interpretability, or sub-optimal performance.

1.2 Multilinear Systems: Structured Nonlinearity

Multilinear systems expand the scope of linear systems by encoding structured nonlinear (multilinear) interactions. Unlike black-box neural networks or similar unstructured approaches, multilinear frameworks:

- Preserve interpretability of resulting models, enabling physical interpretation,
- Allow for memory efficient representations via decompositions, as described in Section 2.3

For the two-room system, this means that adding the door's state as an additional input enables the multilinear framework to capture the nonlinear heat transfer dynamics while maintaining model interpretability.

1.3 Computational Opportunity

The introduction of vectorized instruction sets in modern CPUs present the opportunity to apply the multilinear paradigm to larger problem sets, which have previously been limited to more computationally expensive or less accurate alternatives. This thesis addresses this opportunity by:

- Establishing benchmarks for scalability and performance of simulating multilinear systems
- Exploring and documenting potential algorithmic and computational approaches for the operations involved
- Developing a portable and performant tool for simulating multilinear systems based on empirical research

2 Theoretical Background

In order to illustrate how multilinear modelling extends the linear framework, the necessary core principals will be introduced using the example from Section 1.1:

The temperature of two rooms (T_1, T_2), with one door connecting ($Door_{Open}, U_{Door}$) them is to be predicted. For the purpose of this example heat is only transferred between the two rooms (U_{Wall}), no heat is lost elsewhere. To predict the future temperature of the rooms ($\dot{x_1}, \dot{x_2}$), the system's states, input and parameters are defined in (2.1) with subscript $k+1$ representing the respective state at the next time step k. For the sake of readybility, the notation of continuous time systems ($\dot{x}$ and x) is used to denote the current and next states. The underlying system is a discrete time system, with the next state x_{k+1} being computed from the current state x_k and the input u_k.

$$\begin{aligned} T_1 &\coloneqq x_1 \in \mathbb{R} \\ T_2 &\coloneqq x_2 \in \mathbb{R} \\ Door_{Open} &\coloneqq u_1 \in \{0, 1\} \\ U_{Wall} &\coloneqq f_1 \in \mathbb{R} \\ U_{Door} &\coloneqq f_2 \in \mathbb{R} \end{aligned} \tag{2.1}$$

2.1 Linear Systems

If the door is closed ($u_1 == u_k == 0$), the system behaves purely linear and is described by 2.2.

$$\begin{aligned} \dot{x_1} =& x_1 + f_1(x_2 - x_1) \\ \dot{x_2} =& x_2 + f_2(x_1 - x_2) \end{aligned} \tag{2.2}$$

In order to efficiently compute the state of linear system with $n \in \mathbb{N}$ states and $m \in \mathbb{N}$ inputs, the equation

$$x_{k+1} = x_k A + u_k B \tag{2.3}$$

with $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ is considered.

Applying 2.3 to this system yields the following matrix multiplication.

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \times \begin{bmatrix} 1 - f_1 & f_1 \\ \\ 1 - f_1 & f_1 \end{bmatrix} \tag{2.4}$$

2.2 Multilinear Systems

If the door were to be opened (2.2) the differential equations

$$\begin{aligned} \dot{x_1} =& x_1 + f_1(x_2 - x_1) + u_1 f_2(x_2 - x_1) \\ \dot{x_2} =& x_2 + f_1(x_1 - x_2) + u_1 f_2(x_1 - x_2) \end{aligned} \tag{2.5}$$

fully describe this system for all possible inputs. The newly added terms in 2.5 introduce the multiplicative couplings indicative of multiliinear systems between multiple inputs and states.

2.2.1 Matrix representation

The generalized formulation corresponding to (2.3), retaining equivalent dynamical structure, is expressed as

$$x_{k+1} = Fm(x_k, u_k) \tag{2.6}$$

with the coefficient matrix $F \in \mathbb{R}^{n\times(2^{n+m})}$ and monomial vector $m(x,u) \in \mathbb{R}^{2^{n+m}\times 1}$. Applying this formulation to the example system yields

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} 0 & 1-f_1 & f_1 & 0 & 0 & -f_2 & f_2 & 0 \\ 0 & 1-f_1 & f_1 & 0 & 0 & f_2 & -f_2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ u_1 \\ x_1x_2 \\ x_1u_1 \\ x_2u_1 \\ x_1x_2u_1 \end{bmatrix} \tag{2.7}$$

. The coefficient matrix F is the multilinear equivalent of matrices A and B from 2.3. Vector $\mathbf{m}(\mathbf{x},\mathbf{u})$ comprises all possible combinations of x and u and is computed by

$$\mathbf{m}(\mathbf{x},\mathbf{u}) = \begin{bmatrix} 1 \\ x_1 \end{bmatrix} \otimes \cdots \otimes \begin{bmatrix} 1 \\ x_n \end{bmatrix} \otimes \begin{bmatrix} 1 \\ u_1 \end{bmatrix} \otimes \cdots \otimes \begin{bmatrix} 1 \\ u_m \end{bmatrix}, \tag{2.8}$$

where $\otimes$ denotes the Kronecker product.

2.2.2 Tensor representation

Understanding tensor operations is essential for the computational optimizations developed in this thesis, as the contracted product forms the core computational kernel that will be accelerated through algorithmic and hardware-level optimizations.

Following the mathematical framework established by Samaniego et al. [10], a tensor $\mathsf{X} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ is an N-dimensional arrangement of elements with entries denoted by $x(i_1, i_2, \ldots, i_N)$, where the indices $i_j \in \{1, 2, \ldots, I_j\}$ correspond to each dimension $j = 1, \ldots, N$. Tensors of dimension one and two correspond to vectors and matrices, respectively.

The mathematical foundation linking tensors to multilinear state space models was established in [2], while the computational approach using contracted products was detailed in [3]. In MTI systems, the state transition relies on the contracted product operation between tensors $\mathsf{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N \times J_1 \times \cdots \times J_M}$ and $\mathsf{Y} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, defined as:

$$\mathsf{Z} = \langle \mathsf{X} | \mathsf{Y} \rangle \in \mathbb{R}^{J_1 \times \cdots \times J_N}$$

with elements computed through the summation:

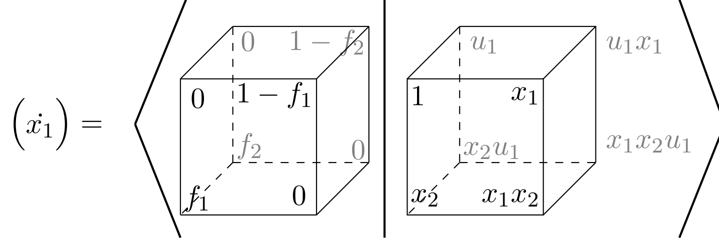
$$z(j_1, \ldots, j_M) = \sum_{i_1=1}^{I_1} \cdots \sum_{i_N=1}^{I_N} y(i_1, \ldots, i_N) x(i_1, \ldots, i_N, j_1, \ldots, j_M)$$

The state equations for inputs $\mathbf{u} \in \mathbb{R}^m$ and states $\mathbf{x} \in \mathbb{R}^n$ are expressed as:

$$\dot{\mathbf{x}} = \langle \mathsf{F} | \mathsf{M}(\mathbf{x}, \mathbf{u}) \rangle \tag{2.9}$$

where the state transition tensor $\mathsf{F} \in \mathbb{R}^{\overbrace{2 \times \cdots \times 2}^{n+m} \times n}$ and monomial tensor $\mathsf{M}(\mathbf{x}, \mathbf{u}) \in \mathbb{R}^{\overbrace{2 \times \cdots \times 2}^{n+m}}$ contain all possible combinations of state and input components [10].

The state equation for x_1 of the example system can thereby visualized by



2.3 Application of tensor decomposition

The software implemented in this thesis targets MTI models comprised of canonically polyadic decomposed (CP) tensors as introduced by Kruppa [4] . An MTI model in CP decomposition can be represented by constructing one factor matrix for each state and input encoding their respective influence on the trajecotry of the system with one

	1	x_1	x_2	u_1	x_1x_2	x_1u_1	x_2u_1	$x_1x_2u_1$
$\mathbf{F}_{x_1} =$	1	0	1	1	0	0	1	0
	0	1	0	0	1	1	0	1
$\mathbf{F}_{x_2} =$	1	1	0	1	0	1	0	0
	0	0	1	0	1	0	1	1
$\mathbf{F}_{u} =$	1	1	1	0	1	0	0	0
	0	0	0	1	0	1	1	1
$\mathbf{F}_{\phi} =$	0	$1-f_1$	f_1	0	0	$-f_2$	f_2	0
	0	$1-f_1$	f_1	0	0	f_2	$-f_2$	0

Figure 2.1: CP Decomposed Sample System with monomial vector

additional factor matrix containing the multilinear coefficients. Following the notation introduced by Kolda and Bader [5] the model can be represented by

$$\mathrm{F} = [\mathbf{F}_{x_1}, ..., \mathbf{F}_{x_n}, \mathbf{F}_{u_1}, ..., \mathbf{F}_{u_m}, \mathbf{F}_{\phi}] \tag{2.10}$$

with the calculation to obtain the next state from current state and inputs can be described as

$$\dot{\mathbf{x}} = \mathbf{F}_{\phi}\left(\left(\mathbf{F}^{\mathrm{T}}_{x_1}\begin{bmatrix}1\\x_1\end{bmatrix}\right) \otimes ... \otimes \left(\mathbf{F}^{\mathrm{T}}_{x_n}\begin{bmatrix}1\\x_n\end{bmatrix}\right) \otimes \left(\mathbf{F}^{\mathrm{T}}_{u_1}\begin{bmatrix}1\\u_1\end{bmatrix}\right) \otimes ... \otimes \left(\mathbf{F}^{\mathrm{T}}_{u_m}\begin{bmatrix}1\\u_m\end{bmatrix}\right)\right) \tag{2.11}$$

with $\otimes$ being the Hadamard (element-wise) product. [6] The equation 2.11 is of particular interest as the optimization of this operation is the main focus of this thesis. Before exploring the computational details all factor matrices apart from $\mathbf{F}_{\phi}$ are normalized such that every column of each matrix add up to an absolute value of 1.

The example system shown in Figure 2.1 already satifies this constraint, no additional computation needs to done in this case. This normalization does allow for substantial memory efficiency optimizations, since the second for of every matrix only contains the complement value for satisfying the normalization constraint and can thusly be ommited.

Combining all remaining rows of the factor matrices into one matrix $\mathbf{F}_U$ results in the compact format of a CP-decomposed MTI model in 1-norm (CPN-1) shown in Fig 2.2, which will be the starting point of the computational optimizations of this thesis.

$$
\begin{array}{r c}
 & \begin{array}{cccccccc} 1 & x_1 & x_2 & u_1 & x_1x_2 & x_1u_1 & x_2u_1 & x_1x_2u_1 \end{array} \\
\mathbf{F}_U = & \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \\
\mathbf{F}_\phi = & \begin{bmatrix} 0 & 1-f_1 & f_1 & 0 & 0 & -f_2 & f_2 & 0 \\ 0 & 1-f_1 & f_1 & 0 & 0 & f_2 & -f_2 & 0 \end{bmatrix}
\end{array}
$$

Figure 2.2: CPN-1 Decomposed Sample System with monomial vector

The CP decomposition applied to 2.5 creates a structured representation where each colored column in matrices $\mathbf{F}_{x_1}$, $\mathbf{F}_{x_2}$, and $\mathbf{F}_u$ encodes a specific combination of states and inputs, directly corresponding to elements of the monomial vector $\mathbf{m}(\mathbf{x}, \mathbf{u})$ from Eq. 2.8. The coefficient matrix $\mathbf{F}_\phi$ completes the represtation of the system's dynamics, with each row defining the evolution of a particular state variable as shown in Fig. 2.1.

2.4 HPC Considerations

High Performance Computing (*HPC*) is a field of Computer Sciences focussing on efficient computation for scientific purposes such as climate simulations or real time controlling. The following chapter will explore the topics of this field relevant to optimization of eMTI simulations.

2.4.1 Compressed Sparse Column Matrix Format

The tensors descibing a eMTI system are - in spite of the CPN-1 decomposition - often very sparse. Due to this fact the models are stored as sparse matrices, enabling both memory- and performance-efficiency by omitting all zero-entries of the matrices. The specific representation used by both MATLAB and the reimplementation of this thesis is the Compressed Sparse Column format (*CSC*) introduced by Tewarson [7]. In this format the non-zero values are stored continoously column-wise with two auxillary arrays. One array stores the rows of all non-zero values, while the second stores the starting index of each column within the values as shown in Fig. 2.3.

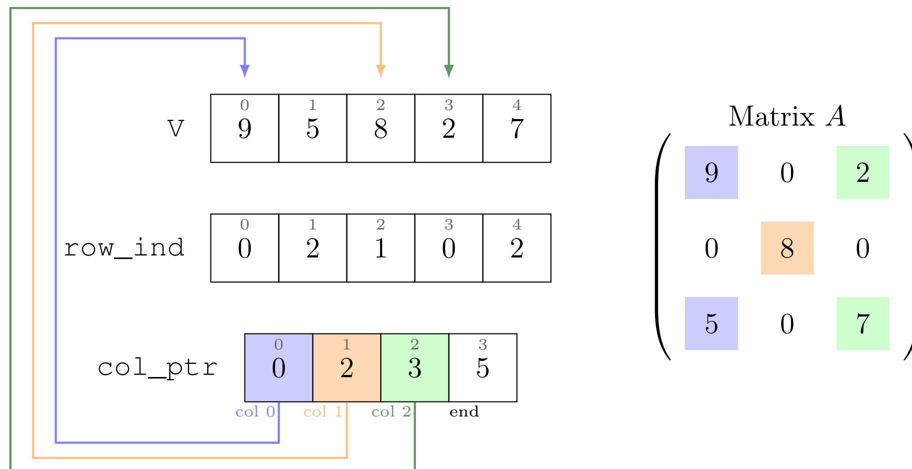


Figure 2.3: CSC Representation in Memory

2.4.2 Single Instruction Multiple Data

Single Instruction, Multiple Data (SIMD) is a parallel processing paradigm focussing on accelerating computation on CPUs, where a single instruction operates on multiple data points simultaneously. For the purpose of this thesis, SIMD operations focussing on double precision floating point numbers extending the x86 instruction set [8] are of particular interest.

First introduced by Intel in 2011 [9] the Advanced Vector Extensions (AVX) are an extension of the x86 instruction set available on modern CPUs facilitating the simulataneous processing of up to eight floating point numbers with the newest iteration being AVX-512. With the goal of maximizing compatibility the methods implemented in this thesis implement the AVX version up to Version AVX2, since AVX-512 is only found on newer high-end CPUs thereby. The results of this thesis are thereby limited to processing capabilities to 4 double precision floating point numbers (256 bit) at a time. A port of the software to AVX-512 feasable without major rewriting of code, but is not within the scope of this work.

When implementing computational routines using AVX instructions, further performance gains can be achieved by arranging the computatations such that SIMD lanes (batches of four double floating point numbers) are processed as a combined multiplication and addition. This is due to the *Fused Multiply and Add* instruction introduced in 2013 with AVX2 [9] being able to process a multiplication followed by an addition with a single instruction and crucially a single rounding operation, thereby simultanously improving

speed and precision.
The listing 2.1 showcases a simple example of such a FMA operation using AVX intrinsics.

```
// Serial (scalar) version of the FMA operation
void serial_fma(double* a, double* b, double* c, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        c[i] = (a[i] * b[i]) + c[i];
    }
}
// Vectorized version using AVX FMA intrinsics for double precision
void vectorized_fma(double* a, double* b, double* c, size_t n) {
    for (size_t i = 0; i < n; i += 4) {
        __m256d vec_a = _mm256_load_pd(a + i);
        __m256d vec_b = _mm256_load_pd(b + i);
        __m256d vec_c = _mm256_load_pd(c + i);
        vec_c = _mm256_fmadd_pd(vec_a, vec_b, vec_c);
        _mm256_store_pd(c + i, vec_c);
    }
}
```

Listing 2.1: Example of AVX implementation of a FMA Operation

2.5 State of research

2.5.1 Current Implementation in MATLAB ®

The basis for the software implemented in this thesis is the software suite dedicated to multilinear systems by Lichtenberg et al. (*MTI-TOOLBOX*) as introduced by Samaniego [10]. The performance of the simulation routine for explicit time-discrete multilinear systems from the *MTI-TOOLBOX* will be serving as a baseline for later benchmarks.
While not extensively optimized for performance, the current implementation already leverages MATLAB's tools for efficiently handling sparse matrices for the neccesary non-linear operations as shown in Listing 2.2.

Listing 2.2: Original Simulation Routine from *MTI-TOOLBOX*

```
@(sys,xu)    ... Lambda function, two inputs
sys.phi *    ...
accumarray( ... for efficient use of sparse
        sys.UColInd,    ... Columns to accumulate
        1 + sys.UDataVec .* ... Data to operate on 1/2
        (xu(sys.URowInd) + sys.SignVec), ...        2/2
        [sys.rowsU 1], ... Output dimensions
        @prod,          ... Accumulation operation
        1               ... Fill-value for missing data
        );
```

3 Requirements

Due to the explorative nature of this thesis' topic and the purely scientific context in which in which it is written (without direct involvement of third party beneficiaries) the requirements described in the following section are exclusively self imposed. The software described in this thesis will, for brevity, be referenced as MSS-SIM.

3.1 Performance

The main objective of MSS-SIM is the acceleration of the computation shown in 2.2, which shall be verified by benchmarking execution times of the original implementation and MSS-SIM.

3.2 Compatibility

In order to maximize the potential impact of improved simulation performance, MSS-SIM shall be executable without the need for external libraries or proprietary dependencies. In order for this cross-platform capability not to impede performance, MSS-SIM shall be compilable with a free and open-source compiler enabling platform-specific optimizations.

3.3 Maintainability

MSS-SIM shall be developed such that future improvements and maintenance are as accessible as possible. This shall be achieved by following a consistent coding style,

extensive in-code comments, dedicated technical documentation and open source publication.

4 Methodology

4.1 Data preparation

For evaluation of the performance of MSS-SIM models generated within the *MTI-TOOLBOX* will be used, as both randomly synthesized models and real world models from active research are available in in the toolbox.

4.1.1 Model selection

For evaluation of the simulation performance, the models should be as simple as possible in order to isolate the operation at hand. Additionally simulation-time of the models should be sufficiently long in order to minimize the influence of potential startup delays and setup routines impertinent to the main calculation. Increasing runtime also contributes to more accurate profiling by decreasing the chance of inaccurate measurements due to undersampling.
Taking the aforementioned considerations into account, parameters for random model generation can be empirically determined (see Table 4.1). The real-world model is first described by Samaniego [11] is an interesting candidate for evaluating real world simulations, but the system described is too small for accurate profiling. To mitigate this, identical copies of the model are concatenated into one bigger composite model. The implementation of this approach is shown in Appendix A.2 resulting in a model with redundant state information but a desirable model size for accurate profiling.

Table 4.1: Comparison of Model Parameters

Model	States	Inputs	Model Rank
Random	700	35	147
Cstr	400	200	204 800

4.2 Benchmarking

In order to evaluate the performance gains of MSS-SIM, simulation paramaters (See Table 4.2) are chosen such that total simulation time serves as a meaningful measure for computational performance.

To ensure comparable results, MATLAB measurements are taken such that only the simulation routine is included in the total simulation time, omitting model creation and other setup routines. When benchmarking MSS-SIM the simulation time is measured within the same boundaries as the baseline measurement in MATLAB for well-founded comparisons.

Table 4.2: Simulation parameters for Benchmarking

Model	No. of timesteps	Simulation Time (s)
Random	9999	6.7
Cstr	60	15.3

4.3 Profiling

While the efficient computation of linear algebra is a very well researched field, in which custom implementations are almost always inferior to established HPC libraries, the same cannot be said about multilinear algebra. Although the *MTI-TOOLBOX* provides a solid basis for computation of such systems, in-depth optimization of the specific computations has not been attempted yet and thereby offers great potential for the results of this thesis. In order to firstly understand and secondly address the bottlenecks during these computations, a detailed insight into the runtime of a simulation routine on a instruction-level is of utmost importance.

Software-Profiling is a discipline dedicated to this exact goal. With the help of a profiling software, software engineers can inspect the runtime of existing programs and analyze every aspect pertinent to performance of said program including runtime of seperate functions within the program, memory bottlenecks and suboptimal utilization of instruction extensions such as SIMD.
The Intel VTune ™ Profiler is a state-of-the-art performance analyzer, widely recognized as a suitable solution for performance profiling in academic and industrial research. [12] [13] [14]

4.3.1 Hotspot Analysis

Hotspot analysis relies on the operating system's timer to sample the instruction pointer in combination with the complete callstack at a set interval of the program in question. At the end of the runtime, the profiler aggregates all recorded samples resulting in a detailed summary of the elapsed time for every function and instruction of the program. This information is crucial to identify areas of code, where spent time could be reduced by algorithmic optimizations or adjustments in technical implementation such as compiler hints.

4.3.2 HPC Performance Characterization

In addition to the overview gained by the hotspot analysis, a more detailed insight about the optimization opportunities on a instruction level is offered by the *HPC Performance Characterization* within the VTune Profiler. This feature provides in-depth informations about the x86 instructions generated during compilation and hardware-level runtime information such as underutilized vectorizations (i.e. AVX2). Additionally the HPC mode offers detailed information about missed cache accesses and similar memory constraints for even more granular optimizations. [15]

4.4 Algorithmic analysis

Analysis of the computational operations required for MSS-SIM is essential for developing effective optimization strategies. A possible implementation in pseudocode is shown in

Algorithm 2, which serves as the foundation for the algorithmic analysis presented in this section.

Examining the core computational kernels reveals several optimization opportunities that will be explored in detail. For readability, explicit references to Algorithm 2 will be omitted throughout this section.

4.4.1 Sparse Matrix Logistics

Lines 1 and 6 are necessitated by the use of underlying sparse matrix formats in combination with the non-linear operations of the algorithm. Since Algorithm 2 is derived from the original MATLAB implementation, these steps will depend heavily on the utilized sparse matrix representation of the implementation.
Line 1 can be omitted in MSS-SIM, since the starting point for the simulation is the raw CSC representation of the system matrices which already contain the information gained by this operation.
Line 6 on the other hand is non-optional since the result of simulating one timestep will always be the dense state vector. For the sparse column-wise compuations a reduction of that state vector according to the respective non-zeroes of F_U is imperative. Whether this reduced state-input vector should be computed in full like in Algorithm 2 or rather in small batches on-the-fly will be investigated in Section 4.4.4.

4.4.2 Precomputation of constants

The one's complement of F_U calculated in line 2 is constant across the length of the operation and can be precomputed, thereby saving valuable simulation time.

4.4.3 Reformulation towards FMA operations

For the purpose of the following optimization approach, some general assumptions about real-world eMTI systems are made: A system with n states and m inputs can be assumed to be of multilinear rank $r >> 1$, since systems $r == 1$ could be represented as a regular linear system and larger r increase the models' capability to capture more complex multilinear dynamics. Real world systems can also be assumed to have a sparse structural matrix F_U, because a dense F_U would mean every state to be correlated with every other

state in every possible multilinear combination. For reference, the real world example system identified by Samaniego used for benchmarking MSS-SIM exhibits a sparsity in F_U of 0.5.
Based on these assumptions the following optimization imperative can be derived: Operations involving xu_k should be isolated if possible, since the combined number of states and inputs will always be magnitudes fewer than the non-zero entries in F_U.

When fully expanded, the operation in line 7 can be expressed as

$$1 - F_U + F_U xu_k \tag{4.1}$$

, demonstrating currently 3 floating point operations (one addition, one subtraction, one multiplication) are required per non-zero entry of F_U. When reformulated as

$$1 + F_U(xu_k - 1) \tag{4.2}$$

the number of operations are unchanged from a mathematical perspective, but will yield considerable performance due to two key computational advantages of this reformulation. By separating xu_k from F_U, only $n + m$ subtractions are necessary instead of $nnz(F_U)$. More importantly, the remaining operations are now in the form $(a \cdot b) + c$, enabling the use of FMA operations as described in Section 2.4.2. This reduces the instructions needed to compute U_{xu} by up to 33%, due to the multiplication and addition being done in a single instruction.

Algorithm 2 Compute next timestep

Require: F_U, F_{Phi},x_0, u (Input Array), N (Number of Steps)
Ensure: $\widehat{x}$ (State Trajectory)

1: $U_{Rows}, U_{Cols}, U_{nnz} \leftarrow \text{find}(F_U)$ ▷ Find non-zero values
2: $U_{nnz,OC} \leftarrow 1 - U_{nnz}$ ▷ Compute one's complement of F_U
3: $x_k \leftarrow x_0$
4: **for** $k \leftarrow 0$ to N **do**
5: $\quad xu_k \leftarrow [x_k(:)\ u(k,:)]$
6: $\quad \tilde{xu}_k \leftarrow xu_k(U_{Rows})$
7: $\quad U_{xu_k} \leftarrow U_{nnz,OC} + U_{nnz} \otimes \tilde{xu}_k$ ▷ $\otimes$ being Hadamard product
8: $\quad$ **for** $i \leftarrow \text{length}(U_{Cols}) - 1$ **do** ▷ For each column of U
9: $\quad\quad l \leftarrow U_{cols}(i) - U_{cols}(i+1)$ ▷ Number of non-zeros for column
10: $\quad\quad v_1, v_2, \ldots, v_l \leftarrow U_{xu_k}(i : l)$
11: $\quad\quad \tilde{x}_{k+1}(i) \leftarrow \prod_{j=1}^{l} v_j$ ▷ Product of grouped values
12: $\quad$ **end for**
13: $\quad x_k = F_{Phi} \cdot \tilde{x}_{k+1}$
14: $\quad \widehat{x}(k) = x_k$
15: **end for**

4.4.4 Loop Fusion

Loop fusion is a optimization technique, by which multiple loops in a program are fused into one loop with a single index variable. It's usually applied by compilers to reduce overhead due to loop control structures. [16] In critical parts of high performance code it can also be applied manually, to additionally achieve performance gains such as improved cache-locality.

This technique has a potential application for MSS-SIM, since the column-specific state-input vector can be either computed before the innermost calculation loop or gathered as needed for each SIMD operation. Precomputing the vector would potentially accelerate the gathering of the state-input vector itself, while incorporating it in the innermost loop would improve cache-locality for the main computation. As already mentioned in Section 4.4.3 the state-input vector is rather small in comparison to F_U, so both variants will be implemented and profiled in Section 5.

4.4.5 Parallelism

In both industrial and scientific practice, the optimization of high-performance computing (HPC) code is predominantly approached through two fundamental and complementary strategies: Vectorization to exploit data-level parallelism within a single core, and parallelization to distribute computation across multiple processing units [17]. The scope of this research is intentionally narrowed to focus on vectorization. This decision is guided by both pragmatism and strategy. From a pragmatic standpoint, the depth required to thoroughly analyze and implement vectorization strategies is more compatible with the timeframe of a master's thesis. Strategically, addressing vectorization first represents a more efficient optimization workflow. As argued by authorities in the field, performance tuning should follow a hierarchy that begins with maximizing single-core performance before scaling out [17]. Any speedup realized by vectorizing code on one core is directly inherited and amplified by every thread in a subsequent parallel implementation on a multi-core CPU. This makes vectorization a prerequisite whose benefits extend into parallel solutions.

That said, closer inspection of Algorithm 2 reveals significant potential for future parallelization. Within a single simulation step, the main workload could be parallelized, as access to the state-input vector xu_k is read-only. A data dependency is introduced only in the final step by the multiplication with F_ϕ, where results from all parallel tasks must be accumulated into the subsequent state vector. This pattern of combining results from many threads via a single associative operation is formally known as a *reduction* [18]. While a naive implementation of a reduction can cause a performance bottleneck, highly efficient parallel algorithms exist that execute with logarithmic time complexity, allowing them to be handled with minimal performance loss [17].

5 Implementation

The software documented in this thesis was implemented in multiple iterations as a result of the explorative nature of the computation and the findings from the optimization process outlined in section 4.3. As the goals of this thesis encompass not only the development of one final software for simulating MTI systems, several of the approaches explored during development will be showcased in this section. This is intended to serve as a basis for potential future research into this topic as well as traceable justification for the final implementation of MSS-SIM.

5.1 Sparse linear algebra library

As outlined in Algorithm 2, only part of the computation is non-linear. A majority of the calculations can be either directly formulated in terms of linear algebra, or simple steps can be taken to enable the use of linear algebra operations. Since the efficient computation of linear algebra operations is a very well researched field since the advent of digital computing, making as much use of those existing solutions is an intuitive approach for reimplementation of this algorithm. The selection of an existing solution for computing linear operations is primarily guided by the imperatives described in Section 3 and the main constraint of compatibility with sparse matrices and an interface for the C language. For this implementation Intel's oneMKL library was choosen, as it satisfies the previously outlined constraints as well as showing highly convincing performance benchmarks. [19] Although it has been shown in scientific literature that custom implementations can significantly outperform even oneMKL [20] in sparse matrix operations, for reasons of scientific inquiry and documentation an implementation will be made and evaluated.
As shown in Algortihm 3 the computation can be written very compactly, due to only step 5 being out of scope for the *oneMKL* library.

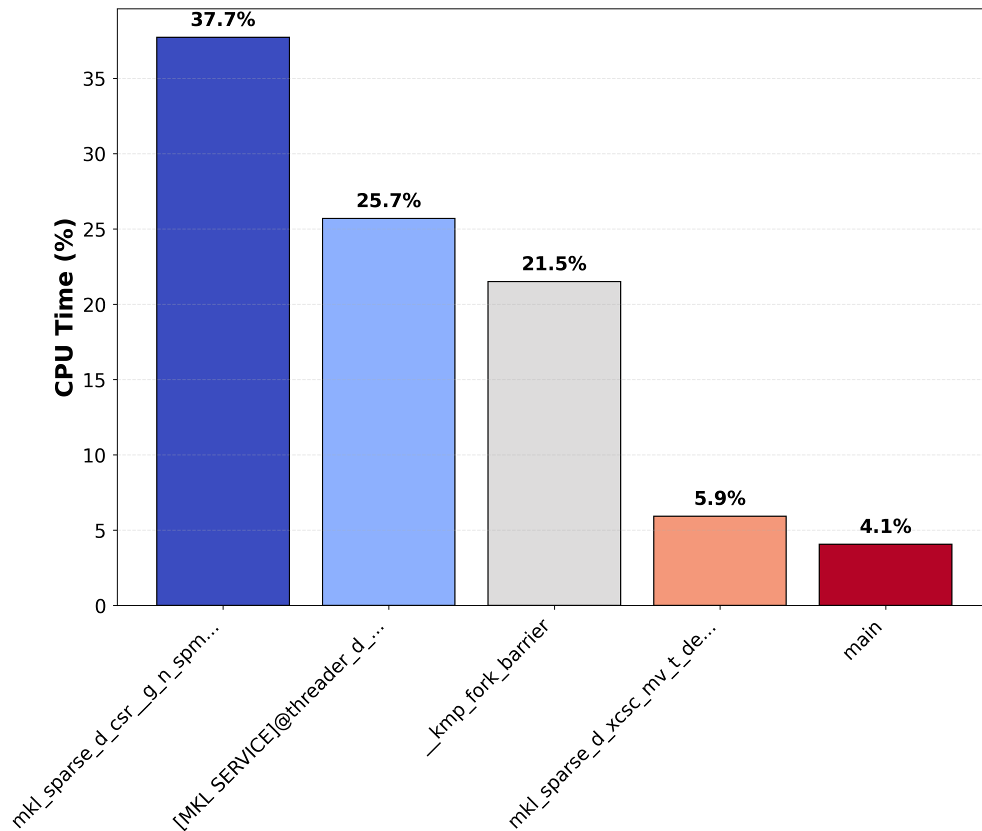


Figure 5.1: Hotspot analysis for oneMKL implementation

Benchmarking results show a significant speedup for both reference models as shown in Table 5.1. When profiling the program as described in Section 4.3, major inefficiencies become obvious. A visualization of the Hotspot analysis can be seen in Figure 5.1, which suggests that a majority of CPU time is spent waiting within the oneMKL threading functions.

Model	**oneMKL**	**MATLAB**
Random	0.74 s	6.7 s
Cstr	0.45 s	15.3 s

Table 5.1: Simulation time of MATLAB and oneMKL implementations

Algorithm 3 Pseudocode for oneMKL implementation

Require: F_U, F_{Phi},x_0, u (Input Array), N (Number of Steps)

Ensure: $\widehat{x}$ (State Trajectory)

1: $xu_k \leftarrow [x0 \; u(0,:)]$
2: $U_{OC} \leftarrow 1 - F_U$
3: $xu_d \leftarrow \text{diag}(xu_k)$
4: **for** $k \leftarrow 0$ to N **do**
5: $\quad \hat{F_U} \leftarrow xu_d \cdot F_U + U_{OC}$
6: $\quad \tilde{F_U} \leftarrow \left[\prod_{i=1}^{n+m} \hat{F}_{U\,i1} \quad \prod_{i=1}^{n+m} \hat{F}_{U\,i2} \quad \cdots \quad \prod_{i=1}^{n+m} \hat{F}_{U\,ir}\right]$ ▷ Only non-library step
7: $\quad \widehat{x}(k) \leftarrow F_{Phi} \cdot \tilde{F_U}^T$
8: $\quad xu_d \leftarrow \text{diag}([\widehat{x}(k) \; u(k,:)])$
9: **end for**

5.2 Vectorized linear algebra library

Due to the column-wise order of operations of the algortihm in combination with the CSC representation of the underlying matrices, the sparse matrix routines are not strictly necessary. The hypothesis motivating this implemntation is that by exploiting the inherent model structure and data representation performance gains can be achieved compared to the prior sparse implementation. With minor restructuring of the sourcecode, the computation can be done solely based on dense vector operations, when applied directly to the data vector of the CSC matrices.
An excerpt from the main loop of this implementation is showcased in Listing 5.1.

```
1  for (size_t i = 0; i < t_num; i++)
2  {
3    memcpy(x_result[i], xu, XRef_n * sizeof(double)); // saving the current x
4    vdPackI(URef_n,URef_vals + i + 1,URef_m,(xu + XRef_n)); // Pack the next
       input vector into xu
5    vdUnpackV(U_nnz,xu,xuUnpacked,U_cols);  // Gather xu into dense vector
6    vdMul(U_nnz,xuUnpacked,U_vals,Uxu_vec); // Element-wise multiplication
7    vdAddI(U_nnz,Uxu_vec, 1,UComp_vals, 1,Uxu_vals, 1); // Add 1-U
8    for (size_t i = 0; i < U_n; i++) // Accumulate rows of Uxu
9    {
10       Uxucum[i] = 1.0;
11       n_coeffs = U_row_ptr[i + 1] - U_row_ptr[i];
12       for (size_t j = 0; j < n_coeffs; j++)
13       {
```

```
14            Uxucum[i] *= Uxu_vals[U_row_ptr[i] + j - 1];
15        }
16    }
17    // Multiply with Phi
18    finalStatus = mkl_sparse_d_mv(SPARSE_OPERATION_NON_TRANSPOSE,1.0,Phi,
        descA,Uxucum,0.0,xu);
19 }
```

Listing 5.1: Excerpt from vectorized implementation

Model	oneMKL vector	sparse oneMKL
Random	0.73 s	0.74 s
Cstr	0.43 s	0.45 s

Table 5.2: Simulation time of oneMKL vector and sparse implementations

As shown in Table 5.2 the change, while substantial in implementation, does not significantly improve performance. The results of profiling as seen in Figure 5.2 reveal that in spite of unchanged performance, the utilization of CPU time is very low due to waiting threads spawned by the oneMKL library.

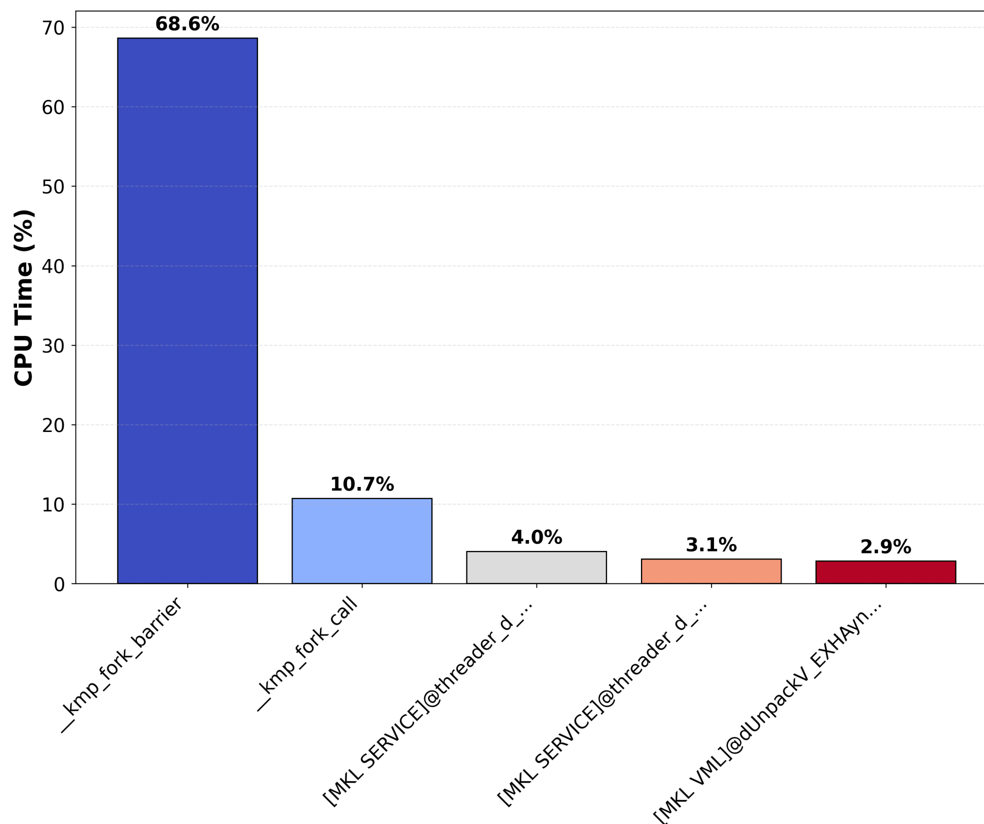


Figure 5.2: Hotspot analysis for oneMKL vectorized implementation

5.3 Custom SIMD Implementation

In order to address the issues discovered during profiling of the implementations involving oneMKL, more control over CPU resources is needed. The following section outlines how the various functionalities provided by the oneMKL library are reimplemented using AVX-2 intrinsics. This will not only increase the control about CPU resources, it will also make the resulting program independent of closed-source dependencies such as oneMKL and enable the compilation using only openly available tools. Due to the significantly increased complexity introduced by the custom implementations, this section will explain the steps taken in more detail than the previous sections. Additionally this will provide a more in-depth overview in the low-level considerations going into computing the operation at hand, thus contributing to one of the goals of this thesis being documenting this previously unexplored feat. As shown in Table 5.3, the custom implementation did

improve upon the oneMKL implementations sigificantly. Closer inspection of the profiling results shown in Figure 5.3 indicate a improved utilization of CPU time, although the amount of time spent on the AVX-2 store instruction suggests room for improvement regarding the details in the SIMD implementation.

Model	Custom SIMD	oneMKL Vector
Random	0.23 s	0.73 s
Cstr	0.36 s	0.43 s

Table 5.3: Simulation time of custom SIMD and oneMKL implementation

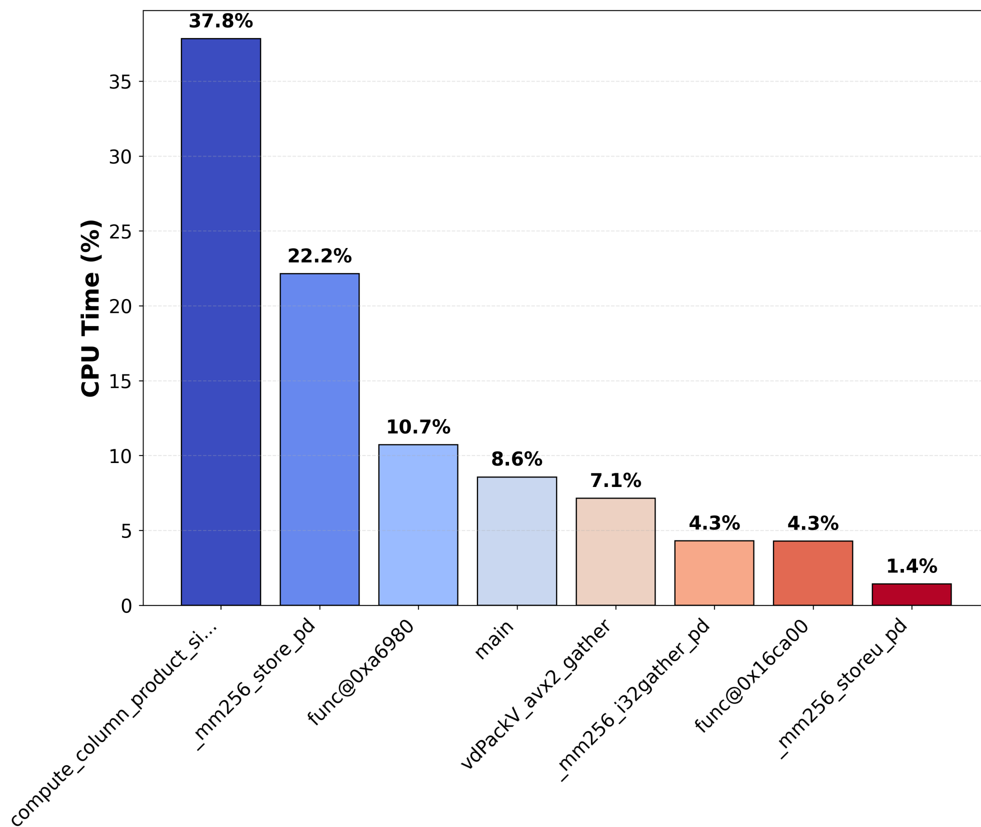


Figure 5.3: Hotspot analysis for custom SIMD implementation

5.3.1 Preprocessor Directives

The AVX-2 instruction set utilized in this implementation offers the ability to compute floating point operations on a 256 bit wide SIMD lane. One SIMD lane can thusly hold either 4 double or 8 single precision floating point numbers. In order to explore the potential performance gains by forgoing precision, MSS-SIM is designed in such that both options will be available to users. To keep the code readable and maintainable this option is provided by incoorporating a preprocessor directive switching between the two different precision modes. This will result in MSS-SIM being comprised of two seperate executables with identical functionalities apart from the conversion to single precision and the necessary adjustments to the computations. By implementing this capability this way, the additional benefit of increased readability of the code is achieved since the AVX-2 intrinsics are quite verbose but the resulting function call via defines will be more concise and interpretable even without deeper knowledge of the AVX-2 syntax. Listing 5.2 shows a subset of the implemented preprocessor directives including type definitions, function calls and number of variables per SIMD lane for later loop indexing.

```
1  #if USING_SINGLE_PRECISION
2      #define PRECISION float
3      #define SIMD_WORD __m256
4      #define SIMD_INDICES __m256i
5      #define SIMD_LOAD(p) _mm256_loadu_ps((p))
6      // Omitted for brevity
7  #else
8      #define PRECISION double
9      #define SIMD_WORD __m256d
10     #define SIMD_INDICES __m128i
11     #define SIMD_LOAD(p) _mm256_loadu_pd((p))
12     // Omitted for brevity
13 #endif
14 #define ELEMENTS_PER_SIMD_LANE (sizeof(__m256d) / sizeof(PRECISION))
```

Listing 5.2: Excerpt from preprocessor directives from MSS-SIM

5.3.2 Aligned data structures

When departing from the oneMKL infrastructure, a substitute for storing the sparse matrices raw data is needed. When implementing these datastructures, mememory alignment is crucial to take into consideration since AVX-2 operations are most perfmant when

operating on 32-bit aligned data.[9] Thus the data structure shown in Listing 5.3 is chosen, incoorporating all necesary data and metadata of a sparse matrix while maintaining alignment and cache-efficient memory layout.

```
typedef struct __attribute__((aligned(32))) {
    int32_t m;            ///< Number of rows
    int32_t n;            ///< Number of columns
    int32_t *col_ptr;     ///< Column pointers
    int32_t *rows;        ///< Row indices
    int32_t nnz;          ///< Number of non-zero values
    PRECISION *vals;      ///< Contiguous array of non-zero values. Can be double or float
} SparseMatrix_t;
```

Listing 5.3: Memory aligned data structure for sparse matrices

5.3.3 SIMD Scalar Add

As a consequence of the reformulation detailed in Section 4.4.3, one of the necessary operations for MSS-SIM is the addition of a scalar to a vector. Listing 5.4 showcases the scalar addition operation and highligting the gained readablilty by the considerations shown in Section 5.3.1. Lines 13-15 also show the workflow of calculations using AVX-2 intrinsics, revolving around loading data from memory into SIMD registers, executing the calculation and saving the result back to memory while considering elements of array with length not divisible by the lane width.

```
void add_scalar_avx2(
    size_t n,
    const PRECISION* input_vector,
    PRECISION scalar,
    PRECISION* output_vector
) {
    size_t i = 0;
    SIMD_WORD scalar_vec = SIMD_SET1(scalar);
    size_t limit_avx2 = n - (n % ELEMENTS_PER_SIMD_LANE);

    for (; i < limit_avx2; i += ELEMENTS_PER_SIMD_LANE)
    {
        SIMD_WORD input_vec = SIMD_LOAD(input_vector + i);
        SIMD_WORD result_vec = SIMD_ADD(input_vec, scalar_vec);
        SIMD_STORE(output_vector + i, result_vec);
    }
```

```
17      for (; i < n; i++)
18      {
19          output_vector[i] = input_vector[i] + scalar;
20      }
21  }
```

Listing 5.4: Scalar Addition SIMD implementation

5.3.4 Gather

One of the crucial operations for the simulation is the preparation of the state-input data for each column's computation, as detailed in Section 4.4.4. The core algorithm requires multiplying the non-zero values from a column of F_U with corresponding, but non-contiguous, elements from the dense state-input vector xu. The gather operation performs the task of collecting these scattered elements, using the matrix's row indices as a map, and packing them into a dense temporary vector suitable for efficient SIMD processing. To maintain high performance throughout the pipeline, this auxiliary operation is also implemented using AVX-2 intrinsics, as shown in Listing 5.5.

```
1   void vdPackV_avx2_gather(
2       int n_indices,
3       const PRECISION* source_array,
4       const int* index_vector,
5       PRECISION* destination_array
6   ) {
7       int i = 0;
8       int limit_avx2 = n_indices - (n_indices % ELEMENTS_PER_SIMD_LANE);
9       for (; i < limit_avx2; i += ELEMENTS_PER_SIMD_LANE)
10      {
11          SIMD_INDICES vidx_vec = SIMD_LOAD_INDICES((const SIMD_INDICES*)(
    index_vector + i));
12          SIMD_WORD gathered_elements = SIMD_GATHER(source_array, vidx_vec,
    sizeof(PRECISION));
13          SIMD_STORE(destination_array + i, gathered_elements);
14      }
15      for (; i < n_indices; ++i)
16      {
17          destination_array[i] = source_array[index_vector[i]];
18      }
19  }
```

Listing 5.5: Gather SIMD implementation

5.3.5 Accumulate

Following the gathering of input data, the accumulate function performs the central computation for each column of the system. This routine implements the critical optimization strategy derived in Section 4.4.3: the reformulation of the element-wise calculation into the FMA-compatible form $(a \cdot b) + c$. As shown in Listing 5.6, the function takes the gathered state-input values and the corresponding non-zero values from the column of F_U and computes the intermediate terms using a Fused Multiply-Add instruction. These terms are then multiplicatively accumulated within a SIMD register to produce a single scalar value representing the final product for that column, as required by the algorithm outlined in Table 4.3. This routine is the primary computational hotspot of the simulation, making its efficient implementation essential to the overall performance of MSS-SIM.

```
static inline PRECISION compute_column_product_avx2_fma(
    int32_t nnz_val,
    const PRECISION* u_vals_ptr,
    const PRECISION* xu_unpacked_ptr
) {
    PRECISION        column_product_scalar = 1.0;
    SIMD_WORD        product_acc_vec = SIMD_SET1((PRECISION) 1.0);
    const SIMD_WORD ones_vec = SIMD_SET1((PRECISION) 1.0);
    int32_t          k = 0;
    int32_t          limit_avx2 = nnz_val - (nnz_val %
ELEMENTS_PER_SIMD_LANE);
    for (; k < limit_avx2; k += ELEMENTS_PER_SIMD_LANE) {
        SIMD_WORD u_vec = SIMD_LOAD(u_vals_ptr + k);
        SIMD_WORD xu_vec = SIMD_LOAD(xu_unpacked_ptr + k);
        SIMD_WORD term_result_vec = SIMD_FMADD(u_vec, xu_vec, ones_vec);
        product_acc_vec = SIMD_MUL(product_acc_vec, term_result_vec);
    }
    PRECISION temp_prod_array[ELEMENTS_PER_SIMD_LANE] __attribute__((
aligned(32)));
    SIMD_STORE(temp_prod_array, product_acc_vec);
    column_product_scalar = temp_prod_array[0] * temp_prod_array[1] *
temp_prod_array[2] * temp_prod_array[3]
    for (; k < nnz_val; ++k) {
        column_product_scalar *= (u_vals_ptr[k] * xu_unpacked_ptr[k] + 1.0)
;
    }
    return column_product_scalar;
}
```

Listing 5.6: Accumulation SIMD implementation

5.4 Custom SIMD Implementation with Loop fusion

The implementation outlined in Section 5.3 exhibited a large amount of CPU time spent in AVX-2 store instructions. This instruction mainly occurs in the gather function, indicating a large overhead of this particular operation. Since the state-input data does not need to be computed previous to the accumulation in whole, the data could also be gathered on-the-fly during the accumulation. This will potentially decrease cache locality during this costly operation, but potentially improve general performance by removing the neccessity of the store instruction completely. The hotspot analysis shown in Figure 5.4 confirms the drastic reduction in CPU time spent in store operations.

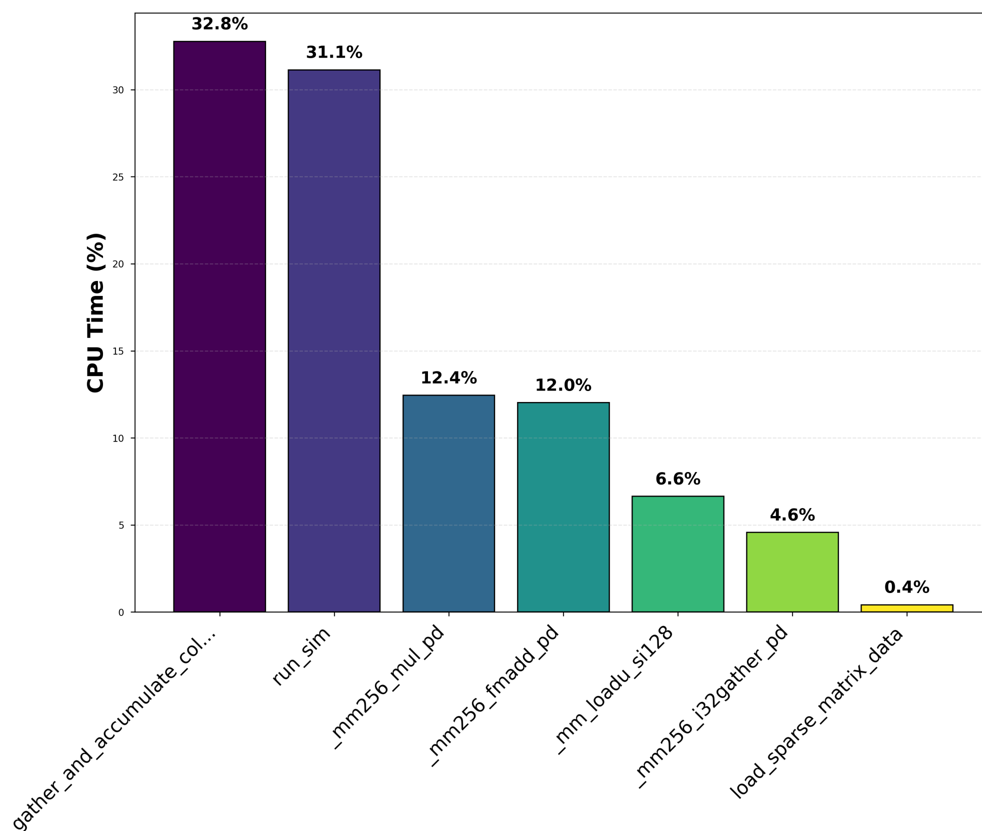


Figure 5.4: Hotspot analysis for custom SIMD implementation with loop fusion

Model	Custom SIMD with loop fusion	Custom SIMD
Random	0.13 s	0.23 s
Cstr	0.21 s	0.36 s

Table 5.4: Simulation time of custom SIMD implementations with and without loop fusion

5.4.1 Combined Gather and Accumulate

The final implementation of the column-wise computation, which embodies the *loop fusion* strategy introduced in Section 5.4, is presented in Listing 5.7. Rather than gathering all required state-input values into an intermediate array before processing, this routine performs the gather operation on-the-fly for each SIMD lane. This fused-loop approach directly mitigates the performance bottleneck identified in the previous implementation, which was attributed to costly memory-write operations (`_mm256_store_pd`) within the stand-alone gather function. By loading data directly into SIMD registers for immediate computation, the need to populate and subsequently re-read from a temporary array is avoided.

The code in Listing 5.7 was modified for publication by omitting comments and removing whitespace. The full source code included in Appendix A.1 is extensively commented and adheres to a project-wide code convention.

```
1  static inline PRECISION gather_and_accumulate_columns_avx2(
2      size_t nnz,
3      const PRECISION* u,
4      const PRECISION* xu,
5      const int* index_vector
6  ) {
7      PRECISION          column_product_scalar = 1.0;
8      SIMD_WORD          product_acc_vec = SIMD_SET1((PRECISION) 1.0);
9      const SIMD_WORD ones = SIMD_SET1((PRECISION) 1.0);
10     size_t            k = 0;
11     PRECISION temp_prod_array[ELEMENTS_PER_SIMD_LANE] __attribute__((
       aligned(32)));
12     size_t limit_avx2 = nnz - (nnz % ELEMENTS_PER_SIMD_LANE);
13     for (; k < limit_avx2; k += ELEMENTS_PER_SIMD_LANE)
14     {
15         const SIMD_WORD u_vec = SIMD_LOAD(u + k);
16         const SIMD_INDICES inds =
17           SIMD_LOAD_INDICES((const SIMD_INDICES*)(index_vector + k));
```

```
18          const SIMD_WORD xu_packed = SIMD_GATHER(xu, inds, sizeof(PRECISION)
    );
19          const SIMD_WORD term_result = SIMD_FMADD(u_vec, xu_packed, ones);
20          product_acc_vec = SIMD_MUL(product_acc_vec, term_result);
21      }
22      SIMD_STORE(temp_prod_array, product_acc_vec);
23      for(size_t i = 0; i < ELEMENTS_PER_SIMD_LANE; i++)
24          column_product_scalar *= temp_prod_array[i];
25      for (; k < nnz; k++)
26          column_product_scalar *= (u[k] * xu[index_vector[k]] + 1.0);
27      return column_product_scalar;
28  }
```

Listing 5.7: Combined Gather and Accumulate SIMD implementation

6 Results

This chapter presents the performance evaluation of the developed implementations against the baseline MATLAB implementation. The results demonstrate significant computational improvements through systematic optimization approaches, achieving speedups of up to 72× while maintaining numerical accuracy.

Note: All performance measurements were conducted on a system equipped with an Intel Core i7-6700HQ processor and 38GB RAM, running Linux 6.15.8-arch1-1.

6.1 Overall Performance Comparison

The primary objective of this thesis was to accelerate the simulation of explicit time-discrete multilinear systems. Table 6.1 summarizes the execution times and speedup factors achieved by each implementation approach for both test models.

Table 6.1: Overall Performance Comparison of All Implementations

Implementation	Random Model (s)	Cstr Model (s)	Avg Speedup
MATLAB Baseline	6.70	15.30	1.0
oneMKL Sparse	0.74	0.45	12.1
oneMKL Vector	0.73	0.43	12.4
Custom SIMD	0.23	0.36	26.1
Custom SIMD + Loop Fusion	0.13	0.21	44.6

The results show a clear progression of performance improvements, with the final implementation achieving an average speedup of 44.6× compared to the MATLAB baseline, notably 72.9x for the real-world model. Figure 6.1 visualizes this performance evolution across all implementation stages.

The Random model consistently shows higher speedup factors than the Cstr model, which can be attributed to the different sparsity patterns and computational characteristics of

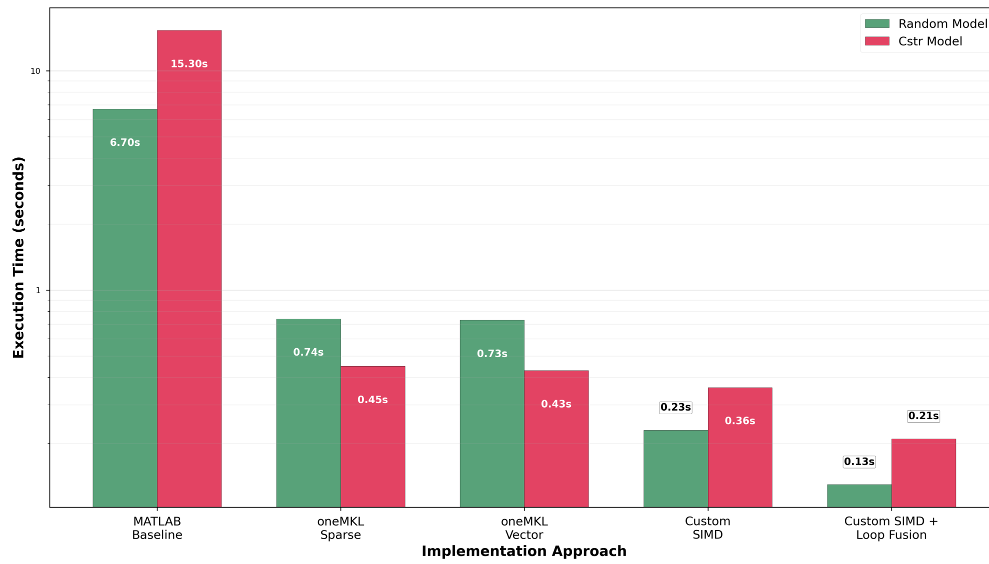


Figure 6.1: Performance progression through implementation stages

these systems. The random model with its lower rank (147 vs 204,800) benefits more significantly from the SIMD optimizations.

6.2 Profiling Analysis Results

Intel VTune profiling analysis reveals the computational bottlenecks in each implementation and validates the optimization strategies employed. The hotspot analysis demonstrates how CPU time utilization improved through the development process.

Figure 6.2 compares the CPU time distribution between the initial oneMKL implementation and the final optimized version, showing the elimination of threading overhead bottlenecks.

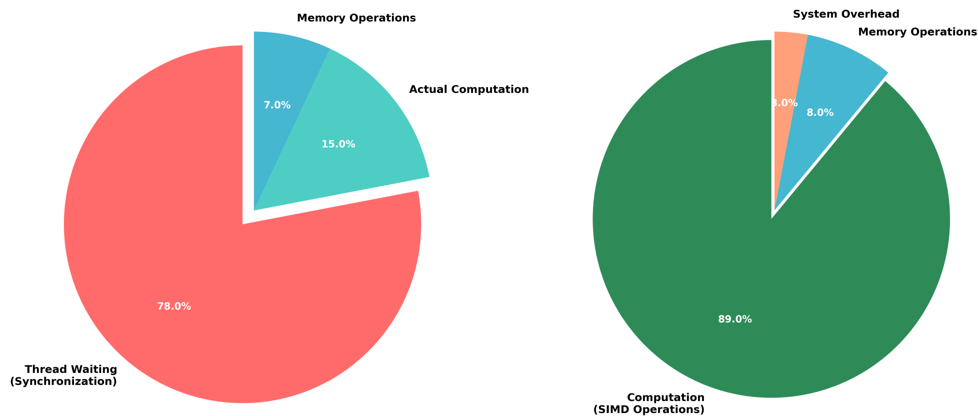


Figure 6.2: CPU time distribution comparison: (left) oneMKL implementation, (right) custom SIMD implementation with loop fusion

The oneMKL implementation suffered from significant threading overhead, with 78% of CPU time spent waiting in thread synchronization routines rather than performing actual computation. The custom SIMD implementation eliminates this bottleneck, achieving 89% of CPU time spent on actual computation.

6.3 Scalability

During the development of MSS-SIM only variations of two different models and thereby model structures were used. While this is necessary for comparability it also limits the applicability of this thesis' findings to a broader selection of models. In order to mitigate this, the performance of MSS-SIM was measured on multiple variations of the real-world model with different sizes. Additionally, to rule out any inefficiencies during set-up or other temporal factors, execution time was measured across a range of different simulation lengths. These measurements confirm that the performance characteristics remain consistent across different operational scales.

The custom implementations consistently achieve the best performance across both model types, with execution times significantly lower than both custom and oneMKL variants. The performance gap between the two custom implementations highlights the effectiveness of loop fusion among the other applied optimizations.

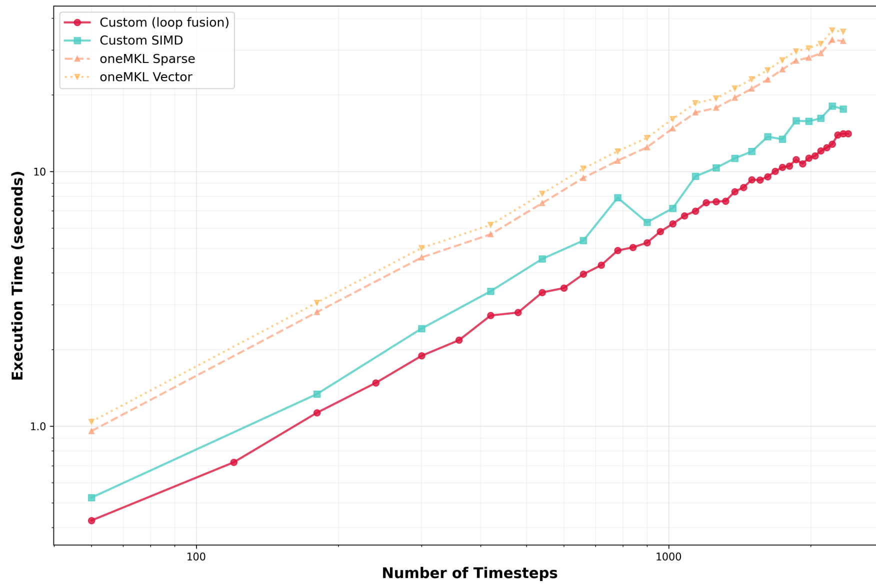


Figure 6.3: Performance comparison for CSTR model implementations demonstrating scalability characteristics across varying timestep ranges.

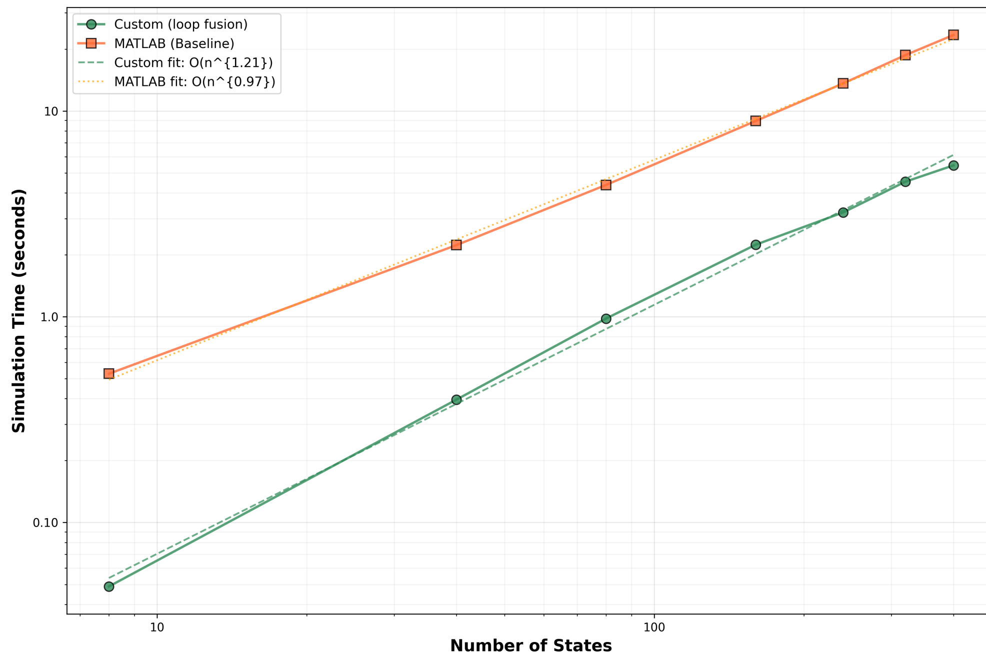


Figure 6.4: Model size scaling comparison between Custom (loop fusion) implementation and MATLAB baseline, showing execution time vs. number of states on logarithmic scales with power-law fit lines.

6.4 Precision Analysis

To explore potential trade-offs between performance and accuracy, both single and double precision implementations were evaluated. Table 6.2 presents the performance characteristics of both variants. The two different implementations do not exhibit significant performance difference, although the single precision version is expected to be more performant due to higher throughput for SIMD operations. This is indicative that the current bottleneck of the implementation is memory access rather than computation.

Table 6.2: Precision vs Performance Trade-off Analysis

Precision	**Random Model (s)**	**Cstr Model (s)**	**Mean Relative Error**
Double	0.13	0.21	5.86×10^{-14}
Single	0.11	0.20	8.03×10^{-5}

7 Discussion and further research

Multilinear modeling as a part of the discipline of control systems is a field of active research and as of publication of this thesis. Most of the research is focussed on expanding and utilizing the mathematical advantages that this field offers to the overarching discipline. This thesis aims to contribute to this collective scientific effort not by further expanding the scope or applicability of multilinear modeling but rather analysing and improving upon the basic operations from a computational perspective. This chapter interprets the significance of the findings and results described in this thesis as well as acknowledge limitations and potential avenues for further research.

The results of this thesis, culminating in a speedup of up to 72x compared to the MATLAB baseline, are not merely an incremental improvement but a potential enabler for advanced control strategies. Such acceleration could allow for the use of these expressive multilinear models in applications that were previously computationally infeasible, such as real-time Model Predictive Control (MPC) with longer prediction horizons or more complex models. By drastically reducing the simulation time, this work helps bridge the gap between complex model formulation and practical, real-world deployment in areas like building automation and power grid management, as outlined in the motivation.

7.1 Algorithmic and Computational Insights

During the algorithmic analysis preceding the implementation of MSS-SIM several key findings contributed to the achieved performance gains, that can utilized in other implementations of multilinear computing:

Sparse matrix representation

The CSC representation for sparse matrices is only one of the commonly used in high performance computing. For the linear operations involved the different common representations don't impact performance significantly. Due to the multilinear

operation mainly operating on columns of F_U, the CSC representation is the preferable choice for future development of computations in this field.
When multiplying the dense input-state vector with columns of systems factor matrix F_U, the non-zeros of F_U are already dense and contiguous in memory. This eliminates the need for additional memory operations, whereas any other sparse matrix reprentation would necessitate further memory oprations for each timestep during simulation.
This novel insight can be utilized advantageously in potential future reimplementations for multilinear modeling.

FMA reformulation

The reformulation of one of the main computational steps as shown in Section 4.4.3 contributed greatly to the performance gains achieved with MSS-SIM. It is also easily applicable to future and current implementation for simulating multilinear systems, as it reduces overall instructions and enables use of efficient SIMD operations on modern CPUs.

Precomputation and loop fusion

Generally, precomputing constants or larger portions of data is advisable from a performance point of view, as is true for computation of the ones complement of F_U. In the case of the column-specific state-input vector profiling has shown that gathering the necessary value as needed during the main computation loop yields higher performance. While contributing to the final version of MSS-SIM, this insight is also applicable for other implementations of this routine being supported by the empirical measurements of this thesis.

Parallelism

Although the implementation of parallelization is outside the scope of this thesis, the analysis of the algorithm resulted in the indicate that the operation is in fact highly parallelizable as highlighted in Section 4.4.5. Future research into this topic can benefit from and sourcecode included in this thesis, as it's modularity and extensive documentation provide a basis for potential investigation of this possibility.

7.2 Portable and performant simulation solution

The software implemented in this thesis is open source and freely available under the *BSD-3* License. As it does not have any proprietary dependencies and is compilable by free and open source software, it can be deployed for accelerating multilinear computations from any other software thats able to execute commandline commands on its host system. At the time of publication the output of the simulation is returned via the standard output. This can easily be adapted without having to edit core-code of MSS-SIM, thusly enabling integration into other software solutions in the future. Especially the integration of the MATLAB environment via its MEX framework presents a promising avenue for future development. Given that MATLAB remains a de facto standard for prototyping and analysis in control systems engineering, creating a MEX interface would dramatically enhance the usability of this work. Such an interface would allow the compiled C code to be called as a native MATLAB function, eliminating the overhead of command-line execution and enabling seamless integration into existing simulation and analysis workflows.

7.3 Conclusion

In conclusion, this thesis successfully demonstrated that through a systematic process of algorithmic analysis, targeted reformulation, and low-level SIMD implementation, the simulation of explicit time-discrete multilinear models can be accelerated by over an order of magnitude. The key contribution is a novel, open-source implementation (MSS-SIM) that significantly outperforms both the original MATLAB code and implementations based on general-purpose HPC libraries. By providing insights into optimization strategies and delivering a freely available, high-performance tool, this work lays a practical foundation for the broader application of multilinear systems in computationally demanding control engineering problems.

Bibliography

[1] W. P. M. H. Heemels, B. De Schutter, and A. Bemporad, "A survey of piecewise affine systems," *IFAC Proceedings Volumes*, vol. 34, no. 14, pp. 427–438, 2001. Analysis and Design of Hybrid Systems 2001 (A postprint of the IFAC Conference, Anaheim, California, USA, 21-22 June 2001).

[2] S. Bächle, *Numerical Solution of Differential-Algebraic Systems Arising in Circuit Simulation*. PhD thesis, Technischen Universität Berlin, 2007.

[3] C. Höger and A. Steinbrecher, "Internalized state-selection: Generation and integration of quasi-linear differential-algebraic equations," in *Proceedings of the 11th International Modelica Conference*, pp. 99–107, 2015.

[4] K. Kruppa, *Multilinear Design of Decentralized Controller Networks for Building Automation Systems*. doctoralthesis, HafenCity Universität Hamburg, 2018.

[5] T. Kolda and B. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, pp. 455–500, 08 2009.

[6] G. Pangalos, A. Eichler, and G. Lichtenberg, "Hybrid multilinear modeling and applications," in *Simulation and Modeling Methodologies, Technologies and Applications* (M. S. Obaidat, S. Koziel, J. Kacprzyk, L. Leifsson, and T. Ören, eds.), (Cham), pp. 71–85, Springer International Publishing, 2015.

[7] R. P. Tewarson, "Solution of a system of simultaneous linear equations with a sparse coefficient matrix," *Numerische Mathematik*, vol. 9, no. 5, pp. 427–432, 1967.

[8] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc., 6th ed., 2020.

[9] A. Fog, "The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers," tech. rep., Technical University of Denmark, 2021. Accessed on: 2025-06-23.

[10] L. Samaniego, E. Uhlenberg, C. Kaufmann, M. Engels, G. Pangalos, C. C. Yáñez, and G. Lichtenberg, "An approach to multi-energy network modeling by multilinear models," in *2024 European Control Conference (ECC)*, pp. 1522–1527, 2024.

[11] X. Li, A. W.-K. Law, and X. Yin, "Partition-based distributed extended kalman filter for large-scale nonlinear processes with application to chemical and wastewater treatment processes," 2024.

[12] A. Marowka, "On performance analysis of a multithreaded application parallelized by different programming models using intel vtune," vol. 6873, pp. 317–331, 10 2011.

[13] M. Malej, M.-A. Lam, F. Shi, and K. Ghosh, "Profiling and optimization of funwave-tvd on high performance computing (hpc) machines," 11 2018.

[14] A. Belkhiri and M. Dagenais, "Analyzing gpu performance in virtualized environments: A case study," *Future Internet*, vol. 16, no. 3, 2024.

[15] Intel Corporation, "Intel® VTune™ Profiler Documentation." https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html, 2025. Accessed: 2025-07-14.

[16] Intel Corporation, "Intel® High Level Synthesis Compiler Pro Edition: Reference Manual. Version 23.1: Loop Fusion." https://www.intel.com/content/www/us/en/docs/programmable/683349/23-1/loop-fusion.html, June 2023. Accessed on 2024-07-21.

[17] G. Hager and G. Wellein, *Introduction to High Performance Computing for S cientists and Engineers*. Boca Raton, FL: CRC Press, 2010.

[18] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. Boston, MA: Addison-Wesley Professional, 2004.

[19] D. A. Augusto, L. M. Carvalho, P. Goldfeld, A. E. F. Muritiba, and M. Souza, "A performance comparison of linear algebra libraries for sparse matrix-vector product," *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*, vol. 3, no. 1, 2015.

[20] K. Stec and P. Stpiczyński, "Performance portability of sparse matrix–vector multiplication implemented using openmp, openacc and sycl," *Future Generation Computer Systems*, 2025.

A Appendix

```
/**
 * @file main.c
 * @author Enrico Uhlenberg (enrico.uhlenberg@gmail.com)
 * @brief Simulation routine for explicit time discrete multilinear models
   in CPN1 decomposition. (MTI models for short)
 * @date 2025-05-19
 * @details
 *  Very brief theory: <br> <br>
 *      The makeup of an MTI model is comparable to a linear state-space
   model (LTI).
 *      A simple  LTI model may consist of two matrices A and B, which
   describe the linear dynamics of the system. (A describing state-state
   realtions and B state-input relations)
 *      MTIs extend this concept by allowing multiple states and inputs to
   have combined influence on the system. In the linear case a difference
   equation (with states x1,x2 and input u) might look like this:
 *          <br>
 *               \f$x1(t+1) = a*x1(k) + b*x2(k) + c*u(k)\f$
 *           <br>
 *      where a in MTI  with the additional multilinear terms could be:
 *          <br>
 *               \f$x1(t+1) = a*x1(k) + b*x2(k) + c*u(k) + d*x1(k)*x2(k) + e
   *x2(k)*u(k) + f*u(k)*x1(k)\f$
 *         <br>
 *
 * Brief description of the operation:
 * Assuming we have system with
 * - n states (x1,x2,...,xn)
 * - m inputs (u1,u2,...,um)
 * - r rank (basically the Number of different multilinear combinations of
   states and inputs)
 * our Model is descibed completetly by the matrices
 * - U ((n+m) x r) containing the information about multilinear relations
 * - Phi (n x r) containing scaling factors, relating the multilinear terms
    to the individual states
```

```
27  *
28  * To obtain a next state vector, the following steps are performed:
29  * 1. The current state vector is appended to the input vector, creating a
       combined state-input vector xu.
30  * 2. This vector is then multilplied (elementwise) with every column of U
31  * 3. For every column j of U, every element if multiplied to create the
       current multilinear terms.
32  * 4. The resulting vector is then right-multiplied with Phi resulting a
       new state vector.
33  *
34  * Note this description is a simplification, detailed descriptions of the
       algorithm can be found in the implementations.
35  *
36  *
37  * The matrices U and Phi can be assumed to be sparse and are stored in CSC
       format. The main performance gains in this software are achieved by
38  * optimizing memory layout and utilizing SIMD operations with this
       specific operation in mind.
39  *
40  * Copyright 2025 Enrico Uhlenberg
41  * Redistribution and use in source and binary forms, with or without
       modification, are permitted provided that the following conditions are
       met:
42  * 1. Redistributions of source code must retain the above copyright notice
       , this list of conditions and the following disclaimer.
43  * 2. Redistributions in binary form must reproduce the above copyright
       notice, this list of conditions and the following disclaimer in the
       documentation and/or other materials provided with the distribution.
44  * 3. Neither the name of the copyright holder nor the names of its
       contributors may be used to endorse or promote products derived from
       this software without specific prior written permission.
45  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
       AS  I S  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
       LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
       PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
       HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
       SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
       LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
       DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
       THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (
       INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
       THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
46  */
47
```

```
48
49 #include <stdio.h>
50 #include <stdlib.h>
51 #include <stdint.h>
52 #include <stdbool.h>
53 #include <stdarg.h>
54 #include <string.h>
55 #include <inttypes.h>
56 #include <time.h>
57 #include <math.h>
58 #include <omp.h>
59 #include <immintrin.h>
60 //#include "mex.h"
61
62
63 //
    ================================================================================
64 // Config
65 //
    ================================================================================
66
67 #define USING_SINGLE_PRECISION 0    ///< Precompiler directive. If set to
    "1" indicating float, the input data will be converted to float.
68 #define DEBUG 1                     ///< Precompiler directive. If set to "1"
    debug messages will be printed to the console.
69 #define MEX 0                       ///< Precompiler directive. If set to "1"
    the code will be compiled as a MEX function for MATLAB.
70 #define oneMKL 1                ///< Precompiler directive. If set to "1" the
    code will use oneMKL for SIMD operations.
71
72 //
    ================================================================================
73 // Constants
74 //
    ================================================================================
75
76 /**
77  * @defgroup PRECISION_SIMD_MACROS
78  *  @brief Macros for SIMD operations
```

```
79  *  @details These macros are used to define the SIMD types and operations
    based on the selected precision.
80  * @{
81  */
82 #if USING_SINGLE_PRECISION
83     #define PRECISION float
84     #define SIMD_WORD __m256
85     #define SIMD_INDICES __m256i
86     #define SIMD_LOAD(p) _mm256_loadu_ps((p))
87     #define SIMD_STORE(p, v) _mm256_storeu_ps((p), (v))
88     #define SIMD_STREAM(p, v) _mm256_stream_ps((p), (v))
89     #define SIMD_SET(src,i,ia) _mm256_set_ps((src[(size_t)((i) + 7) * (ia)
    ]), (src[(size_t)((i) + 6) * (ia)]), (src[(size_t)((i) + 5) * (ia)]), (
    src[(size_t)((i) + 4) * (ia)]),(src[(size_t)((i) + 3) * (ia)]), (src[(
    size_t)((i) + 2) * (ia)]), (src[(size_t)((i) + 1) * (ia)]), (src[(
    size_t)(i) * (ia)]))
90     #define SIMD_FMADD(a,b,c) _mm256_fmadd_ps((a),(b),(c))
91     #define SIMD_GATHER(a,b,c) _mm256_i32gather_ps((a),(b),(c))
92     #define SIMD_MUL(a,b) _mm256_mul_ps((a),(b))
93     #define SIMD_ADD(a,b) _mm256_add_ps((a),(b))
94     #define SIMD_SET1(a) _mm256_set1_ps((a))
95     #define SIMD_LOAD_INDICES(p) _mm256_loadu_si256((p))
96 #else
97     #define PRECISION double
98     #define SIMD_WORD __m256d
99     #define SIMD_INDICES __m128i
100    #define SIMD_LOAD(p) _mm256_loadu_pd((p))
101    #define SIMD_STORE(p, v) _mm256_store_pd((p), (v))
102    #define SIMD_STREAM(p, v) _mm256_stream_pd((p), (v))
103    #define SIMD_SET(src,i,ia) _mm256_set_pd((src[(size_t)((i) + 3) * (ia)
    ]), (src[(size_t)((i) + 2) * (ia)]), (src[(size_t)((i) + 1) * (ia)]), (
    src[(size_t)(i) * (ia)]))
104    #define SIMD_FMADD(a,b,c) _mm256_fmadd_pd((a),(b),(c))
105    #define SIMD_GATHER(a,b,c) _mm256_i32gather_pd((a),(b),(c))
106    #define SIMD_MUL(a,b) _mm256_mul_pd((a),(b))
107    #define SIMD_ADD(a,b) _mm256_add_pd((a),(b))
108    #define SIMD_SET1(a) _mm256_set1_pd((a))
109    #define SIMD_LOAD_INDICES(p) _mm_loadu_si128((p))
110 #endif
111 #define ELEMENTS_PER_SIMD_LANE (sizeof(__m256d) / sizeof(PRECISION))
112
113 /** @} */
114
115
```

```
116
117 // ============================
118 // Typedefs
119 // ============================
120 /**
121  * @brief Struct for storing the sparse matrix in Compressed Sparse Column
      (CSC) format.
122  * @details
123  *      The CSC format stores the non-zeros contiguously in a 1D array, and
      uses two additional arrays to store the row indices and column
      pointers.
124  *      This is  preferred to the otherwise more commen CSR, since data for
      this concrete operation is mostly accessed column-wise.
125  *      For reference, see https://www.intel.com/content/www/us/en/docs/
      onemkl/developer-reference-fortran/2023-0/sparse-blas-csc-matrix-
      storage-format.html
126  */
127 typedef struct __attribute__((aligned(32))) {
128     int32_t m;          ///< Number of rows
129     int32_t n;          ///< Number of columns
130     int32_t *col_ptr;   ///< Column pointers
131     int32_t *rows;      ///< Row indices
132     int32_t nnz;        ///< Number of non-zero values
133     PRECISION *vals;    ///< Contiguous array of non-zero values. Can be
      double or float
134 } SparseMatrix_t;
135
136
137
138
139 // ============================
140 // Function Prototypes
141 // ============================
142
143 /**
144  * @brief Multiplicatively accumulate the values of a column in a sparse
      matrix using OpenMP directives.
145  *
146  * @param nnz_val Number of non-zero values in the column.
147  * @param u_vals_ptr Pointer to the values of the column.
148  * @param xu_unpacked_ptr Pointer to the unpacked state-input vector.
149  * @return The result of the multiplicative accumulation.
150  */
151 #pragma omp declare simd
```

```
152 static inline PRECISION accumulate_columns_omp(
153     int32_t nnz_val,
154     const PRECISION* u_vals_ptr,
155     const PRECISION* xu_unpacked_ptr
156 );
157
158 /**
159  * @brief Gather scattered vector and accumulate the prodcut of that vector
       and a column in a sparse matrix using AVX intrinsics.
160  *
161  * @param nnz Number of non-zero values in the column.
162  * @param u Pointer to the values of the column.
163  * @param xu Pointer to the unpacked state-input vector.
164  * @param index_vector Pointer to the index vector for gathering.
165  * @return The result of the multiplicative accumulation.
166  */
167 static inline PRECISION gather_and_accumulate_columns_avx2(
168     size_t nnz,              // Number of non-zero values in current column
169     const PRECISION* u,      // Pointer to the values of current column
170     const PRECISION* xu,     // Pointer to the unpacked state-input vector
171     const int32_t* index_vector // Pointer to the index vector for
    gathering
172 );
173
174 /**
175  * @brief Copy values from a strided source array to a destination array
      using AVX2 intrinsics.
176  *
177  * @param n Number of elements to copy.
178  * @param src Source array.
179  * @param ia Stride of the source array.
180  * @param dst Destination array.
181  */
182 void strided_gather_avx2(
183     size_t n,
184     const PRECISION* src,
185     size_t ia,
186     PRECISION* dst
187 );
188
189 /**
190  * @brief Add a scalar value to each element of an input vector and store
      the result in an output vector using AVX2 intrinsics.
191  *
```

```
192  * @param n Number of elements in the input vector.
193  * @param input_vector Input vector.
194  * @param scalar Scalar value to add.
195  * @param output_vector Output vector.
196  */
197 void add_scalar_avx2(
198     size_t n,
199     const PRECISION* input_vector,
200     PRECISION scalar,
201     PRECISION* output_vector
202 );
203
204
205 /**
206  * @brief Helper function to calculate the padded size of an array for
        aligned alloc.
207  */
208 static inline size_t calculate_padded_size(
209     size_t num_elements,
210     size_t element_size,
211     size_t alignment
212 );
213
214 /**
215  * @brief Helper function to allocate aligned memory for an array.
216  */
217 static inline void safe_aligned_alloc(
218     void ** ptr,
219     size_t num_elements,
220     size_t element_size,
221     size_t alignment,
222     const char* var_name
223 );
224
225 /**
226  * @brief Helper function to free allocated memory
227  */
228  void cleanup_sparse_matrix(
229     SparseMatrix_t *mat
230 );
231
232 /**
233  * @brief Load sparse matrix data from a binary file in CSC format and
        double precision.
```

```
234  *
235  * @param filename Name of the file to load.
236  * @param mat Pointer to the SparseMatrix_t struct to store the loaded data
       .
237  * @param vals Pointer to the array to store the values.
238  * @return true if loading was successful, false otherwise.
239  */
240 void load_sparse_matrix_data(
241     const char *filename,
242     SparseMatrix_t *mat
243 );
244
245 /**
246  * @brief Allocate memory for the simulation data structures.
247  *
248  * @return true if memory allocation was successful, false otherwise.
249  */
250 void allocate_memory(void);
251
252 /**
253  * @brief Deallocate memory for the simulation data structures.
254  *
255  */
256 void deallocate_memory(void);
257
258 /**
259  * @brief Run the simulation loop.
260  *
261  * This function performs the main simulation loop
262  * @return true if the simulation ran successfully, false otherwise.
263  */
264  void run_sim(void);
265
266 /**
267  * @brief Compare and display simulation results vs reference for first 2
       states and 10 timesteps.
268  *
269  * This function compares the simulated results in x_result with the
       reference results in XRef
270  * and displays them in a formatted table for analysis.
271  */
272 void compare_and_display_results(void);
273
274 /**
```

```
275  * @brief Output complete simulation results as CSV to stdout.
276  *
277  * Optimized for performance with minimal formatting overhead.
278  * Each row represents one timestep, columns represent states.
279  */
280 void output_results_csv_stdout(void);
281
282
283 /**
284  * @brief Print an error message and exit the program.
285  *
286  * @param fmt Format string for the error message.
287  * @param ... Additional arguments for the format string.
288  */
289 void die(const char * fmt, ...);
290
291 /**
292  * @brief Entry point for the MEX function.
293  *
294  * @param nlhs Number of left-hand side arguments (outputs).
295  * @param plhs Array of left-hand side arguments (outputs).
296  * @param nrhs Number of right-hand side arguments (inputs).
297  * @param prhs Array of right-hand side arguments (inputs).
298  */
299
300 //
    ==========================================================================
301 // Globals
302 //
    ==========================================================================
303
304 // Structs for sparse matrices. XRef and URef are admittedly dense, but
     they are stored in a sparse format for consistency
305 static SparseMatrix_t U;
306 static SparseMatrix_t Phi;
307 static SparseMatrix_t XRef;
308 static SparseMatrix_t URef;
309
310 // Number of timesteps to simulate (later on)
311 static size_t t_num = 60;
312
313 // Buffers for storing data without CSC indexing data
```

```
314 // (either because it's a flat array, or because the structure si identical
        to an already existing matrix)
315 static PRECISION *x0                  = NULL;
316 static PRECISION *Uxucum              = NULL;
317 static PRECISION *xu                  = NULL;
318 static PRECISION *xu_minus_1          = NULL;
319 static PRECISION **x_result           = NULL;
320
321 int main(int argc, char *argv[])
322 {
323     const char *model_name = "cstr_concat50n400";
324     bool output_results = false;
325
326     if (argc > 1) {
327         int temp_t_num = atoi(argv[1]);
328         if (temp_t_num > 0) {
329             t_num = (size_t)temp_t_num;
330         } else {
331             printf("Warning: Invalid t_num value '%s'. Using default value
    %zu.\n", argv[1], t_num);
332         }
333     }
334
335     if (argc > 2) {
336         size_t model_len = strlen(argv[2]);
337         if (model_len > 0 && model_len < 100) {
338             bool valid_name = true;
339             for (size_t i = 0; i < model_len; i++) {
340                 char c = argv[2][i];
341                 if (!((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') ||
342                       (c >= '0' && c <= '9') || c == '_')) {
343                     valid_name = false;
344                     break;
345                 }
346             }
347
348             if (valid_name) {
349                 model_name = argv[2];
350             } else {
351                 printf("Warning: Invalid model name '%s'. Using default '%s
     '.\n", argv[2], model_name);
352             }
353         } else {
```

```
354             printf("Warning: Model name too long or empty. Using default '%
    s'.\n", model_name);
355         }
356     }
357 
358     if (argc > 3 && strcmp(argv[3], "--output") == 0) {
359         output_results = true;
360     }
361 
362     if (!output_results) {
363         #if USING_SINGLE_PRECISION
364             printf("INFO: Using SINGLE precision computation path.\n");
365         #else
366             printf("INFO: Using DOUBLE precision computation path.\n");
367         #endif
368 
369         printf("INFO: Using model '%s'\n", model_name);
370     }
371 
372     char file_path[256];
373 
374     snprintf(file_path, sizeof(file_path), "data/%s_U.bin", model_name);
375     load_sparse_matrix_data(file_path, &U);
376 
377     snprintf(file_path, sizeof(file_path), "data/%s_Phi.bin", model_name);
378     load_sparse_matrix_data(file_path, &Phi);
379 
380     snprintf(file_path, sizeof(file_path), "data/%s_uVec.bin", model_name);
381     load_sparse_matrix_data(file_path, &URef);
382 
383     snprintf(file_path, sizeof(file_path), "data/%s_xVec.bin", model_name);
384     load_sparse_matrix_data(file_path, &XRef);
385 
386     allocate_memory();
387 
388     double begin = omp_get_wtime();
389 
390     run_sim();
391 
392     double end = omp_get_wtime();
393 
394     if (output_results) {
395         // Performance mode: only output CSV data
396         output_results_csv_stdout();
```

```
397     } else {
398         // Normal mode: existing output
399         printf("Main loop finished.\n");
400         double time_spent = (double)(end - begin);
401         printf("0: %f \n", x_result[0][0]);
402         printf("1: %f \n", x_result[1][0]);
403         printf("2: %f \n", x_result[2][0]);
404         printf("Simulation time: %f seconds.\n", time_spent);
405 
406         compare_and_display_results();
407     }
408 
409     return EXIT_SUCCESS;
410 }
411 
412 //
    ==========================================================================
413 // Function Implementations
414 //
    ==========================================================================
415 void cleanup_sparse_matrix(
416     SparseMatrix_t *mat
417 ) {
418     if (
419         mat->col_ptr
420         &&
421         mat->rows
422         &&
423         mat->vals
424     ) {
425 
426         free(mat->col_ptr);
427         free(mat->rows);
428         free(mat->vals);
429         mat->col_ptr    = NULL;
430         mat->rows       = NULL;
431         mat->vals       = NULL;
432         mat = NULL;
433     }
434     #if MEX
435     mexErrMsgIdAndTxt("MTISIM:cleanup_sparse_matrix", "Sparse matrix
    cleanup failed.\n");
```

```
436     #endif
437 }
438
439 #pragma omp declare simd
440 static inline PRECISION accumulate_columns_omp(
441     int32_t nnz_val,
442     const PRECISION* u_vals_ptr,
443     const PRECISION* xu_unpacked_ptr
444 ) {
445     PRECISION column_product = 1.0;
446     #pragma omp simd reduction(*:column_product)
447     for (int32_t k = 0; k < nnz_val; ++k) {
448         column_product *= (u_vals_ptr[k] * xu_unpacked_ptr[k] + 1.0);
449     }
450     return column_product;
451 }
452
453 static inline PRECISION gather_and_accumulate_columns_avx2(
454     size_t nnz,             // Number of non-zero values in current column
455     const PRECISION* u,     // Pointer to the values of current column
456     const PRECISION* xu,    // Pointer to the unpacked state-input vector
457     const int* index_vector // Pointer to the index vector for gathering
458 ) {
459     PRECISION        column_product_scalar = 1.0;                  //
    Scalar product accumulator
460     SIMD_WORD        product_acc_vec = SIMD_SET1((PRECISION) 1.0);   // SIMD
     product accumulator
461     const SIMD_WORD ones = SIMD_SET1((PRECISION) 1.0);             //
    Vector of ones for SIMD operations
462     size_t           k = 0;                                        // Loop
     index
463
464     // Buffer for storing the result of the SIMD operations
465     PRECISION temp_prod_array[ELEMENTS_PER_SIMD_LANE] __attribute__((
    aligned(32)));
466
467     // Limit for k, because we can only process full SIMD lanes
468     size_t limit_avx2 = nnz - (nnz % ELEMENTS_PER_SIMD_LANE);
469
470     for (; k < limit_avx2; k += ELEMENTS_PER_SIMD_LANE)
471     {
472         // Load values from U
473         //printf("Loading SIMD vector from u at index %zu\n", k);
```

```
474         //fflush(stdout); // Ensure the message is printed before any
    potential crash
475         const SIMD_WORD u_vec = SIMD_LOAD(u + k);
476         // Load index vector
477         const SIMD_INDICES inds = SIMD_LOAD_INDICES((const SIMD_INDICES*)(
    index_vector + k));
478         // Gather scattered elements
479         const SIMD_WORD xu_packed = SIMD_GATHER(xu, inds, sizeof(PRECISION)
    );
480         // Fused multiply-add operation (U*(xu_minus_one)+1)
481         const SIMD_WORD term_result = SIMD_FMADD(u_vec, xu_packed, ones);
482         // Multiply the result with the accumulator
483         product_acc_vec = SIMD_MUL(product_acc_vec, term_result);
484     }
485 
486     // Copy packed vector to memory for scalar multiplication
487     SIMD_STORE(temp_prod_array, product_acc_vec);
488 
489     // Multiply the SIMD lanes to a scalar, 4 or 8 elements depending on
    precision mode
490     for(size_t i = 0; i < ELEMENTS_PER_SIMD_LANE; i++)
491     {
492         column_product_scalar *= temp_prod_array[i];
493     }
494 
495     // Process the remaining elements that don't fit into a full SIMD lane
496     for (; k < nnz; k++)
497     {
498         column_product_scalar *= (u[k] * xu[index_vector[k]] + 1.0);
499     }
500 
501     return column_product_scalar;
502 }
503 
504 void strided_gather_avx2(
505     size_t n,
506     const PRECISION* src,
507     size_t ia,
508     PRECISION* dst
509 ) {
510     size_t i = 0;
511     size_t limit_avx2 = n - (n % ELEMENTS_PER_SIMD_LANE);
512     for (; i < limit_avx2; i += ELEMENTS_PER_SIMD_LANE)
513     {
```

```
514         SIMD_WORD a_vec = SIMD_SET(src, i, ia);
515         SIMD_STORE(dst + i, a_vec);
516     }
517     for (; i < n; i++)
518     {
519         dst[i] = src[(size_t)i * ia];
520     }
521 }
522 
523 void add_scalar_avx2(
524     size_t n,
525     const PRECISION* input_vector,
526     PRECISION scalar,
527     PRECISION* output_vector
528 ) {
529     size_t i = 0;
530     SIMD_WORD scalar_vec = SIMD_SET1(scalar);
531     size_t limit_avx2 = n - (n % ELEMENTS_PER_SIMD_LANE);
532 
533     for (; i < limit_avx2; i += ELEMENTS_PER_SIMD_LANE)
534     {
535         SIMD_WORD input_vec = SIMD_LOAD(input_vector + i);
536         SIMD_WORD result_vec = SIMD_ADD(input_vec, scalar_vec);
537         SIMD_STORE(output_vector + i, result_vec);
538     }
539     for (; i < n; i++)
540     {
541         output_vector[i] = input_vector[i] + scalar;
542     }
543 }
544 
545 
546 
547 
548 void load_sparse_matrix_data(const char *filename, SparseMatrix_t *mat)
549 {
550     FILE *fp = fopen(filename, "rb");
551     if (!fp)
552     {
553         die("Failed to load matrix file %s", filename);
554     }
555 
556     mat->m =0;
557     mat->n =0;
```

```
558    mat->col_ptr = NULL;
559    mat->rows = NULL;
560    mat->nnz = 0;
561    double *vals = NULL;
562    // Read dimensions
563    if (
564        (fread((void*)&mat->m, sizeof(int32_t), 1, fp) != 1)
565        ||
566        (fread((void*)&mat->n, sizeof(int32_t), 1, fp) != 1)
567    ) {
568        die("Dimension Error in %s", filename);
569    }
570
571    safe_aligned_alloc((void**)&mat->col_ptr ,(size_t)(mat->n) + 1, sizeof(
       int32_t), 64, "col_ptr");
572    if (!mat->col_ptr && ((mat->n) + 1) > 0)
573    {
574        die("Dimension Error in %s", filename);
575    }
576    // Read column pointers
577    if (
578        fread((void*)mat->col_ptr, sizeof(int32_t), (size_t)(mat->n) + 1,
       fp) != (size_t)((mat->n) + 1)
579    )
580    {
581        die("Dimension Error in %s", filename);
582    }
583
584    mat->nnz = (mat->n >= 0 && mat->col_ptr) ? ((mat->col_ptr)[mat->n] - 1)
        : -1;
585    if (mat->nnz < 0)
586    {
587        die("Invalid nnz (%d) for %s\n", mat->nnz, filename);
588    }
589
590    safe_aligned_alloc((void**)&mat->rows ,mat->nnz, sizeof(int32_t), 64, "
       rows");
591    safe_aligned_alloc((void**)&vals ,mat->nnz, sizeof(double), 64, "
       valsdouble");
592
593    if (mat->nnz > 0) {
594        if (!(mat->rows) || !(vals))
595        {
596            die("Data alloc failed for %s\n", filename);
```

```
597         }
598 
599         // Read row indices and values
600         if (
601             (fread((void*)mat->rows, sizeof(int32_t), mat->nnz, fp) != (
    size_t)mat->nnz)
602             ||
603             (fread((void*)vals,     sizeof(double),  mat->nnz, fp) != (
    size_t)mat->nnz)
604         )
605         {
606             die("Data read failed for %s\n", filename);
607         }
608     }
609     fclose(fp);
610 
611     for(int32_t i = 0; i < mat->n + 1; ++i)
612     {
613         mat->col_ptr[i] -= 1; // Convert to 0-based indexing
614     }
615     for(int32_t i = 0; i < mat->nnz; ++i)
616     {
617         mat->rows[i] -= 1; // Convert to 0-based indexing
618     }
619 
620     #if USING_SINGLE_PRECISION
621         safe_aligned_alloc((void**)&mat->vals,mat->nnz, sizeof(float), 64,
    "U_vals");
622         for (int32_t k = 0; k < mat->nnz; ++k)
623         {
624             mat->vals[k] = (float)vals[k];
625         }
626         printf("Data loaded (as double) and converted to float from %s (m=%
    " PRId32 ", n=%" PRId32 ", nnz=%d).\n", filename, mat->m, mat->n, mat->
    nnz);
627     #else
628         mat->vals = vals;
629         printf( "Data loaded (as double) from %s (m=%" PRId32 ", n=%"
    PRId32 ", nnz=%d).\n", filename, mat->m, mat->n, mat->nnz);
630     #endif
631 }
632 
633 static inline size_t calculate_padded_size(size_t num_elements, size_t
    element_size, size_t alignment) {
```

```
634     if (num_elements == 0 || element_size == 0)
635     {
636         return 0;
637     }
638
639     if (alignment == 0)
640     {
641         alignment = 1;
642     }
643     size_t total_size = num_elements * element_size;
644     size_t remainder = total_size % alignment;
645     return (remainder == 0) ? total_size : (total_size + alignment -
        remainder);
646 }
647
648 static inline void safe_aligned_alloc(
649     void ** ptr,
650     size_t num_elements,
651     size_t element_size,
652     size_t alignment,
653     const char* var_name
654 ) {
655     if (num_elements == 0)
656     {
657         *ptr = NULL;
658     }
659
660     if (alignment == 0 || (alignment & (alignment - 1)) != 0 || alignment %
         sizeof(void*) != 0)
661     {
662         die(" Error: Invalid alignment %zu for %s.\n", alignment, var_name)
        ;
663     }
664     size_t padded_size = calculate_padded_size(num_elements, element_size,
        alignment);
665     if (padded_size == 0 && (num_elements * element_size > 0))
666     {
667         die("Error: padded_size is 0 for non-zero request for %s.\n",
        var_name);
668     }
669
670     *ptr = NULL;
671     #ifdef _WIN32
672         *ptr = _aligned_malloc(padded_size, alignment);
```

```
673     #else
674         if (posix_memalign(ptr, alignment, padded_size) != 0)
675         {
676             *ptr = NULL;
677         }
678     #endif
679 
680     if (!ptr && padded_size > 0)
681     {
682         die("Failed to allocate aligned memory for %s.\n", var_name);
683     }
684 }
685 
686 void die(const char * fmt, ...)
687 {
688     #if MEX
689     char errBuf[100];
690     va_list ap;
691     va_start(ap,fmt);
692     sprintf(errBuf,fmt,va_arg(ap,const char *));
693     mexErrMsgIdAndTxt("MTISIM:die",errBuf);
694     va_end(ap);
695     #else
696     va_list ap;
697     va_start(ap,fmt);
698     vprintf(fmt, ap);
699     va_end(ap);
700     exit(EXIT_FAILURE);
701     #endif
702 
703 }
704 
705 
706 
707 void allocate_memory(void)
708 {
709 
710     // Initial state vector
711     safe_aligned_alloc((void**)&x0,XRef.n, sizeof(PRECISION), 64, "x0");
712     if (XRef.vals && x0)
713     {
714         strided_gather_avx2(XRef.n, XRef.vals, XRef.m, x0);
715     }
716 
```

```
717     // Allocate memory for the accumulation vector and the result table
718     safe_aligned_alloc((void**)&Uxucum,U.n, sizeof(PRECISION), 64, "Uxucum"
    );
719     safe_aligned_alloc((void**)&x_result,t_num, sizeof(PRECISION*), 64, "
    x_result table");
720     for (size_t i = 0; i < t_num; i++)
721     {
722         safe_aligned_alloc((void**)&x_result[i],Phi.m, sizeof(PRECISION),
    64, "x_result_row");
723         if (x_result[i])
724         {
725             memset(x_result[i], 0, Phi.m * sizeof(PRECISION));
726         }
727         else
728         {
729             die("Error allocating memory for x_result[%i]", i);
730         }
731     }
732     safe_aligned_alloc((void**)&xu,U.m, sizeof(PRECISION), 64, "xu");
733     if (x0 && x_result[0])
734     {
735         memcpy(x_result[0], x0, XRef.n * sizeof(PRECISION));
736     }
737
738     safe_aligned_alloc((void**)&xu_minus_1,U.m, sizeof(PRECISION), 64, "
    xu_minus_1");
739
740
741
742     if ((U.nnz > 0 && !U.vals)) {
743         die("Memory allocation failed: U.vals is NULL but U.nnz > 0.
    Exiting.\n");
744     }
745     if ((Phi.nnz > 0 && !Phi.vals)) {
746         die("Memory allocation failed: Phi.vals is NULL but Phi.nnz > 0.
    Exiting.\n");
747     }
748     if (!URef.vals) {
749         die("Memory allocation failed: URef.vals is NULL. Exiting.\n");
750     }
751     if (!XRef.vals) {
752         die("Memory allocation failed: XRef.vals is NULL. Exiting.\n");
753     }
754     if (!Uxucum) {
```

```
755         die("Memory allocation failed: Uxucum is NULL. Exiting.\n");
756     }
757     if (!x_result) {
758         die("Memory allocation failed: x_result is NULL. Exiting.\n");
759     }
760     if (!xu) {
761         die("Memory allocation failed: xu is NULL. Exiting.\n");
762     }
763     if (!xu_minus_1) {
764         die("Memory allocation failed: xu_minus_1 is NULL. Exiting.\n");
765     }
766
767 }
768
769 void run_sim(void)
770 {
771     for (size_t i = 1; i < t_num; i++)
772     {
773         // Copy the previous result into the current state vector
774         memcpy(xu, x_result[i-1], XRef.n * sizeof(PRECISION));
775
776         // Append the current input to the state vector
777         strided_gather_avx2(
778             URef.n,                             // Number of elements to gather
779             URef.vals + (i % URef.m) - 1,       // Pointer to the source array
    (URef.vals) Modulo URef.m to cycle through inputs
780             URef.m,                             // Stride of the source array (
    URef.m)
781             (xu + (size_t)XRef.n)               // Pointer to area for inputs
    in the the destination array (xu)
782         );
783
784         // Compute the offset input-state vector
785         add_scalar_avx2(U.m, xu, -1.0, xu_minus_1);
786
787         // Clear the result vector for good measure
788         memset(x_result[i], 0, Phi.m * sizeof(PRECISION));
789
790         //Loop through columns
791         for (int32_t j = 0; j < U.n; ++j)
792         {
793             // col_ptr[j] points to the start of the column j
794             // col_ptr[j+1] points to the end of the column j
```

```
795             const int32_t U_curInd = U.col_ptr[j];                          //
     Index to start of values of column j
796             const int32_t U_nnz = U.col_ptr[j + 1] - U_curInd;              //
     Number of non-zero values in column j
797             const int32_t phi_curInd = Phi.col_ptr[j];                      //
     Index to start of values of column j in Phi
798             const int32_t phi_nnz = Phi.col_ptr[j + 1] - phi_curInd;        //
     Number of non-zero values in column j in Phi
799
800             Uxucum[j] = gather_and_accumulate_columns_avx2(
801                 U_nnz,                                      // Number of non-zero
    values in column j
802                 &U.vals[U_curInd],                          // Pointer to the
    values of column j
803                 xu_minus_1,                                 // Pointer to the state
    -input vector
804                 (const int*)&U.rows[U_curInd]               // Pointer to the index
     vector for gathering
805             );
806
807             // Scale current column j by non-zeros in Phi and additively
    accumulate the result in x_result[i]
808             for(int32_t l = 0; l < phi_nnz; l++)
809             {
810                 int32_t target_row = Phi.rows[phi_curInd + l];
811                 if (target_row >= 0 && target_row < Phi.m)
812                 {
813                     x_result[i][target_row] += Uxucum[j] * Phi.vals[
    phi_curInd + l];
814                 }
815             }
816         }
817     }
818 }
819
820 void deallocate_memory(void)
821 {
822     // Free the allocated memory for the sparse matrices
823     //cleanup_sparse_matrix(&U);
824     //cleanup_sparse_matrix(&Phi);
825     //cleanup_sparse_matrix(&URef);
826     //cleanup_sparse_matrix(&XRef);
827     // Free the allocated memory for the buffers
828     if (x0 != NULL)
```

```
829     {
830         free(x0);
831         x0 = NULL;
832     }
833     if (Uxucum != NULL)
834     {
835         free(Uxucum);
836         Uxucum = NULL;
837 
838     }
839     if (xu != NULL)
840     {
841         free(xu);
842         xu = NULL;
843     }
844     if (xu_minus_1 != NULL)
845     {
846         free(xu_minus_1);
847         xu_minus_1 = NULL;
848     }
849     // Free the result table
850     if (x_result != NULL)
851     {
852         for (size_t i = 0; i < t_num; i++)
853         {
854             if (x_result[i] != NULL)
855             {
856                 free(x_result[i]);
857                 x_result[i] = NULL;
858             }
859         }
860         free(x_result);
861         x_result = NULL;
862     }
863 }
864 
865 void compare_and_display_results(void)
866 {
867     const size_t num_timesteps = 60;
868     const size_t num_states = (size_t)XRef.n;   // Use all states from the
        system
869 
870     PRECISION total_squared_error = 0.0;
871     size_t total_comparisons = 0;
```

```
872
873     // Calculate mean error across all states and timesteps
874     for (size_t state = 0; state < num_states; state++) {
875         for (size_t t = 0; t < num_timesteps && t < t_num; t++) {
876             // Get simulated value
877             PRECISION sim_val = x_result[t][state];
878
879             // Get reference value (stored column-wise in XRef)
880             PRECISION ref_val = XRef.vals[state * XRef.m + t];
881
882             // Calculate squared error
883             PRECISION error = sim_val - ref_val;
884             total_squared_error += error * error;
885             total_comparisons++;
886         }
887     }
888
889     // Calculate root mean square error
890     PRECISION mean_error = sqrt(total_squared_error / total_comparisons);
891
892     printf("Mean error: %.6e\n", mean_error);
893 }
894
895 void output_results_csv_stdout(void)
896 {
897     size_t num_states = (size_t)XRef.n;
898
899     setvbuf(stdout, NULL, _IOFBF, 65536);
900
901     for (size_t t = 0; t < t_num; t++) {
902         for (size_t s = 0; s < num_states; s++) {
903             printf("%.15g", x_result[t][s]);
904             if (s < num_states - 1) {
905                 putchar(',');
906             }
907         }
908         putchar('\n');
909     }
910
911     fflush(stdout);
912 }
913
914 /** @file */
```

Listing A.1: Full Source code for MSS-SIM

Listing A.2: Matlab Function for concatenating multilinear systems from the MTI-TOOLBOX

```
function sysout = conCatMss(sys,n_concat)
    if(n_concat < 1)
        sysout = sys;
        return
    end
    sizephi = size(sys.F.phi);
    rank_old = sizephi(2);
    n = sys.n;
    m = sys.m;
    nm = n + m;
    FU = zeros(nm*(n_concat),rank_old*n_concat);
    Fphi = zeros(n*(n_concat) ,rank_old*n_concat);

    U = full(sys.F.U);
    phi = full(sys.F.phi);

    Fphi(1:n,1:rank_old) = phi;
    for i = 0:n_concat-1
        UStateStart = i*n+1;
        UStateEnd = ((i+1)*n);
        UInputStart = (n_concat)*n+1 + i*m;
        UInputEnd =  (n_concat)*n + i*m + m;
        FU(UStateStart:UStateEnd,i*rank_old+1:(i+1)*rank_old) =
            U(1:n,:);
        FU(UInputStart:UInputEnd,i*rank_old+1:(i+1)*rank_old) =
            U(n+1:end,:);

        Fphi(i*n+1:(i+1)*n,i*rank_old+1:(i+1)*rank_old) = phi;
    end
    sysout = mss(CPN1(sparse(FU),sparse(Fphi)),sys.ts);
end
```

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original