IMPERIAL

Imperial College London

Imperial-X

Efficient Computations In Sequence Models: Finding a Sweet Spot Between Memory Efficiency And Parallelism In Sequence Length

Author:
Maxime Vergnaud

Supervisor:
Cristopher Salvi

September 2025

Abstract

While Transformers dominate modern sequence modelling through parallelisable attention mechanisms, they struggle with state-tracking tasks that recurrent neural networks handle naturally. In recent years, state-space models (SSMs) have emerged as a compelling alternative, offering linear computational complexity while maintaining comparable performance on sequence modelling tasks. Moreover, recent work has shown that several SSMs can be reformulated within a continuous-time framework using Linear Neural Controlled Differential Equations (LNCDEs), enabling evaluation between time states via matrix exponentials, which can be efficiently parallelised using associative parallel scans.

This thesis investigates the evaluation of these matrix exponentials using Taylor series and Padé approximation methods, while analysing the theoretical complexity and GPU implementations of the underlying matrix multiplications they rely on. We investigate tiling strategies for structured matrices including block-diagonal, Walsh-Hadamard, and sparse representations that preserve maximal expressivity while reducing computational overhead. Our evaluation demonstrates that the Padé approximation combined with scaling and squaring provides the best balance of accuracy and computational efficiency, consistently outperforming Taylor series methods, and speed-ups of up to 22% are achieved through polynomial evaluation optimisations using the Paterson-Stockmeyer algorithm. In addition, adaptive step sizing methods based on the Magnus expansion are introduced to handle problems with stiff dynamics where fixed data-driven observation intervals are insufficient to guarantee stable evaluation, although their current implementation fails to show advantages over fixed step methods.

Acknowledgements: I would like to express my sincere gratitude to my supervisor Cristopher Salvi for his invaluable guidance and support throughout the writing of this project. His work on structured linear controlled differential equations served as the fundamental motivation for this thesis, and his expertise and mentorship have been instrumental in shaping both the direction and quality of my work. I am also grateful to Nicola Gnecco for serving as second marker and providing valuable feedback.

Contents

1 Introduction

Sequence-to-sequence models are central to modern machine learning, providing a framework for transforming input sequences into output sequences in domains such as natural language processing and time-series forecasting, where data is of a sequential nature. Historically, recurrent neural networks (RNNs) were the dominant architecture for sequence modelling, but their reliance on recurrence-processing limited their scalability for long sequences. In recent years, Transformers [56] have taken over the space, improving computational efficiency by allowing parallel training with GPUs, with their self-attention mechanism allowing models to simultaneously attend to the entire input sequence at once, rather than relying on sequential recurrence. State-space models (SSMs)[20], a class of linear RNN, have emerged as another alternative. Unlike Transformers, SSMs treat sequences as linear dynamical systems, and can be trained using parallel scans, algorithms that reformulate sequential state updates into associative operations which are computable in parallel over the sequence.

Although Transformers and SSMs allow parallelisable sequence modelling, both struggle with state-tracking, the ability to effectively capture and update the compressed representation of the information contained in the input as it is processed sequentially. This results in both architectures being unable to perform certain state-tracking tasks, such as permutation composition, which can easily be done by single layer RNNs ([36], [37]). This limits the use cases of SSMs, as permutation composition is a key component of several real-world problems such as code evaluation, tracking chess moves and tracking characters in a narrative [36].

In recent years, advances have addressed this challenge through expressive recurrences, mechanisms that allow for input-sensitive state-tracking in parallel. For instance, Linear Neural Controlled Differential Equations (LNCDEs) [9] utilise continuous time dynamics governed by controlled differential equations, decomposing state evolution into associative parallel scans. Similarly, Input-Dependent S4 (IDS4) [36] extend the expressive power of SSMs by parametrising transition matrices as input-dependent functions.

Models such as LNCDEs and IDS4 rely on computing iterated products of matrix exponentials when approximating continuous-time dynamics, parallelising these matrix products through log-depth computation graphs. However, the computational cost associated to these methods scales cubically with hidden dimension, d_h compared to quadratically for SSMs. To address these computational and memory overhead issues, structured matrix parametrisations such as block-diagonal, sparse, and Hadamard representations can be employed, which crucially retain the state-tracking and parallelisable properties whilst reducing parameter count and computational cost [57].

The most efficient way to compute such matrix exponentials for different matrix structures has been an area of extensive research for decades, since Moler and Van Loan's pioneering 1978 paper [38], which examines nineteen different methods for computing the matrix exponential, analysing their numerical stability, computational cost, and generality. Out of the methods studied, the scaling and squaring technique has become the method of choice in modern deep learning. This preference stems largely from the influential work of Nicholas Higham [25], who rigorously analysed and enhanced the method's stability and efficiency by integrating scaling and squaring with Padé approximations, making it highly effective across a wide range of matrices.

When implementing the Padé approximation on modern graphics processing units (GPUs), the primary computational bottleneck arises from matrix multiplication operations. Optimising these operations has been extensively studied in the 20th century, with numerous algorithms being developed to minimise the computational complexity of the calculations. In practice, however, many of the fastest algorithms developed historically have not yet been fully adapted to modern GPU architectures, often due to large memory overheads and limited compatibility with the parallelism and memory hierarchy optimisations that GPUs require. Instead, a technique known as tiling has become the standard approach. Tiling breaks down large matrices into smaller, manageable blocks and aims to optimise matrix multiplications by improving data locality and reuse.

In this paper, we focus on the theoretical foundations and computational strategies underlying the evaluation phase of models similar to LNCDEs, aiming to optimise both memory efficiency and parallelisability when computing matrix exponentials. All code for the experiments can be found at `https://github.com/MazzimV/Thesis`.

2 Background

2.1 Transformers

Let $x \in \mathbb{R}^{N \times d}$ represent the matrix of N feature vectors, each with dimension d. Transformers are functions $T : \mathbb{R}^{N \times d} \to \mathbb{R}^{N \times d}$ made up by composing transformer layers. These layers can be described as

$$T_l = f_l(A_l(x) + x), \tag{1}$$

where A_l is the self-attention function which acts across the length of the sequence, and f_l is a two-layer feedforward network which acts on each row independently [29].

2.1.1 Attention [56]

For each token in a sequence, the self-attention mechanism determines how much attention to give to every other token based on the similarity of their representations. This is computed by taking the dot product between the token's projected query vector and the key vectors of all other tokens, scaling the result by the square root of the dimension, and applying a softmax to produce normalised attention weights. The query, key and value matrices are obtained by projecting x using three learnable matrices W_Q, W_K and W_V where,

$$Q = xW_Q, \quad K = xW_K, \quad V = xW_V,$$

and

$$A_l(x) = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V,$$

where the softmax function is applied row by row. This parallel form allows the softmax to be applied to each row of the input matrix simultaneously, exploiting the architecture of modern GPUs.

2.1.2 Linear Attention

Whilst attention achieves strong performance, its $\mathcal{O}(N^2)$ computational complexity becomes prohibitive for long sequences. To address this scalability limitation, Katharopoulos et al. [29] reformulated transformers as linear RNNs, replacing the softmax function with a polynomial feature map ϕ which transforms the query and key vectors to enable the approximation:

$$\exp\left(\frac{Q_i K_j^T}{\sqrt{D}}\right) \approx \phi(Q_i)^T \phi(K_j),$$

resulting in

$$A_l^{(i)}(x) = \frac{\sum_{j=1}^N \phi(Q_i)^T \phi(K_j) V_j}{\sum_{j=1}^N \phi(Q_i)^T \phi(K_j)} = \frac{\phi(Q_i)^T \sum_{j=1}^N \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^N \phi(K_j)}. \tag{2}$$

From this final form, it is easy to see that the memory complexity is $\mathcal{O}(N)$ as $\sum_{j=1}^N \phi(K_j)V_j^T$ and $\sum_{j=1}^N \phi(K_j)$ can be computed once and applied to each query position.

One of the main uses of Transformers is in auto-regressive modelling, where the objective is to predict each token based solely on previous tokens in the sequence. This is accomplished by masking future positions in attention calculations so that each output depends only on preceding positions. After applying this causal mask, Equation 2 becomes:

$$A_l^{(i)}(x) = \frac{\phi(Q_i)^T \sum_{j=1}^i \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^i \phi(K_j)}.$$

2.1.3 Reformulating Transformers as RNNs

With the masking introduced, the transformer effectively becomes a model which takes an input, modifies a hidden state depending on previous inputs and predicts an output, similarly to an RNN. In fact, Equation 1 can be reformulated in the following way [29]:

$$\begin{aligned}
s_0 &= 0, \\
z_0 &= 0, \\
s_i &= s_{i-1} + \phi(x_i W_K)(x_i W_V)^T, \\
z_i &= z_{i-1} + \phi(x_i W_K), \\
\tilde{T}_l^{(i)} &= f_l\left(\frac{\phi(x_i W_Q)^T s_i}{\phi(x_i W_Q)^T z_i} + x_i\right).
\end{aligned}$$

In this formulation, s_i and z_i function as hidden states that are updated incrementally as model takes new inputs. This establishes that transformers, when causally masked, operate as linear RNNs through sequential state updates.

Although this RNN formulation demonstrates linear complexity, it suffers from the fundamental challenge of how to initialise the transformation matrices used to avoid unstable dynamics such as vanishing gradients during the training process. SSMs address this limitation by leveraging structured initialisations which provide stable parameters.

2.2 State-Space Models

SSMs are a type of linear RNN that can be reformulated through different equivalent mathematical representations. Originally derived from control theory, SSMs can be expressed through continuous differential equations which can be discretised to yield recurrent or convolutional formulations for sequential time processing applications. In recent years, SSMs have become an increasingly prominent alternative to transformer architectures due to their ability to reach linear computational complexity whilst maintaining comparative performance on sequence modelling tasks, with the 2023 Mamba architecture outperforming transformers on a range of language modelling benchmarks [19].

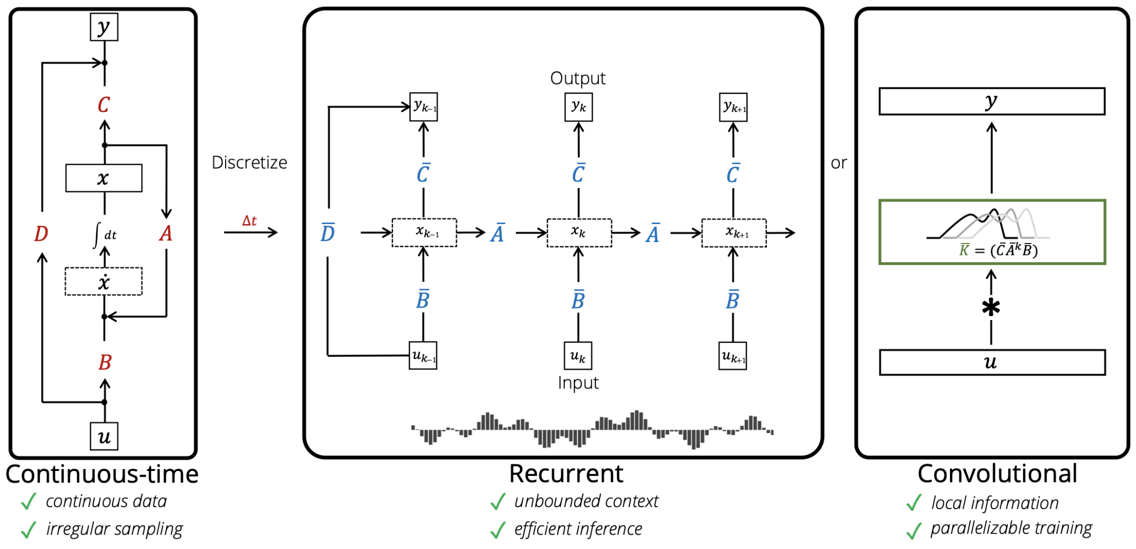

Figure 1: Three Formulations of SSMs [21]

[20] **Continuous Formulation:** SSMs are defined by the following continuous differential equation:

$$
\begin{aligned}
x'(t) &= Ax(t) + Bu(t) \\
y(t) &= Cx(t) + Du(t),
\end{aligned}
$$

where a one dimensional input $u(t)$ is mapped to an N dimensional latent space $x(t)$ before being projected to a one dimensional output signal $y(t)$. Furthermore, re term Du is often viewed a skip connection and omitted from formulations, as it is easily computed [20]. We will thus proceed with $D = 0$.

[20] **Recurrent Formulation:** Whilst the continuous formulation provides the theoretical foundation for SSMs, practical implementation requires discrete-time computation. This is done by discretising the continuous form to obtain a recurrent reformulation, mapping a discrete one dimensional input sequence u instead of the continuous input $u(t)$:

$$
\begin{aligned}
x_k &= \bar{A}x_{k-1} + \bar{B}u_k \\
y_k &= \bar{C}x_k,
\end{aligned}
$$

where each of $\bar{A}, \bar{B}$ and $\bar{C}$ are obtained from closed formulas in A, B and C.

[20] **Convolutional Formulation:** Due to their sequential dependencies, the recurrent formulation of SSMs is computationally inefficient for training on modern parallel hardware architectures. Consequently, research efforts have focused on developing convolutional representations of SSMs that enable parallel computation during training. By unrolling the recurrent formulation provided above, we obtain:

$$
\begin{array}{llll}
x_0 = \bar{B}u_0, & x_1 = \bar{A}\bar{B}u_0 + \bar{B}u_1, & x_2 = \bar{A}^2\bar{B}u_0 + \bar{A}\bar{B}u_1 + \bar{B}u_2, & \dots \\
y_0 = \bar{C}\bar{B}u_0, & y_1 = \bar{C}\bar{A}\bar{B}u_0 + \bar{C}\bar{B}u_1, & y_2 = \bar{C}\bar{A}^2\bar{B}u_0 + \bar{C}\bar{A}\bar{B}u_1 + \bar{C}\bar{B}u_2, & \dots
\end{array}
$$

yielding,

$$
y_k = \bar{C}\bar{A}^k\bar{B}u_0 + \bar{C}\bar{A}^{k-1}\bar{B}u_1 + \cdots + \bar{C}\bar{A}\bar{B}u_{k-1} + \bar{C}\bar{B}u_k.
$$

Thus, we can obtain an explicit formula for the convolutional kernel K, namely:

$$K := (\bar{C}\bar{A}^i\bar{B})_{i\in[L]} = (\bar{C}\bar{B}, \bar{C}\bar{A}\bar{B}, \dots, \bar{C}\bar{A}^{L-1}\bar{B}).$$

Hence, the output of the SSM can be obtained by applying the convolution filter to the input u, resulting in $y = u * K$.

A key factor contributing to the superior performance of modern SSMs is their use of input-dependent state transition matrices, which enable them to dynamically select which information to preserve and which to discard based on the contextual content of the input sequence. Furthermore, they are fundamentally characterised by recurrent formulations that exhibit linear dynamics in the hidden state space while permitting non-linear transformations of the input. This design enables their analysis through a continuous-time framework using linear controlled differential equations [9].

2.3 Linear Controlled Differential Equations

Definition 2.1 (Linear Controlled Differential Equation [57]). Let $\omega : [0, T] \to \mathbb{R}^{d_\omega}$ be a path with bounded variation, $A_i \in \mathbb{R}^{d_h \times d_h}$ be linear vector fields for the channels $i = 1, \dots, d_\omega$ and $h : [0, T] \to \mathbb{R}^{d_h}$ be the solution path. A linear controlled differential equation takes the form

$$dh_s = \sum_{i=1}^{d_\omega} A_i h_s d\omega_s^i.$$

Definition 2.2 (Neural Controlled Differential Equation [33]). Suppose we have a multi-variate time series with a set of observations $\{(t_i, x_i)\}_{i=0}^n$ with $x \in \mathbb{R}^v$, which may be sampled at irregular time intervals. Let $X : [t_0, t_n] \to \mathbb{R}^{v+1}$ be a continuous interpolation such that $X_{t_i} = (x_i, t_i)$.

A neural controlled differential equation is a time series model that takes the form:

$$h_t = h_{t_0} + \int_{t_0}^{t} f_\theta(h_s)\, dX_s,$$

where $h_{t_0} = \zeta_\phi(x_0, t_0)$. Here, $f_\theta : \mathbb{R}^w \to \mathbb{R}^{w\times(v+1)}$ and $\zeta_\phi : \mathbb{R}^{v+1} \to \mathbb{R}^w$ are neural networks depending on parameters θ and ϕ respectively and w is the dimension of the hidden layer. The output of the model is then obtained by applying a linear map to the hidden layer h_t.

It has been shown that the action of a linear map on the output of a NCDE is a universal approximator of continuous functions $C(\mathbb{R}^v, \mathbb{R})$ [33]. This property, coupled with their ability to handle non-uniform sampling, makes LNDCEs a strong option for modelling complex temporal dynamics.

2.3.1 Reformulating SSMs as LNCDEs

Recently, several SSM layers have been reformulated as linear NCDEs [9], including S4 and S6, the selective SSM behind the Mamba architecture [19]. The generalised form for a LNCDE [57] is given by:

$$h_{t_n} = h_{t_0} + \int_{t_0}^{t_n} \sum_{i=1}^{d_\omega} A_\theta^i h_s\, d\omega_s^{X,i}. \tag{3}$$

Here, A_θ^i are learnable weight matrices depending on parameters θ and ω_s^X are paths depending on the input X. Under this general form, universal approximation is obtained when taking A_θ^i to be dense matrices [9]. However, the computational cost of calculating $A_\theta^i h_s$ for these dense matrices renders them unusable for large models. As a result, A^i_θ are taken to be diagonal matrices in S4 and S6, which significantly reduces the expressive capacity of the model, compromising both state-tracking ability and universal approximation guarantees [36]. As a response, research has explored the use of structured matrices that preserve maximal expressivity whilst remaining computationally feasible in practice [57]. We look into this further in § 5.

2.3.2 Approximating LNCDEs

In order to use LNCDEs such as Equation 3 with discrete data, it is possible to use finite-difference approximation to replace the continuous paths ω_s with a linear interpolation of the discrete observations $\{\omega_{t_j}\}_{j=0}^n$ [57]. This is done by approximating the differential $\mathrm{d}\omega_s$ on each interval $s \in [t_j, t_{j+1}]$ by the finite difference:

$$\frac{\omega^i_{t_{j+1}} - \omega^i_{t_j}}{t_{j+1} - t_j},$$

where $\omega^i_{t_j}$ represents the i-th component of the interpolated path at time t_j. Thus the integrand

$$\mathrm{d}h_s = \sum_{i=1}^{d_\omega} A^i h_s \,\mathrm{d}\omega^i_s,$$

in Equation 3 can be rewritten as

$$\mathrm{d}\tilde{h}_s = \left(\sum_{i=1}^{d_\omega} \frac{\omega^i_{t_{j+1}} - \omega^i_{t_j}}{t_{j+1} - t_j} A^i \right) \tilde{h}_s \,\mathrm{d}s,$$

with solution

$$\tilde{h}_{t_{j+1}} = \exp\left(\sum_{i=1}^{d_\omega} \left(\omega^i_{t_{j+1}} - \omega^i_{t_j} \right) A^i \right) \tilde{h}_{t_j},$$

over each interval $[t_j, t_{j+1}]$. This form can easily be composed using associative parallel scans (§ 7.1, [6]), computing the exponential part for each $j \in [0, n-1]$.

3 Matrix Multiplication Algorithms

The evaluation of LNCDE outputs fundamentally relies on composing matrix exponentials through parallel scan operations. Regardless of the specific approximation method employed for computing the matrix exponential, the core underlying operation remains the calculation of the powers of the state transition matrix

$$\sum_{i=1}^{d_\omega} \left(\omega^i_{t_{j+1}} - \omega^i_{t_j} \right) A^i$$

through matrix multiplication. Given that matrix multiplication is a fundamental part of various different applications of computer science, including machine learning, computer graphics, image processing, and solving linear systems, LNCDEs can capitalise on decades of research into finding efficient matrix multiplication algorithms.

One of the first major breakthroughs in the space came in 1969, when Volker Strassen demonstrated that matrices could be multiplied in $\mathcal{O}(n^{2.807})$ time [55], breaking the cubic $\mathcal{O}(n^3)$ barrier, which had been considered the only viable method prior to the discovery. Following this finding, numerous algorithmic methods were developed throughout the 1970s and 1980s attempting to further reduce this exponent, culminating in a notable advance achieved by Coppersmith and Winograd, who reduced the cost to $\mathcal{O}(n^{2.376})$ in 1987 [10]. More recently, this method was improved upon by Andrew James Stothers [54], Virginia Vassilevska Williams [61] and Le Gall [17], reducing the complexity to $\mathcal{O}(n^{2.373})$. Finding the lowest theoretical value of this exponent remains an open problem, and the asymptotic complexity estimate conjectured to be $\mathcal{O}(n^{2+\epsilon})$ for some arbitrary $\epsilon > 0$ remains unproven [53].

In recent years, ways of tackling the issue of fast matrix multiplication have shifted from more traditional analytical and algebraic approaches to computational and automated discovery methods, using reinforcement learning [15] and custom search algorithms [30]. In May 2025, Google DeepMind's evolutionary algorithm, AlphaEvolve [43], was able to achieve a significant milestone, discovering new tensor decomposition algorithms, including the first known rank-48 algorithm for the multiplication of two 4×4 complex-valued matrices over a field with characteristic 0, improving Strassen's long-standing result of rank 49.

3.1 Theoretical Complexity of Matrix Multiplication

The development of efficient matrix multiplication algorithms benefits from understanding the operation's fundamental mathematical structure. Matrix multiplication can be represented as a tensor product, a form that provides insight into why certain algorithmic approaches achieve better computational complexity than others. The complete mathematical derivation, along with all necessary definitions, are provided in Appendix A. To better understand the theoretical benefits of various matrix multiplication algorithms from the tensor perspective, we introduce the concept of tensor rank.

Definition 3.1 (Tensor Rank)**.** Let V, W, and U be finite-dimensional vector spaces over a field $\mathbb{F}$. For a tensor $T \in V^* \otimes W^* \otimes U$, the rank of T, denoted $R(T)$ is defined as:

$$R(T) = \min \left\{ r \in \mathbb{N} : T = \sum_{i=1}^{r} \varphi_i \otimes \psi_i \otimes u_i \right\}$$

where $\varphi_i \in V^*$, $\psi_i \in W^*$, and $u_i \in U$ for all $i \in 1, 2, \ldots, r$.

As it turns out, tensor rank can be utilised to obtain the lowest number of variable, variable multiplications required for a given matrix multiplication. This is formalised in the below lemma.

Lemma 3.2. *Let $T_{\langle m,k,n \rangle} \in (F^{m\times k})^* \otimes (F^{k\times n})^* \otimes F^{m\times n}$ be the tensor representing the matrix multiplication $AB = C$. Suppose that $T_{\langle m,k,n \rangle}$ admits a decomposition of the form:*

$$T_{\langle m,k,n \rangle} = \sum_{i=1}^{r} \varphi_i \otimes \psi_i \otimes C_i,$$

where $\varphi_i \in (F^{m\times k})^*$, $\psi_i \in (F^{k\times n})^*$, and $C_i \in F^{m\times n}$. Then $R(T)$ is the lowest number of scalar multiplications required to compute the matrix product.

Proof. Define $s_i := \varphi_i(A), t_i := \psi_i(B)$. We can now compute the scalar product $u_i := s_i \cdot t_i$ yielding:

$$C = AB = T_{<m,k,n>} = \sum_{i=1}^{r} u_i C_i.$$

$\square$

Remark. In algebraic complexity, multiplications between variables (such as entries of matrices) are considered the primary source of computational cost as they require more operations at hardware level [59]. In comparison, additions and multiplication by a scalar constant are often treated as negligible. In Lemma 3.2, the term *scalar multiplication* specifically refers to variable-variable multiplications, which dominate the complexity.

3.2 Naive Algorithm

Example 3.3 (Naive 2×2 Algorithm)**.** Let A, B and $C = AB$ be the following 2×2 matrices:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, \quad C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22}. \end{bmatrix}.$$

. Using the naive algorithm for matrix multiplication, each entry of C is computed as follows:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22}. \end{aligned}$$

Each product $a_{ij}b_{jk}$ is a multiplication between two variables. Exactly 8 such products are required to obtain the output C. By the universal property of tensor products, the bilinear map

$$\mu : \mathbb{F}^{2\times 2} \times \mathbb{F}^{2\times 2} \to \mathbb{F}^{2\times 2}, \quad (A, B) \mapsto AB$$

can be uniquely extended to a linear map $\tilde{\mu} : \mathbb{F}^{2\times 2} \otimes \mathbb{F}^{2\times 2} \to \mathbb{F}^{2\times 2}$, which by Theorem A.5, corresponds to a tensor $T_{\langle 2,2,2\rangle} \in (\mathbb{F}^{2\times 2})^* \otimes (\mathbb{F}^{2\times 2})^* \otimes \mathbb{F}^{2\times 2}$.

This tensor can be written as a sum of 8 tensors, where each term has the form $T_i = \phi_i \otimes \psi_i \otimes C_i$, where $\phi_i \in (\mathbb{F}^{2\times 2})^*$ extracts a single entry from A, $\psi_i \in (\mathbb{F}^{2\times 2})^*$ extracts a single entry from B, and $C_i \in \mathbb{F}^{2\times 2}$ is a matrix with a single 1 in the (p, q)-entry and 0 elsewhere, indicating where the product contributes in C. Explicitly:

i	$\boldsymbol{\varphi_i(A)}$	$\boldsymbol{\psi_i(B)}$	$\boldsymbol{C_i}$ (contribution matrix)
1	a_{11}	b_{11}	$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$
2	a_{12}	b_{21}	$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$
3	a_{11}	b_{12}	$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
4	a_{12}	b_{22}	$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
5	a_{21}	b_{11}	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$
6	a_{22}	b_{21}	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$
7	a_{21}	b_{12}	$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$
8	a_{22}	b_{22}	$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$

Table 1: Naive 2×2 matrix multiplication as a decomposition into rank-one tensors

We now turn to Strassen's famous algorithm for matrix multiplication [55], approaching it as a tensor decomposition by seeking the minimal rank required.

3.3 Strassen's Algorithm

Example 3.4 (Strassen's 2×2 Algorithm)**.** Let A, B and $C = AB$ be the same 2×2 matrices as in Example 3.3. Strassen discovered a reformulation of the standard matrix multiplication algorithm, reducing the number of scalar multiplications from the naive 8 to just 7.

$$\begin{aligned}
m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_2 &= (a_{21} + a_{22})b_{11} \\
m_3 &= a_{11}(b_{12} - b_{22}) \\
m_4 &= a_{22}(b_{21} - b_{11}) \\
m_5 &= (a_{11} + a_{12})b_{22} \\
m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
m_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}$$

where the output matrix C is given by:

$$\begin{aligned}
c_{11} &= m_1 + m_4 - m_5 + m_7 \\
c_{12} &= m_3 + m_5 \\
c_{21} &= m_2 + m_4 \\
c_{22} &= m_1 - m_2 + m_3 + m_6.
\end{aligned}$$

Notice that each m_i represents a rank-one tensor of the form $\phi_i \otimes \psi_i \otimes C_i$, where $\phi_i \in (F^{2\times 2})^*$, $\psi_i \in (F^{2\times 2})^*$, and $C_i \in F^{2\times 2}$, capturing how a single scalar multiplication between linear combinations of entries from A and B contributes to specific entries in the output matrix C. Explicitly:

i	$\varphi_i(A)$	$\psi_i(B)$	C_i
1	$a_{11} + a_{22}$	$b_{11} + b_{22}$	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
2	$a_{21} + a_{22}$	b_{11}	$\begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix}$
3	a_{11}	$b_{12} - b_{22}$	$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$
4	a_{22}	$b_{21} - b_{11}$	$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$
5	$a_{11} + a_{12}$	b_{22}	$\begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}$
6	$a_{21} - a_{11}$	$b_{11} + b_{12}$	$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$
7	$a_{12} - a_{22}$	$b_{21} + b_{22}$	$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$

Table 2: Strassen's algorithm as a decomposition into rank-one tensors

Strassen's algorithm can also be applied to larger $2^n \times 2^n$ matrices. The procedure involves splitting the matrix into 2 $2^n/2 \times 2^n/2$ blocks and recursively applying the algorithm described above to each block, down to the base case.

Example 3.5 (Strassen's $2^n \times 2^n$ Algorithm)**.** Let A,B and $C = AB$ be square matrices of size $2^n \times 2^n$, where $n > 1$. Then the matrices A, B and C can be divides into four equal sized blocks such that:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22}, \end{bmatrix}$$

where each A_{ij}, B_{ij}, C_{ij} is a $2^{n-1} \times 2^{n-1}$ submatrix. The multiplication $C = AB$ can then be computed recursively applying Strassen's 7 multiplications as follows:

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

where the output matrix C is given by:

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}$$

Each of the multiplications to obtain M_i involves multiplying smaller $2^{n-1} \times 2^{n-1}$ matrices, meaning the algorithm continues recursively until base case $n = 1$, where Example 3.4 can be used directly. This reduces the number of multiplications required from $(2^n)^3 = 8^n$ to 7^n resulting in significant speed-up for large matrix sizes.

4 GEMM on CPU and GPU

Despite the theoretical advancements discussed in § 3, most of the fastest known matrix multiplication algorithms that achieve low complexity are not practically feasible on modern hardware. Their implementations often involve large constant terms and memory overhead that make them less efficient than optimised versions of the classical cubic algorithm.

In all exponential approximation methods implemented on CPUs and GPUs, General Matrix Multiplication (GEMM) consistently serves as a fundamental computational building block. GEMM is a generalised form of matrix multiplication used in high-performance libraries such as NVIDIA's cuBLAS (CUDA Basic Linear Algebra Subprograms), PyTorch, TensorFlow and several others.

Definition 4.1 (GEMM)**.** General Matrix-Matrix multiple performs one of the matrix operations:

$$C := \alpha \cdot op(A) \cdot op(B) + \beta \cdot C,$$

where $op(X)$ is either X, X^T or X^H (conjugate transpose) and $A \in \mathbb{C}^{M \times K}, B \in \mathbb{C}^{K \times N}, C \in \mathbb{C}^{M \times N}$ are matrices of compatible dimensions.

When performing GEMM on modern processing units, the primary objectives include maximising memory hierarchy utilisation, minimising data movement costs and exploiting parallelism and hardware-specific acceleration. The tiling algorithm is a fundamental strategy employed on both CPUs and GPUs to achieve these goals.

4.1 Tiling Implementation

In practice, GEMM is implemented onto CPU's and GPU's by utilising an algorithm called tiling. This consists of dividing the input matrices into smaller submatrices or tiles, allowing efficient use of the memory hierarchy and facilitating parallel computation. By processing tiles that fit in faster memory levels, such as caches on CPUs and shared memory on GPUs, tiling reduces data movement latency, saving computational time. A visualisation of the tiling process is provided in Figure 2.

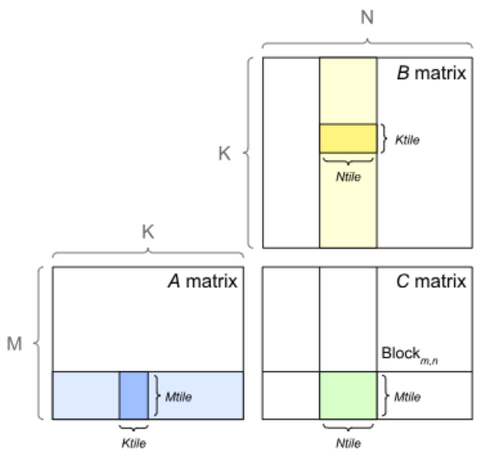

Figure 2: Tiling Algorithm [45]

In this setup, each output block in C is computed by loading in the required values from the necessary blocks of matrices A and B and performing FMA. As is clear from Figure 2, all sub-blocks in C are independent of one another, allowing us to parallelise computations. In fact, each output tile C_{ij} is computed via the equation:

$$C_{mn} = \sum_{k=0}^{K-1} A_{mk}B_{kn}. \tag{4}$$

To demonstrate this process in detail, we now consider a concrete example:

Example 4.2. Suppose we have three matrices A, B, C with dimensions 4×4 where $C = AB$, tiles into 2×2 blocks. Then we have:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}, \quad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}, \quad C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix},$$

Thus using Equation 4,

$$\begin{aligned} C_{00} &= A_{00}B_{00} + A_{01}B_{10} \\ C_{01} &= A_{00}B_{01} + A_{01}B_{11} \\ C_{10} &= A_{10}B_{00} + A_{11}B_{10} \\ C_{11} &= A_{10}B_{01} + A_{11}B_{11}. \end{aligned}$$

From the above, it is easy to see that tiles such as A_{00} are reused across multiple C tiles, in this case C_{00} and $C_{0,1}$.This reuse drastically reduces data movement overhead when the tiles are retained in fast memory, such as caches or shared memory. As matrix sizes grow, these benefits become more pronounced, and enable more efficient utilisation of the processing unit's computational resources.

4.2 Tiling on CPU

On CPUs, the tiling algorithm is primarily used to optimise cache usage, aiming to structure data access patterns such that frequently used tiles remain in the fastest cache possible. Modern CPUs feature multiple levels of cache hierarchy, commonly consisting of L1, L2 and L3 caches. Each successive level of the cache stores larger amounts of data but is located progressively further from the CPU registers where computations occur. This increasing distance results in higher data movement latencies for caches farther from the registers. Efficient tiling implementations organise the tiles of the matrices being used within the fastest possible cache, reducing slower memory accesses.

When performing tiling on a CPU, tiles are processed using nested loops, with the outer loop iterating over tiles of the matrix and the inner loop iterating over individual elements within those tiles. This structure enhances temporal and spatial locality, meaning that data within caches can be reused before being evicted and contiguous memory locations can be accessed efficiently.

Some amount of parallelisation is possible on CPUs, and loop tiling is often combined with multi-threading techniques such as OpenMP which distribute computations of different tiles across CPU cores in parallel. CPUs also allow for SIMD (Single Instruction, Multiple Data) vectorisation, which allows each core to perform operations on multiple data points simultaneously.

Unlike GPUs which allow programmers to control which data is stored in shared memory, CPUs rely on a hardware-managed cache hierarchy. As a result, tile sizes must be carefully chosen to align with cache sizes and characteristics to maximise utilisation and minimise memory accesses.

4.2.1 CPU Architecture Overview

CPUs are designed for versatile computation, balancing fast single-thread performance with multi-core parallelism. CPUs use a memory hierarchy to optimise data access patterns and mitigate the problem of the ”memory wall”, where processors outperform memory bandwidth. Table 3 details this hierarchy for the Intel Core i7-14700K, serving as a representative example to highlight the general orders of magnitude for the storage capacities and latencies of the different components of a CPU.

Type	**Storage**	**Latency**
Global memory (DDR5)	Up to 192 GB	Hundreds of cycles
L3 cache	33 MB	Tens of cycles
L2 cache	28 MB	Tens of cycles
L1 cache	1.5 MB	A few cycles
Registers	256 $\times$ 64-bit per core	One cycle

Table 3: CPU Memory Hierarchy for Intel Core i7-14700K [28]

The fundamental processing units on a CPU is the core, which contains specific resources for instruction handling. Cores can execute multiple threads, the smallest unit of execution, at once. This gives the CPU the ability to perform several instructions simultaneously. When a thread is executed, Arithmetic Logic Units (ALUs) carry out the arithmetic operations, such as those those required for GEMM, whilst vector processing units execute SIMD operations on multiple elements at once.

During thread execution, data is stored in registers, which are the fastest storage accessible by ALUs. The hierarchical memory system mentioned above then helps the CPU access frequently used data efficiently, where multiple levels of cache act as intermediate storage layers between registers and main memory.

4.2.2 Strassen's Algorithm on CPU

Whilst Strassen's algorithm remains unfeasible for small matrix sizes due to the overhead introduced in intermediate calculations, a hybrid approach can exploit its advantages for large matrices where $N \geq 2048$. Modern implementations such as Intel MKL and OpenBLAS combine the recursive block decomposition until matrices of size 64 to 256 are achieved, reverting to classical CPU GEMM on these smaller matrices, obtaining performance increases of up to 25-30% for single-precision matrices of size $N \geq 8192$ on multi-core architectures [27].

4.3 Tiling on GPU

On GPUs, the matrices A and B are stored in global memory. The output matrix is then divided into tiles of a specified size (§ 4.3.3) and the tiles assigned to a thread block (a collection of threads) to compute (§ 4.3.6). Each thread block then loads the necessary tile of A and B into shared memory, allowing the tile to be reused easily. Partial results are stored in registers after computation by threads.

The tiling algorithm recursively divides the main tile into smaller sub-tiles. Each warp, consisting of 32 threads, is assigned one of these sub-tiles. Within the warp, each individual thread computes a small sub-block, typically a 4×4 or 8×8 section of the final output. The data is accessed by the threads from shared memory and temporarily stored in their registers, where the GPU performs the arithmetic operations. The results are then written back to global memory upon completion of the computation.

4.3.1 GPU Architecture Overview

Modern GPUs are built to manage data-intensive workloads. Their hierarchical architecture is composed of several components that facilitate the storage, transfer and usage of data in an efficient manner. Table 4 provides an example of the memory hierarchy of an NVIDIA GeForce RTX 4080 Laptop GPU.

Type	**Storage**	**Latency**
Global memory	12 GB	Hundreds of cycles
L2 cache	48 MB	Tens of cycles
Shared memory and/or L1 cache	128 KB per thread block	A few cycles
Registers	256×32-bit per thread	One cycle

Table 4: GPU Memory Hierarchy for Nvidia GeForce RTX 4080 Laptop GPU[40]

The execution on GPUs is carried out by Streaming Multiprocessors (SMs), that contain the hardware resources for parallel computation to take place. Each SM is responsible for managing hundreds of threads. Threads are handled by the GPU's hardware cores (also known as stream processors), of which there are several on each SM. As in CPUs, ALUs perform arithmetic operations. Data within cores are stored in registers, which are the smallest and fastest private data storage form.

Similarly to CPUs, each GPU contains caches and shared memory to reduce latency and reduce the need to fetch data from the slower global memory. Shared memory and L1 caches provides access to frequently used data and are private to each SM, whilst the L2 cache is shared between all SMs, acting as a larger, intermediate buffer between the SMs and the global memory. The global memory acts as the main memory module of the GPU, offering high capacity with slow latencies.

4.3.2 Tensor Cores

Modern NVIDIA GPUs contain specialised hardware units called Tensor Cores, that are designed to accelerate matrix multiplication and mixed-precision computations for small sized matrices. These features are automatically used when running kernels in cuBLAS, and under the hood in PyTorch.

Tensor Cores are able to perform FMA operations on matrices in a single clock cycle, due to their ability to combine multiplication-accumulate (MAC) operations into a single step at rapid speed. Tensor cores in NVIDIA's 1st generation V100 are able to perform 64 floating-point FMA mixed-precision operations per clock cycle, with the number increasing to up to 256 operations on the A100 and up to 1024 operations for the latest H100, depending on the type of precision data used [44].

Tensor cores require reduced-precision data types (FP16, BF16, INT8, or TF32) to maximize computational throughput and memory efficiency. These formats use fewer bits than FP32/FP64, enabling higher parallelism and lower memory bandwidth usage. Additionally, matrices must meet architecture-specific alignment requirements. For the most recent NVIDIA GPU architectures (A100, H100), matrices with dimensions that are multiples of 16 and 32 should be used to maximise performance [44].

4.3.3 Tile Dimensions

When selecting the size of the tiles used in the algorithm, the main consideration is the trade-off between data reuse and parallel utilisation of the GPU. For larger tiles, more data is held within shared memory allowing for fewer global memory accesses at the cost of reducing the number of concurrent thread blocks that can run on each SM. However, using larger tiles decreases the overall number of parallelisable tiles, which may result in suboptimal occupancy of the GPU. This trade-off is particularly significant for small matrices, as sufficiently large matrices can utilise the largest available tiles while still achieving full GPU occupancy, simply due to their scale [45].

Recent research has focused on optimising tiling algorithms, yielding significant performance improvements. Google DeepMind's latest evolutionary algorithm, AlphaEvolve, has been used to discover optimal tile dimensions tailored to specific hardware, speeding up the training of the Google Gemini LLMs by 1% [43]. When performing GEMM on NVIDIA GPUs with cuBLAS, pre-compiled hardware-specific kernels are used alongside heuristic-based tile selection. These heuristics, implemented as hard-coded rule-based systems, determine the optimal tile sizes based on the dimensions of matrices used, data precision types and capabilities of the GPU hardware using empirically gathered data.

4.3.4 Tile Quantisation

When the dimensions of the matrices used in GEMM are not divisible by the size of the thread blocks used, tile quantisation occurs. This is when threads are idle, or equivalently, compute padded values added for dimensional consistency.

Example 4.3. [45] In Figure 3, a simple example is provided of 128×128 thread block tiles for 256×256 and 256×257 matrices respectively. In the second case, six thread blocks are launched compared to only four in the first case. As all threads within the same block must execute the same FMA operation, the second matrix must be padded with additional zeros to a multiple of the thread block size, in this case adding 127 columns. This results in the vast majority of threads in the extra blocks performing redundant arithmetic operations. For the given example,

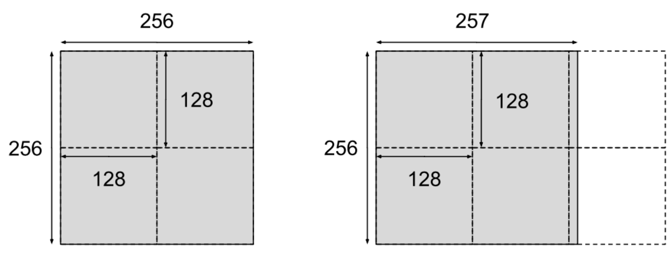

Figure 3: Example of Tiling Quantisation [45]

the number of arithmetic operations increases by 50%, when only $256 \times 257/256 \times 256 = 0.39\%$ additional operations are actually required.

4.3.5 Wave Quantisation and Stream-K Scheduling

Another issue arises when the number of tiles is quantized to match the number of multiprocessors on the GPU, a process known as wave quantization. To explain this process we begin by defining the wave size as the number of thread blocks that can simultaneously be executed on a GPUs SMs.

During wave quantisation, the thread blocks launched do not fully utilise the available capacity of the SMs. Despite this underutilisation, the execution time remains similar to full waves, when all SMs are in use.

Example 4.4. [45] Consider the tiling setup from Figure 2. We use $M = 2304$ and $K = 4096$ with thread blocks of size 256×128, where one thread block can be executed per SM. This leads to a wave size of 108 tiles that can be executed in a single wave cycle. The number of tiles per column is $2304/256 = 9$. When $N = 1536$, the number of tiles per row is $1536/128 = 12$, resulting in $9 \times 12 = 108$ total tiles, or one full wave. When $1536 < N \leq 1664$, there are 13 tiles per row, resulting in $9 \times 13 = 117$ total tiles. This results in one full wave and a tail wave of only 9 tiles. This tail wave has comparable execution time to the full wave, but uses only $9/108 = 8.33\%$ of the GPUs SMs.

In response to this issue, NVIDIA researchers discovered Stream-K processing, which optimises the distribution of computation across GPUs [47]. In Stream-K processing, each output tile is divided into k-iterations, before dividing all k-iterations across all tiles evenly between all available cores in a single wave cycle. This differs to the traditional tiling approach in which each core is responsible for computing all k-iterations to compute the required output tile.

Example 4.5. Suppose we have the same setup as in Example 4.4, which results in an inefficient tail wave, taking $N = 1664$. This results in 117 total tiles. To find the number of k-iterations per tile, we divide K (the original K dimension) by 128, the second dimension along our output tiles. This yields

$$\text{k-Iterations per tile} = 4096/128 = 32,$$

resulting in $117 \times 32 = 3744$ total k-iterations over all tiles. Thus, to divide the work over all 118 available SMs:

$$\text{k-Iterations per SM} = 3744/108 = 34.67.$$

This means that some SMs will get 34 k-iterations, whilst some will get 35. More precisely, 72 SMs will get 35 iterations, and 36 SMs will get 34 iterations ($72 \times 35 + 36 \times 34 = 2,520 + 1,224 = 3,744$). This results in a single wave utilising near-optimal GPU capacity, compared to around 54.17% efficiency for Example 4.4.

In Stream-K processing, SMs perform k-iterations that may span over several output tiles. Thus, SMs must coordinate with each other to produce complete tiles. In fact, the SM responsible for computing the first k-iteration of each output tile becomes responsible for producing the final output tile result by loading partial sums from all SMs that contribute to the same tile and accumulating them with its own computed results. This accumulation overhead is computationally minimal when compared to the huge number of MAC operations computed for a given output tile.

Although Steam-K allows for significant improvements in GPU utilisation, achieving optimal performance with the method is highly dependent on tailoring the configuration to the size of the matrices used as well as the specific GPU architecture. As a result, Stream-K ++ was introduced by researchers at Advanced Micro Devices (AMD), to find an efficient way of selecting the most suitable Stream-K configuration for a given setup [51]. Stream-K ++ expands the original Stream-K algorithm from 3 to 7 distinct scheduling policies, using Bloom filters to eliminate the large majority of unsuitable configurations before selecting the optimal workload distribution strategy. The exact methodology using Bloom filters is explained in Appendix C.

The results show that whilst standard data-parallel policies are optimal for most matrix sizes, Stream-K methods demonstrate significant performance advantages for certain edge cases where matrix dimensions result in highly inefficient tail waves. In fact for certain scenarios, performance improvements of over 40% were observed for Stream-K methods [51]

4.3.6 Assigning Output Tiles to Thread Blocks

In the tiling process described, each output tile is assigned to an individual thread block for computation. There are several methods to how these assignments are made, with the overarching goal of maximising data locality for efficient reuse.

In recent years, extensive research has investigated various methods for assigning tiles, with particular attention given to space-filling curves such as Morton (Z-order) curves [42], Peano curves [2], and Hilbert curves [8]. However, these are not implemented in practice, with cuBLAS and other common libraries using naive sequential row-major assignment.

4.4 Performance Bottlenecks

To analyse the performance of the GEMM function on a GPU, we follow NVIDIA's framework [45], which we summarise below. In general, performance is constrained by either memory bandwidth or compute bandwidth.

4.4.1 Double Buffering

One of the performance bottlenecks in GEMM performed on GPUs is the latency introduced by accessing global memory. As demonstrated in Table 4, this latency can often be hundreds of cycles long, magnitudes of order longer than FMA operations. Thus, if data loading and computation are performed sequentially, large inefficiencies can be introduced to the pipeline when computational units sit idle whilst waiting for data to be accessed.

In order to mitigate this issue, GPUs use two buffers that alternate between loading data and executing computations, with both of these processes occurring simultaneously. In fact, public libraries such as NVIDIA's CUTLASS uses double buffering at each stage in the data loading process [31].

4.4.2 Memory Bandwidth vs Compute Bandwidth

Assuming that different threads can be overlapped, the total time for a function to be executed is $\max(T_{\text{mem}}, T_{\text{math}})$ where T_{mem} and T_{math} represent the time spent in accessing the memory and performing mathematical operations respectively. The larger of these two values dictates the limiting factor in performance. We say that the function is memory limited if $T_{\text{mem}} > T_{\text{math}}$, and that it is math limited if $T_{\text{math}} > T_{\text{mem}}$.The memory time is the number of bytes accessed from the memory divided by the GPU's memory bandwidth and the mathematical time is the number of operations required divided by the GPU's mathematical bandwidth.

Therefore, to better understand whether the bottleneck on performance is caused by the memory or mathematical bandwidth, we can rearrange the equation

$$\begin{aligned} T_{\text{math}} > T_{\text{mem}} &\implies \#\text{ops}/\text{BW}_{\text{math}} > \#\text{bytes}/\text{BW}_{\text{mem}} \\ &\implies \#\text{ops}/\#\text{bytes} > \text{BW}_{\text{math}}/\text{BW}_{\text{mem}}. \end{aligned}$$

The right-hand side is a constant for any given processor, called the ops:byte ratio. The left-hand side, known as arithmetic intensity, varies with the algorithm and its implementation, allowing us to identify whether performance is limited by computation or memory.

Notice that applying GEMM for the dimensions from Definition 4, the product AB has $M \times N$ values, each of which are obtained by taking a dot-product of K elements. Therefore, $M \times N \times K$ fused multiply-adds are required. Thus, we have:

$$\text{Arithmetic Intensity} = \frac{\text{No. of FLOPS}}{\text{No. of bytes accessed}} = \frac{2MNK}{2(MK + NK + MN)} = \frac{MNK}{MK + NK + MN}.$$

5 Tiling Algorithms for Structured Matrices

State-transition matrices in the LNCDE method of § 2.3.2 [57] can adopt several different structural forms. In order to achieve the fundamental goal of sequence models in achieving both computational efficiency and universal approximation capabilities, we introduce the following theorem:

Theorem 5.1. *[57] Consider $\mathcal{X}$ to be the space of time-augmented continuous paths with bounded variation on $[0, T]$, where all paths share a common starting point. This space is equipped with the 1-variation topology. Define $\mathcal{F}$ as the family of LNCDEs incorporating a linear readout layer ℓ_{θ_2}. Each function $f_\theta : \mathcal{X} \to \mathbb{R}$ in this family takes the form*

$$\omega \mapsto \ell_{\theta_2}(h_{t_n}) = \ell_{\theta_2}\left(h_{t_0} + \int_{t_0}^{t_n} \sum_{i=1}^{d_\omega} A^i_{\theta_1} h_s \, d\omega^i_s\right),$$

for $\omega \in \mathcal{X}$. In this setting, $\mathcal{F}$ is maximally expressive [32]. In fact, when $A^i_{\theta_1}$ are taken to be dense matrices, maximally expressivity is obtained. However, when these state-transition matrices are taken to be diagonal matrices, such as in S4 [20] and Mamba [19], maximal expressivity is lost [9].

In the theorem, expressivity refers to the breadth of functions a model can approximate, with maximal expressivity (or universal approximation) ensuring that any continuous function on a compact set can be approximated to arbitrary precision given appropriate parameters. Explicit definitions are provided in Appendix B. The theorem suggests that an effective compromise exists in structured matrix forms that exceed diagonal matrices in expressivity whilst reducing parameter count in comparison to dense alternatives with the aim of improving computational efficiency. Walker et al., found that block-diagonal, Walsh-Hadamard and sparse matrix parametrisations achieve precisely this balance, maintaining maximal probabilistic expressivity [57].

For our use case, it is thus important to consider how the tiling algorithm can be sped up when dealing with matrices of these structures. Modern NVIDIA libraries such as cuSPARSE and cuTLASS aim to achieve this through existing tiling algorithms that help exploit patterns in structured matrices, skipping redundant operations and improving data locality, when compared to naive dense GEMM methods. In the following sub-sections, we explore ways to optimise GEMM for the three different forms of state-transition matrices.

5.1 Block-Sparse Matrices

Black-sparse matrices are structured such that each dimension of the matrix can be partitioned into square blocks, with only a subset of these blocks containing dense values. Block-diagonal matrices are a specific subset of block-sparse matrices which consist of dense blocks alongside the diagonal of the matrix. This form is represented by the notation

$$A^i_\theta = \mathrm{BlockDiag}(B^i_{\theta,1}, B^i_{\theta,2}, \ldots, B^i_{\theta,k}),$$

where k and b are the number and size of the blocks respectively.

5.1.1 Blocked-Ellpack Formatting

The general method used to perform GEMM on a block-sparse matrix and a dense matrix uses Blocked-Ellpack formatting [63], which we define as:

Definition 5.2. The Blocked-Ellpack format of a matrix is made up of two 2D arrays, as visualised in Figure 4 below. The right of the two arrays contains the dense blocks of the original matrix, stored in consecutive blocks. The left array contains the indices of the non-zero blocks.

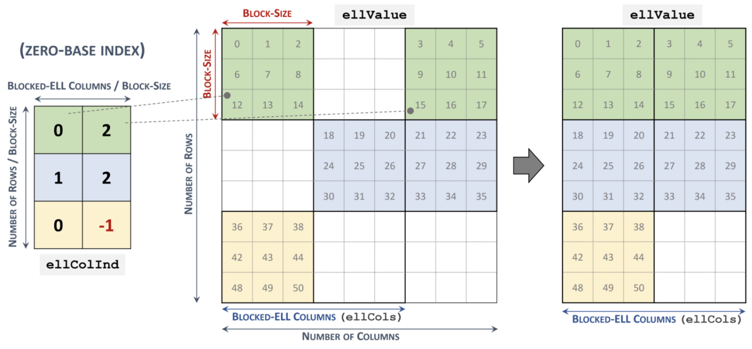
(ZERO-BASE INDEX)
BLOCKED-ELL COLUMNS / BLOCK-SIZE
NUMBER OF ROWS / BLOCK-SIZE
ellColInd
BLOCK-SIZE
ellValue
NUMBER OF ROWS
BLOCKED-ELL COLUMNS (ellCols)
NUMBER OF COLUMNS

Figure 4: Blocked-Ellpack format [63]

5.1.2 Sparse Tensor Cores

Recent NVIDIA GPUs such as the A100 and H100 feature Sparse Tensor Cores that accelerate computations on matrices with 2:4 structured sparsity, where at least two of every four contiguous values are zero [3]. However, this fine-grained sparsity optimisation does not benefit general block-structured matrices such as the block-diagonal forms discussed above. For these forms, NVIDIA's cuSPARSE library can automatically optimise calculations by recognizing the block diagonal structure, avoiding redundant operations on zero blocks without requiring manual format conversion.

General block-sparse matrices require conversion to Blocked-Ellpack formats to achieve performance optimisation. The acceleration mechanism eliminates zero blocks from memory storage entirely, restricting computational operations and memory transfers to non-zero blocks only.

5.1.3 Batched Matrix Multiplication

Another, simpler method called batched matrix multiplication can be used in PyTorch with `torch.bmm` to multiply block-diagonal matrices effectively. Instead of constructing and storing the entire large sparse matrix including many zero blocks, this approach represents the block-diagonal matrix as a batch of smaller dense blocks where each block corresponds to one of the diagonal sub-matrices.

By representing block-diagonal matrices as tensors of dense sub-matrices, `torch.bmm` is able to perform independent dense matrix multiplications for all blocks in parallel, avoiding redundant storage and computations for zero-blocks. As a result, this method significantly reduces computational cost and memory usage, enabling faster GEMM.

5.2 Walsh-Hadamard Matrices

A Hadamard matrix H_n of order 2^n is an $2^n \times 2^n$ matrix contained entries ± 1, with mutually orthogonal rows and columns. Equivalently,

$$H_n H_n^\top = 2^n I_{2^n}.$$

Walsh-Hadamard are built recursively with $H_0 = 1$ and

$$H_n = \begin{pmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{pmatrix}.$$

Multiplication of a Walsh-Hadamard matrix with a vector can be optimised on GPUs using the Fast Walsh-Hadamard Transforms (FWHT) algorithm, which we now introduce.

5.2.1 Fast Walsh-Hadamard Transform Algorithm

The FWHT algorithm [16] performs a recursive algorithm to calculate the multiplication of a Walsh-Hadamard matrix and an input vector a. The algorithm is provided in Algorithm 1.

The algorithm works by applying a series of iterative butterfly operations, which takes two numbers and replaces them with their sum and difference. The following example is provided to understand this more clearly:

Example 5.3 (4 Element FWHT)**.** Let the input vector $a = [1, 2, 3, 4]^T$. Suppose we want to compute $x = H_2 a$ where H_2 is the 4×4 Hadamard matrix defined recursively.

Iteration 1 ($h = 1$): Apply butterfly operations to pairs at distance 1:

$$\begin{aligned}(a[0], a[1]) &= (1, 2) \to (1 + 2, 1 - 2) = (3, -1) \\ (a[2], a[3]) &= (3, 4) \to (3 + 4, 3 - 4) = (7, -1) \\ &\Rightarrow x = [3, -1, 7, -1]^T\end{aligned}$$

Algorithm 1 Fast Walsh-Hadamard Transform

```
Require: Vector $a$ of length $N = 2^k$
Ensure: $H_k \times a$
 1: $h \leftarrow 1$
 2: while $h < N$ do
 3:     for $i = 0$ to $N - 1$ by $2h$ do
 4:         for $j = i$ to $i + h - 1$ do
 5:             $x \leftarrow a[j]$
 6:             $y \leftarrow a[j + h]$
 7:             $a[j] \leftarrow x + y$
 8:             $a[j + h] \leftarrow x - y$
 9:         end for
10:     end for
11:     $h \leftarrow 2h$
12: end while
13: return $a$                                        ▷ Transformed array
```

Iteration 2 ($h = 2$): Apply butterfly operations to pairs at distance 2:

$$\begin{aligned}(a[0], a[2]) &= (3, 7) \to (3 + 7, 3 - 7) = (10, -4)\\ (a[1], a[3]) &= (-1, -1) \to (-1 + (-1), -1 - (-1)) = (-2, 0)\\ &\Rightarrow x = [10, -2, -4, 0]^T\end{aligned}$$

Thus we obtain the final result $x = [10, -2, -4, 0]^T = H_2 a$.

The FWHT algorithm can be applied to matrices in similar manner, performing iterations across each row. For an $m \times n$ matrix, the algorithm has $n \log(n)$ floating point operations per row, repeated across m rows. The algorithm thus reduces complexity from $\mathcal{O}(mn^2)$ to $\mathcal{O}(mn \log(n))$.

5.2.2 FWHT on GPU

When performing the algorithm on GPU, the butterfly operations in each inner loop can be parallelised since they operate on non-overlapping pairs of elements. When a is a matrix, operations on each row can also be parallelised. However, the outer loop iterations must be computed sequentially as each iteration depends on the outputs from the previous iteration.

Memory access challenges arise when assigning threads to butterfly operations. If each thread is responsible for processing a single pair of elements, threads must communicate between themselves as pairings change across iterations. For instance, in Example 5.3 the thread working on $(a[0], a[1])$ in the first iteration will process $(a[0], a[2])$ in the second iteration, requiring the output $a[2]$, which was computed by a different thread in the previous iteration.

When assigning one pair of elements per thread, a warp of 32 threads can process up to 64 elements, while a thread block of up to 1024 threads can handle 2048 elements. Within these limits, communication between threads can be efficiently managed through shared memory. However, for larger Walsh-Hadamard transforms, either multiple elements must be assigned per thread, or expensive cross thread block communication is required [1]. To address these challenges, the Dao AI Lab has developed optimised implementations of the FWHT algorithm tailored for efficient execution on CUDA, resulting in substantial speed-ups [11].

5.2.3 Utilising Tensor Cores for FWHT

Whilst the FWHT algorithm is primarily viewed as an alternative to matrix multiplication, recent work has demonstrated that modern NVIDIA Tensor Cores can actually accelerate Hadamard transforms through matrix operations, as introduced in the 2024 HadaCore kernel [1].

The HadaCore algorithm replaces the butterfly operations used in the FWHT method with 16×16 matrix multiplications performed on Tensor Cores. For vectors of length 256, the input vector is reshaped into a square 16×16 matrix and multiplied by H_4. This result is then transposed and multiplied again by H_4, before transposing the result and reshaping back to the original vector form. For larger vectors, the input is partitioned into 256 sized chunks that can be processed in parallel across the GPUs warps, utilising shared memory, and the algorithm is applied on each of these chunks individually.

The algorithm also handles cases where the vector is not a power of 16 by factorising $\text{Length}(a) = 2^m \cdot 16^n$ where $0 < m < 4$. The process then applies the same n iterations as in the case where the input is a power of 16 adding an extra iteration by using the 16×16 matrix with H_m repeated across the diagonal and zeros elsewhere.

The HadaCore method results in significant speed-up on large input vectors whilst maintaining numerical accuracy, demonstrating its best performance on the A100 GPU in comparison to the H100.

5.3 Sparse Matrices

In order to build a sparse representation, the state transition matrix A^i_θ is taken to be the sparse matrix with $\mathcal{O}(d_h^{1+\epsilon})$ non-zero entries, where $0 < \epsilon < 1$. These entries are randomly selected treating each entry as an independent Bernoulli trial, where each element has probability $d_h^{\epsilon-1}$ of being retained.

Optimising sparse matrix multiplication on GPUs is challenging due to the irregular distribution of non-zero elements in sparse matrices. This leads to inefficient memory access patterns, poor cache utilisation, and uneven distribution of computational loads across GPU threads.

Recently, NVIDIA have introduced cuSPARSE, a library with specialised routines for sparse matrices. These kernels are optimised for matrices with high sparsity ratios ($\geq 70\%$), particularly those stored in Compressed Sparse Row (CSR) formatting.

5.3.1 Compressed Sparse Row Formatting

The CSR format has become the predominant storage scheme used in most modern SpGEMM methods due its efficiency in represent sparse matrices in a simple structure. CSR represents matrices by storing row offsets, column indices of non-zero elements, and their corresponding values. This is visualised in Figure 5.

By storing only non-zero elements and their positional information, CSR reduces memory requirements and enables efficient memory accesses during computation. However, despite its common use across modern SpGEMM algorithms, the CSR format requires manual optimisation, with few existing techniques to automatically adapt the storage layout to exploit sparsity patterns.

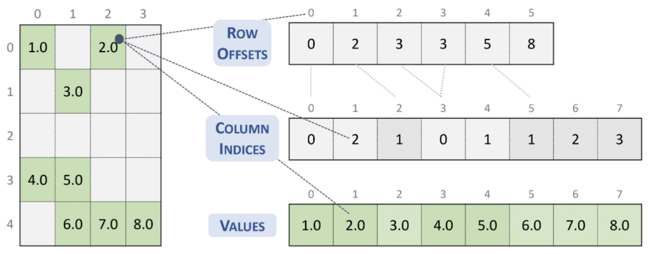

Figure 5: CSR Format [46]

When performing Sparse GEMM (SpGEMM) on GPUs, most methods use Gustavsons' row-row formulation [41] as their starting point. We introduce this algorithm in the following section.

5.3.2 Gustavson's Algorithm for Sparse GEMM

Gustavson's algorithm utilises the indenpendence of the rows of the output matrix $C = AB$, allowing operations to be parallelised.

Algorithm 2 Gustavson's Row-Row SpGEMM Algorithm [22]

```
Require: Sparse matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$ in CSR format
Ensure: Sparse matrix $C = AB \in \mathbb{R}^{m \times n}$
 1: for $i = 0$ to $m - 1$ do in parallel
 2:     predict size of $c_{i*}$ intermediate products        ▷ Estimate nonzeros in result row
 3:     Allocate $c_{i*}$ to memory
 4:     for each nonzero entry $a_{ij}$ in $a_{i*}$ do
 5:         for each nonzero entry $b_{jk}$ in $b_{j*}$ do
 6:             $value \leftarrow a_{ij} \cdot b_{jk}$
 7:             if $c_{ik} \notin c_{i*}$ then
 8:                 insert $c_{ik}$ to $c_{i*}$
 9:                 $c_{ik} \leftarrow value$
10:             else
11:                 $c_{ik} \leftarrow c_{ik} + value$
12:             end if
13:         end for
14:     end for
15: end for
16: return $C$
```

Whilst the above algorithm takes advantage of parallelisation, issues can arise due to the rows of the output C requiring varying computational loads, depending on the sparsity of the input matrices. As the number of non-zero entries of C is not known before performing the algorithm, a large amount of space can also be required to store intermediate results. Furthermore, the unknown positions of non-zero elements in C make it hard to optimise the accumulation of outputs [41].

In response to the mentioned issues, recent research has looked into ways of optimising tiling algorithms with the primary objective of improving GPU utilisation and addressing load balancing challenges faced in SpGEMM. Several innovative methods have been developed which were considered state-of-the-art at the time of their publication. In the following subsections, we look into the most prominent of these methods.

5.3.3 Ac-SpGEMM, 2019 [62]

Ac-SpGEMM introduces a chunk-based GEMM approach, focusing on achieving four goals: to allow for computations in on chip memory, coherent memory accesses, independence of row length and deterministic bit results. Ac-SpGEMM implements a four-stage process to achieve these.

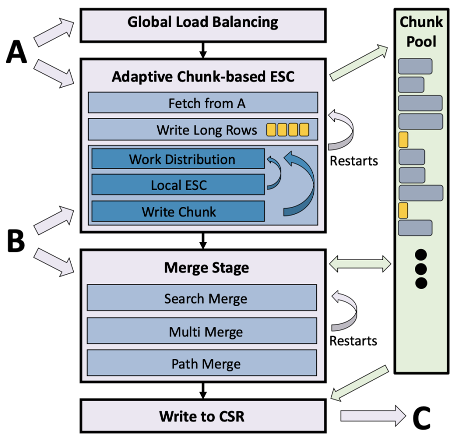

Figure 6: Ac-SpGEMM Framework

In the first stage, non-zeros of the matrix A are uniformly split across thread blocks, providing static memory requirements. The second step, where the key innovation occurs, performs chunk-based Expand-Sort-Compress (ESC) several times, rather than only once. This is the process where intermediate products of $A_{ik} \cdot B_{kj}$ are produced (Expand), these products are sorted by row and column indices (Sort), and products with identical indices are combined to obtain final results (Compress). ESC uses the shared memory local to each thread-block alongside a dynamic work distribution system to decide how much data must be fetched from matrix B for each iteration, ensuring the thread block's local memory stays full but doesn't overflow. The third stage merges rows that are shared across multiple chunks to generate the final result using three specialised merging strategies, each handling row splits over a different number of chunks. The fourth stage allocates the output matrix C, performing a prefix-sum over the rows to obtain the row pointer array as well as the exact memory requirements for allocation of the values and column id arrays. The data is then copied in parallel, iterating over all chunks.

5.3.4 spECK, 2020 [48]

spECK introduces a framework with six stages, aimed at adaptively combining multiple strategies depending on the sparsity patterns of the input matrices used.

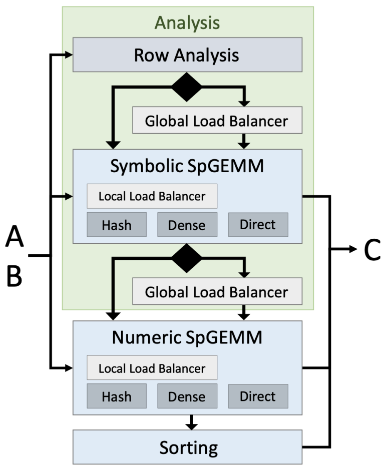

Figure 7: spECK Framework

In the first step, spECK performs an analysis on the order of the number of non-zeros of matrix A ($\text{nnz}(A)$). For each row in A, the algorithm obtains the number of total products required, the longest referenced row of B, and the minimum and maximum indices of non-zero elements in these referenced rows. The second step decides whether to use load balancing, depending on the variation in memory requirements for each output row in C, using the maximum/average ratio. Six kernel configurations are used to optimise thread-block memory utilisation, skipping load balancing altogether for uniform matrices where the overheads outweigh the benefits (under a certain ratio). Step three performs Symbolic GEMM, determining exact output matrix dimensions and memory requirements without computing values. Step four determines the optimal number of threads per row of B for each block, balancing the maximum number of iterations against the number of rows being processed to improve thread utilisation. Step five computes numeric GEMM using the same accumulation strategy as in stage three and step six sorts the results where necessary and generates the final output matrix C.

5.3.5 TileSpGEMM, 2022 [41]

TileSpGEMM decomposes sparse matrices into 16×16 sparse tiles that are stored in CSR format with bit masks and performs a three step algorithm to improve the computation of SpGEMM. An overview of these steps is provided below.

The algorithm begins by performing symbolic GEMM, determining which tile positions in the output matrix C will contain non-zero values. This enables efficient memory allocation for only dense tiles. In the second step, for each output tile, the algorithm uses binary search to find intersecting dense tiles A_{ik} and B_{kj} and generates bit masks to determine the exact sparsity pattern and allocate memory accordingly. Finally, output values are computed calculating values only for positions where non-zeros are found in each tile of C. If the tile being computed contains $< 75\%$ sparsity, a sparse accumulator is used, traversing non-zero values of A_{ik} and B_{kj} individually. If the sparsity is higher than this threshold, operations are performed on the whole 16×16 block in shared memory, as in dense matrix multiplication.

5.3.6 ApSpGEMM, 2025 [64]

ApSpGEMM introduces a four-step adaptive panel-based approach for CPU-GPU collaboration for SpGEMM.

In the first step, ApSpGEMM performs analysis on the non-zero elements of the input matrices, obtaining the number of non-zeros in each row of matrices A and B, and the number of non-zeros in each column of A using prefix sum operations on the CSR format. The second step splits matrices into panels, ordering them in descending sparsity order. Rows exceeding the sparsity threshold over 0.05 are classified as dense, and the rest as sparse, resulting in three operation types: sparse $\times$ sparse, sparse $\times$ dense and dense $\times$ dense. In step three, computation strategies are optimised for each operation type. Step four implements the collaboration between CPU and GPU, allocating higher sparsity panels to GPU and lower sparsity panels to CPU, allocation around $60 - 65\%$ of computation to the former. Computation-transfer overlap strategies are also introduced to reduce communication latencies between the cores.

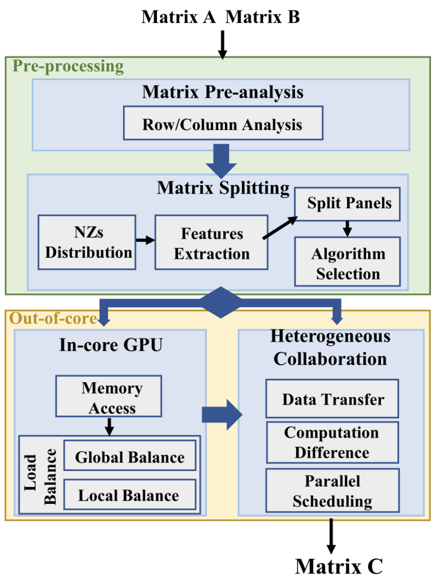

Figure 8: ApSpGEMM Framework

Now that we have explored algorithms for GEMM on GPU for structured matrices, we can understand how these forms are used in computing matrix exponentials, as required in sequence models.

6 Series Methods for Computing the Matrix Exponential

Having examined the core computation of matrix multiplication that underlies all matrix exponential algorithms, we now turn to the exponential operation itself, which is central to obtaining the solution of LNCDEs over time. In their widely-cited paper, Moler and Van Loan investigated nineteen methods for these computations [39]. These approaches vary significantly in their computational complexity, numerical stability, and suitability for different matrix structures. We begin our analysis of these methods by defining the exponential of a matrix as a convergent power series.

Definition 6.1. Let A be an $n \times n$ matrix. The matrix exponential $\exp(tA)$ is defined by the convergent power series:

$$\exp(tA) = I + tA + \frac{t^2A^2}{2!} + \frac{t^3A^3}{3!} + \cdots$$

6.1 Taylor Series Approximation

In previous works [57], the first-order approximation

$$\exp\left(\Delta t_j \sum_{i=1}^{d_X} X^i_{t_j} A^i_\theta\right) \approx I_{d_h} + \Delta t_j \sum_{i=1}^{d_X} X^i_{t_j} A^i_\theta$$

has been used to approximate the matrix exponential calculated in § 2.3.2, where the intervals t_j are fixed to align with the time of the observation data and:

$$\frac{\omega^i_{t_{j+1}} - \omega^i_{t_j}}{t_{j+1} - t_j} = X^i_{t_j}, \tag{5}$$

where $X_{t_j} = (x_{t_j}, t_j)$.

This is a specific case of a Taylor series approximation for exponentiating matrices. Taylor series methods truncate the infinite power series in the original definition to a finite degree, introducing approximation error which depends on both the order of truncation and the properties of the underlying matrix. We define the Taylor series approximation of $\exp(tA)$ below:

Definition 6.2 (Taylor Series Approximation)**.** The Taylor Series approximation of $\exp(tA)$ to the k^{th} degree is defined by

$$T_k(tA) = \sum_{j=0}^{k} (tA)^j / j! \tag{6}$$

In Taylor series approximation methods, deciding which degree to truncate the sum at is critical for efficiency. The truncation degree k must balance computational cost against approximation accuracy, as higher degrees require computing additional matrix powers A^j which become increasingly expensive.

6.1.1 Error Bounds

To understand the convergence behaviour the Taylor series for $\exp(tA)$, we look at the error between the infinite series and truncated series. More explicitly, the truncation error is:

$$\exp(tA) - T_k(tA) = \sum_{j=k+1}^{\infty} \frac{(tA)^j}{j!}.$$

To quantify this error, taking $\|\cdot\|$ to be a matrix norm, we can use the triangle inequality and the submultiplicativity of matrix norms to get the bound:

$$\left\| \sum_{j=k+1}^{\infty} \frac{(tA)^j}{j!} \right\| \leq \sum_{j=k+1}^{\infty} \left\| \frac{tA^j}{j!} \right\| \leq \sum_{j=k+1}^{\infty} \frac{\|tA\|^j}{j!} = \exp(\|tA\|) - \sum_{j=0}^{k} \frac{\|tA\|^j}{j!},$$

where the final term is the truncation error of the exponential $\exp(\|tA\|)$. We now utilise Lagrange's Remainder form, which states that

$$\exp(\|tA\|) - \sum_{k=0}^{n} \frac{\|tA\|^k}{k!} = \frac{\exp(\xi)}{(n+1)!} \|tA\|^{n+1}$$

for some $0 \leq \xi \leq \|tA\|$, where $\xi \in \mathbb{R}$. Thus, as the exponential function monotonically increases over real numbers, $\exp(\xi) \leq \exp(\|tA\|)$, and so we obtain the bound

$$\|\exp(tA) - T_k(tA)\| \leq \frac{\|tA\|^{k+1}}{(k+1)!} \exp(\|tA\|).$$

From this form, we are able to estimate the smallest required k for the truncation error to achieve a desired error tolerance. Furthermore, we can see that for small $\|tA\|$, the error decreases quickly for small k. However, when $\|tA\|$ is large, the error term may initially grow until k is large enough for the factorial in the denominator to dominate. To address these convergence challenges while maintaining computational efficiency, rational approximation methods such as Padé approximation offer an alternative to polynomial truncation.

6.2 Padé Approximation

The Padé approximation expresses functions as the ratio of two polynomials. The explicit definition is provided below:

Definition 6.3 (Padé Approximation)**.** The r_{km} Padé approximation to $\exp(tA)$ is defined by:

$$r_{k,m}(tA) = [q_{k,m}(tA)]^{-1} p_{k,m}(tA),$$

where

$$p_{k,m}(tA) = \sum_{j=0}^{k} \frac{(k+m-j)!k!}{(k+m)!j!(k-j)!}(tA)^j, \quad q_{k,m}(tA) = \sum_{j=0}^{m} \frac{(k+m-j)!m!}{(k+m)!j!(m-j)!}(-tA)^j.$$

By construction, $r_{km}(tA)$ matches the first $k+m+1$ terms of the Taylor series for $\exp(tA)$, but generally achieves better practical accuracy than the equivalent-order Taylor truncation T_{k+m} ([4], [26]). In addition, the direct Taylor approach requires computing matrix powers up to A^{k+m}. This is significantly more computationally expensive than when using Padé approximants, where the highest power of A required is $A^{\max(k,m)}$.

6.2.1 Diagonal Padé Approximants

In practice, diagonal Padé approximants are preferred over non-diagonal ones. This is due to the fact that for $k < m$, the largest power calculated is $A^{\max(k,m)}$ in r_{km}, which in this case is A^m, achieving an approximant of order $k+m$. However, the same cost is required to compute r_{mm} for an approximation of order $2m > k+m$. The same applies for $k > m$. If all eigenvalues of A lie in the left half plane, approximants with $k > m$ can have larger rounding errors due to cancellation (§ 6.4) and those with $k < m$ can have larger rounding errors due to ill-conditioned denominator matrices q_{km}. Furthermore, for $k > m$, the Padé approximants are unbounded for large t, when they should converge to 0 as $t \to \infty$, leading to incorrect asymptotic behaviour. Diagonal Padé approximants mitigate these issues by remaining bounded as t grows large, providing a numerically stable compromise [39].

Example 6.4. The diagonal Padé approximation of degree [3/3] of tA is given by

$$\exp(tA) \approx r_{3,3}(tA) = \left(120I - 60\,tA + 12\,(tA)^2 - (tA)^3\right)^{-1}\left(120I + 60\,tA + 12\,(tA)^2 + (tA)^3\right)$$

6.3 Scaling and Squaring

For both approximation methods discussed, numerical roundoff errors and computational complexity tend to increase with the magnitude of $\|tA\|$ or with a larger spectral spread of A [39]. To address these challenges, the exponential identity

$$\exp(A) = (\exp(A/\sigma))^{\sigma},$$

where $A \in \mathbb{C}^{n\times n}$ and $\sigma \in \mathbb{C}$, is leveraged in the scaling and squaring method. In this approach, $\sigma = 2^s$ is chosen where $s \in \mathbb{N}$ is the smallest power of two for which $\|A\|/\sigma \leq 1$. This ensures that $\exp(A/\sigma)$ can be accurately and efficiently approximated using Padé or Taylor expansions, after which $\exp(A)$ is recovered through repeated squaring. The scaling and squaring method has become by far the most widely used due to its implementation in SciPy and MATLAB [25].

6.3.1 Padé Approximation in Scaling and Squaring

When combined with the scaling method, diagonal Padé approximants have been shown to reduce computational cost compared to non-diagonal approximants of the same polynomial degree [7].

When using such approximants, the two parameters k and s must be chosen to obtain r_{kk} and σ. Moler and Van Loan [39] showed that if $\|tA\| \leq 2^{s-1}$, then

$$r_{kk}(tA/2^s)^{2^s} = \exp(tA + E),$$

where

$$\frac{\|E\|}{\|tA\|} \leq 8\left(\frac{\|tA\|}{2^s}\right)^{2k}\left(\frac{(k!)^2}{(2k)!(2k+1)!}\right).$$

This relative error bound enables the selection of parameters k and s to ensure the error remains within any specified tolerance ϵ. Furthermore, due to the fact that $r_{kk}(tA/2^s)^{2^s}$ requires around $(k + s + 1/3)n^3$ flops to be evaluated, the optimal pair (k, s) can be selected from the list of pairs that satisfy the specified error tolerance to minimise the sum $k + s$.

In fact, it has been shown that for $\|A/\sigma\| \leq \epsilon_k$ for $k \in \{3, 5, 7, 9, 13\}$, as summarised in Table 5 below, using a polynomial of degree k is enough to guarantee that the relative error associated to the method is below the unit round-off in IEEE double-precision arithmetic [25].

k	ϵ_k
3	$1.495585217958292 \times 10^{-2}$
5	$2.539398330063230 \times 10^{-1}$
7	$9.504178996162932 \times 10^{-1}$
9	$2.097847961257068 \times 10^{0}$
13	$5.371920351148152 \times 10^{0}$

Table 5: Threshold values ϵ_k for Padé approximants of degree k [25].

Another method that has been explored involves using parallel processing on a GPU cluster with a CUDA-aware Message Passing Interface (MPI), enabling communication between GPUs. In this setup, increasing the number of computer nodes, each equipped with one or more GPUs, led to a reduction in computation time for large transition matrices, with $d_h \geq 816$. However, for smaller transition matrices, the advantage of using multiple nodes diminished, with single-node parallelisation being found to be marginally more efficient [34].

6.4 Catastrophic Cancellation

Both Taylor Series and Padé approximation methods can suffer from numerical instability when intermediate computations of large size compared to the final output are required. This instability primarily manifests through catastrophic cancellation, where accuracy is lost due to limited-floating point precision. To demonstrate this more clearly, the below example is used:

Example 6.5. [39] Suppose that ignoring efficiency, we keep adding terms to Equation 6 until $\mathrm{fl}[T_K(A)] = \mathrm{fl}[T_{K+1}(A)]$; that is, the added term is so small relative to the accumulated sum that it falls below machine precision and no longer affects the computed result.

When applying $T_k(A)$ as the approximation of $\exp(A)$ on the matrix

$$A = \begin{bmatrix} -49 & 24 \\ -64 & 31 \end{bmatrix}$$

the output

$$\begin{bmatrix} 22.25880 & 1.432766 \\ -61.49931 & -3.474280 \end{bmatrix} \neq \begin{bmatrix} -0.735759 & 0.551819 \\ -1.471518 & 1.103638 \end{bmatrix} \approx \exp(A)$$

was obtained on the IBM 370 computer. Upon analysis of intermediate calculations, the authors observed that the matrices $A^{16}/16!$ and $A^{17}/17!$ had elements of opposite signs of order of magnitude 10^6 and 10^7. However, as the relative precision on the computer used was only 10^{-5}, the absolute error of these intermediate terms exceeded the magnitude of the final result. This led to a significant loss of accuracy due to the truncation of floating point arithmetic.

Although the scaling and squaring method reduces approximation error and computational complexity when $\|tA\|$ is large, the algorithm introduces a different form of roundoff error, which occurs when t/s is said to be *under the hump.* This phenomenon arises when

$$\|\exp(tA)\| \ll \|\exp(tA/\sigma)\|^{\sigma}.$$

When this is the case, the intermediate computations during the squaring phases involve matrices with norms that can be significantly larger than the final result, causing catastrophic calculation in a similar manner to Example 6.5.

This phenomenon can be understood through the lens of matrix conditioning. Since

$$(\exp(tA/\sigma))^{-1}(\exp(tA/\sigma))^2 = \exp(tA/\sigma),$$

using the submultiplicativity of matrix norms, we have

$$\begin{aligned} \mathrm{cond}(\exp(tA/\sigma)) = \|\exp(tA/\sigma)\| \cdot \|(\exp(tA/\sigma))^{-1}\| &\geq \|\exp(tA/\sigma)\| \cdot \frac{\|\exp(tA/\sigma)\|}{\|(\exp(tA/\sigma))^2\|} \\ &= \frac{\|\exp(tA/\sigma)\|^2}{\|(\exp(tA/\sigma))^2\|}. \end{aligned}$$

Thus, when $\|\exp(tA/\sigma)\|^2 \gg \|(\exp(tA/\sigma))^2\|$, the inequality shows that $\exp(tA/\sigma)$ must be poorly conditioned.

7 Evaluation in Sequence Models

In sequence models, outputs computed over several time steps frequently depend on associative operations. In our particular setting, this involves the multiplication of matrix exponentials, as discussed in § 2.3.2. Suppose we are tasked with finding the output of a sequence model over T time steps. A naive recursive approach would incur a computational cost of $\mathcal{O}(T)$, as each matrix exponential would be computed and multiplied in sequence. In 1990, Guy Blelloch introduced an algorithm known as the associative parallel scan [6], which reduces the overall complexity to $\mathcal{O}(\log(T))$, exploiting the associativity of underlying operations using parallel computations.

7.1 Associative Parallel Scans [6]

Associative parallel scans inherently rely on prefix operations, which are defined below:

Definition 7.1 (Prefix Operation)**.** Given a sequence $[x_1, x_2, \dots, x_T]$ and an associative operator $\odot$, define the prefix operation as:

$$_T = x_1 \odot x_2 \odot \cdots \odot x_T.$$

The explicit algorithm for the scan is provided in Algorithm 3, adapted to take matrix multiplication as the operator on matrix exponentials.

Example 7.2. Suppose our sequence is $[1, 5, 4, 7]$ and our operator is addition. Then the prefix sums are

$$[1, 1+5, 1+5+4, 1+5+4+7] = [1, 6, 10, 17].$$

Algorithm 3 Blelloch's Parallel Prefix Scan (Adapted for Matrix Exponentials)

```
Require: Array of matrices $A[0 \dots T-1] \in \mathbb{R}^{n \times n}$ where $T$ is a power of 2
Ensure: Exclusive prefix products $E[0 \dots T-1]$ where $E[i] = \prod_{j=0}^{i-1} \exp(A_j)$ for $i \geq 1$, $E[0] = I$
 1: Step 1: Compute Matrix Exponentials
 2: for $i = 0$ to $T-1$ do in parallel
 3:     $E[i] \leftarrow \exp(A[i])$                               ▷ Parallel matrix exponential
 4: end for
    Step 2: Up-Sweep Phase
 5: for $d = 0$ to $\log_2 T - 1$ do
 6:     for all $k = 0$ to $T-1$ by $2^{d+1}$ do in parallel
 7:         $E[k + 2^{d+1} - 1] \leftarrow E[k + 2^d - 1] \cdot E[k + 2^{d+1} - 1]$
 8:     end for
 9: end for
    Step 3: Down-Sweep Phase
10: $E[T-1] \leftarrow I$                                          ▷ Set identity matrix at root
11: for $d = \log_2 T - 1$ down to 0 do
12:     for all $k = 0$ to $T-1$ by $2^{d+1}$ do in parallel
13:         $temp \leftarrow E[k + 2^d - 1]$
14:         $E[k + 2^d - 1] \leftarrow E[k + 2^{d+1} - 1]$
15:         $E[k + 2^{d+1} - 1] \leftarrow temp \cdot E[k + 2^{d+1} - 1]$
16:     end for
17: end for
18: return $E[0 \dots T-1]$                         ▷ Exclusive matrix exponential products
```

Example 7.3. Suppose our sequence is

$$\left[\begin{bmatrix}1 & 2\\0 & 1\end{bmatrix}, \begin{bmatrix}2 & 0\\1 & 2\end{bmatrix}, \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\right]$$

and our operator is matrix multiplication. Then the prefix products are:

$$\left[\begin{bmatrix}1 & 2\\0 & 1\end{bmatrix}, \begin{bmatrix}1 & 2\\0 & 1\end{bmatrix}\begin{bmatrix}2 & 0\\1 & 2\end{bmatrix}, \begin{bmatrix}1 & 2\\0 & 1\end{bmatrix}\begin{bmatrix}2 & 0\\1 & 2\end{bmatrix}\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\right] = \left[\begin{bmatrix}1 & 2\\0 & 1\end{bmatrix}, \begin{bmatrix}4 & 4\\1 & 2\end{bmatrix}, \begin{bmatrix}4 & 4\\2 & 1\end{bmatrix}\right].$$

It is important to notice that this algorithm is an exclusive scan, meaning for an input of size $T+1$ (time steps $0, 1, \ldots, T$), each position $E[i]$ contains the exponential product of matrices from time steps 0 through $i-1$. The first value $E[0]$ is the identity matrix, and the final value $E[T]$ contains the product up to time step $T-1$. Thus in practice, an input of size $T+2$ would be required to compute the output at time T.

While Blelloch's algorithm is mathematically efficient, its practical performance on GPUs depends on various factors, such as hardware characteristics and memory hierarchy behaviour. In practice, implementations such as `jax.lax.associative_scan` adopt a hybrid approach that combines the theoretical advantages of Blelloch's algorithm with additional optimisations tailored to modern hardware to exploit parallelism and mitigate memory bottlenecks. The following sections examine these implementation challenges and discuss strategies employed to address them in high-performance computing environments.

7.1.1 Bank Conflicts

On modern NVIDIA GPUs, shared on-chip memory is divided into smaller chunks called banks. There are usually 32 such banks, to match the number of threads in a warp. Each one of these banks can serve only one access per clock cycle. Consequently, if multiple threads within a warp attempt to access pieces of information that map to the same bank, a bank conflict occurs. These conflicting accesses are serialised, requiring multiple cycles to complete, which can significantly degrade performance [24].

To understand this better, it is important to note that data stored in memory is accessed by index. The memory bank to which the data at index i is mapped is determined by i mod (number of banks). This means that data is cyclically accessed to banks in index i, until the total number of banks is reached. In parallel algorithms, threads frequently access data at regular strided intervals. This predictable pattern causes multiple threads to target memory addresses that map to the same shared memory bank. The below example is provided to visualise this phenomenon more clearly:

Example 7.4. Consider a GPU shared memory system with 32 memory banks and a warp of 32 threads executing in parallel. Each thread accesses an element in shared memory, where the bank number is determined by:

$$\text{Bank Number} = \text{Address mod } 32$$

Suppose each thread t accesses the shared memory element at index $i_t = t \times 2$, i.e., threads access data with a stride of 2. Table 6 shows the bank assignments for threads 0 through 8:

Thread ID t	Accessed Index $i_t = 2t$	Bank Number $(2t) \bmod 32$
0	0	0
1	2	2
2	4	4
3	6	6
$\vdots$	$\vdots$	$\vdots$
15	30	30
16	32	0
17	34	2
$\vdots$	$\vdots$	$\vdots$
31	62	30

Table 6: Bank mapping for threads accessing shared memory with stride 2

As seen in the table above, several bank conflicts occur (e.g. indexes 0 and 32 map to bank 0). This forces the hardware to serialise these accesses, reducing parallel efficiency.

In order to solve this issue, padding is used to increase spacing between elements in the shared memory to ensure that they land in different banks. More specifically, the index

$$i \leftarrow i + \left\lfloor \frac{i}{\text{number of banks}} \right\rfloor .$$

This padding ensures that all indices from Example 7.4 are mapped to distinct banks, allowing simultaneous access by all threads within a single clock cycle and thereby eliminating bank conflicts.

7.1.2 Dealing with Longer Sequences

When handling longer sequences, parallel scan algorithms are extended in hierarchical fashion. The process begins by partitioning the input array into multiple segments, each of which is processed by an independent thread block. Each block is responsible for handling a fixed number of elements (1,024 elements on NVIDIA 8 GPUs [24]), and scans are performed locally within each block.

Within each block, the scan operations are performed by thread warps, avoiding bank conflicts as described in § 7.1.1, before a block-wide scan combines the partial results from each warp. Finally, these results are combined across all blocks to produces the completed scan.

7.2 Optimising the Padé Approximation

When computing the numerator $p_{n,n}$ (and similarly for the denominator $q_{n,n}$) associated to the diagonal Padé approximant $r_{n,n}$, evaluation is required of polynomials of the form

$$p_{n,n}(A) = c_0 I + c_1 A + c_2 A^2 + c_3 A^3 \cdots + c_n A^n. \tag{7}$$

Computing each power of the matrix A individually requires storing all intermediate powers up to A^n, which can lead to significant memory overheads that scale proportionally with the size of matrix used. This can be become a bottleneck for large scale problems. In order to mitigate this issue, methods have looked into reducing the number of matrices stored by reusing previously computed powers through efficient recursions. One such approach is Horner's Method.

7.2.1 Horner's Method

Horner's method factors polynomials into a nested form, allowing sequential computation of the powers of a matrix, storing only a single temporary matrix variable. For the numerator (and denominator) used in Padé Approximations such as in Equation 7, this yields the form

$$p_{n,n} = c_0 I + A\left(c_1 I + A\left(c_2 I + \cdots + A\left(c_{n-1} I + c_n A\right)\right)\right).$$

The method begins by starting with the highest degree coefficient multiplied by the identity matrix, setting result $= c_n I$. Terms are then sequentially evaluated through the calculation result $= A \cdot$ result $+ c_{n-1} I$, repeating the iteration until all terms are processed. In this way, only one matrix is stored to hold the running result, whilst the number of multiplications remains the same as in the naive method.

Whilst Horner's method reduces the issue of memory overhead, it does not optimise the number of multiplication steps, as it performs them sequentially without reusing intermediate calculations. The Paterson-Stockmeyer (PS) method [49] addresses this limitation, reducing the total number of matrix multiplications required to evaluate the required polynomial. Modern frameworks leverage this method to evaluate both Padé and Taylor Series approximants in modern implementations of the scaling and squaring method ([14], [52]).

7.2.2 Paterson-Stockmeyer Method

In this section, we follow the framework provided by Fasi and Higham [14]. Suppose we have a polynomial $p(X) = \sum_{i=0}^{k} \alpha_i X^i$. This can be rewritten as $p(X) = \sum_{i=0}^{\mu} B_i(X) X^{\nu i}$, where the block size $\nu \leq k$ is a positive integer, $\mu = \lfloor k/\nu \rfloor$ and

$$B_i(X) = \begin{cases} \alpha_{\nu i+\nu-1} X^{\nu-1} + \cdots + \alpha_{\nu i+1} X + \alpha_{\nu i} I, & i = 0, \ldots, \mu - 1, \\ \alpha_k X^{k-\nu\mu} + \cdots + \alpha_{\nu\mu+1} X + \alpha_{\nu\mu} I, & i = \mu, \end{cases}$$

To understand this more clearly, the following example is provided:

Example 7.5. Suppose $\nu = 3$ and that we have the polynomial

$$p(X) = \alpha_0 + \alpha_1 X + \alpha_2 X^2 + \alpha_3 X^3 + \alpha_4 X^4 + \alpha_5 X^5.$$

Then, we have $\mu = \lfloor 5/3 \rfloor = 1$, meaning we have two blocks, corresponding to $i = 0, 1$. Then, we can rewrite the polynomial as

$$p(X) = B_0(X) + B_1(X) X^3 = (\alpha_0 + \alpha_1 X + \alpha_2 X^2) + (\alpha_3 + \alpha_4 X + \alpha_5 X^2) X^3.$$

When diagonal Padé approximants are used, the method can lead to significant reductions in the number of matrix multiplications used by exploiting the identity $p_{n,n}(A) = q_{n,n}(-A)$. In this way, $p_{n,n}(A)$ can be rewritten as

$$p_{n,n}(A) = \sum_{i=0}^{n} \alpha_i A^i = \sum_{i=0}^{\lfloor n/2 \rfloor} \alpha_{2i} A^{2i} + A \sum_{i=0}^{\lceil n/2-1 \rceil} \alpha_{2i+1} A^{2i} =: U_e + U_o,$$

where U_e and U_o represent the even and odd terms of the sum respectively. The same process can be applied to obtain $q_{n,n}(A) = U_e - U_o$. With this method, one and $\nu - 1$ multiplications are required to obtain A^2 and the first ν powers of A^2 respectively. One further multiplication by A is needed to obtain the term U_o. Thus, it can be shown that taking $\nu = \lfloor \sqrt{n - 1/2} \rfloor$ or $\nu = \lceil \sqrt{n - 1/2} \rceil$ minimises the total number of matrix multiplications required for diagonal Padé approximants [14].

7.2.3 Inverting $q_{n,n}$

After $p_{n,n}(A)$ and $q_{n,n}(A)$ are obtained, the output $r_{n,n}(A) = [q_{n,n}(A)]^{-1}p_{n,n}(A)$, is evaluated. In practice, solving the linear system $q_{n,n}(A)r_{n,n}(A) = p_{n,n}(A)$ is preferred over direct inversion. This is due to inversion being more computationally expensive and numerically unstable for ill-conditioned matrices. Modern frameworks such as JAX and MATLAB solve this linear system using LU decomposition with partial pivoting to ensure that the largest available pivot is selected at each elimination step, reducing the magnitude of the multipliers to prevent error amplification. The below example is provided to visualise how LU decomposition of $q_{n,n}(A)$ is used to obtain the Padé Approximant $r_{n,n}(A)$:

Example 7.6 (Evaluating $r_{n,n}$)**.** Suppose we have

$$p_{n,n}(A) = \begin{bmatrix} 12 & 6 \\ 4 & 9 \end{bmatrix} \quad \text{and} \quad q_{n,n}(A) = \begin{bmatrix} 6 & 2 \\ 1 & 4 \end{bmatrix}.$$

We begin by obtaining the LU decomposition of $q_{n,n}(A)$. As the first row already contains the largest leading entry, no pivoting is required. Thus, performing forward elimination:

$$\text{Row } 2 \to \text{Row } 2 - \frac{1}{6} \times \text{Row } 1$$

yielding

$$L_q = \begin{bmatrix} 1 & 0 \\ \frac{1}{6} & 1 \end{bmatrix}, \quad U_q = \begin{bmatrix} 6 & 2 \\ 0 & \frac{11}{3} \end{bmatrix}.$$

We can now solve

$$L_q U_q r_{n,n}(A) = p_{n,n}(A)$$

First, we solve $L_q Y = p_{n,n}(A)$ for Y:

$$\begin{bmatrix} 1 & 0 \\ \frac{1}{6} & 1 \end{bmatrix} Y = \begin{bmatrix} 12 & 6 \\ 4 & 9 \end{bmatrix}$$

Forward substitution yields $Y = \begin{bmatrix} 12 & 6 \\ 2 & 8 \end{bmatrix}$.
Next, we solve $U_q r_{n,n}(A) = Y$:

$$\begin{bmatrix} 6 & 2 \\ 0 & \frac{11}{3} \end{bmatrix} r_{n,n}(A) = \begin{bmatrix} 12 & 6 \\ 2 & 8 \end{bmatrix}$$

Backward substitution gives the final result:

$$r_{n,n}(A) = \begin{bmatrix} \frac{20}{11} & -\frac{3}{11} \\ \frac{6}{11} & \frac{24}{11} \end{bmatrix}.$$

7.3 Adaptive Step Sizing

When evaluating outputs across several time steps in LNCDEs, the time intervals used to update the hidden state in previous works are dictated solely by the frequency of the discrete observations [57]. Whilst such observation-driven methods work well for smoothly varying paths, issues can arise when the underlying system exhibits stiff dynamics. In stiff systems, the state transition matrices can have eigenvalues with vastly different magnitudes, requiring smaller step sizes to maintain numerical stability of the matrix exponential computation, regardless of the input sampling rate. Furthermore, irregular sampling typical of medical data or sensor networks can create large intervals where approximating the matrix exponential accurately becomes challenging. As a result, accuracy can be lost when observation-controlled step sizes are too large relative to the numerical requirements of the matrix exponential evaluation.

A key advantage of linear systems is that matrix exponentials satisfy the time additivity property, meaning that for a constant matrix M,

$$\exp(tM) = \exp(t_1 M)\exp(t_2 M)$$

where $t_1 + t_2 = t$. This property enables exact decomposition of large time intervals into smaller sub-intervals, thereby yielding more stable and accurate numerical evaluations.

When selecting optimal time intervals to use for these evaluations, traditional step-size controllers operate under the assumption that computational cost decreases monotonically with step size, as larger steps require fewer evaluations over a fixed integration interval. This assumption holds for explicit numerical methods, such as Runge-Kutta schemes, where the computational work per step remains constant regardless of step size. However, the assumption becomes invalid for iterative matrix exponential approximation methods, including the scaling and squaring approach we have studied [13]. For this method, the computational cost exhibits a dependence on step size through the spectral properties of the scaled matrix, due to the number of scaling and squaring steps required for matrices with larger norm.

As a response to this problem, adaptive step size methods have been developed that dynamically adjust time intervals based on local system dynamics, ensuring both computational efficiency in smooth regions and numerical accuracy where rapid changes or stiff behaviour demand finer discretisation. For linear systems solved via exponentials, this adaptation can be implemented using Magnus-based exponential integrators, which preserve the structure of the flow while allowing local error estimation.

7.3.1 Magnus Expansion for Linear ODEs

Consider a non-autonomous linear ODE

$$\frac{d}{dt}h(t) = A(t)\,h(t), \quad h(t_0) = h_0,$$

where $A(t) \in \mathbb{R}^{n\times n}$ is a time-dependent system matrix. The Magnus expansion [35] expresses the solution in the form

$$h(t) = \exp\big(\Omega(t, t_0)\big)\, h_0,$$

where $\Omega(t, t_0)$ is given by a convergent infinite series involving integrals and nested commutators of $A(t)$:

$$\begin{aligned}\Omega(t, t_0) = &\int_{t_0}^{t} A(\tau_1)\, d\tau_1 + \frac{1}{2}\int_{t_0}^{t}\int_{t_0}^{\tau_1} [A(\tau_1), A(\tau_2)]\, d\tau_2\, d\tau_1 \\ &+ \frac{1}{6}\int_{t_0}^{t}\int_{t_0}^{\tau_1}\int_{t_0}^{\tau_2} \Big([A(\tau_1), [A(\tau_2), A(\tau_3)]] + [A(\tau_3), [A(\tau_2), A(\tau_1)]]\Big)\, d\tau_3\, d\tau_2\, d\tau_1 \\ &+ \cdots\end{aligned}$$

where the matrix commutator is defined as $[X, Y] = XY - YX$.

The implementation of Magnus expansion methods requires three key stages: series truncation to a computationally feasible order, numerical approximation of the integrals within the truncated expansion, and evaluation of the resulting matrix exponential [5]. Since computing many terms in the series becomes computationally expensive, truncations are typically applied after a small number of terms. The most commonly used truncation retains only the first term and approximates the integral with the midpoint rule, yielding the second-order Magnus method (Magnus–2):

$$\Omega^{[2]}(t+\Delta t, t) = \Delta t\, A\left(t + \frac{\Delta t}{2}\right), \tag{8}$$

so that the solution update becomes

$$h(t+\Delta t) = \exp\left(\Delta t\, A(t + \tfrac{\Delta t}{2})\right) h(t).$$

This method is second-order accurate [5] and requires only a single evaluation of A at the midpoint of the interval. Higher-order Magnus integrators (Magnus–4, Magnus–6, etc.) incorporate additional commutator terms using Gaussian quadrature rules for evaluating time integrals. While these higher-order methods involve extra evaluations of $A(t)$ and explicit computation of commutators per step, they enable larger step sizes for a given accuracy and can include embedded formulas for adaptive step size control. Consequently, although a single step is more expensive, the total computational cost to achieve a desired accuracy can be lower than using multiple small steps with a lower-order method.

This computational trade-off becomes particularly relevant in sequence models, where hidden states must be updated across multiple time intervals. For such updates, the Magnus expansion can be unrolled to yield:

$$\begin{aligned} h(t_n) &= \exp\big(\Omega(t_n, t_{n-1})\big) h_{n-1} \\ &= \exp\big(\Omega(t_n, t_{n-1})\big) \exp\big(\Omega(t_{n-1}, t_{n-2})\big) \dots \exp\big(\Omega(t_1, t_0)\big) h_0. \end{aligned}$$

The central challenge thus becomes selecting optimal time intervals that align with the system's underlying dynamics.

7.3.2 Two Adaptive Magnus Methods

We now describe two adaptive step sizing strategies that build on the Magnus expansion to adjust the step size dynamically: a straightforward step-doubling method, and an embedded Magnus approach.

Step-Doubling Method Our first method uses the second order Magnus-2 integrator and estimates the local truncation error by comparing one full step versus two half-steps.

The full step of size Δt is computed as

$$h^{\text{coarse}}_{t_{n+1}} = \exp\left(\Delta t\, A\big(t_n + \tfrac{\Delta t}{2}\big)\right) h_{t_n},$$

and the two half-steps of size $\Delta t/2$ as

$$h_{\text{half}} = \exp\left(\tfrac{\Delta t}{2}\, A(t_n + \tfrac{\Delta t}{4})\right) h_{t_n}, \quad h^{\text{fine}}_{t_{n+1}} = \exp\left(\tfrac{\Delta t}{2}\, A(t_n + \tfrac{3\Delta t}{4})\right) h_{\text{half}},$$

where $t_{n+1} = t_n + \Delta t$.

Following the framework used by Hairer et Al. [23], the step is accepted if

$$|h^{\text{fine}}_{t_{n+1},i} - h^{\text{coarse}}_{t_{n+1},i}| \le \text{sc}_i, \quad \text{where} \quad \text{sc}_i = \text{Atol}_i + \max\left(|h_{t_n,i}|, |h^{\text{fine}}_{t_{n+1},i}|\right) \cdot \text{Rtol}_i,$$

for each component i. Furthermore, the local error can estimated using:

$$e \approx \frac{h^{\text{fine}}_{t_{n+1}} - h^{\text{coarse}}_{t_{n+1}}}{\text{sc}}.$$

If the step is accepted, the solution is updated as $h_{t_{n+1}} = h_{\text{fine}}$, and the next step size is chosen by multiplying the current step size Δt by a factor

$$\text{factor} = \text{safety} \cdot \left(\frac{1}{\|e\|} \right)^{1/(p+1)}, \tag{9}$$

where $p = 2$ is the order of Magnus method used. This factor is restricted to lie within a given range (for example between 0.2 and 5), to prevent the step size from increasing or decreasing too rapidly. If the step is rejected, h_n is retained, and the step is retried using the newly computed step size.

This step-doubling approach is conceptually related to Richardson extrapolation (Appendix D), which removes the leading-order local truncation error by combining solutions computed at different step sizes. While we do not apply the correction formula, the difference between the coarse and fine updates provides a proportional error estimate for adaptive step-size control.

Embedded Magnus Method: Our second method uses a fourth-order Magnus integrator for the main update, while embedding a second-order Magnus solution for local error estimation, resulting in only two evaluations of $A(t)$ per step, compared to three for the step-doubling method.

For a step of size Δt from t_n, define the 2-point Gauss-Legendre nodes:

$$t_{n,\pm} = t_n + \frac{\Delta t}{2} \left(1 \pm \frac{1}{\sqrt{3}} \right)$$

and evaluate

$$A_\pm = A(t_{n,\pm}).$$

The second-order Magnus term is obtained from Equation 8, and the fourth-order Magnus term (fully derived in Appendix E.2) is [5]:

$$\Omega^{[4]} = \Omega^{[2]} - \frac{\sqrt{3}\Delta t^2}{12} [A_-, A_+].$$

Similarly to the first method, the error is checked component-wise against tolerances:

$$|h^{\text{high}}_{t_{n+1}} - h^{\text{low}}_{t_{n+1}}| \leq sc_i = \text{Atol}_i + \max\left(|h_{t_n,i}|, |h^{\text{high}}_{t_{n+1},i}| \right) \cdot \text{Rtol}_i$$

where the local error is estimated as the scaled difference between the high and low order solutions:

$$e = \frac{h^{\text{high}}_{t_{n+1}} - h^{\text{low}}_{t_{n+1}}}{\text{sc}} = \frac{\exp(\Omega^{[4]})h_{t_n} - \exp(\Omega^{[2]})h_{t_n}}{\text{sc}},$$

and the solution updated as $h_{t_{n+1}} = h^{\text{high}}_{t_{n+1}}$ if the step is accepted. The next step size is adapted using the same factor as in Equation 9, using $p = 4$ rather than $p = 2$. The newly computed step size is used in the same manner as the previous method when step size is rejected.

7.3.3 Starting Step Size

While we have now explored two methods for adapting step size, we still need to determine the initial step size. A poor initial guess can lead to excessive rejections or unnecessarily small steps, reducing computational efficiency. To address this, strategies have been developed to estimate a reasonable starting step size based on local properties of the system. We adopt an approach introduced by Gladwell et al. [18] (reformulated in [23]), which uses approximations of the first and second derivatives of the solution:

Let p denote the order of the Magnus method. We begin by setting

$$d_0 = \|h_{t_0}\|, \quad d_1 = \|f_{t_0}\| = \|A(t_0)h_{t_0}\|,$$

As a first guess, we set

$$\tilde{\Delta}t = 0.01 \cdot \frac{d_0}{d_1},$$

or $h_0 = 10^{-6}$ if d_0 or $d_1 < 10^{-5}$. Next, perform one Euler step, $h_{t_1} = h_{t_0} + \tilde{\Delta}tA(t_0)h_{t_0}$ and compute $f_{t_1} = A(t_0 + \tilde{\Delta}t)h_{t_1}$. We can now estimate the second derivative:

$$d_2 = \frac{\|f_1 - f_0\|}{\tilde{\Delta}t}.$$

and obtain a second guess

$$\hat{\Delta}t = \left(\frac{0.01}{\max(d_1, d_2)}\right)^{1/(p+1)},$$

taking $h_1 = \max(10^{-6}, h_0 \cdot 10^{-3})$ if $\max(d_1, d_2) \leq 10^{-15}$. Finally, we take the starting step size as

$$\Delta t = \min(100 \cdot \tilde{\Delta}t, \hat{\Delta}t).$$

This procedure can be applied directly to both the step-doubling Magnus-2 method and the embedded Magnus-4 method, varying only the order of p used.

7.3.4 Combining Magnus Expansion with Scaling and Squaring

For our LNCDE use case, the adaptive Magnus integrators described above can be combined with the Padé approximation scaling and squaring method. This is applied to the path-dependent, weighted state transition matrix, defined piecewise over each observation interval as

$$G(t) = \sum_{i=1}^{d_\omega} \frac{\omega^i_{j+1} - \omega^i_j}{t_{j+1} - t_j} A^i \quad \text{for } t \in [t_j, t_{j+1}]$$

over all observation intervals. The Magnus integrator of desired order can then be obtained from the infinite Magnus series for $G(t)$, before the scaling and squaring steps are applied to ensure numerical stability for large norms.

To incorporate scaling and squaring into the two adaptive Magnus methods described in § 7.3.2, an extra step is added before evaluating the matrix exponentials during the hidden state update. Although it may initially appear that this would significantly increase the computational cost, since the step-doubling method requires exponentials for three matrices (one full step and two half-steps) and the embedded method requires exponentials for two matrices, this overhead can be reduced by choosing the same scaling factor for all matrices in a given step, meaning the scaling and squaring steps can be computed in parallel on the GPU.

8 Experiments

8.1 Matrix Exponential Algorithms

Having developed the necessary components for evaluating linear CDEs, we now turn to performing numerical experiments. In particular, we compare different evaluation methods for linear CDEs in order to assess their accuracy and efficiency.

As explored in § 2.3.2, the key operation required to evaluate a sequence model at different time steps is

$$\tilde{h}_{t_{j+1}} = \exp\left(\sum_{i=1}^{d_\omega}\left(\omega^i_{t_{j+1}} - \omega^i_{t_j}\right)A^i\right)\tilde{h}_{t_j}. \tag{10}$$

The focus of our experiments therefore lies in evaluating the efficiency and accuracy of different matrix exponential methods when applied to the linear CDE reformulation of SSMs.

In our first experiment, we begin by constructing the collection of A^i system matrices that govern the evolution of the hidden state. For simplicity, we generate these to be dense with entries drawn from a Gaussian distribution and scaled by the hidden dimension, taking $d_h = d_\omega = 32$ as used by Walker et Al. [57] in their dense experiments. To simulate the types of input signals encountered in practice, we generate sample Brownian motion paths under an irregular sampling scheme. We start with a uniform time grid on the interval $[0, 60]$ where the number of grid points is treated as an experiment variable. Small random perturbations are then added to the intermediate grid points, and the resulting times are sorted to produce irregularly spaced observation points, before generating Brownian increments $\Delta\omega \sim \mathcal{N}(0,\, \Delta t\, I_{d_\omega})$ to dictate the evolution of the path over the interval. This approach mirrors the linear CDE setup from § 2.3.2 where measurements are collected at irregular intervals and continuous paths are reconstructed through linear interpolation between data points. The different matrix exponential methods are then used to calculate the exact solution over each time interval, and associative parallel scans are used to parallelise the computations over multiple time steps.

The methods evaluated include the Padé approximation with and without scaling and squaring, the truncated Taylor series expansion, the Runge–Kutta method, and Euler's method. For reference, we treat Euler's method with a very large number of sub-steps (100,000 per interval)

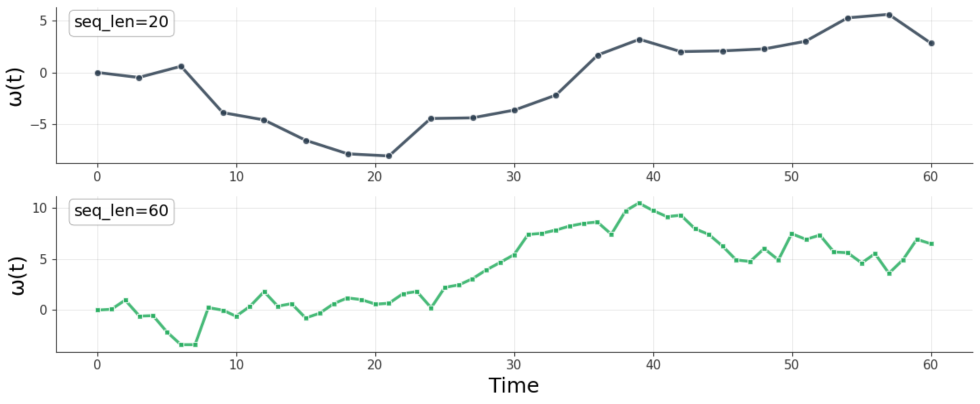

Figure 9: Visualisation of two randomly generated paths using sequence length of 20 (top) and 60 (bottom). A higher sequence length corresponds to more subintervals over the whole interval $[0, 60]$.

Table 7: Average and standard deviation of relative error to Euler method with $100,000$ sub-steps per interval across sequence lengths for different matrix exponential methods across 10 runs.

Method	**5**	**10**	**20**	**30**	**60**	**100**
Series						
S&S 3	1.18e-02 ± 5.70e-03	9.61e-03 ± 4.34e-03	8.26e-03 ± 4.04e-03	6.70e-03 ± 1.60e-03	5.77e-03 ± 2.22e-03	7.53e-03 ± 2.54e-03
S&S 5	1.18e-02 ± 5.70e-03	9.61e-03 ± 4.34e-03	8.26e-03 ± 4.04e-03	6.70e-03 ± 1.60e-03	5.77e-03 ± 2.22e-03	7.53e-03 ± 2.54e-03
S&S 7	1.18e-02 ± 5.70e-03	9.61e-03 ± 4.34e-03	8.26e-03 ± 4.04e-03	6.70e-03 ± 1.60e-03	5.77e-03 ± 2.22e-03	7.53e-03 ± 2.54e-03
S&S 9	1.18e-02 ± 5.70e-03	9.61e-03 ± 4.34e-03	8.26e-03 ± 4.04e-03	6.70e-03 ± 1.60e-03	5.77e-03 ± 2.22e-03	7.53e-03 ± 2.54e-03
S&S 13	1.18e-02 ± 5.70e-03	9.61e-03 ± 4.34e-03	8.26e-03 ± 4.04e-03	6.70e-03 ± 1.60e-03	5.77e-03 ± 2.22e-03	7.53e-03 ± 2.54e-03
Padé 3	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	9.76e-01 ± 6.92e-02	inf ± nan
Padé 5	1.00e+00 ± 7.02e-17	1.00e+00 ± 0.00e+00	1.00e+00 ± 4.70e-11	1.20e+00 ± 5.16e-01	5.56e+03 ± 1.57e+04	5.95e-01 ± 5.44e-01
Padé 7	1.00e+00 ± 1.17e-13	1.00e+00 ± 1.56e-10	4.46e+00 ± 9.78e+00	3.24e+01 ± 5.20e+01	5.21e-02 ± 5.55e-02	1.25e-02 ± 1.31e-02
Padé 9	1.00e+00 ± 1.01e-09	1.05e+00 ± 1.38e-01	2.69e+00 ± 3.88e+00	4.23e-02 ± 4.58e-02	5.87e-03 ± 2.19e-03	7.54e-03 ± 2.54e-03
Padé 13	9.98e-01 ± 2.69e-02	6.82e-01 ± 1.55e+00	8.33e-03 ± 4.03e-03	6.70e-03 ± 1.60e-03	5.77e-03 ± 2.22e-03	7.53e-03 ± 2.54e-03
Taylor 1	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00
Taylor 2	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00
Taylor 5	1.00e+00 ± 9.40e-12	1.00e+00 ± 1.11e-12	1.00e+00 ± 4.26e-11	1.00e+00 ± 1.04e-09	1.00e+00 ± 9.33e-03	1.74e+00 ± 2.24e+00
Taylor 10	1.01e+00 ± 1.05e-02	9.85e+00 ± 2.65e+01	5.59e+01 ± 1.42e+02	2.28e+00 ± 1.02e+00	4.46e+00 ± 7.78e+00	6.12e-01 ± 6.66e-01
Taylor 20	1.41e+03 ± 4.17e+03	2.15e+03 ± 6.44e+03	8.46e-01 ± 1.53e+00	3.30e-02 ± 2.95e-02	5.52e-03 ± 2.21e-03	7.51e-03 ± 2.56e-03
Taylor 26	2.51e+03 ± 7.53e+03	3.27e+01 ± 9.71e+01	3.42e-02 ± 8.24e-02	6.90e-03 ± 1.48e-03	5.77e-03 ± 2.22e-03	7.53e-03 ± 2.54e-03
Numerical						
RK4 10	1.25e+00 ± 7.69e-01	7.12e-01 ± 1.20e-01	5.67e-01 ± 6.31e-01	1.50e-01 ± 8.87e-02	9.13e-02 ± 6.12e-02	3.57e-02 ± 2.97e-02
RK4 100	1.01e-02 ± 3.65e-03	9.32e-03 ± 4.35e-03	8.12e-03 ± 3.92e-03	6.67e-03 ± 1.59e-03	5.76e-03 ± 2.21e-03	7.53e-03 ± 2.54e-03
Euler 10	1.00e+00 ± 4.17e-13	1.00e+00 ± 3.51e-16	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00	1.00e+00 ± 0.00e+00
Euler 100	9.91e-01 ± 2.18e-02	9.95e-01 ± 1.62e-02	9.79e-01 ± 5.88e-02	1.00e+00 ± 1.42e-03	9.97e-01 ± 5.34e-03	9.96e-01 ± 9.87e-03

as the ground truth solution. The relative errors are computed as:

$$\text{Relative Error} = \frac{\|h_{\text{method}} - h_{\text{Euler}}\|_2}{\|h_{\text{Euler}}\|_2},$$

and are shown in Table 7 and the varying computation time in Figure 10, where S&S n represents the scaling and squaring algorithm with the $[n, n]$ diagonal approximation, Padé n the pure Padé method (without scaling and squaring) with the $[n, n]$ diagonal approximation, Taylor n the n-th order Taylor expansion, and the number accompanying the numerical methods dictates the number of sub-steps per interval.

The results demonstrate the advantages of the scaling and squaring method with Padé approximation for series methods, with all five methods outperforming the Taylor series and pure Padé methods on the chosen time intervals. As expected, the Taylor and pure Padé methods of low order show consistently poor accuracy across all sequence lengths, with relative errors around 1 that make them unsuitable for practical applications. As the sequence length increases, the associated time increments diminish, thereby improving the performance of higher-order methods. This effect is likely to arise due to shorter increments yielding reduced norms in the path-weighted summations of the system matrices in Equation 10, improving stability for the methods. The Runge-Kutta and Euler methods are used in comparison to demonstrate the rate of convergence of numerical methods with RK4 100 performing exceptionally well across all sequence lengths.

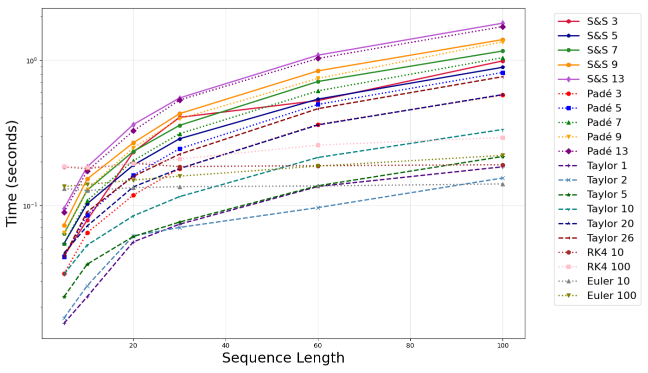

Figure 10: Plot of sequence length vs time for different matrix exponential evaluation methods across 10 runs.

The time plots illustrate the expected scaling of computational cost with sequence length across the different methods. In particular, the scaling-and-squaring approach incurs higher runtimes, as anticipated, due to the additional overhead introduced by the repeated scaling operations. The relatively flat timing curves for numerical methods result from JAX's `lax.scan` implementation, which sequentially processes operations with efficient compiled code, making per-operation costs negligible compared to compilation overhead. Overall, the runtimes exhibit the expected behaviour, with an increasing Taylor and Padé order resulting in a larger number of matrix multiplications, consistent with the higher observed runtimes.

8.1.1 Effect of Norm on Series Methods

As earlier mentioned, numerical roundoff errors and computational complexity increase with the magnitude of the generator matrix

$$\sum_{i=1}^{d_\omega} \left(\omega^i_{t_{j+1}} - \omega^i_{t_j} \right) A^i.$$

In order to isolate the effect of the norm of this exponent on the accuracy of the approximation, independent of variations in the underlying experimental setup, we introduce a scaling factor that directly controls its magnitude. By systematically adjusting this factor, we are able to investigate how the size of the exponent influences the resulting relative error. The scaling factors are generated on a logarithmic scale ranging from 10^{-7} to 1, with twenty evenly spaced values in the log space and sequence length is fixed to 10. The 1-norm is taken for all 10 evaluations, corresponding to the intervals determined by sequence length, and averaged to obtain a measure of norm size.

The results demonstrate distinct performance regimes across generator norm magnitudes. At norms smaller than 10^{-3}, all methods achieve high levels of accuracy, meaning computational efficiency can be prioritised without having to sacrifice accuracy. For norms greater than 1, Padé approximants with scaling and squaring demonstrate their theoretical advantage over the other methods. All scaling and squaring methods sustain reasonable accuracy even at large generator norms, whereas Taylor and pure Padé methods suffer catastrophic degradation regardless of their order. Notably, Taylor 26 and Padé 13 perform worse than low-order scaling and squaring approximants, illustrating that polynomial degree alone cannot overcome inherent stability limitations. Overall, the relative error of all methods increases with generator norm as expected.

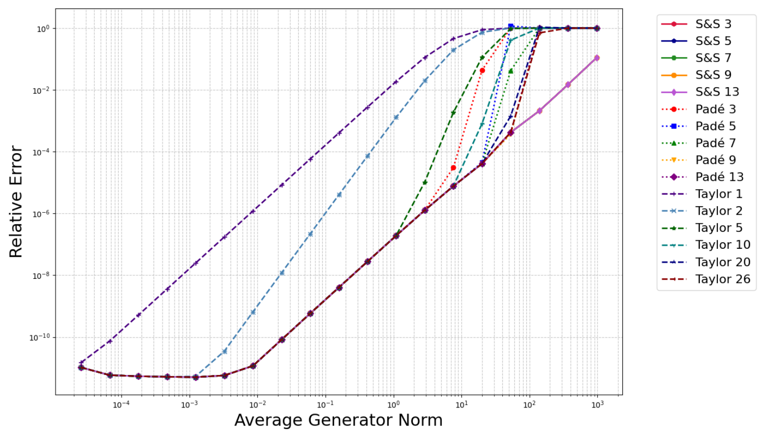

Figure 11: Plot of average generator norm across all timesteps vs relative error for series matrix exponential evaluation methods across 10 runs.

Table 8: Runtime (ms) comparison of Horner and Patterson-Stockmeyer methods varying matrix size from 16×16 to 256×256 on Padé $[13, 13]$ approximation.

Method	**16 × 16**	**32 × 32**	**64 × 64**	**128 × 128**	**256 × 256**	**Speedup**
Naive Padé	9.92±1.05	9.95±1.06	10.23±1.20	10.14±1.04	10.22±1.05	1.00x
Horner	8.74±1.10	8.76±0.96	8.95±1.07	8.96±1.03	8.95±0.99	1.13x
PS $\nu = 3$	8.17±1.00	8.33±1.01	8.41±1.04	8.35±0.93	8.55±1.03	1.19x
PS $\nu = 4$	**8.08±1.05**	**8.05±0.94**	**8.21±1.02**	**8.23±0.97**	**8.36±1.01**	**1.22x**

8.2 Optimising the Padé Approximation

As discussed in § 7.2, certain frameworks attempt to minimise the number of matrix multiplications that occur in the evaluation of the Padé approximation, with the aim of increasing computational efficiency. We compare the use of the Horner and Patterson-Stockmeyer method in this section. All implementations from this point onwards will use the diagonal Padé $r_{13,13}$ approximation due its common practical adoption.

Our Horner method implementation evaluates polynomials in A^2 rather than successive powers of A, allowing us to apply Horner's rule separately to the even and odd coefficients to obtain U_e, U_o before making use of the identity:

$$p_{n,n}(A) = \sum_{i=0}^{n} \alpha_i A^i = \sum_{i=0}^{\lfloor n/2 \rfloor} \alpha_{2i} A^{2i} + A \sum_{i=0}^{\lceil n/2-1 \rceil} \alpha_{2i+1} A^{2i} =: U_e + U_o,$$

where $q_{n,n}(A) = U_e - U_o$. Two variations of the Paterson-Stockmeyer method are implemented with different blocking parameters, dictated by the two optimal choices for ν, namely $\nu = \lfloor \sqrt{n - 1/2} \rfloor = 3$ and $\nu = \lceil \sqrt{n - 1/2} \rceil = 4$. Following Fasi and Highams' method in [14], this results in $\mu = 2$ (3 blocks) and $\mu = 1$ (2 blocks) respectively. JAX's `jnp.linalg.solve` is then used to solve the linear system $q_{n,n}(A) r_{n,n}(A) = p_{n,n}(A)$, ensuring consistency across the methods. The naive implementation taken as benchmark follows the straightforward approach of directly computing successive matrix powers from the Padé approximation definition.

Performing $5,000$ runs on randomly generated matrices with regular, stiff and skew-symmetric structures, Table 8 demonstrates the speed-up obtained using both methods. The Patterson-Stockmeyer method with $\nu = 4$ performs best, obtaining 22% acceleration over the naive implementation. Using the parameter $\nu = 3$ gives a slightly lower speed-up at 19%, with Horner's method outperforming by only 13%.

8.3 Adaptive Step Sizing

Given the clear superiority of the scaling and squaring method with Padé approximation over alternative approaches, our investigation can be extended to examining how adaptive step size control can be employed when facing stiff underlying dynamics.

To establish a fair comparison between adaptive and fixed step size methods, consider the practical scenario where high-frequency data results in a very large number of observations, creating many piecewise linear segments following the interpolation step. In such cases, evaluating the matrix exponential separately for each individual interval becomes computationally prohibitive due to the volume of exponential calculations required. Under these conditions, fixed step methods can span multiple intervals within a single step, substantially decreasing the number of matrix exponential evaluations required. This computational approach can then be directly

Table 9: Comparison of fixed step and adaptive methods with stiff underlying system dynamics over 10 runs

Stiffness Factor = 1

dt_fixed	**Method**	**Rel Error**	**Time (s)**
1	Fixed step	3.97e-4 ± 7.02e-5	2.3e-1 ± 1.3e-2
2	Fixed step	5.54e-4 ± 6.29e-5	3.3e-1 ± 4.5e-1
5	Fixed step	8.72e-4 ± 9.74e-5	1.6e-1 ± 4.8e-2
10	Fixed step	1.09e-3 ± 2.04e-4	1.35e-1 ± 1.6e-2
Adaptive	Adaptive step doubling	3.51e-2 ± 1.26e-2	7.79e1 ± 3.78e0
	Embedded Magnus	2.00e-1 ± 6.74e-2	2.00e0 ± 1.6e-1

Stiffness Factor = 5

dt_fixed	**Method**	**Rel Error**	**Time (s)**
1	Fixed step	9.73e-3 ± 1.41e-3	2.26e-1 ± 4.0e-3
2	Fixed step	1.34e-2 ± 1.60e-3	1.78e-1 ± 7.2e-3
5	Fixed step	2.15e-2 ± 2.59e-3	1.47e-1 ± 1.0e-2
10	Fixed step	2.66e-2 ± 3.84e-3	1.29e-1 ± 3.0e-3
Adaptive	Adaptive step doubling	1.57e-1 ± 5.46e-2	8.25e1 ± 5.44e0
	Embedded Magnus	1.78e-1 ± 5.95e-2	1.51e1 ± 1.71e0

Stiffness Factor = 10

dt_fixed	**Method**	**Rel Error**	**Time (s)**
1	Fixed step	3.77e-2 ± 5.61e-3	2.33e-1 ± 1.0e-2
2	Fixed step	5.16e-2 ± 7.62e-3	1.79e-1 ± 4.1e-3
5	Fixed step	8.37e-2 ± 1.26e-2	1.49e-1 ± 1.1e-2
10	Fixed step	1.02e-1 ± 1.07e-2	1.26e-1 ± 3.9e-3
Adaptive	Adaptive step doubling	3.07e-1 ± 2.45e-1	9.34e1 ± 4.62e0
	Embedded Magnus	2.22e-1 ± 7.99e-2	5.73e1 ± 4.03e0

Stiffness Factor = 20

dt_fixed	**Method**	**Rel Error**	**Time (s)**
1	Fixed step	1.49e-1 ± 3.87e-2	2.46e-1 ± 1.6e-2
2	Fixed step	1.95e-1 ± 3.67e-2	1.85e-1 ± 7.5e-3
5	Fixed step	3.14e-1 ± 6.15e-2	1.48e-1 ± 3.3e-3
10	Fixed step	3.69e-1 ± 5.35e-2	1.33e-1 ± 4.0e-3
Adaptive	Adaptive step doubling	6.00e-1 ± 2.63e-1	1.05e2 ± 7.78e0
	Embedded Magnus	3.41e-1 ± 6.41e-2	7.60e1 ± 3.16e0

Stiffness Factor = 50

dt_fixed	**Method**	**Rel Error**	**Time (s)**
1	Fixed step	8.73e-1 ± 2.38e-1	4.73e-1 ± 7.17e-1
2	Fixed step	9.04e-1 ± 1.19e-1	3.20e-1 ± 4.33e-1
5	Fixed step	9.62e-1 ± 4.93e-2	2.84e-1 ± 4.21e-1
10	Fixed step	9.80e-1 ± 2.95e-2	1.33e-1 ± 2.17e-2
Adaptive	Adaptive step doubling	1.87e0 ± 1.59e0	1.24e2 ± 1.14e1
	Embedded Magnus	1.30e0 ± 1.38e0	8.54e1 ± 3.79e0

compared against adaptive methods to determine which strategy delivers superior performance.

Our final experiment compares the two adaptive step size methods introduced in § 7.3.2 with the starting step size from § 7.3.3 against a fixed step method, where scaling and squaring with Padé approximation is used as the exponential evaluation method and linear interpolations of the path are taken between points dictated by the fixed intervals. In this experiment, we use a similar setup to our first experiment, keeping $d_h = d_\omega = 32$ and generating sample paths in the same manner. We take the interval to be $[0, 10]$ for computational speed, and increase the number of generated observation points to 200. Due to the smaller intervals, we reduce the number of steps taken to compute the Euler ground truth to $1,000$ for experimental speed. We can then control the number of fixed intervals as an input parameter, where the total time domain $[0, 10]$ is partitioned into equally sized sub-intervals.

In order to generate an underlying system with stiff dynamics, the A_i matrices are constructed through eigenvalue decomposition, where each matrix is formed as $A_i = Q_i D_i Q_i^{-1}$ with Q_i being a random orthogonal matrix obtained via QR decomposition of a Gaussian random matrix and D_i containing eigenvalues that span linearly from the negative of stiffness factor to 0, where this factor is taken as an input variable. Rather than using purely diagonal matrices, this construction rotates the stiffness structure into different random orientations for each input channel, ensuring that each A_i matrix possesses the desired eigenvalue spectrum while having distinct eigenvector directions, creating a more realistic scenario where stiff dynamics are coupled across state variables in different ways for each input.

In this way, we vary the number of equally sized time intervals taken for the fixed method between 1 and 100, and vary the stiffness factor for all methods, using the values $1, 5, 10, 20$ and 50. The results are shown in Table 9. For any given stiffness factor, we can see that increasing step size for a constant number of data points results in monotonically increasing error for the fixed step methods. Whilst the adaptive methods show competitive accuracy, they fail to outperform the fixed step methods for any stiffness factor tested, with relative error increasing with stiffness. Furthermore, both adaptive schemes exhibit extremely high computational costs, with execution times several orders of magnitude above the fixed step methods for all stiffness factors. These results indicate that adaptive Magnus-type solvers offer little advantage over their fixed-step counterparts as currently implemented.

The limited effectiveness of adaptive step methods may stem from their inability to properly account for discontinuities that occur within the adaptive intervals themselves. For instance, consider an adaptive step spanning $[0, 1]$ where the path transitions from one linear segment to another at $t = 0.1$. When employing step-doubling techniques, the evaluations of the generator matrix occur at $t = 0.25$, $t = 0.5$, and $t = 0.75$, which all fall within the second interval. Since the generator remains constant across these evaluation points due to its piecewise continuity, the estimated local truncation error becomes zero, leading to automatic step acceptance despite the method neglecting the discontinuity from the initial segment. This introduces approximation errors that accumulate throughout the method, as the adaptive mechanism lacks the capability to handle the piecewise structure within individual time steps.

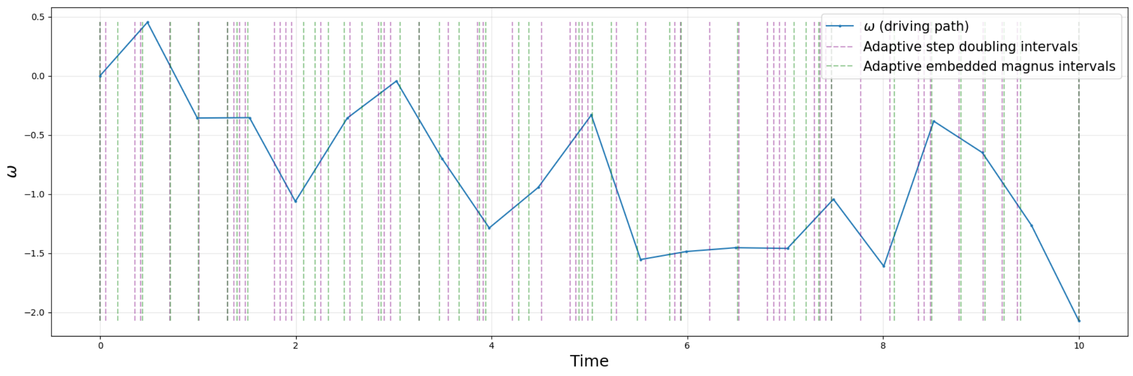

Figure 12: Plot of driving path and adaptive intervals for both adaptive methods with 20 observation points with stiffness value of 50.

9 Discussion

9.1 Project Overview

Our project examined how sequence models like LNCDEs can be evaluated efficiently by analysing the matrix exponential operations at their core. We decomposed these exponentials into their fundamental GEMM components and studied both the theory and their practical GPU implementations using tiling approaches, covering both standard dense matrices and structured forms. Building on this foundation, we investigated series methods for matrix exponential evaluation, with particular attention to their integration and optimisation within sequence models.

The key takeaway from this project is that the scaling and squaring with Padé approximation method obtains the best balance between accuracy and computational complexity out of the methods studied. Whilst Taylor series and pure Padé methods can perform competitively when exponentiating matrices with small norms, stability issues quickly arise when norm is increased.

While adaptive step sizing methods can enhance accuracy for systems with smooth dynamics, the LNCDE framework presents challenges due to discontinuities inherent in the linear interpolation between discrete data points. Notably, fixed step size methods achieved superior accuracy to adaptive approaches in all system scenarios even for large step sizes, while offering significantly superior computational efficiency.

9.2 Limitations & Future Work

Despite achieving superior theoretical complexity, the advanced matrix multiplication algorithms examined in § 3 remain impractical on modern processing units due to irregular memory access patterns and poor alignment with GPU architectures. The dominance of simpler tiling approaches reflects a fundamental disconnect between algorithmic complexity theory and the reality of modern hardware limitations, representing a significant research opportunity.

All experiments in this thesis use dense matrices by design, though structured alternatives could easily be substituted. The block-diagonal form is of particular interest due to its straightforward implementation, significant computational advantages (up to $45\times$) speed-up as shown in Appendix F.2.1, and crucially, is closed under matrix multiplication, unlike the Walsh-Hadamard and general sparse forms studied by Walker et al. [57]. More generally, sparse matrices present implementation challenges for LNCDE applications, as the absence of standardised benchmarks and the complexity of optimising sparse algorithms for the specific sparsity patterns arising from matrix exponentials make them less practically appealing. The development of efficient exponential methods for sparse forms represents an active research area with significant implications for computational efficiency across numerous domains.

Additionally, all experiments use FP64 precision to prevent round-off error accumulation across time steps, which disables Tensor Core acceleration. While this ensures numerical accuracy, it sacrifices the $9\times$ speed-up in Appendix F.1.1 representing a fundamental trade-off between accuracy and speed. However, even FP64 precision cannot eliminate catastrophic cancellation in cases where intermediate computations during matrix exponential evaluation have magnitudes orders of magnitude larger than the final result during the scaling and squaring process. Production implementations could achieve significantly faster performance through mixed-precision strategies that selectively use higher precision only where error propagation is critical.

A fundamental limitation of adaptive step size methods arises when discontinuities in the piecewise linear driving path occur within a proposed integration interval. The sequential nature of these algorithms, which rely on prior step sizes presents opportunities for future optimisation research.

10 Conclusion

This thesis has examined the computational foundations underlying efficient sequence model evaluation, with particular emphasis on Linear Neural Controlled Differential Equations and their dependence on matrix exponential computations. Through our comprehensive analysis of matrix multiplication algorithms, GPU optimisation techniques, and methods for computing matrix exponentials, we have established that the Padé approximation combined with the scaling and squaring method provides the best balance between numerical accuracy and computational efficiency of the studied methods for matrix exponential evaluation, consistently outperforming Taylor series and pure Padé methods of equivalent order. Further optimisations through Horner's method and the Paterson-Stockmeyer algorithm provided meaningful computational improvements, achieving up to 22% speed-up in polynomial evaluation. Additionally, we demonstrated that adaptive step sizing methods can be used for the evaluation of LNCDEs. However, their high computational cost currently limits practical use, and in their present form they failed to outperform fixed-step approaches.

Use of Artificial Intelligence: I acknowledge the use of artificial intelligence tools for reformulating certain sentences to improve clarity and for assistance with the formatting of graphs and tables throughout this thesis.

References

[1] Agarwal, K., Astra, R., Hoque, A., Srivatsa, M., Ganti, R., Wright, L., and Chen, S. (2024). HadaCore: Tensor Core Accelerated Hadamard Transform Kernel.

[2] Bader, M. and Zenger, C. (2006). Cache oblivious matrix multiplication using an element ordering based on a Peano curve. *Linear Algebra and its Applications*, 417(2):301–313.

[3] Bai, H. and Li, Y. (2023). Structured Sparsity in the NVIDIA Ampere Architecture and Applications in Search Engines.

[4] Benistant, F. (2022). Taylor Series, Pade Approximants, and Neural Networks.

[5] Blanes, S., Casas, F., Oteo, J. A., and Ros, J. (2009). The Magnus expansion and some of its applications. *Physics Reports*, 470(5):151–238.

[6] Blelloch, G. E. (1990). Prefix Sums and Their Applications.

[7] Brezinski, C. (1980). *Padé-Type Approximation and General Orthogonal Polynomials*, volume 50 of *International Series of Numerical Mathematics / Internationale Schriftenreihe zur Numerischen Mathematik / Série internationale d'Analyse numérique*. Birkhäuser, Basel.

[8] Böhm, C. (2020). Space-filling Curves for High-performance Data Mining. arXiv:2008.01684 [cs].

[9] Cirone, N. M., Orvieto, A., Walker, B., Salvi, C., and Lyons, T. (2025). Theoretical Foundations of Deep Selective State-Space Models. arXiv:2402.19047 [cs].

[10] Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 1–6, New York, NY, USA. Association for Computing Machinery.

[11] Dao, T. (2024). Fast Hadamard Transform in CUDA, with a PyTorch interface. original-date: 2023-11-29T01:24:07Z.

[12] Davis, T. A. and Hu, Y. (2011). The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25.

[13] Deka, P. J. and Einkemmer, L. (2022). Efficient adaptive step size control for exponential integrators. *Computers & Mathematics with Applications*, 123:59–74. arXiv:2102.02524 [math].

[14] Fasi, M. and Higham, N. J. (2019). An Arbitrary Precision Scaling and Squaring Algorithm for the Matrix Exponential. *SIAM Journal on Matrix Analysis and Applications*, 40(4):1233–1256. Publisher: Society for Industrial and Applied Mathematics.

[15] Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R Ruiz, F. J., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., and Kohli, P. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53.

[16] Fino, B. and Algazi, R. (1976). Unified Matrix Treatment of the Fast Walsh-Hadamard Transform | IEEE Journals & Magazine | IEEE Xplore.

[17] Gall, F. L. (2014). Powers of Tensors and Fast Matrix Multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, pages 23–23. arXiv:1401.7714 [cs].

[18] Gladwell, I., Shampine, L. F., and Brankin, R. W. (1987). Automatic selection of the initial step size for an ODE solver. *Journal of Computational and Applied Mathematics*, 18(2):175–192.

[19] Gu, A. and Dao, T. (2024). Mamba: Linear-Time Sequence Modeling with Selective State Spaces. arXiv:2312.00752 [cs].

[20] Gu, A., Goel, K., and Ré, C. (2022). Efficiently Modeling Long Sequences with Structured State Spaces. arXiv:2111.00396 [cs].

[21] Gu, A., Johnson, I., Goel, K., Saab, K., Dao, T., Rudra, A., and Ré, C. (2021). Combining Recurrent, Convolutional, and Continuous-time Models with Linear State-Space Layers. arXiv:2110.13985 [cs].

[22] Gustavson, F. G. (1978). Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269.

[23] Hairer, E., Norsett, S., and Wanner, G. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 8.

[24] Harris, M., Sengupta, S., and Owens, J. D. (2024). Chapter 39. Parallel Prefix Sum (Scan) with CUDA.

[25] Higham, N. J. (2005). The Scaling and Squaring Method for the Matrix Exponential Revisited. *SIAM Journal on Matrix Analysis and Applications*, 26(4):1179–1193. Publisher: Society for Industrial and Applied Mathematics.

[26] Higham, N. J. (2008). 10. Matrix Exponential. In *Functions of Matrices*, Other Titles in Applied Mathematics, pages 233–267. Society for Industrial and Applied Mathematics.

[27] Huang, J., Smith, T. M., Henry, G. M., and Van De Geijn, R. A. (2016). Strassen's Algorithm Reloaded. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 690–701, Salt Lake City, UT. IEEE.

[28] Intel (2023). Intel® Core™ i7 processor 14700K (33M Cache, up to 5.60 GHz) - Product Specifications.

[29] Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. (2020). Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. arXiv:2006.16236 [cs].

[30] Kauers, M. and Moosbauer, J. (2022). Flip Graphs for Matrix Multiplication. arXiv:2212.01175 [cs].

[31] Kerr, A., Merrill, D., Demouth, J., and Tran (2017). CUTLASS: Fast Linear Algebra in CUDA C++.

[32] Kidger, P. (2022). On Neural Differential Equations. arXiv:2202.02435 [cs].

[33] Kidger, P., Morrill, J., Foster, J., and Lyons, T. (2020). Neural Controlled Differential Equations for Irregular Time Series. arXiv:2005.08926 [cs].

[34] Ledoh, M., R. J. and Reza, P. (2019). Parallelization of Padé Approximation of Matrix Exponential with CUDA-Aware MPI. In *2019 5th International Conference on Science and Technology (ICST)*, volume 1, pages 1–6.

[35] Magnus, W. (1954). On the exponential solution of differential equations for a linear operator. *Communications on Pure and Applied Mathematics*, 7(4):649–673. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpa.3160070404.

[36] Merrill, W., Petty, J., and Sabharwal, A. (2025). The Illusion of State in State-Space Models. arXiv:2404.08819 [cs].

[37] Merrill, W. and Sabharwal, A. (2023). The Parallelism Tradeoff: Limitations of Log-Precision Transformers. arXiv:2207.00729 [cs].

[38] Moler, C. and Van Loan, C. (1978). Nineteen Dubious Ways to Compute the Exponential of a Matrix. *SIAM Review*, 20(4):801–836.

[39] Moler, C. and Van Loan, C. (2003). Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. *SIAM Review*, 45(1):3–49.

[40] Murray, L. (2024). Matrix Multiplication On GPU: Part 2, Tiling.

[41] Niu, Y., Lu, Z., Ji, H., Song, S., Jin, Z., and Liu, W. (2022). TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–106, Seoul Republic of Korea. ACM.

[42] Nocentino, A. E. and Rhodes, P. J. (2010). Optimizing memory access on GPUs using morton order indexing. In *Proceedings of the 48th Annual Southeast Regional Conference*, pages 1–4, Oxford Mississippi. ACM.

[43] Novikov, A., Vu, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. (2025). AlphaEvolve: A coding agent for scientific and algorithmic discovery.

[44] NVIDIA (2022). NVIDIA H100 Tensor Core GPU Architecture.

[45] NVIDIA (2024). Matrix Multiplication Background User's Guide - NVIDIA Docs.

[46] NVIDIA (2025). 1. Introduction — cuSPARSE 12.9 documentation.

[47] Osama, M., Merrill, D., Cecka, C., Garland, M., and Owens, J. D. (2023). Stream-K: Work-centric Parallel Decomposition for Dense Matrix-Matrix Multiplication on the GPU.

[48] Parger, M., Winter, M., Mlakar, D., and Steinberger, M. (2020). spECK: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 362–375, San Diego California. ACM.

[49] Paterson, M. S. and Stockmeyer, L. J. (1973). On the Number of Nonscalar Multiplications Necessary to Evaluate Polynomials. *SIAM Journal on Computing*, 2(1):60–66. Publisher: Society for Industrial and Applied Mathematics.

[50] Richardson, L. F. and Gaunt, J. A. (1997). VIII. The deferred approach to the limit. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 226(636-646):299–361. Publisher: Royal Society.

[51] Sadasivan, H., Osama, M., Podkorytov, M., Huang, C., and Liu, J. (2024). Stream-K++: Adaptive GPU GEMM Kernel Scheduling and Selection using Bloom Filters. arXiv:2408.11417 [cs].

[52] Sastre, J., Ibánez, J., Defez, E., and Ruiz, P. (2015). New Scaling-Squaring Taylor Algorithms for Computing the Matrix Exponential. *SIAM Journal on Scientific Computing*, 37(1):A439–A455. Publisher: Society for Industrial and Applied Mathematics.

[53] Smirnov, A. V. (2013). The bilinear complexity and practical algorithms for matrix multiplication. *Computational Mathematics and Mathematical Physics*, 53(12):1781–1795.

[54] Stothers, A. J. (2010). On the complexity of matrix multiplication. Accepted: 2011-02-01T10:14:31Z Publisher: The University of Edinburgh.

[55] Strassen, V. (1969). Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13:354–356.

[56] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2023). Attention Is All You Need. arXiv:1706.03762 [cs].

[57] Walker, B., Yang, L., Cirone, N. M., Salvi, C., and Lyons, T. (2025). Structured Linear CDEs: Maximally Expressive and Parallel-in-Time Sequence Models. arXiv:2505.17761 [cs].

[58] Wikipedia (2023). Bloom filter. Page Version ID: 1296914652.

[59] Wikipedia (2025). Computational complexity of matrix multiplication. Page Version ID: 1281194461.

[60] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. (2007). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 1–12, New York, NY, USA. Association for Computing Machinery.

[61] Williams, V. V. (2014). Multiplying matrices in O(n2.373) time.

[62] Winter, M., Mlakar, D., Zayer, R., Seidel, H.-P., and Steinberger, M. (2019). Adaptive sparse matrix-matrix multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 68–81, Washington District of Columbia. ACM.

[63] Yamagushi, T. and Busato, F. (2021). Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores.

[64] Yao, D., Zhao, S., Liu, T., Wu, G., and Jin, H. (2025). ApSpGEMM: Accelerating Large-scale SpGEMM with Heterogeneous Collaboration and Adaptive Panel. *ACM Transactions on Architecture and Code Optimization*, 22(1):1–23.

A Tensor Representation of Matrix Multiplication

Matrix multiplication can naturally be represented as a tensor, encapsulating its multilinear structure. This perspective enables the application of tensor decompositions and rank techniques, which provide useful tools for optimising matrix multiplication algorithms. In this appendix, we formalise these concepts, demonstrating how they explain the theoretical computational complexity of the algorithms introduced in § 3.

Definition A.1 (Tensor Product)**.** The tensor product of two vector spaces V and W is a vector space $V \otimes W$ together with a bilinear map $\otimes : V \times W \to V \otimes W$ satisfying the universal property that for every vector space U and bilinear map $\phi : V \times W \to U$, there is a unique linear map $\tilde{\phi} : V \otimes W \to U$ such that:

$$\phi(v, w) = \tilde{\phi}(v \otimes w) \quad \forall v \in V, w \in W.$$

In order for this definition to be meaningful, we must construct a vector space $V \otimes W$ and a bilinear map $\otimes$ that satisfy this universal property. We do so using a quotient of the free vector space.

Definition A.2 (Free Vector Space)**.** The free vector space $\mathbb{F}^{(V \times W)}$ is the vector space with basis $V \times W$, consisting of all finite formal sums:

$$\sum_{i=1}^{n} \alpha_i (v_i, w_i), \quad \alpha_i \in \mathbb{F}, (v_i, w_i) \in V \times W$$

with component-wise addition and scalar multiplication.

Construction. *Let $R \subseteq \mathbb{F}^{(V \times W)}$ be the subspace generated by the set of all vectors of the form:*

$$\begin{aligned} (v_1 + v_2, w) - (v_1, w) - (v_2, w), \\ (v, w_1 + w_2) - (v, w_1) - (v, w_2), \\ (\alpha v, w) - \alpha(v, w), \\ (v, \alpha w) - \alpha(v, w), \end{aligned}$$

for all $v, v_1, v_2 \in V$, $w, w_1, w_2 \in W$, and $\alpha \in \mathbb{F}$. The tensor product $V \otimes W$ is the quotient space:

$$V \otimes W := \mathbb{F}^{(V \times W)} / R.$$

Elements of $V \otimes W$ are equivalence classes $[(v, w)]$, denoted $v \otimes w$, and are often called pure tensors. Furthermore, the map $\otimes : V \times W \to V \otimes W$ is bilinear by construction.

Having constructed the tensor product space $V \otimes W$, we now verify that the universal property of the tensor product holds for the given construction.

Lemma A.3 (Universal Property of the Tensor Product)**.** *Let V, W and U be vector spaces. Then the linear map defined in Definition A.1 exists for the given construction, and is unique up to isomorphism.*

Proof. Let $\phi : V \times W \to U$ be a bilinear map. We want to construct a unique linear map $\tilde{\phi} : V \otimes W \to U$ such that:

$$\tilde{\phi}(v \otimes w) = \phi(v, w), \quad \forall v \in V, w \in W.$$

Existence: Define $\Phi : \mathbb{F}^{(V \times W)} \to U$ as the linear extension of ϕ to the free space by:

$$\Phi\left(\sum \alpha_i (v_i, w_i)\right) := \sum \alpha_i \phi(v_i, w_i).$$

Note that Φ sends all elements of the subspace R to 0 due to the bilinearity of ϕ. Thus Φ descends to the quotient $V \otimes W$, and we can define:

$$\tilde{\phi}([v, w]) := \Phi((v, w)).$$

Thus, the induced map $\tilde{\phi}$ satisfies:

$$\tilde{\phi}\left(\sum \alpha_i (v_i \otimes w_i)\right) = \sum \alpha_i \phi(v_i, w_i),$$

which is equivalent to the required condition from Definition A.1.

Uniqueness: Suppose $\psi : V \otimes W \to U$ is any linear map satisfying $\psi(v \otimes w) = \phi(v, w)$ for all $v \in V, w \in W$. We know by construction that each element of $V \otimes W$ is a finite sum of simple tensors of the form:

$$\sum_{i=1}^{n} \alpha_i (v_i \otimes w_i).$$

Thus we have:

$$\begin{aligned}\psi\left(\sum_{i=1}^{n} \alpha_i (v_i \otimes w_i)\right) &= \sum_{i=1}^{n} \alpha_i \psi(v_i \otimes w_i) \\ &= \sum_{i=1}^{n} \alpha_i \phi(v_i, w_i) \\ &= \tilde{\phi}\left(\sum_{i=1}^{n} \alpha_i (v_i \otimes w_i)\right),\end{aligned}$$

showing that that ψ and $\tilde{\phi}$ agree on all sums of simple tensors. As both are linear, and coincide on the spanning set $\{v \otimes w : v \in V, w \in W\}$, they must be equal. □

The universal property can be extended to the trilinear case. This is a key step to formulate matrix multiplication as a tensor.

Proposition A.4. *Let V, W, U be finite dimensional vector spaces over a field $\mathbb{F}$. Every trilinear map $T : V \times W \times U^* \to \mathbb{F}$ corresponds uniquely to an element $\tilde{T} \in V^* \otimes W^* \otimes U$. Equivalently, there is a natural isomorphism:*

$$\mathrm{Tril}(V \times W \times U^*, \mathbb{F}) \cong V^* \otimes W^* \otimes U.$$

Proof. Let $T : V \times W \times U^* \to \mathbb{F}$ be trilinear. For fixed $f \in U^*$, define:

$$T_f(v, w) := T(v, w, f).$$

It is easy to see that $T_f : V \times W \to \mathbb{F}$ is bilinear and the map $f \mapsto T_f$ is linear. Now, define $\tilde{T} : V \times W \to (U^*)^*$ as:

$$\tilde{T}(v, w)(f) := T_f(v, w).$$

Thus, there exists an isomorphism

$$\mathrm{Tril}(V \times W \times U^*, \mathbb{F}) \cong \mathrm{Bil}(V \times W, (U^*)^*) \cong \mathrm{Bil}(V \times W, U) \cong \mathrm{Lin}(V \otimes W, U),$$

where the penultimate isomorphism comes from the fact that $(U^*)^* \cong U$ as U is finite dimensional, and the ultimate isomorphism from the universal property. Thus there is some linear map $\hat{T} : V \otimes W \to U$ such that $\tilde{T}(v, w) = \hat{T}(v \otimes w)$.

Now for $v \in V, w \in W, f \in U^*$, define the linear functional $\Phi_T : V \otimes W \otimes U^* \to \mathbb{F}$ by:

$$\Phi_T(v \otimes w \otimes f) := f(\hat{T}(v \otimes w)).$$

This gives an isomorphism:,

$$\text{Lin}(V \otimes W, U) \cong \text{Lin}(V \otimes W \otimes U^*, \mathbb{F}) = (V \otimes W \otimes U^*)^*.$$

Next, define the map $\Psi : V^* \times W^* \to (V \otimes W)^*$ by:

$$\Psi(\phi \otimes \psi)(v \otimes w) = \phi(v)\psi(w).$$

This map is bilinear in ϕ and ψ. By the universal property of tensor products, this extends uniquely to a linear map $\tilde{\Psi} : V^* \otimes W^* \to (V \otimes W)^*$. Due to finite dimensionality of V and W, this map is an isomorphism giving,

$$(V \otimes W)^* \cong V^* \otimes W^*,$$

meaning that

$$(V \otimes W \otimes U^*)^* \cong V^* \otimes W^* \otimes (U^*)^* \cong V^* \otimes W^* \otimes U,$$

yielding the desired result. □

We now specialize to the case of matrices to obtain our main result.

Theorem A.5 (Matrix Multiplication as a Tensor). *Let $\langle m, k, n\rangle$ denote the multiplication of an $m \times k$ matrix A with an $k \times n$ matrix B to produce an $m \times n$ matrix C. Then the bilinear map:*

$$(A, B) \mapsto C = AB$$

can be represented by a tensor $T_{\langle m,k,n\rangle} \in (\mathbb{F}^{m\times k})^* \otimes (\mathbb{F}^{k\times n})^* \otimes \mathbb{F}^{m\times n}$ where $\mathbb{F}$ is a base field and $*$ denotes the dual space.

Proof. Given the bilinear map $\mu : \mathbb{F}^{m\times k} \times \mathbb{F}^{k\times n} \to \mathbb{F}^{m\times n}$, we know there exists a unique linear map $\tilde{\mu} : \mathbb{F}^{m\times k} \otimes \mathbb{F}^{k\times n} \to \mathbb{F}^{m\times n}$ from the universal property. From Proposition A.4, we know that

$$\tilde{\mu} \in \text{Lin}(F^{m\times k} \otimes F^{k\times n}, F^{m\times n}) \cong (F^{m\times k})^* \otimes (F^{k\times n})^* \otimes F^{m\times n},$$

which yields the result. □

B Expressivity

In this Appendix, we use the framework and definitions introduced by Walker et al. [57].

Definition B.1 (Maximal Expressivity). Let $\mathcal{X}$ be a topological space and let $\mathcal{F} = \{f_\theta : \mathcal{X} \to \mathbb{R} \mid \theta \in \Theta\}$ denote a family of real-valued functions on $\mathcal{X}$ parameterised by a set Θ. The family $\mathcal{F}$ is said to be maximally expressive (or universal) if, for any compact subset $\mathcal{K} \subset \mathcal{X}$ and any continuous function $f : \mathcal{K} \to \mathbb{R}$, there exists a sequence of functions in $\mathcal{F}$ that converges uniformly to f on $\mathcal{K}$. Equivalently:

$$\forall \epsilon > 0, \exists \theta \in \Theta \quad \text{s.t.} \quad \sup_{x\in\mathcal{K}} |f(x) - f_\theta(x)| < \epsilon.$$

Taking $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{F}$ to be the class of single-hidden-layer feed-forward networks yields the the Universal Approximation Theorem. Expressivity can also be viewed in an alternative way by introducing the concept of probabilistic expressivity:

Definition B.2 (Maximal Probabilistic Expressivity)**.** Let $\mathcal{X}$ be a topological space, $N \in \mathbb{N}$, and $\mathcal{F}^N = \{f_\theta^N : \mathcal{X} \to \mathbb{R} \mid \theta \in \Theta_N\}$ denote a family of real-valued functions on $\mathcal{X}$ where each function is defined by

$$f_\theta^N(\omega) = \ell_{\theta_2}(\tilde{f}_{\theta_1}^N(\omega)),$$

for $\omega \in \mathcal{X}$. Here, $\tilde{f}_{\theta_1}^N : \mathcal{X} \to \mathbb{R}^N$ represents a feature map, $\ell_{\theta_2} : \mathbb{R}^N \to \mathbb{R}$ is a linear readout function, with parameters $\theta_1 \in \Theta_1^N$, $\theta_2 \in \Theta_2^N$, and $\Theta_N = \Theta_1^N \cup \Theta_2^N$. Given a sequence of probability measures $\mathbb{P}_N$ defined on Θ_1^N such that $\theta_1 \sim \mathbb{P}_N$, the family $\mathcal{F}$ is said to have maximal probabilistic expressivity if, for every compact subset $\mathcal{K} \subset \mathcal{X}$, every continuous function $f : \mathcal{K} \to \mathbb{R}$, and every $\epsilon > 0$, then:

$$\lim_{N\to\infty} \mathbb{P}_N \left\{ \exists \ell_{\theta_2} \text{ s.t. } \sup_{\omega\in\mathcal{K}} |f(\omega) - f_\theta^N(\omega)| < \epsilon \right\} = 1.$$

This latter definition offers greater practical utility, as it ensures that for sufficiently large model dimensions, almost all random parameter initialisations will admit some linear readout capable of achieving the desired approximation quality.

When using LNCDEs, $N = d_h$ and each matrix $A_{\theta_1}^i$ is drawn from its own probability distribution $\mathbb{P}_{d_h}^i$ and

$$\tilde{f}_{\theta_1}^{d_h}(\omega) = h_{t_0} + \int_{t_0}^{t_n} \sum_{i=1}^{d_\omega} A_{\theta_1}^i h_s \, d\omega_s^i.$$

C Bloom Filters for Stream K++

Bloom filters are efficient data structures that check whether an element is a member of a set with perfect accuracy for negative results but potential false positives. When queried for an element, the two possible outputs of a Bloom filter are that the element "may be in the set" or "is definitely not in the set".

A Bloom filter consists of an m-bit array initially filled with zeros and k independent hash functions. To add an element to the set, each hash function maps the element to a position in the bit array (0 to m-1), and all k corresponding bits are set to 1. When an element is queried, the same k has functions are applied. If any of the resulting k bit positions contain a 0, the element is known to be absent from the set. If all k positions contain a 1, the element may be in the set. The hash functions must be independent and uniformly distributed across the entire array to minimise false positive rates and ensure reliable performance.
To visualise this more clearly, consider the following example

Example C.1. Suppose we have a set of three elements $\{x, y, z\}$ with three hash functions h_1, h_2, h_3 (we leave these undefined, but several options can be taken). Initialise the empty Bloom filter to be an 18 length array of only zeros. Next, for each element in {x,y,z}, apply all three hash functions and set the corresponding bit positions to 1, where each hash output is taken modulo 18 to ensure it falls within the array bounds.
In this case we have:

$$\begin{aligned}
&h_1(x) = 1 \pmod{18} \qquad &&h_1(y) = 4 \pmod{18} \qquad &&h_1(z) = 3 \pmod{18}\\
&h_2(x) = 5 \pmod{18} &&h_2(y) = 11 \pmod{18} &&h_2(z) = 5 \pmod{18}\\
&h_3(x) = 13 \pmod{18} &&h_3(y) = 16 \pmod{18} &&h_3(z) = 11 \pmod{18},
\end{aligned}$$

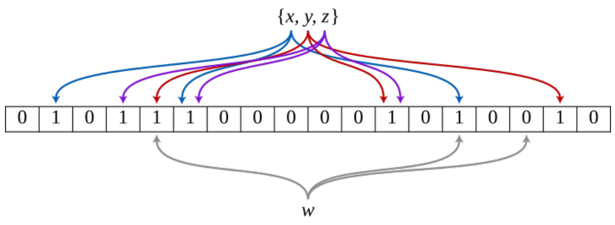

Figure 13: Bloom Filter on a Set of 3 Elements with $k = 3$ [58]

meaning we set the bit positions at indices $1, 3, 4, 5, 11, 13$ and 16 to 1. To test the inclusion of element w in the set, we apply the three hash functions to w, obtaining:

$$\begin{aligned} h_1(w) &= 4 \pmod{18} \\ h_2(w) &= 13 \pmod{18} \\ h_3(w) &= 15 \pmod{18}. \end{aligned}$$

Inspecting the above figure, we see that these indices correspond to a $1, 1$ and 0 respectively. Thus, as one of the bit positions contains a 0, we conclude with certainty that w is not in the set.

C.1 Methodology for Stream-K ++

For Stream-K ++, 7 policies are used, ranging from the basic all data-parallel approach, to batches of stream-k processing (0-6 batches) followed by a data-parallel batch. These 7 policies are initially used on the dataset of 923 (M, K, N) matrix dimensions, and an optimal policy is chosen, depending on which policy produces the shortest processing time.

In the second step, 7 empty Bloom filters are initialised, each corresponding to one of the distinct polices. The 923 matrix permutations are then processed by applying 7 hash functions to determine bit positions in the Bloom filter corresponding to the optimal policy. For instance if a matrix of specific size exhibits best performance on the 2nd policy, the hash functions are applied to the matrix size and resulting indices are filled in only the 2nd Bloom filter.

In the last step, matrix sizes are queried following the same procedure as in Example C.1, across all 7 Bloom filters. The policies with associated Bloom filters with 1s in all queried bit positions are then chosen as the list of possible policies. These candidate policies are then benchmarked to find the actual optimal choice.

D Accuracy-Based Control Using Richardson Extrapolation [50]

Richardson's method improves the rate of convergence of a sequence of estimates by combining results obtained with different step sizes to eliminate leading order error terms. His framework, reformulated by Hairer et Al. [23] is provided below:

Suppose that we have an initial value (x_0, y_0) and step size h. We compute two steps, using an approximation method of order p, and obtain the results y_1 and y_2. We then compute, starting from (x_0, y_0), one big step with step size $2h$ to obtain the solution w. The error of y_1 is

$$e_1 = y(x_0 + h) - y_1 = C \cdot h^{p+1} + \mathcal{O}(h^{p+2})$$

We can also obtain the error for the second step which is composed of the transported error of the first step and the local error of the second step, obtaining:

$$\begin{aligned} e_2 = y(x_0 + 2h) - y_2 &= (I + \mathcal{O}(h))\, Ch^{p+1} + (C + \mathcal{O}(h))\, h^{p+1} + \mathcal{O}(h^{p+2}) \\ &= 2Ch^{p+1} + \mathcal{O}(h^{p+2}). \end{aligned}$$

For the big step w, we have:

$$y(x_0 + 2h) - w = C(2h)^{p+1} + \mathcal{O}(h^{p+2}).$$

Combining the two above results, we obtain:

$$y(x_0 + 2h) - y_2 = \frac{y_2 - w}{2^p - 1} + \mathcal{O}(h^{p+2}).$$

Thus, using this result, we can obtain a better result for y_2, by finding an approximation of order $p + 1$:

$$\hat{y}_2 = y_2 + \frac{y_2 - w}{2^p - 1}$$

In traditional ODE solvers, this corrected value $\hat{y}_2$ is used to improve the solution's order of accuracy by cancelling the leading local truncation error. In our setting, LNCDEs with piecewise-linear control paths have exact interval-wise solutions in theory, so there is no time-discretisation error. Instead, we apply the step-doubling comparison $y_2 - w$ to estimate the numerical error in computing the matrix exponential. This serves as the basis for the first method in § 7.3.2, where the estimate is used solely to decide whether the step size should be reduced, rather than to choose the corrected $\hat{y}_2$.

E Gauss-Legendre Quadrature for Magnus Expansion

In order to evaluate truncations of the Magnus expansion, the numerical evaluation of (nested) integrals is required. For instance, the fourth-order Magnus expansion is given by

$$\Omega^{[4]}(t, t_0) = \int_{t_0}^{t} A(\tau_1) d\tau_1 - \frac{1}{2} \int_{t_0}^{t} \int_{t_0}^{\tau_1} [A(\tau_1), A(\tau_2)] d\tau_2 d\tau_1.$$

Accurate numerical approximation of these integrals is essential to achieve the desired order of accuracy. One standard approach is to use quadrature rules, which approximate integrals by weighted sums of function evaluations at specific points.

A quadrature rule approximates an integral of a function f over an interval $[a, b]$ as

$$\int_a^b f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i),$$

where $x_i \in [a, b]$ are the nodes and w_i are the corresponding weights.

E.1 Gauss-Legendre Quadrature

For the numerical evaluation of integrals arising in truncated Magnus expansions, Gauss-Legendre quadrature provides an efficient approach. This method constructs quadrature nodes x_i from the roots of the n-th degree Legendre polynomial $P_n(x)$. The resulting quadrature rule achieves exact integration for polynomials of degree at most $2n - 1$, making it particularly suitable for approximating the integrals that appear in truncated Magnus expansions with a minimal number of function evaluations.

E.1.1 Legendre Polynomials

Legendre polynomials $P_n(x)$ are polynomials of degree n defined on the interval $[-1,1]$ that satisfy the Legendre differential equation:

$$\frac{d}{dx}\Big[(1-x^2)\frac{dP_n}{dx}\Big] + n(n+1)P_n(x) = 0, \quad n = 0,1,2,\dots$$

The first few Legendre polynomials are:

$$\begin{aligned}
P_0(x) &= 1,\\
P_1(x) &= x,\\
P_2(x) &= \frac{1}{2}(3x^2-1),\\
P_3(x) &= \frac{1}{2}(5x^3-3x)
\end{aligned}$$

The roots of $P_n(x)$ lie in $(-1,1)$ and are used as nodes for the n-point Gauss-Legendre quadrature. Thus, the quadrature is naturally defined on the interval $[-1,1]$:

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i),$$

where x_i are the roots of $P_n(x)$ and w_i are the corresponding weights.

E.2 Finding the Nodes for Fourth-Order Magnus Expansion

For a fourth-order Magnus integrator, a 2-point Gauss-Legendre quadrature is sufficient, as the nested commutator integral is cubic in τ after expansion. For a step from t_n to t_{n+1} with $\Delta t = t_{n+1} - t_n$, the nodes and weights are

$$t_{n,\pm} = t_n + \frac{\Delta t}{2}\Big(1 \pm \frac{1}{\sqrt{3}}\Big), \quad w_\pm = \frac{\Delta t}{2}.$$

We can then use these points to approximate the nested integral

$$\int_{t_n}^{t_{n+1}} \int_{t_n}^{\tau_1} [A(\tau_1), A(\tau_2)]\, d\tau_2 d\tau_1.$$

Theorem E.1. *[5] The fourth order Magnus term can be written as*

$$\Omega^{[4]} = \Omega^{[2]} - \frac{\sqrt{3}\,\Delta t^2}{12}[A_-, A_+].$$

Proof. On the step $[t_n, t_{n+1}]$, approximate $A(\tau)$ by its first order Taylor expansion:

$$A(\tau) \approx A(t_n) + (\tau - t_n)A'(t_n).$$

Consider the nested commutator integral

$$I = \int_{t_n}^{t_{n+1}} \int_{t_n}^{\tau_1} [A(\tau_1), A(\tau_2)]\, d\tau_2\, d\tau_1.$$

Expanding the commutator,

$$\begin{aligned}
[A(\tau_1), A(\tau_2)] &= [A(t_n) + (\tau_1 - t_n)A'(t_n),\ A(t_n) + (\tau_2 - t_n)A'(t_n)]\\
&= (\tau_2 - \tau_1)[A(t_n), A'(t_n)].
\end{aligned}$$

Thus,

$$\int_{t_n}^{\tau_1} [A(\tau_1), A(\tau_2)]\, d\tau_2 = [A(t_n), A'(t_n)] \int_{t_n}^{\tau_1} (\tau_2 - \tau_1)\, d\tau_2 = -\tfrac{1}{2}(\tau_1 - t_n)^2 [A(t_n), A'(t_n)].$$

Integrating in τ_1,

$$I = \int_{t_n}^{t_{n+1}} -\tfrac{1}{2}(\tau_1 - t_n)^2 [A(t_n), A'(t_n)]\, d\tau_1 = -\frac{\Delta t^3}{6}\, [A(t_n), A'(t_n)], \tag{11}$$

where $\Delta t = t_{n+1} - t_n$.

Now, evaluate $[A_-, A_+]$ at the two Gauss-Legendre nodes

$$t_{n,\pm} = t_n + \frac{\Delta t}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right), \qquad A_\pm = A(t_{n,\pm}).$$

With the first order Taylor series,

$$A_- = A(t_n) + (t_{n,-} - t_n)A'(t_n), \quad A_+ = A(t_n) + (t_{n,+} - t_n)A'(t_n),$$

hence

$$\begin{aligned}[A_-, A_+] &= \left((t_{n,+} - t_n) - (t_{n,-} - t_n)\right)[A(t_n), A'(t_n)] \\ &= \frac{\Delta t}{\sqrt{3}}[A(t_n), A'(t_n)]\end{aligned}$$

Therefore,

$$[A(t_n), A'(t_n)] = \frac{\sqrt{3}}{\Delta t}[A_-, A_+].$$

Substituting into Equation 11 gives

$$I = -\frac{\sqrt{3}\,\Delta t^2}{6}[A_-, A_+].$$

Thus, multiplying by the factor of 1/2, we obtain

$$\Omega^{[4]} = \Omega^{[2]} - \frac{\sqrt{3}\,\Delta t^2}{12}[A_-, A_+],$$

as required. □

F Additional Experiments

This appendix is provided to demonstrate all additional experiments that were carried out over the period of research. All experiments in this thesis are conducted with an A100 NVIDIA GPU.

F.1 GEMM on GPU

In this subsection, matrices with random entries are sampled from the standard normal distribution. All implementations are performed using PyTorch.

F.1.1 Tensor Core Activation

As discussed in § 4.3.2, Tensor Cores are activated on modern GPU architectures when reduced-precision data types are used. To test the performance increase when Tensor Cores are activated, two square 4096×4096 matrices were used to perform matrix multiplication with different data types. Note that FP16 is not supported for CPUs, so only FP32 was used for this case.

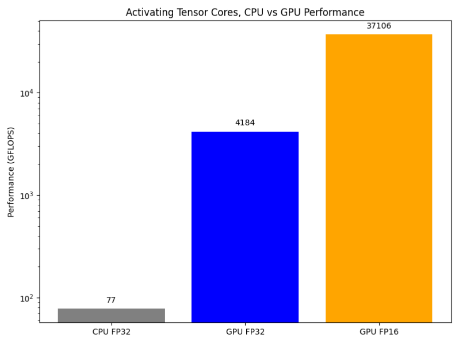

Figure 14: Performance for $M = K = N = 4096$ GEMM on T4 GPU

The GPU achieved significant speed-up over CPU for FP32 operations, with even greater acceleration observed when using FP16 precision on GPU due to Tensor Core activation. In fact, activation of the Tensor Cores when FP16 data was used led to a roughly 9 times performance increase, with GFLOPS rising from $4,184$ to $37,106$. The CPU only achieved 77 GFLOPS for FP32 data.

F.1.2 Basic GEMM

To examine the performance of GEMM for different sizes of square matrices, $M = K = N$ were varied between 2^5 and 2^{14}, using the FP16 data type to ensure Tensor Core utilisation.

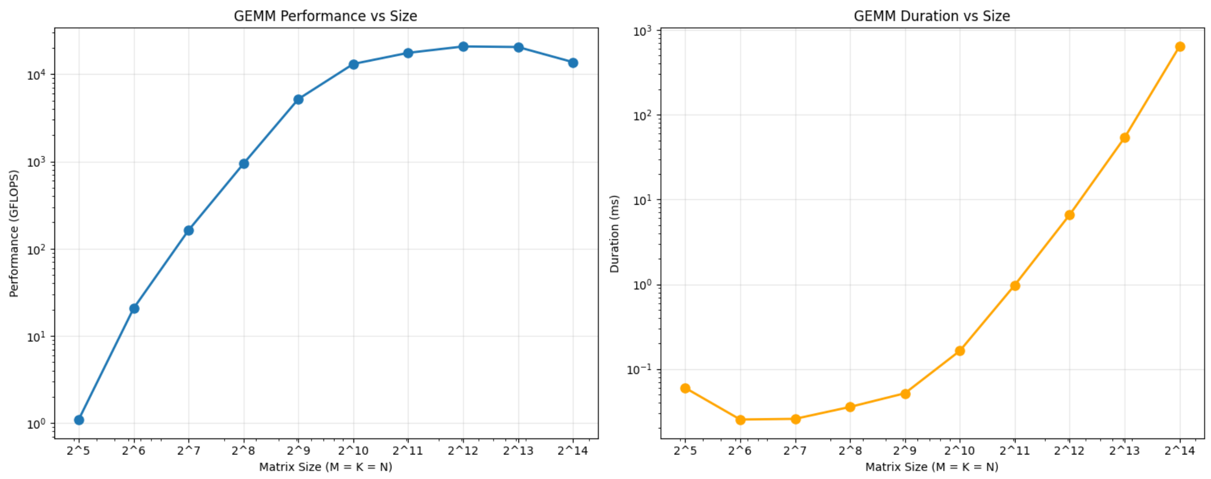

Figure 15: Performance and Duration for GEMM on T4 GPU

When performing basic GEMM in the setup described above, performance increased with matrix size before up to the threshold of 2^{12}, before tailing off slightly for larger dimensions. The duration began by decreasing slightly with matrix size between matrices with sizes 2^5 and 2^6, before monotonically increasing with matrix size.

This behaviour reflects the three phases of GPU utilisation. Initially small matrices are not sufficiently large to utilise the full GPU capacity, resulting in fixed overheads dominating actual computational time. As the size of the matrices grow, so does the utilisation of the GPU, until the maximal threshold is hit, where memory accesses begin to cause issues.

F.1.3 CPU vs GPU GEMM

To compare performance on CPU and GPU, $M = K = N$ were varied between 2^5 and 2^{12}, using the FP16 data type.

Matrix Size	CPU (GFLOPS)	GPU (GFLOPS)	GPU Speedup
32	9.8	1.7	0.2×
64	42.4	13.2	0.3×
128	70.1	106.7	1.5×
256	114.4	764.0	6.7×
512	127.6	4,638.6	36.4×
1,024	137.2	12,473.1	90.9×
2,048	122.4	16,153.2	132.0×
4,096	110.7	19,938.4	180.1×

Table 10: CPU vs GPU Matrix Multiplication Performance on T4 GPU

Based on the results, we can see that large speed-ups were obtained for large matrix sizes, with speed-up increasing monotonically with size. For the two smallest sizes of 32 and 64, the CPU outperformed the GPU significantly, with speed-ups of 0.2× and 0.3× obtained for the two respective cases. The crossover point occurred for matrix size 128, where the GPU began to show its computational advantages, with a 1.5× speed-up. This advantage increased to over 180× speed-up for the largest matrix size tested of 4,096.

The performance of the GPU increased monotonically over the matrix sizes, indicating that matrices were not large enough to cause memory access problems. A performance of almost 20,000 GFLOPS was obtained for matrix size 4,096. The CPUs performance increased until matrix size 1,024, reaching a peak of 137.2 GFLOPS, before slightly dropping off for the two largest matrix sizes.

F.2 Structured GEMM

F.2.1 Block-Diagonal GEMM

To test the speed-up of using block-diagonal matrices over regular dense ones, performance on randomly generated block-diagonal square matrices with eight blocks (where each block has size $N/8$) was examined, comparing PyTorch's standard dense matrix multiplication and `torch.bmm` for batched GEMM. Times were averaged over 50 trials on the T4 GPU.

The results show significant performance increases for the batched GEMM method, with speed-up increasing with matrix size, achieving over 45 times the performance of dense GEMM for matrices of size 8192×8192 and 16384×16384. Some amount of speed-up, however minimal, was observed for every matrix sized used.

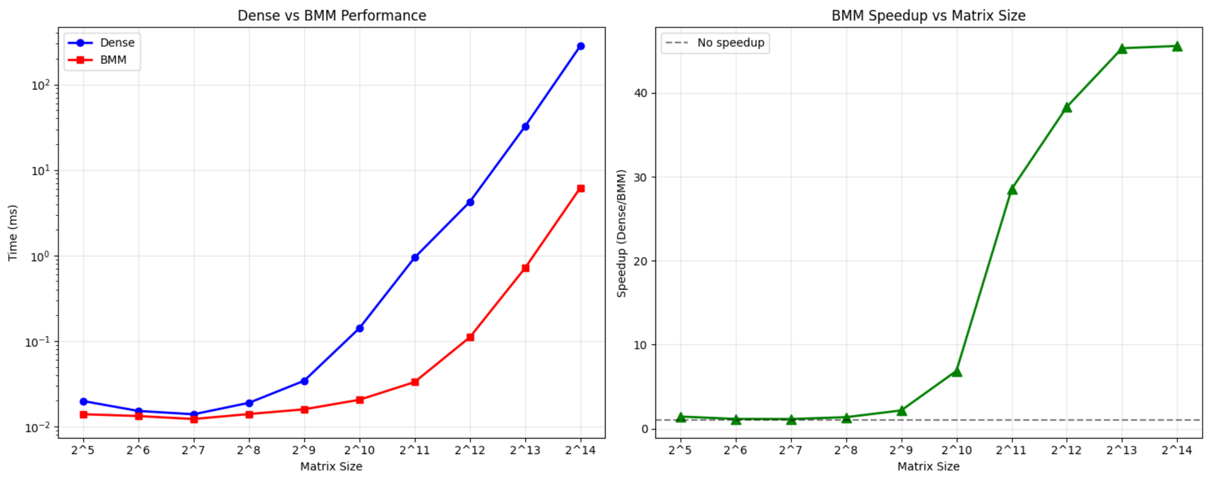

Figure 16: Dense vs BMM Performance and Speed-up on T4 GPU

F.2.2 Hadamard Methods

In order to test the various Hadamard matrix multiplication or FWHT equivalent methods, naive FWHT (Algorithm 1), Dao AI Labs' `hadamard_transform`, and HadaCore's `faster _hadamard_transform` were implemented on square matrices with size 2^n for $5 \leq n \leq 15$. The speed of the respective methods were compared to those for dense GEMM, averaging times over 30 trials on an A100 GPU, which is a requirement for the Dao AI Labs and HadaCore methods.
The results demonstrate that both the Dao AI and HadaCore implementations achieved substantial performance improvements over the naive FWHT implementation and dense GEMM approaches across all tested matrix sizes. The Dao AI method consistently outperformed dense matrix multiplication by factors ranging from 2.60 for smaller matrices (2^5) to over 440.36 for the largest tested size (2^{15}). The speed-up showed a clear scaling effect, getting progressively larger with matrix size.

In line with research, HadaCore achieved even better performance than the Dao AI method, delivering higher speed-ups across all matrix sizes. These speed-ups ranged from 3.39 (2^5) to 511.18 (2^{14}) over the basic dense GEMM method.

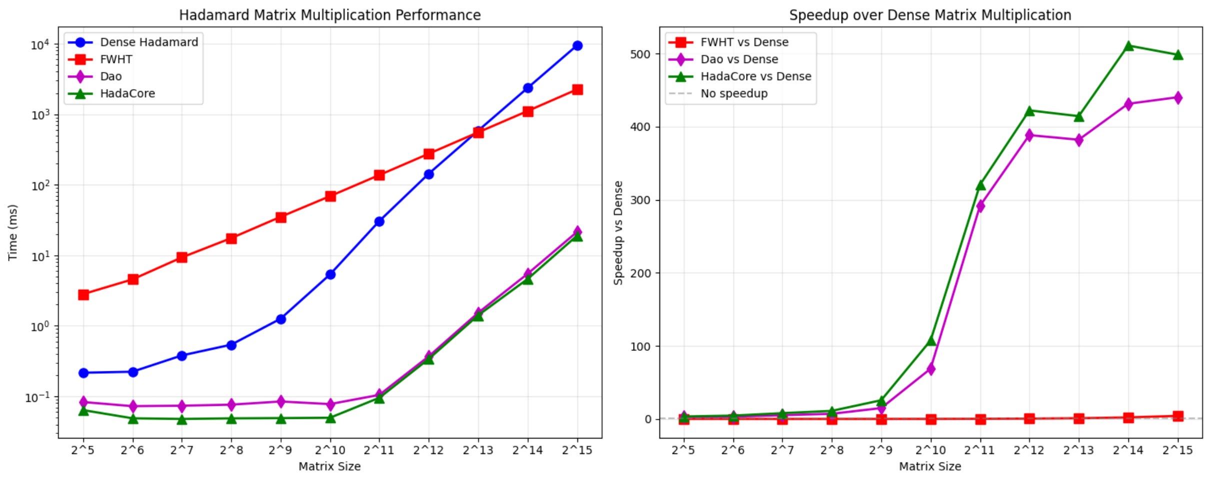

Figure 17: Performance and Speed-up of Hadamard Methods on A100 GPU

Size	Dense (ms)	FWHT (ms)	Dao AI (ms)	HadaCore (ms)	FWHT Speed-up	Dao AI Speed-up	HadaCore Speed-up
32	0.216	2.816	0.083	0.064	0.08×	2.60×	3.39×
64	0.224	4.554	0.073	0.049	0.05×	3.08×	4.57×
128	0.381	9.338	0.074	0.048	0.04×	5.16×	7.95×
256	0.541	17.521	0.076	0.049	0.03×	7.08×	11.08×
512	1.261	35.098	0.085	0.049	0.04×	14.90×	25.63×
1024	5.348	68.907	0.078	0.050	0.08×	68.80×	107.45×
2048	30.641	137.431	0.105	0.096	0.22×	291.71×	320.55×
4096	143.550	275.516	0.369	0.340	0.52×	388.55×	422.40×
8192	584.514	553.534	1.529	1.410	1.06×	382.22×	414.48×
16384	2352.505	1109.291	5.453	4.602	2.12×	431.41×	511.18×
32768	9423.112	2243.483	21.398	18.897	4.20×	440.36×	498.66×

Table 11: Performance Comparison of Hadamard Matrix Multiplication Methods on A100 GPU

In contrast, naive implementations of the FWHT exhibited poor performance on smaller sizes, underperforming dense GEMM for matrices of sizes 2^5 to 2^{12}. However, consistent with the scaling behaviour observed in other fast transform methods, FWHT began to demonstrate slightly better performance at larger matrix sizes, achieving modest speed-ups of 2.12× and 4.20× for matrices of size 2^{14} and 2^{15} respectively. This transition point around 2^{13} suggests that the algorithmic complexity advantages of the FWHT only become apparent when the overhead costs are distributed across sufficiently large computational workloads.

F.2.3 Sparse GEMM

The SuiteSparse Matrix Collection, previously known as the University of Florida Sparse Matrix Collection, is a comprehensive repository of sparse matrices used as benchmarks across a wide range of application domains [12]. 12 matrices from this set were used to evaluate the TileSpGEMM algorithm compared to cuSPARSE implementation for $C = A^2$, using the GitHub for the TileSpGEMM [41] and spECK [48] papers due to their ease of implementation. These 12 matrices represent a commonly used dataset in sparse matrix research, which was first utilised by William's et al [60]. Note that all matrices used in the dataset are square matrices, which allowedd us to compute A^2. A visualisation of the distributions of non-zeros in two matrices in the dataset are provided in Figure 18. An A100 NVIDIA GPU was used for this experiment.

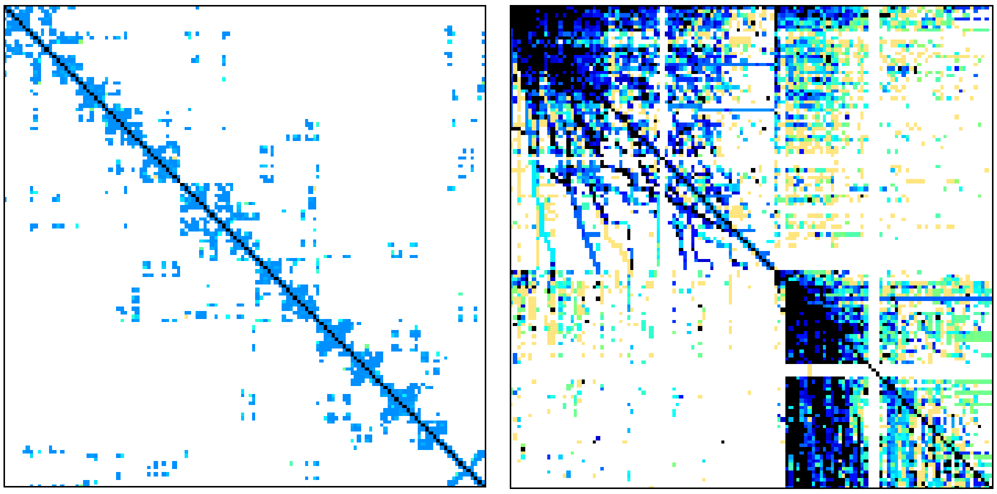
Figure 18: Non-zero Distribution Patterns in pdb1HYS and webbase-1M Matrices [12]

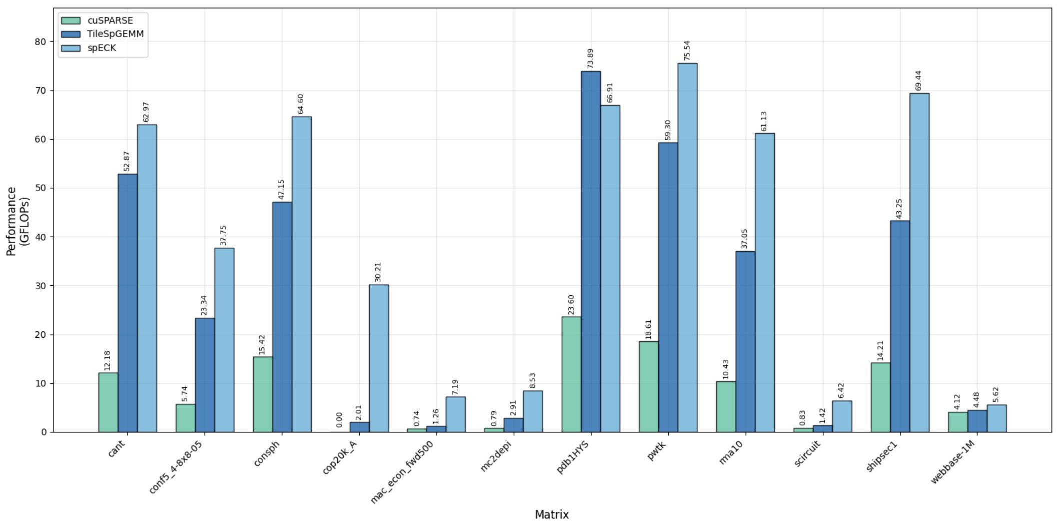

Figure 19: Performance of spECK, TileSpGEMM and cuSPARSE on A100 GPU

Analysing the results obtained, spECK consistently outperformed both TileSpGEMM and cuSPARSE on 11 out of 12 matrices, with only pdb1HYS showing TileSpGEMM achieving slightly higher GFLOPS (73.89 vs 66.91) and cuSPARSE underperforming both other methods on all 12 matrices. spECK demonstrated particularly impressive performance improvements on certain matrices where TileSpGEMM struggled, such as cop20k$_A$, $achieving 30.21 GFLOPS compared to TileSpGEMM$

The spECK method achieved highest performance on the pwtk matrix, achieving 75.54 GFLOPS, followed closely by shipsec1 at 69.44 GFLOPS and pdb1HYS at 66.91 GFLOPS. TileSpGEMM achieved its highest performance on the pdb1HYS matrix at 73.89 GFLOPS, compared to 23.60 GFLOPS for cuSPARSE. The lowest performance for spECK occurred on the webbase-1M matrix at 5.62 GFLOPS, while TileSpGEMM showed its poorest performance on the cop20k_A matrix at 2.01 GFLOPS, where cuSPARSE failed altogether.

Matrix	**dim(A)**	**nnz(A)**	**nnz(C)**	**FLOPs Required**	**spECK (ms)**	**TileSpGEMM (ms)**	**cuSPARSE (ms)**
cant	62K	4.0M	17.4M	539.0M	8.56	10.19	44.27
consph	83K	6.0M	26.5M	927.7M	14.36	19.68	60.17
pdb1HYS	36K	4.3M	19.6M	1.1B	16.44	15.03	47.07
cop20k_A	121K	2.6M	18.7M	159.8M	5.29	79.64	FAILED
mac_econ_fwd500	207K	1.3M	6.7M	15.1M	2.10	12.03	20.55
rma10	47K	2.4M	7.9M	313.0M	5.12	8.45	30.01
scircuit	171K	1.0M	5.2M	17.4M	2.71	12.23	20.83
shipsec1	141K	7.8M	24.1M	901.3M	12.98	20.84	63.42
pwtk	218K	11.6M	32.8M	1.3B	17.21	21.11	67.28
webbase-1M	1.0M	3.1M	51.1M	139.0M	24.73	31.06	33.75
conf5_4-8x8-05	49K	1.9M	10.9M	149.5M	3.96	6.41	26.03
mc2depi	526K	2.1M	5.2M	16.8M	1.97	5.76	21.25

Table 12: Performance of spECK, TileSpGEMM, and cuSPARSE on Sparse Matrix Subset