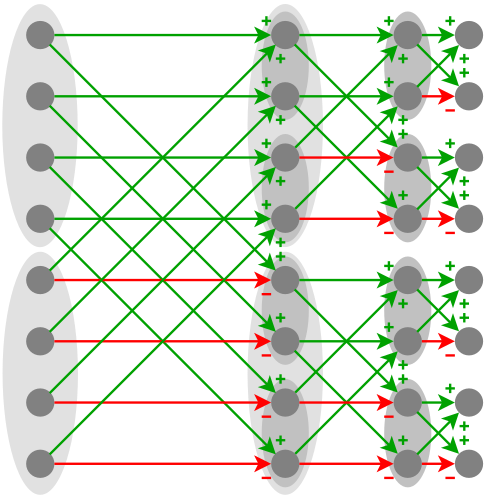

Efficient And Numerically Stable GPU Implementations of The Fast Walsh-Hadamard Transform

Exploring The Practical Applicability of a New Fast Walsh-Hadamard Transform Algorithm and Methods to Improve Rounding Errors When Calculating Walsh-Hadamard Transforms

Master's thesis in Complex Adaptive Systems

JOEL ANDERSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Efficient And Numerically Stable GPU Implementations of The Fast Walsh-Hadamard Transform

Exploring The Practical Applicability of a New Fast Walsh-Hadamard Transform Algorithm and Methods to Improve Rounding Errors When Calculating Walsh-Hadamard Transforms

JOEL ANDERSSON

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Efficient And Numerically Stable GPU Implementations of The Fast Walsh-Hadamard Transform
Exploring The Practical Applicability of a New Fast Walsh-Hadamard Transform Algorithm and Methods to Improve Rounding Errors When Calculating Walsh-Hadamard Transforms
JOEL ANDERSSON

Supervisor: Matti Karppa, Computer Science and Engineering
Examiner: Graham Kemp, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Data flow of the FWHT on an input vector of size 8

Typeset in LaTeX
Gothenburg, Sweden 2025

Efficient And Numerically Stable GPU Implementations of The Fast Walsh-Hadamard Transform
Exploring The Practical Applicability of a New Fast Walsh-Hadamard Transform Algorithm and Methods to Improve Rounding Errors When Calculating Walsh-Hadamard Transforms
Joel Andersson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Walsh-Hadamard Transform is a fundamental mathematical operation used for many applications. Recent findings from its applicability within Convolutional Neural Networks (CNNs) has sparked a lot of attention, leading to very efficient GPU implementations of the Fast Walsh-Hadamard Transform (FWHT) algorithm. However, current state-of-the-art FWHT implementations are purpose built for usage within CNNs, thus the input size is heavily limited and the precision supported is low, reducing the number of alternative applications. Additionally, recent algorithmic improvements by Alman and Rao have not been used in these implementations. This thesis tackles all of these problems, presenting GPU implementations of both the *Folklore* and *Alman & Rao* algorithms. The implementations support input sizes of up to 2^{63} and the numerical formats: FP64, FP32, FP16 and BF16. Both implementations suffer heavily from being memory bound, only outperforming Nvidia's implementation of the computationally heavier Fast Fourier Transform on L4/L40s GPUs when using FP64. We find no benefit of using *Alman & Rao*'s algorithm. Additionally, we employ techniques heavily inspired by Kahan and Neumaier summation, empirically reducing relative errors on a wide range of real world applications. For a given numerical format, the median reduction of the relative error over all experiments is 30% and 75% respectively, while using *Alman & Rao*'s algorithm leads to a 50% increase of the relative error. All implementations are available as free software: https://github.com/erwinia42/FWHT.

Keywords: FWHT, WHT, GPU, CUDA, Kahan, Neumaier, Floating Point Errors

Acknowledgements

I would like to thank my supervisor Matti Karppa, both for the excellent insights during this thesis and for giving me the opportunity to previously work on *Pagh*'s algorithm, which led to this idea.
I would also like to thank the Computer Science and Engineering department for providing me access to the computational resources required for this project.

Joel Andersson, Gothenburg, June 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis:

FWHT	Fast Walsh-Hadamard Transform
WHT	Walsh-Hadamard Transform
FFT	Fast Fourier Transform
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FP16	16-Bit Floating Point (IEEE 754 binary16)
FP32	32-Bit Floating Point (IEEE 754 binary32)
FP64	64-Bit Floating Point (IEEE 754 binary64)
BF16	16-Bit Brain Floating Point
RAM	Random-Access Memory
VRAM	Video Random-Access Memory
SM	Streaming Multiprocessor

Contents

List of Figures

List of Tables

1

Introduction

The Walsh-Hadamard Transform (WHT), and the algorithm for computation of it in $O(N \log N)$ time, the Fast Walsh-Hadamard Transform (FWHT) is a mathematical operation on $N = 2^m$ numbers equivalent to Matrix-Vector multiplication with the Hadamard matrix H_m of size $2^m \times 2^m$. It sees widespread use in many fields, including image analysis, quantum computation and, more recently, machine learning [1]. Additionally, The FWHT is equivalent to the multidimensional Fast Fourier Transform (FFT) of size $2 \times 2 \times ... \times 2$ (m dimensions). In implementations where it is possible to change from the FFT to the FWHT, it is beneficial to use the latter, due to the FWHT being computable with only addition and subtraction, requiring no multiplication and no use of complex numbers. This can lead to a significant advantage in performance, as shown by Andersson and Karppa in relation to an approximate matrix multiplication algorithm by Rasmus Pagh [2][3].

Improving the performance of FWHT computation could lead to improvements of many other algorithms, and is a very active research area, in large part due to newfound uses of the FWHT in machine learning. As recently as December 2024, Agarwal et al. published HadaCore [4], a GPU implementation of the FWHT outperforming existing state of the art implementations like the implementation from Dao AI lab [5] that is used by PyTorch. However, Both HadaCore and Dao AI lab's implementation suffer from several limitations. Firstly, they are only implemented for low precision numerical formats with little consideration for numerical stability and are therefore not suitable for some scientific use cases. Secondly, the size of the input is limited to 2^{15} entries, a limitation arising from the fact that the implementations focus on the machine learning use case where a huge amount of small transforms need to be computed [1].

In this project, we are more concerned with computing very large transforms. This approach is less explored, there is some related previous work by Bikov and Bouyukliev [6], but even they only achieve a theoretical maximum size of 2^{20}, and only showed results for sizes up to 2^{18}. We achieve several implementations capable of (in theory) input sizes up to 2^{63}, and show real results for input sizes up to 2^{35}. We also try to make use of recent theoretical improvements in the computation of the FWHT by Alman and Rao [7]. Their algorithm has to our knowledge not previously been practically implemented for GPU execution, or any other platform for that matter.

Additionally we address numerical stability concerns with implementations that

draw heavy inspiration from existing common techniques to mitigate floating point rounding errors when computing sums, namely Kahan summation [8] and Neumaier summation [9]. We also produce a 64-bit floating point implementation, increasing numerical stability simply by increasing the accuracy of the numerical format used, and show that this approach is quite efficient, even on GPUs with no hardware support for FP64 operations.

When it comes to the theoretical improvements from Alman and Rao [7], they showed that the FWHT computation of size 8 requires only 23 operations, a reduction of the previously best known operation count of 24. While they expand their operation set slightly, all of the operations are still fast, as they only introduce bit shifts. We improve on the algorithm, in the sense that we turn the recursive algorithm presented by Alman and Rao into a iterative algorithm much more suited for GPU implementation. We then show that the implementation performs equivalently to the *Folklore* based algorithm in most cases. However, it decreases numerical stability.

We show that the stabilized versions of the *Folklore* based algorithm increase numerical stability in many use cases. Specifically we target the Block Walsh-Hadamard Transform (BWHT) layers from the paper *Fast Walsh-Hadamard Transform and Smooth-Thresholding Based Binary Layers in Deep Neural Networks* by Pan, Dabawi and Cetin [1] and Approximate Matrix Multiplication from work by Pagh [3] and Andersson and Karppa [2]. For both use cases, and using several different input distributions, the techniques decrease the mean relative error by up to 75%.

1.1 Aim of Study

We divided the goal of the project into three categories: "Numerical Stability", "Alman & Rao's Algorithm" and "Larger Input Size". The main goal of the project was to answer the following questions related to those three categories:

- Numerical Stability – Is it possible to increase numerical stability of GPU implementations of the FWHT with existing methods without losing too much performance?
- Alman and Rao's Algorithm – Is it possible to show real world performance gains using Alman and Rao's algorithm for computation of the FWHT compared to the currently used *Folklore* algorithm? How is the numerical stability impacted?
- Larger Input Sizes – Is it possible to make an efficient implementation of the FWHT that lifts the current input size limitations imposed by other implementations such as HadaCore, Dao AI lab and Bikov and Bouyukliev?

1.1.1 Numerical stability

We empirically evaluate the numerical stability (the precision of the output) of all implementations over a wide array of real-world use cases, drawn from both the

usage of the FWHT in BWHT layers in CNNs [1] and Pagh's Algorithm [2][3]. We compare the results to a highly stable 128-bit floating point implementation and show that using techniques heavily inspired by Kahan summation [8] and Neumaier summation [9] in the implementations boost numerical stability without impacting performance heavily.

1.1.2 Alman and Rao's Algorithm

As Alman and Rao argue in their paper [7], since their algorithm only uses bit-shifts and additions it should be possible to implement it fairly efficiently. With a speedup of $1 - \frac{23}{24} \approx 1 - 0.96 = 4\%$ for transforms of size 8, this might seem too insignificant to show any meaningful speedup in implementation, but as the result can be used recursively, in theory it could make a significant difference for large transforms. As an example, if the input size is $2^{21} = 8^7$, we would expect a theoretical speedup of $1 - \left(\frac{23}{24}\right)^7 \approx 1 - 0.74 = 26\%$. While we do not achieve an improvement in practice, we produce an implementation that performs equivalently to the *Folklore* implementation.

1.1.3 Larger Input Sizes

With different design choices than current state of the art, comes different challenges. Recent improvements have focused more on parallelizing computation over the different transforms than over the individual transforms [4][5]. This means they can compute entire transforms within a single CUDA thread-block, requiring no synchronization. We develop an algorithm that scales to theoretically infinite size, only bounded by the size of the index type, which in our implementations is a 64 bit unsigned integer, allowing transform sizes of sizes up to 2^{63}. In practice the much tighter bound is the available memory on the GPU, and thus we can't show results for transforms that large. The largest transforms computed in this thesis were of size 2^{35}.

1.2 Limitations

As the project has limited scope time and resource wise, we must prioritize where to spend time when making implementations. This leads to the following intentional limitations in the design of the algorithms

- We limit ourselves to single GPU implementations, as we expect our implementations will be used as subroutines in larger applications, that themselves may be multi-GPU.
- We also limit the implementations to input sizes that are strictly powers of 2. This simplifies implementation and isn't seen as too huge of a sacrifice as if the input is not, the user may always pad the input with 0:s to get to the next power of 2. It is also not properly defined how to take a FWHT of a input vector which size is not a power of 2.

- As previously mentioned, we limit ourselves to using FP16, FP32, BF16 and FP64 numerical formats.
- The usual ordering of the FWHT is the Hadamard ordered FWHT, also known as the natural ordered FWHT. There also exists another common variation, the Walsh ordered FWHT. This project is only interested in the Hadamard ordered FWHT, as this is the more commonly used of the two.
- All implementations will be made in CUDA [10], as all testbenches available for the project have Nvidia GPUs. We will not provide any implementation for other frameworks, e.g OpenCL [11] or ROCm [12].

1.3 Social and Ethical Aspects

While the technical aspects of this work aim simply aims to enhance computational efficiency and numerical precision of the FWHT, it is important to consider the broader impact on society this could have.

1.3.1 Accessibility and Inclusion

One potential benefit of this work is improving access to high-performance computational tools for smaller organizations or researchers who lack the resources to develop such implementations themselves. By releasing the implementation as free software, this project enables a wide audience to benefit from the work.

1.3.2 Environmental Impact

The work done in this project contributes to computational efficiency, which can reduce energy consumption for large-scale computations. GPUs are known for their high power requirements, and optimizing algorithms to minimize computational complexity and energy usage can lead to less energy being used to perform the same calculations.

On the other hand, the potential scalability of this project to larger input sizes might encourage more computationally intensive tasks, which could, in turn, lead to higher energy consumption in certain applications. This could also simply be a consequence of improving performance of current algorithms, leading to more widespread use of said algorithms.

1.3.3 Machine Learning

The FWHT has become a tool in machine learning, especially for convolutional neural networks dealing with e.g. image processing [1]. These models have both positive and negative applications. While a better FWHT implementation might be used to improve medical imaging, it could also be applied to surveillance technologies or the optimization of weapons systems. By openly sharing this implementation, we have limited control over its use, and risk that the implementations are used to

improve ethically questionable technology. Ultimately, we believe that the potential of improving beneficial technology is worth taking this risk.

2

Technical Background

The required background for the project can be divided into three areas, the Walsh-Hadamard transform, floating point numerical formats and CUDA architecture.

2.1 Kroenecker product

The *Kronecker product* is an operation on two matrices that produces a larger matrix with a structured block form. Given an $m \times n$ matrix A and a $p \times q$ matrix B, their Kronecker product, denoted as $A \otimes B$, results in an $(mp) \times (nq)$ matrix. Each element a_{ij} of A is replaced by the matrix $a_{ij}B$, effectively scaling B by each corresponding entry of A [13].

Example: Consider the matrices

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix}$$

The Kronecker product $A \otimes B$ is computed as follows:

$$A \otimes B = \begin{bmatrix} 1 \cdot B & 2 \cdot B \\ 3 \cdot B & 4 \cdot B \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}$$

2.1.1 Properties of the Kronecker Product

The Kronecker product satisfies several important properties [14], the ones relevant for this thesis are:

Associativity: For any three matrices A, B, and C, the Kronecker product is associative:

$$(A \otimes B) \otimes C = A \otimes (B \otimes C).$$

Non-commutativity: In general, the Kronecker product is **not** commutative, meaning:

$$A \otimes B \neq B \otimes A.$$

Mixed-product property: If A, B, C, D are matrices of appropriate sizes such that the matrix multiplications AC and BD are defined, then:

$$(A \otimes B)(C \otimes D) = (AC) \otimes (BD).$$

2.2 Walsh-Hadamard Transform

The Walsh-Hadamard Transform (WHT), is an operation transforming data into *Hadamard-space* to enable simpler handling of data for certain operations. The definition of the WHT is a Matrix-Vector multiplication with the Hadamard matrix. The Hadamard matrix H_n of square size $n = 2^m$ is recursively defined through the Kronecker product

$$H_1 = \begin{bmatrix} 1 \end{bmatrix}$$
$$H_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$
$$H_n = H_2 \otimes H_{n/2}$$

Or alternatively as a block matrix of smaller Hadamard matrices

$$H_n = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{n/2} & H_{n/2} \\ H_{n/2} & -H_{n/2} \end{bmatrix}$$

For the remainder of the thesis, the scaling factor is omitted for convenience (if not specified otherwise) as it can simply be applied to all elements as a scalar multiplication with $\frac{1}{\sqrt{2}^{\log_2(n)}}$ after the transform has completed. The Hadamard matrix is orthogonal, which gives it the nice property that the inverse matrix H_n^{-1} is simply the Hadamard matrix of the same size. This also means that the Walsh-Hadamard transform is its own inverse, which in turn means an implementation of the FWHT enables transformation both to and from *Hadamard-space*.

2.2.1 Fast Walsh-Hadamard Transform

Obviously, it is possible to compute the Walsh-Hadamard Transform on an input of size n as the full Matrix-Vector multiplication with the explicit Hadamard matrix of size $n \times n$. However, this is computationally inefficient, taking $\Theta(n^2)$ time, and also takes up a lot of space by explicitly storing the Hadamard matrix. To address this issue, we may decompose the Hadamard matrix using the mixed-product property of the Kroenecker product [15].

$$H_n = H_2 \otimes H_{n/2} = (I_2 H_2) \otimes (H_{n/2} I_{n/2}) = (I_2 \otimes H_{n/2})(H_2 \otimes I_{n/2})$$

$$H_n = \begin{bmatrix} H_{n/2} & 0_{n/2} \\ 0_{n/2} & H_{n/2} \end{bmatrix} \begin{bmatrix} I_{n/2} & I_{n/2} \\ I_{n/2} & -I_{n/2} \end{bmatrix}$$

Multiplying the input vector with the right matrix can be done in $O(n)$ time, as the matrix contains exactly 2 non-zero elements per row. Additionally, for an input

vector v every element $j \leq \frac{n}{2}$ can be paired up with element $j + \frac{n}{2}$ with the entries of the output vector v' being

$$v'_j \leftarrow v_j + v_{j+\frac{n}{2}}$$

$$v'_{j+\frac{n}{2}} \leftarrow v_j - v_{j+\frac{n}{2}}$$

The fact that only a pair of values is needed to compute the new value of both the entries, means the transform can very easily be computed in-place. The computation of the FWHT is very similar to the Fast Fourier Transform (FFT), which also relies on this structure, called *radix-2 butterflies* [16]. The only difference being that in the FFT, a multiplication with a complex root of unity ω_n^j is applied to the second entry.

FWHT Radix-2 Butterfly

$$\begin{aligned} x &\leftarrow v_j \\ y &\leftarrow v_{j+\frac{n}{2}} \\ v_j &\leftarrow x + y \\ v_{j+\frac{n}{2}} &\leftarrow x - y \end{aligned}$$

FFT Radix-2 Butterfly

$$\begin{aligned} x &\leftarrow v_j \\ y &\leftarrow v_{j+\frac{n}{2}} \cdot \omega_n^j \\ v_j &\leftarrow x + y \\ v_{j+\frac{n}{2}} &\leftarrow x - y \end{aligned}$$

Finally, to apply the left matrix, we recursively apply the transform of size $n/2$ to the two halfs of the vector. As this needs to be done $\log_2(n)$ times, the final running time for the algorithm is $O(n \log n)$. This approach written out as pseudo-code is provided in Algorithm 1, along with an illustrative example in Figure 2.1.

Algorithm 1 Recursive FWHT

```
1:  function FWHT(x, n, s)
2:      if n = 1 then
3:          return
4:      end if
5:      m ← n/2
6:      for i ← s to s + m − 1 do
7:          a ← x[i]
8:          b ← x[i + m]
9:          x[i] ← a + b
10:         x[i + m] ← a − b
11:     end for
12:     FWHT(x[s : s + m − 1], m, s)
13:     FWHT(x[s + m : n], m, s + m)
14: end function
```

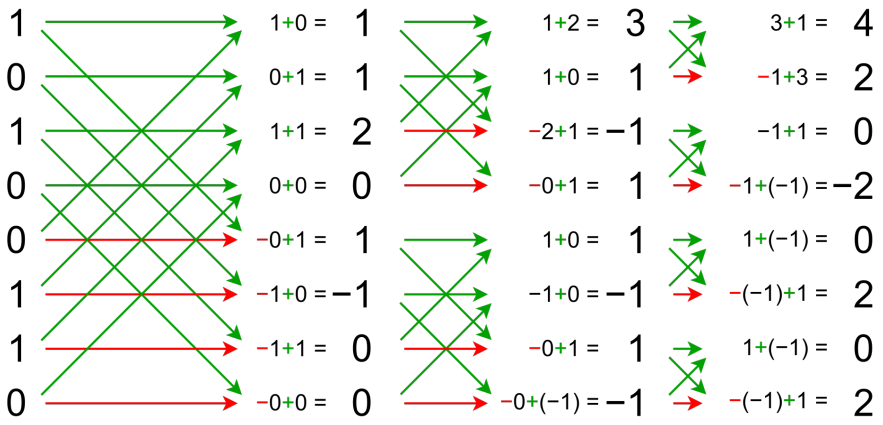

Figure 2.1: Example calculation of the FWHT using Algorithm 1 with the input vector {1, 0, 1, 0, 0, 1, 1, 0}. Green arrows signify addition while red arrows signify subtraction.

The approach for Algorithm 1 applies the largest *butterflies* first. But by slightly modifying the decomposition of the Hadamard matrix from before we can show that we can also start with the smallest *butterflies* and then increase the size.

$$H_n = H_2 \otimes H_{n/2} = (H_2 I_2) \otimes (I_{n/2} H_{n/2}) = (H_2 \otimes I_{n/2})(I_2 \otimes H_{n/2})$$

$$H_n = \begin{bmatrix} I_{n/2} & I_{n/2} \\ I_{n/2} & -I_{n/2} \end{bmatrix} \begin{bmatrix} H_{n/2} & 0_{n/2} \\ 0_{n/2} & H_{n/2} \end{bmatrix}$$

Thus we could first recurse and then apply the large butterflies. Additionally, if we reapply the rule several times, we get H_n as a product of $\log_2(n)$ sparse matrices of size $n \times n$.

$$H_n = (H_2 \otimes I_{n/2})(I_2 \otimes ((H_2 \otimes I_{n/4})(I_2 \otimes H_{n/4})) =$$

$$(H_2 \otimes I_{n/2})(I_2 I_2 \otimes ((H_2 \otimes I_{n/4})(I_2 \otimes H_{n/4})) =$$

$$(H_2 \otimes I_{n/2})(I_2 \otimes H_2 \otimes I_{n/4})(I_2 \otimes I_2 \otimes H_{n/4})) =$$

$$(H_2 \otimes I_{n/2})(I_2 \otimes H_2 \otimes I_{n/4})(I_4 \otimes H_{n/4}) = ...$$

$$H_n = (H_2 \otimes I_{n/2})(I_2 \otimes H_2 \otimes I_{n/4})...(I_{n/4} \otimes H_2 \otimes I_2)(I_{n/2} \otimes H_2)$$

This decomposition gives us a non-recursive approach, for which pseudo-code is provided as Algorithm 2. Again, note that due to the sparsity of identity matrices, every term in the product is a matrix which has exactly 2 non-zero entries per row, thus every matrix-vector multiplication is computable in $O(n)$ time, meaning Algorithm 2 also runs in $O(n \log n)$ time. This is the usual way to order computations in actual implementations of the FWHT. This algorithm is often called the *Folklore* FWHT algorithm.

Algorithm 2 Non-Recursive FWHT (Folklore FWHT)

```
1: function FWHT(x, n)
2:     m ← 1
3:     while m < n do
4:         for i ← 0 to n − 1 by 2 · m do
5:             for j ← i to i + m − 1 do
6:                 a ← x[j]
7:                 b ← x[j + m]
8:                 x[j] ← a + b
9:                 x[j + m] ← a − b
10:            end for
11:        end for
12:        m ← 2 · m
13:    end while
14: end function
```

2.2.2 Alman and Rao's Algorithm

The *Folklore* implementation was considered state of the art for many decades, but recently Alman and Rao published an improvement where they compute a transform of size 8 with 23 operations, instead of the 24 needed when using the *Folklore* algorithm [7]. Additionally, the operations used are only additions, subtractions and bit shifts [7], these are all fast operations on modern hardware, including Nvidia GPUs.

Alman and Rao's algorithm is based on decomposing the H_8 matrix into a sum of a sparse matrix and a matrix with low rank.

$$H_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}}_{\text{low rank}} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 & 0 & 0 & 2 \\ 0 & 2 & 2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 2 & 0 & 2 & 2 & 0 \end{bmatrix}}_{\text{sparse}}$$

To simplify notation, we call the elements of the input vector of length 8: a, b, c, d, e, f, g, h. First, double all entries except a ($b^* = 2b, c^* = 2c, \cdots, h^* = 2h$). We may then compute the matrix-vector multiplication by making intermediary sums

$$B_1 = b^* + c^*$$

,

$$B_2 = d^* + h^*$$

,

$$B_3 = f^* + g^*$$

Then, for the low rank matrix, we compute $tot = B_1 + B_2 + B_3 + e^* = 2(b + c + d + e + f + g + h)$, we then divide tot by 2, or in practice, bit-shift it one step to the right, $tot^* = \frac{tot}{2}$. Considering only the low rank matrix, the first entry of the output vector is then simply $a + tot^*$, and the remaining entries are $a - tot^*$, call the second sum $diff$. To combine with the sparse matrix, compute additional intermediary sums using $diff$, $D = diff + d^*$, $E = diff + e^*$, $H = diff + h^*$. Finally, these sums can be used to compute the remaining entries of the output vector, call these $b', c', d', e', f', g', h'$.

$$b' = E + c^* + g^*$$
$$c' = E + b^* + f^*$$
$$d' = E + B_2$$
$$e' = D + B_1$$
$$f' = H + c^* + f^*$$
$$g' = H + b^* + g^*$$
$$h' = D + B_3$$

Using this, we make 30 operations, but 7 of these are doubling input entries. If we disregard these, we get the claimed 23 operations. What remains is to show that we can construct a recursive implementation where all entries in the input except the first is already multiplied by 2. To this end, Alman and Rao keep track of how much each entry should be scaled with in the base case, and as this is a power of 2, the final scaling can always be computed as a single bit shift. Finally, if the size does not allow computation with only H_8 matrices, the base case is to employ *Folklore* when the recursion gets to a size ≤ 4. This makes the total operation count for a transform of size N using Alman and Rao's algorithm

$$\left(\frac{23}{24} N \log N + \frac{N}{24}(\log N \ \% \ 3) + N - 1\right) \ [7]$$

The first term is due to the H_8 applications, the second from application of 0, 1 or 2 steps of the *Folklore* algorithm and $N - 1$ from having to scale all elements except the first entry in the vector. The recursive algorithm is provided as Algorithm 3 [7].

2.3 Floating Point Numerical Formats

To store fractional numbers with infinite precision would require infinite memory, therefore some standards on how to store fractional numbers in a given number of bits have existed for a long time. This is accomplished through floating point numbers, that store the number using a coefficient (mantissa) $M \in [1, c)$ and an exponent E for some given base c. As an example, to store x we find M and E such that

$$x \approx M \cdot c^E$$

The most common standard used for floating point numbers is IEEE 754 [17]. The standard defines several floating number representations of a fractional numbers

Algorithm 3 Alman and Rao's implementation of FWHT [7]

function $FWHT(x, k, n)$
  if $n \leq 4$ **then**
    Scale the inputs by 2^k
    Use folklore $n \log n$ FWHT
    return
  end if
  $a \leftarrow FWHT(x[j]_{j=0}^{N/8-1}, k, n/8)$ ▷ a, b, c, d, e, f, g, h are vectors of length $n/8$
  $b \leftarrow FWHT(x[j]_{j=N/8}^{N/4-1}, k+1, n/8)$
  $c \leftarrow FWHT(x[j]_{j=N/4}^{3N/8-1}, k+1, n/8)$
  $d \leftarrow FWHT(x[j]_{j=3N/8}^{N/2-1}, k+1, n/8)$
  $e \leftarrow FWHT(x[j]_{j=N/2}^{5N/8-1}, k+1, n/8)$
  $f \leftarrow FWHT(x[j]_{j=5N/8}^{3N/4-1}, k+1, n/8)$
  $g \leftarrow FWHT(x[j]_{j=3N/4}^{7N/8-1}, k+1, n/8)$
  $h \leftarrow FWHT(x[j]_{j=7N/8}^{N-1}, k+1, n/8)$
  $B_1 \leftarrow b + c$
  $B_2 \leftarrow d + h$
  $B_3 \leftarrow f + g$
  $tot \leftarrow (B_1 + B_2 + B_3 + e)/2$
  $diff \leftarrow a - tot$
  $D \leftarrow diff + d$
  $E \leftarrow diff + e$
  $H \leftarrow diff + h$
  $x[j]_{j=0}^{N/8-1} \leftarrow a + tot$
  $x[j]_{j=N/8}^{N/4-1} \leftarrow E + c + g$
  $x[j]_{j=N/4}^{3N/8-1} \leftarrow E + b + f$
  $x[j]_{j=3N/8}^{N/2-1} \leftarrow E + B_2$
  $x[j]_{j=N/2}^{5N/8-1} \leftarrow D + B_1$
  $x[j]_{j=5N/8}^{3N/4-1} \leftarrow H + c + f$
  $x[j]_{j=3N/4}^{7N/8-1} \leftarrow H + b + g$
  $x[j]_{j=7N/8}^{N-1} \leftarrow D + B_3$
end function

with varying size and base. For this project we only use the formats using base 2, and the background thus exclusively discusses those. These standards define a representation consisting of three parts, a sign bit, an exponent and a mantissa. The sign bit is simply one bit deciding the sign of the number, with a 1 indicating a negative number. The mantissa field is interpreted as a fractional binary number, if we have k bits in the mantissa, with the i-th bit denoted m_i, we get a coefficient M.

$$M = 1 + \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^{i+1} m_i$$

The leading 1 comes from the standard defining an implicit bit set to 1 at the start of the mantissa. The exponent, if given over n bits, is interpreted as an integer E_I, and together with the "bias" E_{bias} defined by the standard gives the expression for E

$$E = E_I - E_{bias}$$

Two special exponents are defined by the standard, if every bit is set to 1, the number becomes either $\pm\infty$ if all mantissa bits are 0, or NaN if any mantissa bit is 1. Additionally, if every exponent bit is set to 0, the exponent remains $E = 1 - E_{bias}$, but with the implicit bit in the mantissa set to 0. This is done to more accurately represent numbers close to 0. Numbers in this format are called *subnormal*, while numbers with the implicit bit set to 1 are called *normal*. The final representation of a decimal number using floating point numbers becomes

$$\begin{cases} (-1)^s \cdot 2^{E_I - E_{bias}} \cdot (1 + \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^{i+1} m_i) \text{ if } E_I \neq 0 \\ (-1)^s \cdot 2^{1 - E_{bias}} \cdot (0 + \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^{i+1} m_i) \text{ if } E_I = 0 \end{cases}$$

One key feature of this definition is that the relative error remains constant for all *normal* numbers as the distance between two normal numbers with exponent $E = E_I - E_{bias}$ and k bits of mantissa is $\delta = 2^{E-k}$. This gives a relative error of at most 2^{-k}. This is usually called the *Interval Machine Epsilon*, but for the rest of the thesis, we only call it the *Machine Epsilon*, or ε.

The maximum and minimum value of the exponents, and thus the range of representable values is determined by the size of the exponent field, while the precision of the format is determined by the size of the mantissa field. Thus there exists a trade off between range and precision. The partitions decided by IEEE 754 and the subsequent range and precision of each format is shown in Table 2.1 and Table 2.2. The tables also includes the *BFloat16* (BF16) format, which relies on the same principles as IEEE 754 floating point numbers, but partitions the available bits differently, it was introduced by Google Brain as a method to increase computational capabilities in AI applications without reducing the representable range compared to IEEE 754 binary32 (FP32) [18] and has widespread hardware support.

Table 2.1: Floating point numerical formats: exponent and mantissa characteristics.

Format	E bits	M bits	Bias
binary128 (FP128/Quad)	15	112	16383
binary64 (FP64/Double)	11	52	1023
binary32 (FP32/Float)	8	23	127
binary16 (FP16/Half)	5	10	15
BFloat16 (BF16)	8	7	127

Table 2.2: Floating point numerical formats: value ranges and relative precision.

Format	Maximum Value	Minimum Value	Machine Epsilon
binary128 (FP128/Quad)	$\approx 10^{4932}$	$\approx 10^{-4966}$	$\approx 10^{-34}$
binary64 (FP64/Double)	$\approx 10^{308}$	$\approx 10^{-324}$	$\approx 10^{-16}$
binary32 (FP32/Float)	$\approx 10^{38}$	$\approx 10^{-45}$	$\approx 10^{-7}$
binary16 (FP16/Half)	$\approx 10^{5}$	$\approx 10^{-8}$	$\approx 10^{-3}$
BFloat16 (BF16)	$\approx 10^{38}$	$\approx 10^{-45}$	$\approx 10^{-2}$

2.3.1 Error Propagation

The usage of floating point numbers gives rise to rounding errors for each entry, these errors will propagate through calculations, and can accumulate to such a degree that the final answer varies by much more than the given precision of the numerical format used. To illustrate the problem, summation of arrays is often used, and as this is highly relevant when calculating the FWHT, we will do so here as well. When analyzing the quality of an algorithm, the most relevant factor is the relative error, given as the absolute error divided by the absolute value of the real value. Notation wise, we will use a * to indicate the number in floating point representation, and ε to denote the relative error of the numerical format itself, often called machine epsilon. We express the relative error E_r when summing n values as

$$E_r = \frac{E_a}{|S_n|} = \frac{|S_n^* - S_n|}{|S_n|}$$

$$S_n = \sum_{i=1}^{n} x_i$$

Additionally, we put a $fl()$ around any expression that is rounded.

There are many ways to reduce errors when summing arrays, the methods explored in this thesis are Kahan summation and the closely related Neumaier summation. Additionally, the idea of Pairwise summation is relevant as it's implicitly used by the FWHT.

2.3.1.1 Naive Summation

Naive summation, given as Algorithm 4, is our baseline. Analyzing the error bound can be done by recursive expansion. Denote S_k as the sum of the k first elements in a sum containing n elements [19].

$$S_k^* = fl(S_{k+1}^* + x_k) = (S_{k-1}^* + x_k)(1 + \delta_k), \;\; |\delta_k| \leq \varepsilon$$

$$S_n^* = (x_1 + x_2)\prod_{k=2}^{n}(1 + \delta_k) + \sum_{i=3}^{n} x_i \prod_{k=i}^{n}(1 + \delta_k)$$

To simplify calculations, define and bound θ_n

$$\prod_{i=1}^{n}(1 + \delta_i) = 1 + \theta_n, \; |\theta_n| \leq \frac{n\varepsilon}{1 - n\varepsilon}$$

Continue simplification:

$$S_n^* = (x_1 + x_2)(1 + \theta_{n-1}) + \sum_{i=3}^{n} x_i(1 + \theta_{n-i+1})$$

Now we can bound absolute error by:

$$|E_a| = \left|(x_1 + x_2)\theta_{n-1} + \sum_{i=3}^{n} x_i\theta_{n-i+1}\right| \leq |\theta_n| \sum_{i=1}^{n} |x_i| = \frac{n\varepsilon}{1 - n\varepsilon} \sum_{i=1}^{n} |x_i|$$

Which gives a relative error bound of:

$$|E_r| \leq \frac{n\varepsilon}{1 - n\varepsilon} \cdot \frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|} = \frac{n\varepsilon}{1 - n\varepsilon} \cdot \kappa(X)$$

The second fraction is a measurement of the inherent difficulty of stabilizing a given sum, and is called the *Conditioning number* [19], it will show up again in the other summation techniques. It is denoted by $\kappa(X)$ from now on. An important remark here is that for sequences where all numbers have the same sign, the conditioning number becomes 1 and disappears, but if the final result is close to 0, the conditioning number goes to infinity and the bounds are practically useless.

Algorithm 4 Naive Summation

```
1: function NaiveSum(X = x_1, x_2, ..., x_n)
2:     S ← 0.0
3:     for i ← 1 to n do
4:         S ← S + x_i
5:     end for
6:     return S
7: end function
```

2.3.1.2 Kahan Summation

Kahan summation is a summation technique that iteratively applies summations with a running error term e, which is calculated by subtracting contributions from the sum in more than 1 step. Let S_x denote the sum after x entries and y the next term. Then $S^*_{x+1} = fl(S_x+y)$. To recover the error e we compute $e = (S^*_{x+1}-S_x)-y$. This will be a good estimate of the error, assuming $|S_x| \geq |y|$, as the subtraction of the two large values leaves smaller values to subtract in the second step. In fact, when working in base 2 this error perfectly captures the rounding error $S^*_{x+1} - e^* = S_x + y$ [19]. Pseudocode for Kahan summation is given as Algorithm 5

Algorithm 5 Kahan Summation

```
1: function KahanSum(X = x_1, x_2, ..., x_n)
2:     S ← 0.0
3:     e ← 0.0
4:     for i ← 1 to n do
5:         y ← x_i − e
6:         t ← S + y
7:         e ← (t − S) − y
8:         S ← t
9:     end for
10:    return S
11: end function
```

Even though the error is captured perfectly, the reapplication of it ($y \leftarrow x_i - e$) is not, and we are also assuming that $|S_x| \geq |y|$, which while true in many cases, especially if the summands are all positive, is not always a good assumption to make. Nonetheless, Knuth [20] shows that the error is bounded by

$$E_a \leq (2\varepsilon + O(n\varepsilon^2)) \sum_{i=1}^{n} |x_i|$$

and thus the relative error is bounded by

$$E_r \leq (2\varepsilon + O(n\varepsilon^2)) \cdot \kappa(X)$$

2.3.1.3 Neumaier Summation

A simple way to address the issue of assuming the summands are smaller than the running sum, proposed by Neumaier [9], is to check if the condition is met, and reverse the order of subtractions if it is not met. The pseudocode for this is provided as Algorithm 6. There are no direct theoretical benefits to the error bound, but it does always produce at least as good of a result as Kahan summation, and in some sequences it performs better, e.g $[1, 10^{100}, 1, -10^{100}]$ gives 0 with Kahan summation, and the correct answer 2 with Neumaier summation.

Algorithm 6 Neumaier Summation

```
1: function NeumaierSum(X = x_1, x_2, ..., x_n)
2:     S ← 0.0
3:     e ← 0.0
4:     for i ← 1 to n do
5:         y ← x_i − e
6:         t ← S + y
7:         if |S| ≥ |y| then
8:             e ← (t − S) − y
9:         else
10:            e ← (t − y) − S
11:        end if
12:        S ← t
13:    end for
14:    return S
15: end function
```

2.3.1.4 Pairwise Summation

Another simple approach to error reduction is to apply sums pairwise instead of iteratively, as an example, we turn the sum

$$(((x_1 + x_2) + x_3) + x_4)$$

into

$$((x_1 + x_2) + (x_3 + x_4))$$

More precisely, the summing algorithm is given as Algorithm 7

Algorithm 7 Pairwise Summation

```
1: function PairSum(X = {x_1, x_2, ..., x_n})
2:     if |X| == 1 then
3:         return x_1
4:     end if
5:     X_1 ← x_1, x_2, ..., x_{n/2}
6:     X_2 ← x_{n/2+1}, x_{n/2+2}, ..., x_n
7:     S ← PairSum(X_1) + PairSum(X_2)
8:     return S
9: end function
```

We can derive theoretical guarantees in a similar manner as for Naive summation, but we need some additional notation. If we view the summations as a tree of $\log n$ layers, with $\frac{n}{2^k}$ summations in layer k, we can denote the partial sums at this layer as $S_{i:k}, i \in \{1, 2, ..., \frac{n}{2^k}\}$. For convenience we assume that the sum always consists of 2^m entries for some positive integer m. An example of a summation of 8 entries is provided as Figure 2.2.

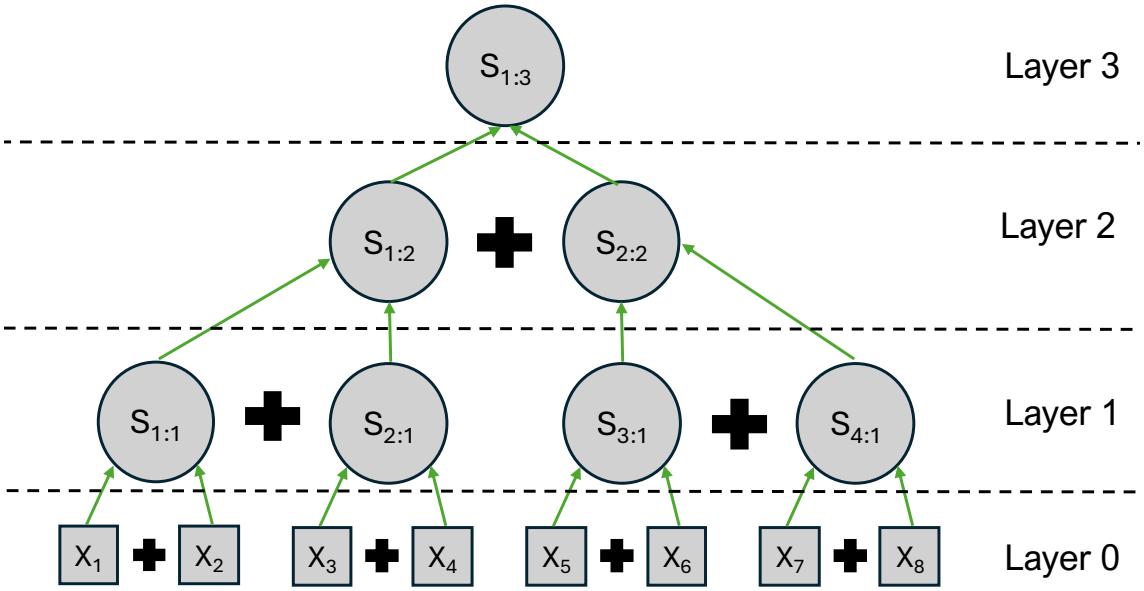

Figure 2.2: Illustration of setup for pairwise summation on a vector of length 8, including notation of partial sums. The final sum will be the lone node at the top layer.

In general, the partial sum is then calculated as

$$S_{i:1} = x_{2i-1} + x_{2i}$$

$$S_{i:k} = S_{2i-1:k-1} + S_{2i:k-1}$$

and the floating point representation as

$$S^*_{i:1} = fl(x_{2i-1} + x_{2i}) = (x_{2i-1} + x_{2i})(1 + \delta_{i:1}),\ |\delta_{i:1}| \leq \varepsilon$$

$$S^*_{i:k} = fl((S^*_{2i-1:k-1} + S^*_{2i:k-1})) = (S^*_{2i-1:k-1} + S^*_{2i:k-1})(1 + \delta_{i:k}),\ |\delta_{i:k}| \leq \varepsilon$$

Applying this recursively we get an expression for the result $S_{1:\log n}$

$$S_{1:\log n} =$$

$$\begin{aligned}
&x_1(1+\delta_{1:1})(1+\delta_{1:2})(1+\delta_{1:3})\cdots(1+\delta_{1:\log n-1})(1+\delta_{1:\log n})+\\
&x_2(1+\delta_{1:1})(1+\delta_{1:2})(1+\delta_{1:3})\cdots(1+\delta_{1:\log n-1})(1+\delta_{1:\log n})+\\
&x_3(1+\delta_{2:1})(1+\delta_{1:2})(1+\delta_{1:3})\cdots(1+\delta_{1:\log n-1})(1+\delta_{1:\log n})+\\
&x_4(1+\delta_{2:1})(1+\delta_{1:2})(1+\delta_{1:3})\cdots(1+\delta_{1:\log n-1})(1+\delta_{1:\log n})+\\
&x_5(1+\delta_{3:1})(1+\delta_{2:2})(1+\delta_{1:3})\cdots(1+\delta_{1:\log n-1})(1+\delta_{1:\log n})+\\
&\vdots\\
&x_n(1+\delta_{\frac{n}{2}:1})(1+\delta_{\frac{n}{4}:2})(1+\delta_{\frac{n}{8}:3})\cdots(1+\delta_{2:\log n-1})(1+\delta_{1:\log n})
\end{aligned}$$

More concisely, we can express the sum as

$$S_{1:\log n} = \sum_{i=1}^{n} x_i \prod_{j=1}^{\log n} (1 + \delta_{(\lfloor \frac{i-1}{2^j} \rfloor + 1):j})$$

Bounding the product the same way as for naive summation, we get a tighter bound

$$\prod_{j=1}^{\log n} (1 + \delta_{(\lfloor \frac{i-1}{2^j} \rfloor + 1):j}) = 1 + \theta_i,\ |\theta_i| \leq \frac{(\log n)\varepsilon}{1 - (\log n)\varepsilon}$$

$$S_{1:\log n} = \sum_{i=1}^{n} x_i(1 + \theta_i)$$

Which leads to a tighter error bound

$$|E_a| = \left| \sum_{i=1}^{n} x_i \theta_i \right| \leq \frac{(\log n)\varepsilon}{1 - (\log n)\varepsilon} \sum_{i=1}^{n} |x_i|$$

$$|E_r| \leq \frac{(\log n)\varepsilon}{1 - (\log n)\varepsilon} \cdot \kappa(X)$$

Since the FWHT is built upon recursive additions, the transform is stabilized inherently by exactly this technique. This means that the transform is already fairly stable when $\kappa(X)$ is small.

2.4 CUDA Model

The idea of GPU computation is highly parallelized execution across many cores. Programming such a device can become very complicated, thus Nvidia developed the *Compute Unified Device Architecture* (CUDA) model [10], which abstracts away several layers for the programmer when working with their GPUs. The main idea of CUDA is *Single Instruction Multiple Data* (SIMD) execution, meaning that on a hardware level the same instruction is executed over groups of 32 threads, called warps, where each thread may hold different data. On a software level, threads are bundled into blocks that all execute the same instructions, without explicit synchronization between threads in the block. As warps are used on a hardware level, there is a benefit to use block sizes that are multiples of 32. The blocks are assigned to Streaming Multiprocessors (SMs) that schedule execution of the threads onto the available CUDA Cores. While the threads within a block aren't implicitly synchronized, CUDA provides easy explicit synchronization of threads in a block through the *___syncthreads()* directive. However, the blocks themselves must be able to independently execute, with no possibility of inter-block synchronization.

2.4.1 Memory Hierarchy

Due to the very high computational capabilities of GPUs, memory constraints are often the bottleneck when implementing algorithms. It is therefore important to understand the memory model used in CUDA. There are several levels of data

memory, in decreasing order of distance to the CUDA cores, they are: host memory (RAM), global memory (VRAM), L2 Cache and L1 Cache/Shared Memory (SRAM). In addition to these, there are also several levels of instruction memory, which are not discussed in this thesis. The host memory refers to the RAM accessible by the CPU, this memory is physically very far away from the GPU, and connected to the GPU with low bandwidth. To use data stored in host memory it must first be transferred to the global memory of the GPU. The global memory is much larger than any other part of the memory hierarchy, and is placed outside of the GPU chip, but on the same circuit board. On the GPU chip itself are L2 caches, and in each SM there is a small memory unit that serve as both L1 cache and shared memory. Shared memory is shared by all threads in a single block. The size of the shared memory is configurable through the CUDA API, with the remainder of the memory unit serving as L1 cache, at least 1 KB of data must be configured for this purpose [10]. This means that even when the data resides on the GPU, the implementation may be memory bound as data needs to move from the global memory to shared memory before it can be used.

2.4.2 SM Occupancy

As previously mentioned, there is an upper limit on the number of blocks that may be assigned to a single SM, it differs based on architecture (L4/L40s: 24, H100: 32) [10]. However, this limit is often not hit in practice, as other factors also limit the number of blocks per SM. The limiting factors relevant for this thesis are shared memory and warps. To achieve optimal performance, it is recommended to have a kernel be limited by the number of warps per SM, as this gives the SM the best chance to hide memory latencies by switching what warp is executing [10]. To give a measure of how well this is achieved, there is a concept of SM *Occupancy*, calculated as

$$\textbf{Occupancy} = \frac{\textbf{Theoretical Warps per SM}}{\textbf{Maximum Warps per SM}}$$

The maximum amount of warps that may be scheduled onto the same SM varies by architecture (L4/L40s: 48, H100: 64), and is part of the tuning guides for each architecture, the tuning guide also includes data on the SMs register file size, the maximum number of registers per thread and the shared memory capacity of the SM.

3

Method

For all implementations, the work was carried out in three stages: "Development", "Numerical Stability Evaluation" and "Performance Evaluation". While they are presented separately here, some testing of the implementation was of course necessary during development as well. However, the method chapter intends to describe how the empirical evaluation that is presented in the results was performed.

3.1 Minerva Cluster

The Minerva Cluster is a compute cluster owned and maintained by Chalmers Computer Science and Engineering department. All empirical evaluations were conducted on the nodes of the Minerva Cluster. The cluster features seven GPU accelerated nodes in three different configurations, all equipped with Intel Xeon processors and powerful Nvidia GPUs, the technical details of all setups can be found in Table 3.1, with further details about the GPUs in Table 3.2 and Table 3.3.

3.2 Main Implementation Idea

One of the main goals of the project was to expand the allowed input size of the implementation. The approached used to accomplish this was to construct a kernel that dynamically reinterprets the data based on the progress that has been made by previous computation. For a transform of size n, we say we need to compute $\log n$ steps, where each step corresponds to applying one of the matrices in the product decomposition of the Hadamard matrix. In other words, each step applies all radix-2 butterflies with size 2^{steps_made+1}.

As applying the steps one by one would be very inefficient, we want each CUDA thread-block to compute as many steps as possible. The implementation therefore fetches data from global memory, loads it into a chunk of shared memory and does

Table 3.1: Technical data of Testbenches

#Nodes	CPU	RAM	GPU
4	Intel Xeon GOLD 6548N (2 × 64 cores)	512 GiB	8× Nvidia L4
2	Intel Xeon GOLD 6548N (2 × 64 cores)	512 GiB	4× Nvidia L40s
1	Intel Xeon GOLD 6548N (2 × 64 cores)	1 TiB	4× Nvidia H100 NVL

Table 3.2: Technical data of GPUs

GPU	VRAM	Cuda Cores	SMs	Tensor Cores
L4	24 GiB	7424	60	240
L40s	48 GiB	18176	142	568
H100 NVL	94 GiB	14592	114	456

Table 3.3: Compute Capabilities of GPUs (TFLOPS)

GPU	FP64	FP64 Tensor Core (TC)	FP32	FP16 TC	BF16 TC
L4	N/A	N/A	30.3	242	242
L40s	N/A	N/A	91.6	733	733
H100 NVL	30	60	60	1671	1671

block_steps many steps locally, before writing the data back to global memory. The problem that remains is, what data should each block load into memory to compute step *steps_made + 1* to *steps_made + block_steps*?

Essentially, we need to fetch 2^{block_steps} elements, at stride 2^{steps_made}, with each block starting at

$$a \cdot 2^{block\_steps} + b$$

$$a \in \left\{0, 1, ..., \frac{n}{2^{block\_steps} \cdot 2^{steps\_made}}\right\}, b \in \{0, 1, ..., 2^{steps\_made}\}$$

This can be visualized by treating the data as a matrix in row-major format with n entries and 2^{steps_made} columns. Thus at the start we have a column vector, and split it into chunks of size 2^{block_steps}. With any arbitrary number of steps made, we have 2^{steps_made} columns, and for each column, we split it into chunks of size 2^{block_steps}. In the last kernel call, we may be in a situation where we have to make less than *block_size* steps, meaning each column of the matrix becomes it's own chunk. In this situation we can load several chunks into one block, and stop computation after the correct amount of steps have been made, this reduces the number of blocks used, boosting performance, but pragmatically it is the same as if each small chunk would get assigned to its own block.

The idea is visualized in Figure 3.1. Here we assume the blocks may only compute a transform of size 3, meaning $block\_steps = 2$. In the actual implementation, the sizes are much bigger.

Given a kernel that splits the data into these chunks and applies *block_steps* steps, we may write pseudocode that more closely resembles the actual implementations. This high level implementation is given as Algorithm 8 and it describes the host code for all implementations[1].

[1]In the actual implementations various additional information must be passed along to the kernel as well, this varies by implementation, but is not discussed here, as it doesn't impact the implementation idea

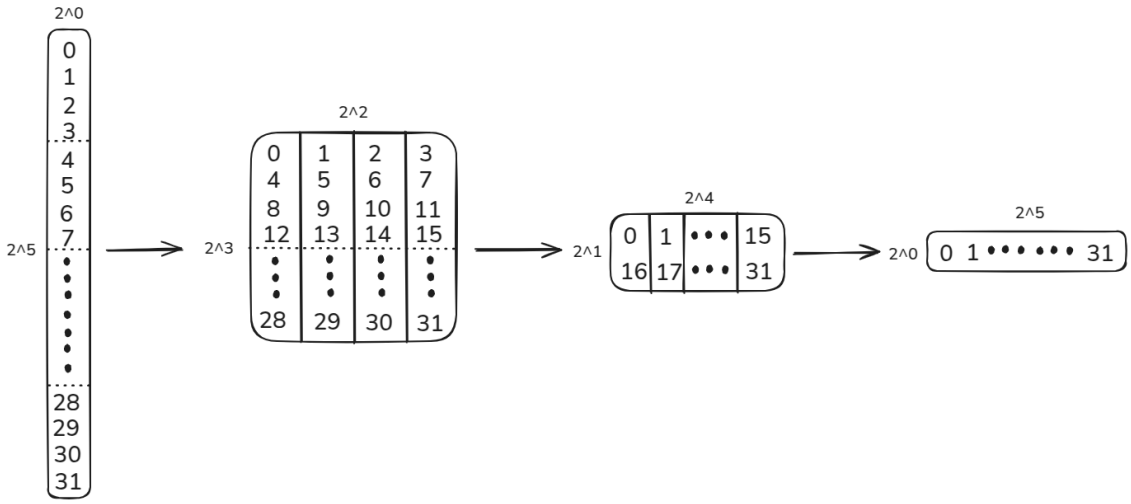

Figure 3.1: Visualization of idea through computation of a transform of size 2^5 with *block_steps* $= 2$. Numbers indicate index of item, in each step, the items are grouped into chunks of 4 elements, then a Hadamard transform is applied to the data in each chunk. When all blocks have finished computation, the elements are regrouped by reinterpreting the data as a wider matrix, before computation continues. This process repeats until what was initially a column vector of size 2^5, has become a row vector of size 2^5. This way we can compute the larger transform as a sequence of smaller transform.

Algorithm 8 Host Pseudocode For All Implementations

```
1:  function FWHT(x, n)
2:      Copy host memory at x to device memory at device_x
3:      total_steps ← log n
4:      steps_made ← 0
5:      block_steps ← Implementation Specific Constant
6:      while steps_made + block_steps ≤ total_steps do
7:          KERNEL(device_x, steps_made, block_steps)
8:          CUDADEVICESYNCHRONIZE()        ▷ Wait for kernel to finish executing
9:          steps_made ← steps_made + block_steps
10:     end while
11:     remaining_steps ← total_steps − steps_made
12:     if remaining_steps > 0 then
13:         KERNEL(device_x, steps_made, remaining_steps)
14:         CUDADEVICESYNCHRONIZE()        ▷ Wait for kernel to finish executing
15:     end if
16:     Copy device memory at device_x to host memory at x
17: end function
```

3.3 Folklore Implementation

The folklore implementation was built in two variations, firstly, a simple variant in which each thread maps uniquely to one index in the array, meaning each thread processes one element. This naturally leads to a limit on the block_size corresponding to the maximum number of threads per block, which for most GPUs is 1024 [10]. This means that the number of steps the kernel can make is at most $\log_2(1024) = 10$, in practice, blocks of this size are impractical, as it heavily limits the amount of blocks that may be assigned to each SM. The most commonly used block sizes are $128, 256$ and 512. Corresponding to *block_steps* values of $7, 8$ and 9. For experimental evaluation, the block size was set at 256 as this showed the best performance during initial testing.

The second variation is instead restricted by the shared memory usage per block, which is limited to 32 KiB/Block on most Nvidia GPUs[2]. This increases the size of the chunks greatly, especially for smaller datatypes. Allowing 2^{12} doubles, 2^{13} floats or 2^{14} halfs/bfloat16:s, corresponding to *block_steps* values of 12, 13 and 14. It also decouples the size of the chunks from the size of the blocks, meaning we are more free to set the block size. However, for experimental evaluation the block size for this variation was also set at 256.

3.3.1 Kahan stabilized

We can't directly apply Kahan summation to all sums, as we are not doing iterative addition. However, we can draw heavy inspiration from the ideas, and construct a similar stabilization technique for the FWHT. The main difference to iterative summation is that both the summands in each summation are error prone as they are constructed from previous sums. To address this, we look at the problem from a per butterfly perspective. Here we have four terms, A and B, the summands of the current summation, and e_A and e_B, the errors corresponding to the two summands from previous steps (how these are calculated is described later). The goal is to calculate A' and B', the values for the next step of the transform at the same index as A and B. They are calculated as

$$A' = A + B - (e_A + e_B)$$

$$B' = A - B - (e_A - e_B)$$

The concrete order of operations is to first combine the errors, then the "main" summands, and finally add the error to that result.

$$A' = (A + B) - (e_A + e_B)$$

$$B' = (A - B) - (e_A - e_B)$$

This procedure is illustrated in Figure 3.2. The errors are stored in a separate vector of equivalent size to the input vector, and tracks the current error estimate for the entry at the corresponding index.

[2]This is not entirely true, it is possible to reconfigure the limit to larger values. However, this serves little purpose, as increasing it to the next power of 2 (64 KiB) would limit occupancy greatly.

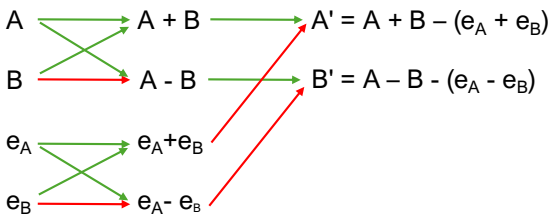

Figure 3.2: Example of a butterfly operation using the Kahan stabilized FWHT algorithm

To calculate the new errors e'_A, e'_B, we undo the summation in the following steps

$$e'_A = ((A' - A) - B) + (e_A + e_B)$$

$$e'_B = ((B' - A) + B) + (e_A - e_B)$$

3.3.2 Neumaier stabilized

The error retrieval in the Kahan stabilized implementation is good in the cases where the following holds $|A|' \geq |A| \geq |B| \geq |e_A + e_B|$ and $|B'| \geq |A| \geq |B| \geq |e_A - e_B|$. However, this is of course not always the case. To handle this, add conditions on the ordering of summation, just like in Neumaier summation, to ensure that summands of similar absolute size are summed first. Practically, we first make a reasonable assumption that $|e_A + e_B|$ and $|e_A - e_B|$ are small in comparison to the other terms, then we sort the three remaining terms A'/B', A and B by absolute size and sum the largest values first. The errors may then expressed in the following manner

$$e'_A = \begin{cases} ((A' - A) - B) + (e_A + e_B), & \text{if } |A'|, |A| \geq |B| \\ ((A' - B) - A) + (e_A + e_B), & \text{if } |A'|, |B| \geq |A| \\ ((-A - B) + A') + (e_A + e_B), & \text{if } |B|, |A| \geq |A'| \end{cases}$$

$$e'_B = \begin{cases} ((B' - A) + B) + (e_A - e_B), & \text{if } |B'|, |A| \geq |B| \\ ((B' + B) - A) + (e_A - e_B), & \text{if } |B'|, |B| \geq |A| \\ ((-A + B) + B') + (e_A - e_B) & \text{if } |A|, |B| \geq |B'| \end{cases}$$

3.4 Alman Implementation

While Alman and Rao provide good arguments as to why their theoretical results should be implementable efficiently, there were two main problems with the algorithm left to solve to make an actual implementation on GPU hardware. Firstly, the algorithm provided is recursive, reducing performance unnecessarily, and making it hard to implement in CUDA, which requires a split up block structure. This problem was rather easily solved by turning the algorithm into a non-recursive one. Initially we assumed we knew the scaling factors for each element, and had access to a scaling function based on the index (and the number of initial *Folklore*_steps)

$alman\_scale(x, i, folklore\_steps)$. The non-recursive implementation in this case is provided as Algorithm 9.

Implementing scaling was more problematic, while it is true that the scaling itself can be applied in constant time per element, just as argued by Alman and Rao, they provide no solution for how to determine how much each element should be scaled. Even in their own recursive implementation they disregard the integer summation when increasing the exponent for 7 out of the 8 recursive calls. Accounting for this operation, the computational advantage disappears. While it is not a fundamental operation, and thus Alman and Rao's theoretical result remains true, practically we found no way around this. Thus we settled on using a scaling function which runs in $O(\log n)$. The pseudocode for *alman_scale* is provided as Algorithm 10, it breaks down the recursive pattern by observing that the exponent will increase by 1 for each triplet of bits that is non-zero. Mathematically, we take each index modulo 8, if the result is not 0, we increase the exponent. We then integer divide the index by 8, and repeat the process until the index reaches 0.

3.5 Numerical Stability Evaluation

All implementations were tested against a 128-bit floating point CPU implementation, which is considered to output the true answer to a very low fault-tolerance. Measurements for largest deviation, mean deviation and mean deviation were collected for all implementations over a wide range of input classes, described in Section 3.5.1. For every class of inputs, measurements for all four relevant representations were collected, meaning, in descending order of precision FP64, FP32, FP16 and BF16. Due to the small range of the FP16 representation, some experiments do not show data for FP16 for all sizes of input. Data was collected for all input sizes that are powers of 2, from $n = 2^3$ to $n = 2^{25}$. For every size the same input data is used by all numerical formats (up to rounding).

3.5.1 Input Data Selection

When selecting input data, we mainly concern ourselves with two use cases of the FWHT, Pagh's matrix multiplication [2][3] and the substitution of 1×1 convolutional layers in MobileNetV2 [1]. In the first use case, we expect fairly sparse vectors as input. These vectors are constructed by selecting n entries of a matrix (either a row or a column), applying a random sign to each entry, and dispersing them out to a vector of size cn, with $c \geq 2$. For the experiments in this thesis, we chose $c = 8$ as this was frequently used in the previous work [2]. This means at least $\frac{7}{8}$ of the entries in the input vector to the FWHT will be 0. The matrix we draw the values from could contain any distribution of values, we decided to use normal distribution and uniform selection from {-1, 1} to reflect previous work on the topic [2].

For the second use case we examine the layers of MobileNetV2 [21], and find that before every 1×1 convolutional layer, there is either a normalization layer, or a normalization layer followed by a weightless ReLU layer. This means the input is clearly defined as either normal distributed, or normal distributed with all values $<$

Algorithm 9 Non-Recursive Alman FWHT

```
1:  function FWHT(x, n)
2:      m ← 1
3:      step ← 0
4:      folklore_steps ← log_2 n % 3
5:      alman_steps ← ⌊log_2 n/3⌋
6:      while step < folklore_steps do
7:          for i ← 0 to n − 1 by 2 · m do
8:              for j ← i to i + m − 1 do
9:                  a ← x[j]
10:                 b ← x[j + m]
11:                 x[j] ← a + b
12:                 x[j + m] ← a − b
13:             end for
14:         end for
15:         m ← 2 · m
16:         step ← step + 1
17:     end while
18:     for i ← 0 to n − 1 do
19:         alman_scale(x, i, folklore_steps)
20:     end for
21:     step ← 0
22:     while step < alman_steps do
23:         for i ← 0 to n − 1 by 8 · m do
24:             for j ← i to i + m − 1 do
25:                 a ← x[j]
26:                 b ← x[j + m]
27:                 c ← x[j + 2m]
28:                 d ← x[j + 3m]
29:                 e ← x[j + 4m]
30:                 f ← x[j + 5m]
31:                 g ← x[j + 6m]
32:                 h ← x[j + 7m]
33:                 B_1 ← b + c
34:                 B_2 ← d + h
35:                 B_3 ← f + g
36:                 tot ← (B_1 + B_2 + B_3 + e)/2
37:                 diff ← a − tot
38:                 D ← diff + d
39:                 E ← diff + e
40:                 H ← diff + h
41:                 x[j] ← a + tot
42:                 x[j + m] ← E + c + g
43:                 x[j + 2m] ← E + b + f
44:                 x[j + 3m] ← E + B_2
45:                 x[j + 4m] ← D + B_1
46:                 x[j + 5m] ← H + c + f
47:                 x[j + 6m] ← H + b + g
48:                 x[j + 7m] ← D + B_3
49:             end for
50:         end for
51:         m ← 8 · m
52:         step ← step + 1
53:     end while
54: end function
```

Algorithm 10 Alman Scaling Using Bit-shifts

```
function alman_scale(x, i, folklore_steps)
    exponent ← 0
    i_original ← i
    while i > 0 do
        if i & 7 ≠ 0 then                          ▷ bitwise-and
            exponent ← exponent + 1
        end if
        i ← i >> 3                                 ▷ bitwise right shift
    end while
    x[i_original] << exponent
end function
```

Table 3.4: Input Distributions (PAGH scattering: Values are drawn from the given distribution and added to random indices in a zero vector with a random sign, repeating $n/8$ times.)

Short Name	Description
PMONE	Randomly selects either -1 or 1 for each entry with equal probability
NORM	Samples from a standard normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$
RELU_NORM	Applies the ReLU function (`x = max(0, x)`) to values drawn from NORM
PAGH_NORM	Applies PAGH scattering of values drawn from NORM
PAGH_PMONE	Applies PAGH scattering of values drawn from PMONE

0 set to 0. This reasoning was further verified experimentally by measuring the activation values before each 1×1 convolutional layer when giving the network an input from it's test dataset and plotting a Kernel Density Estimation of the result. This is shown in Figure 3.3.

In addition to these practical selections, we want to test the implementation on one additional dense input. For this we chose to draw each input uniformly at random from the set $\{-1, 1\}$.

Note that for all rows of the Hadamard matrix except the first, exactly half of the entries are 1, with the remaining entries being -1. This means that if the entries in the input vector are drawn independently from each other the expected output is 0 for all entries except the first, no matter the distribution. This means that the conditioning number is infinite in expectation, and thus the theoretical guarantees presented previously do not hold.

To summarize, this gives us 5 input classes, which are given a short name in Table 3.4. These names will be used to refer to each distribution from now on.

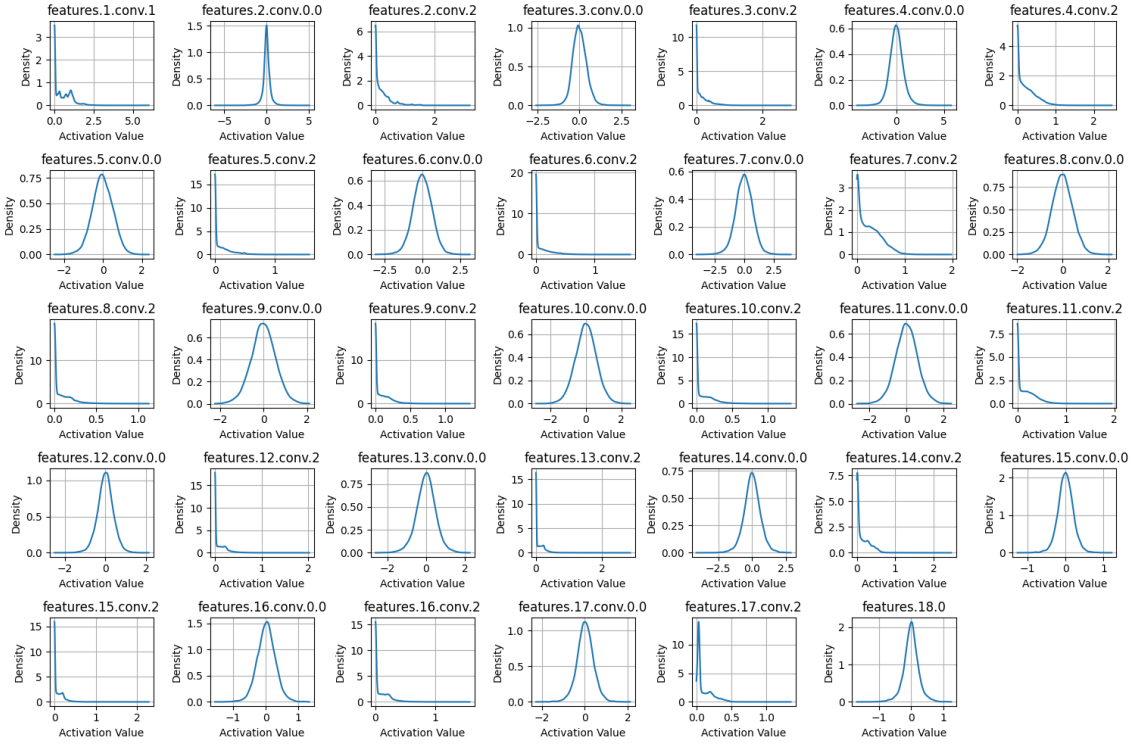

Figure 3.3: Kernel Density Estimation (KDE) plot of activation values before each 1×1 convolution layer in MobileNetV2. The analysis confirms that activations are either normally distributed or follow a rectified normal distribution (normal distribution with values < 0 set to 0), as expected by the analysis of the layers.

3.5.2 Operation Selection

In addition to populating the input vector with values drawn from various distributions, we also want to test the stability when the transforms are part of larger operations. To achieve this we use each distribution to perform 4 different tests. The two first tests are general tests of accuracy after performing one transform, and after two transforms + normalization. The later shouldn't change the input vector at all, since the transform is it's own inverse. For more practical applications we consider performing operations on the vectors in *Hadamard-space* and then transforming the vector back. We once again use examples from the two previously mentioned use cases, the BWHT layer and Pagh multiplication. For the BWHT layer, the operation applied in the Hadamard space is a smoothing function φ_T. This function depends on a parameter T, which is learned in real networks but is set to 1.0 in our experiments.

$$\varphi_T(x) = \begin{cases} x - T, & \text{if } x > T \\ 0, & \text{if } |x| \leq T \\ x + T, & \text{if } x < -T \end{cases}$$

The second practical experiment is performing an XOR Convolution using the FWHT, this is one of the core mechanics of Pagh multiplication, and therefore highly relevant. The procedure does the following: take two vectors A and B, transform both using the FWHT, then apply element wise multiplication of the two resulting vectors. Finally transform that result to get the XOR convolution of A and B. Without the transforms, XOR convolution is calculated by taking a sum of n products for each value C_i in the output vector.

$$C_i = \sum_{j \texttt{ XOR k = i}} A_j \cdot B_k$$

The naive summation requires $O(n^2)$ operations to produce all the entries of C, while the FWHT method only requires $O(n \log n)$ operations.

Table 3.5 concisely explains the 4 experimental setups and gives each experiment a short name that will be used from now on.

3.6 Performance Evaluation

All implementations, in all supported numerical formats, were tested on all available GPU setups in the Minerva cluster. As a baseline to compare against, we use *cuFFT*, computing a real to complex Fast Fourier Transform of equal size. While it doesn't compute the same transform, it is very similar, requiring almost the exact same data movements, but computing with complex numbers, and applying complex roots of unity to the butterflies, making it computationally heavier. We chose to compare against a real to complex transform instead of a complex to complex transforms, as it requires an equivalent amount of input and output data. This is also what is compared in the previous work on Pagh's algorithm [2].

The timing data gathered includes 100 runs for sizes $n \leq 2^{25}$ and 10 runs for sizes $2^{26} \leq n$. The difference is to ensure stable measurements for small sizes, where the

Table 3.5: Description and short-names of all experiments that were ran using the FWHT implementations

Short Name	Description
One-Way	Fill one vector of a given size with values from a given distribution and apply FWHT without normalization.
Two-Way	Fill one vector of a given size with values from a given distribution and apply FWHT twice to the vector, with normalization.
Smoothed	Fill one vector of a given size with values from a given distribution and apply the FWHT, then apply a Smoothing function φ_T with $T = 1.0$ before applying another FWHT and normalization.
XOR Conv	Fill two vectors of equivalent given size with values from a given distribution and apply the FWHT to both, then apply element wise multiplication between the two resulting vectors. Finally transform the result back and apply normalization.

variation in runtime was observed to be much greater. In both cases, we discard the first result as a cold-cache run, and collect the median runtime of the remaining runs. Due to VRAM memory constraints, the sizes of the transforms are limited to $\frac{2^{34}}{sizeof(data\_type)}$ using a L4 GPU, $\frac{2^{35}}{sizeof(data\_type)}$ using a L40s GPU and $\frac{2^{36}}{sizeof(data\_type)}$ using a H100 GPU. Additionally, *cuFFT* only seems to support transform sizes up to 2^{30}. The stabilized versions use $2n \cdot sizeof(data\_type)$ bytes, as in addition to the main vector, they maintain an error vector of size n. This means results for these implementations stop 1 power of 2 earlier than the *Folklore* and *Alman & Rao* implementations.

4

Results

The results chapter presents comparative data of numerical stability and/or runtime for selected groups of implementations. Much more data was collected than is presented here. The complete dataset is available as a JSON file in the public GitHub repository.[1]

4.1 Numerical Stability Results

Here we present comparative data from the three folklore based implementations, using no stabilization, Kahan stabilization and Neumaier stabilization as well as the Alman based implementation. We refer to these four implementations as: *Folklore*, *Kahan*, *Neumaier* and *Alman & Rao*. The result presented is mean relative error, where relative error is measured as difference from the true value divided by the magnitude of the true value $e = \frac{|V^* - V|}{|V|}$. The true value is calculated by performing the experiment using a 128-bit CPU Implementation. The input is drawn from one of the distributions discussed in Section 3.5.1 and the experimental setup is one of the setups discussed in Section 3.5.2. The data is presented over 4 Figures: 4.1, 4.2, 4.3 and 4.4, showing the results for FP64, FP32, FP16 and BF16 respectively. For a lot of experimental setups, the range of FP16 is too small to handle the experiment without overflow, in these cases, no data point is recorded, and thus the graphs stop early. Any plots that show just the line for a machine epsilon indicate that all implementations get exactly the correct answer for all input sizes.

While the results are fairly varied depending on operation and input distribution, two general claims about the data can be made. Firstly, relative to the machine epsilon, the experiments show very similar data for all numerical formats. Secondly, for almost all data points the mean relative error compared to the baseline folklore implementation is reduced by employing Kahan summation, and reduced further by employing Neumaier summation. On the other hand, using Alman's algorithm increases the error. This claim is quantified in Table 4.1, where the median error reduction over all experiments is reported. The results show roughly a 30% reduction when using *Kahan*, a 75% reduction when using *Neumaier* and a 50% increase when using *Alman & Rao*, the reduction/increase does not seem to vary significantly between numerical formats, but increases with transform size.

[1] https://github.com/erwinia42/FWHT

Table 4.1: Reduction of relative error compared to baseline, taken as median over all experiments of same size for each data type and implementation.

	Double			Float			Half			BFloat16		
$\log_2$(Size)	Kah	Neu	Alm	Kah	Neu	Alm	Kah	Neu	Alm	Kah	Neu	Alm
3	0.0%	**44.8%**	0.0%	0.0%	**0.0%**	-8.6%	0.0%	**0.0%**	-21.6%	0.0%	**8.0%**	0.0%
4	6.2%	**17.7%**	-4.5%	2.9%	**10.8%**	0.0%	-8.2%	**0.0%**	-49.0%	0.0%	**8.4%**	-30.9%
5	28.5%	**50.1%**	-47.9%	0.8%	**22.2%**	-24.6%	24.7%	**36.9%**	-37.6%	31.7%	**46.8%**	-1.7%
6	26.9%	**46.2%**	-88.1%	14.7%	**44.1%**	-41.1%	4.9%	**46.3%**	-47.3%	27.7%	**52.5%**	-14.2%
7	27.4%	**61.6%**	-40.5%	17.1%	**52.2%**	-70.4%	24.7%	**53.3%**	-37.9%	25.7%	**59.6%**	-19.9%
8	39.9%	**71.7%**	-6.0%	24.5%	**45.5%**	-68.7%	20.8%	**55.5%**	-8.5%	31.0%	**58.0%**	-29.7%
9	35.6%	**72.4%**	-29.7%	25.6%	**61.4%**	-47.1%	9.0%	**60.4%**	-44.0%	12.1%	**51.8%**	-59.6%
10	29.9%	**70.9%**	-28.0%	28.7%	**70.9%**	-43.2%				24.7%	**64.4%**	-46.6%
11	25.3%	**65.9%**	-46.3%	23.3%	**63.6%**	-35.7%				30.3%	**67.8%**	-43.5%
12	24.6%	**72.8%**	-31.4%	21.8%	**65.9%**	-32.7%				20.6%	**62.3%**	-64.6%
13	33.0%	**72.9%**	-37.7%	29.0%	**67.0%**	-38.4%				29.7%	**72.9%**	-44.7%
14	33.8%	**76.3%**	-46.7%	33.4%	**70.0%**	-43.1%				31.9%	**69.1%**	-53.0%
15	32.1%	**74.4%**	-51.9%	26.4%	**72.5%**	-42.2%				29.4%	**74.3%**	-46.1%
16	32.6%	**76.4%**	-51.0%	27.2%	**71.9%**	-36.5%				29.5%	**69.6%**	-50.9%
17	30.3%	**77.3%**	-40.0%	32.7%	**73.3%**	-27.4%				30.9%	**69.3%**	-52.9%
18	29.5%	**77.3%**	-52.1%	34.8%	**74.5%**	-46.7%				27.3%	**65.3%**	-48.4%
19	37.3%	**79.8%**	-31.2%	37.0%	**76.3%**	-26.5%				27.3%	**71.6%**	-68.9%
20	36.8%	**79.9%**	-33.0%	35.4%	**76.7%**	-35.0%				29.8%	**74.7%**	-50.2%
21	27.6%	**76.8%**	-40.4%	31.9%	**80.3%**	-8.4%				31.7%	**78.6%**	-54.2%
22	31.9%	**78.3%**	-34.5%	35.1%	**76.3%**	-40.2%				29.2%	**75.9%**	-54.4%
23	35.5%	**78.2%**	-46.4%	31.6%	**76.4%**	-46.9%				30.0%	**77.6%**	-51.8%
24	38.8%	**81.1%**	-28.2%	38.7%	**79.0%**	-42.4%				30.4%	**76.6%**	-48.9%
25	35.6%	**83.9%**	-45.7%	32.6%	**76.4%**	-46.6%				31.0%	**79.3%**	-51.0%

For all numerical representations there is an anomalous result for XOR conv using Pagh_Norm and sizes between 2^6 and 2^8, most likely best explained by some entry being extremely close to 0, making very small absolute errors appear huge in terms of relative error. Investigating the data further we find that the mean absolute error remains very small for the affected data points, thus this will not be discussed further.

4.2 Performance Results

We present measurements of median runtime for the two main variations of the FWHT, *Folklore* and *Alman & Rao*, as well as a baseline comparison of *cuFFT*, Nvidia's implementation of the FFT [22]. We present both the runtime including the data movement between CPU and GPU, as well as the runtime of just the GPU kernel. Additionally, we show the performance of *cuFFT* and *Alman & Rao* compared to *Folklore*.

Figures 4.5, 4.6 and 4.7 are relevant to the scenario where the data that is to be transformed resides on Host memory and therefore includes both the data movement of the input from Host to GPU and data movement of the result back to the Host. On all GPUs *Folklore* and *Alman & Rao* perform very similarly. they are also extremely close to the lower bound of *Memcpy*, simply moving the input to the GPU and then back to the Host.

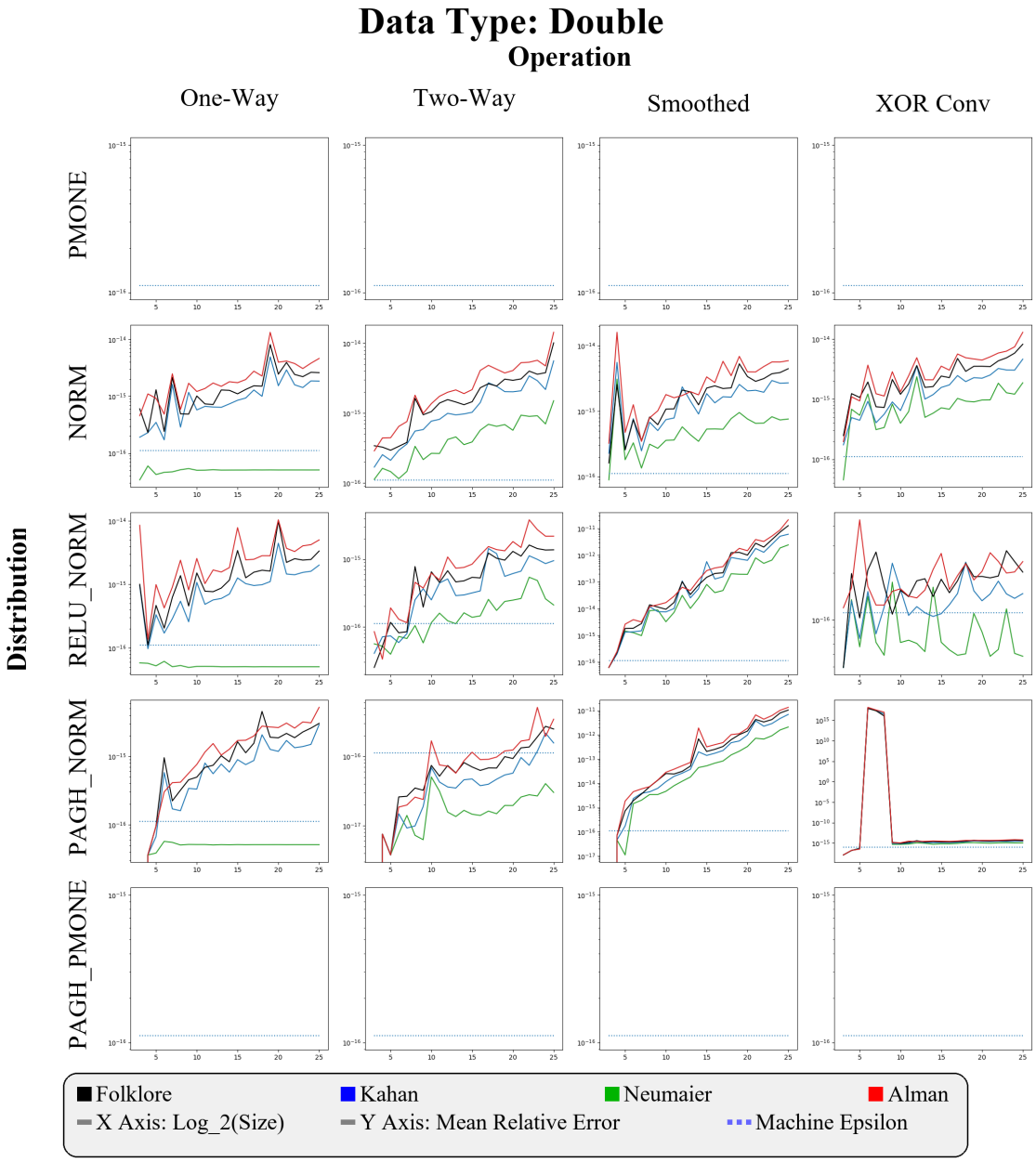

Figure 4.1: Mean relative error results from all numerical stability experimental setups using all available input distributions and FP64 implementations. Sorted column wise by experiment and row wise by distribution. Data collected for all sizes that are powers of 2, from 2^3 to 2^{25} and for the three folklore implementations, using no stabilization, Kahan stabilization and Neumaier stabilization respectively

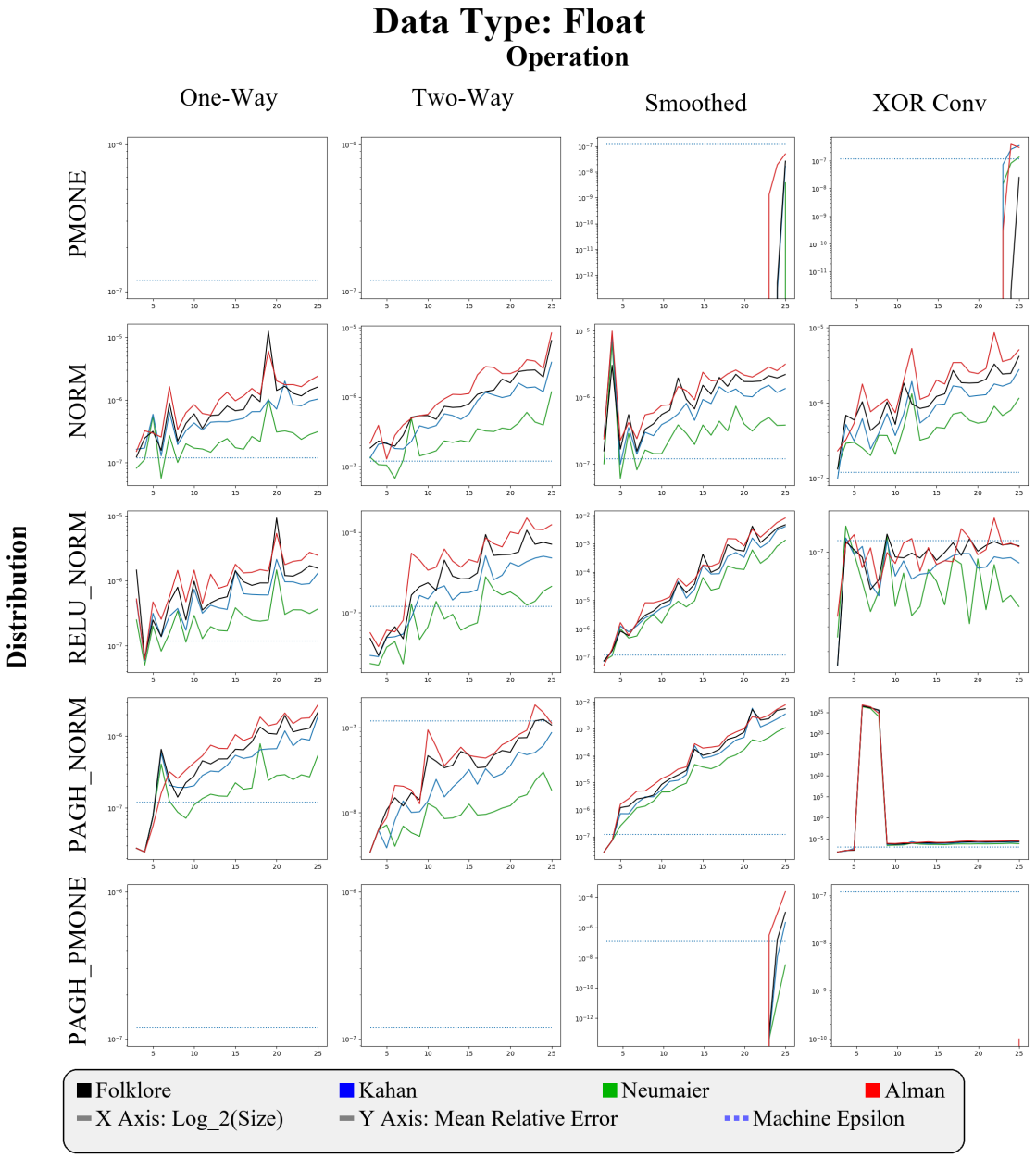

Figure 4.2: Mean relative error results from all numerical stability experimental setups using all available input distributions and FP32 implementations. Sorted column wise by experiment and row wise by distribution. Data collected for all sizes that are powers of 2, from 2^3 to 2^{25} and for the three folklore implementations, using no stabilization, Kahan stabilization and Neumaier stabilization respectively

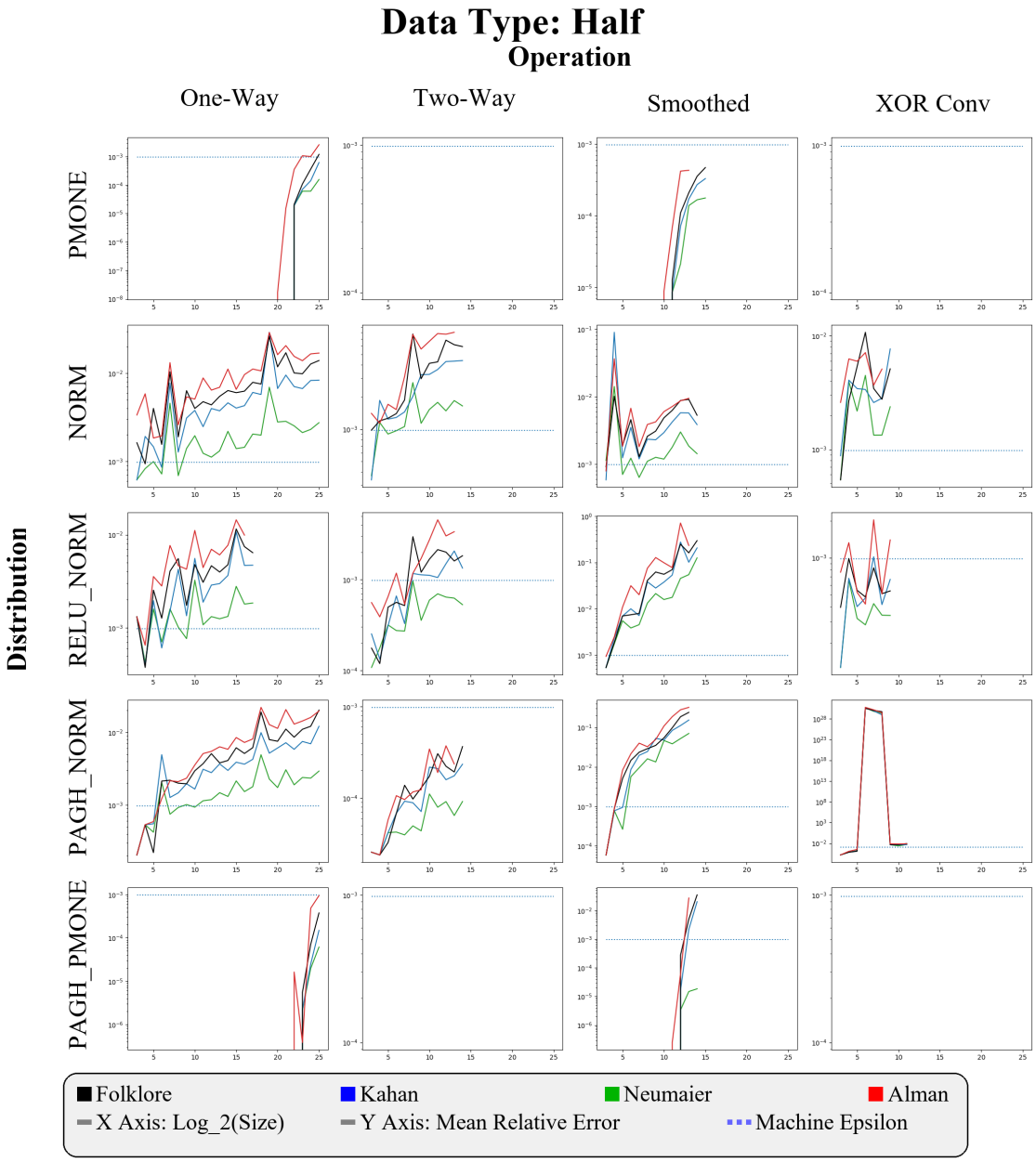

Figure 4.3: Mean relative error results from all numerical stability experimental setups using all available input distributions and FP16 implementations. Sorted column wise by experiment and row wise by distribution. Data collected for all sizes that are powers of 2, from 2^3 to 2^{25} and for the three folklore implementations, using no stabilization, Kahan stabilization and Neumaier stabilization respectively

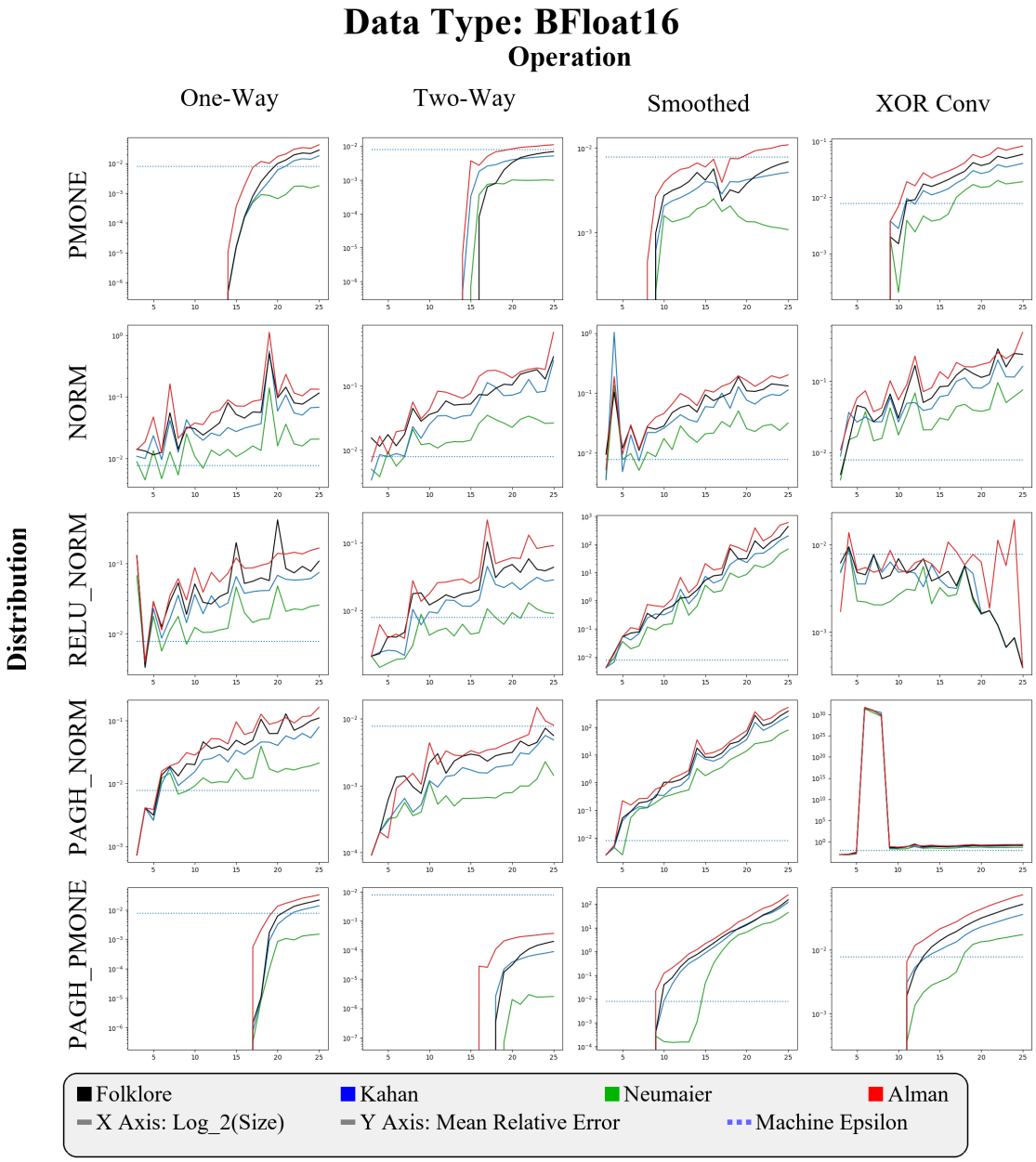

Figure 4.4: Mean relative error results from all numerical stability experimental setups using all available input distributions and BF16 implementations. Sorted column wise by experiment and row wise by distribution. Data collected for all sizes that are powers of 2, from 2^3 to 2^{25} and for the three folklore implementations, using no stabilization, Kahan stabilization and Neumaier stabilization respectively

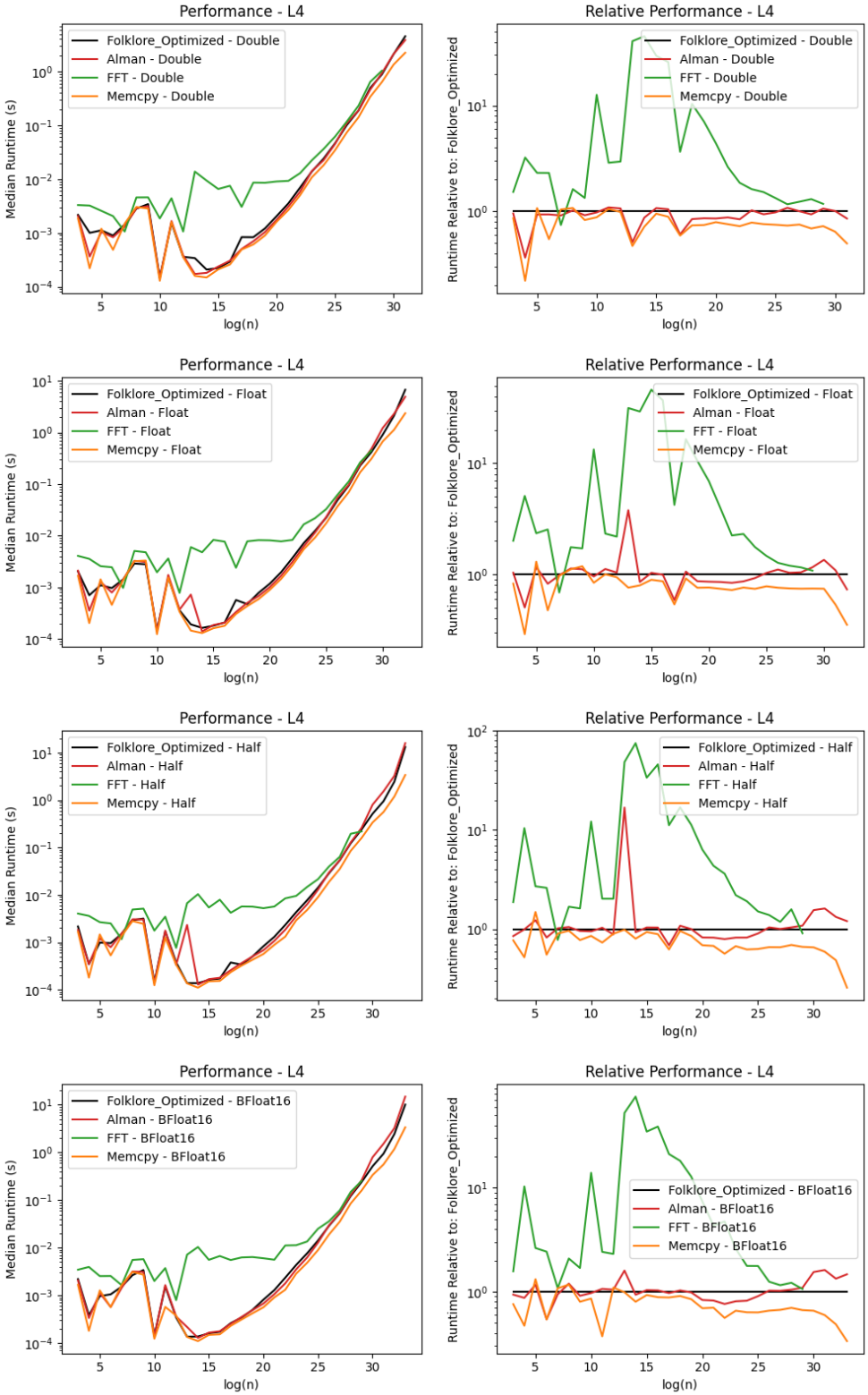

Figure 4.5: Performance measurements using Nvidia L4 for Double, Float, Half and BFloat16. The time presented is the median runtime and includes Memcpy time

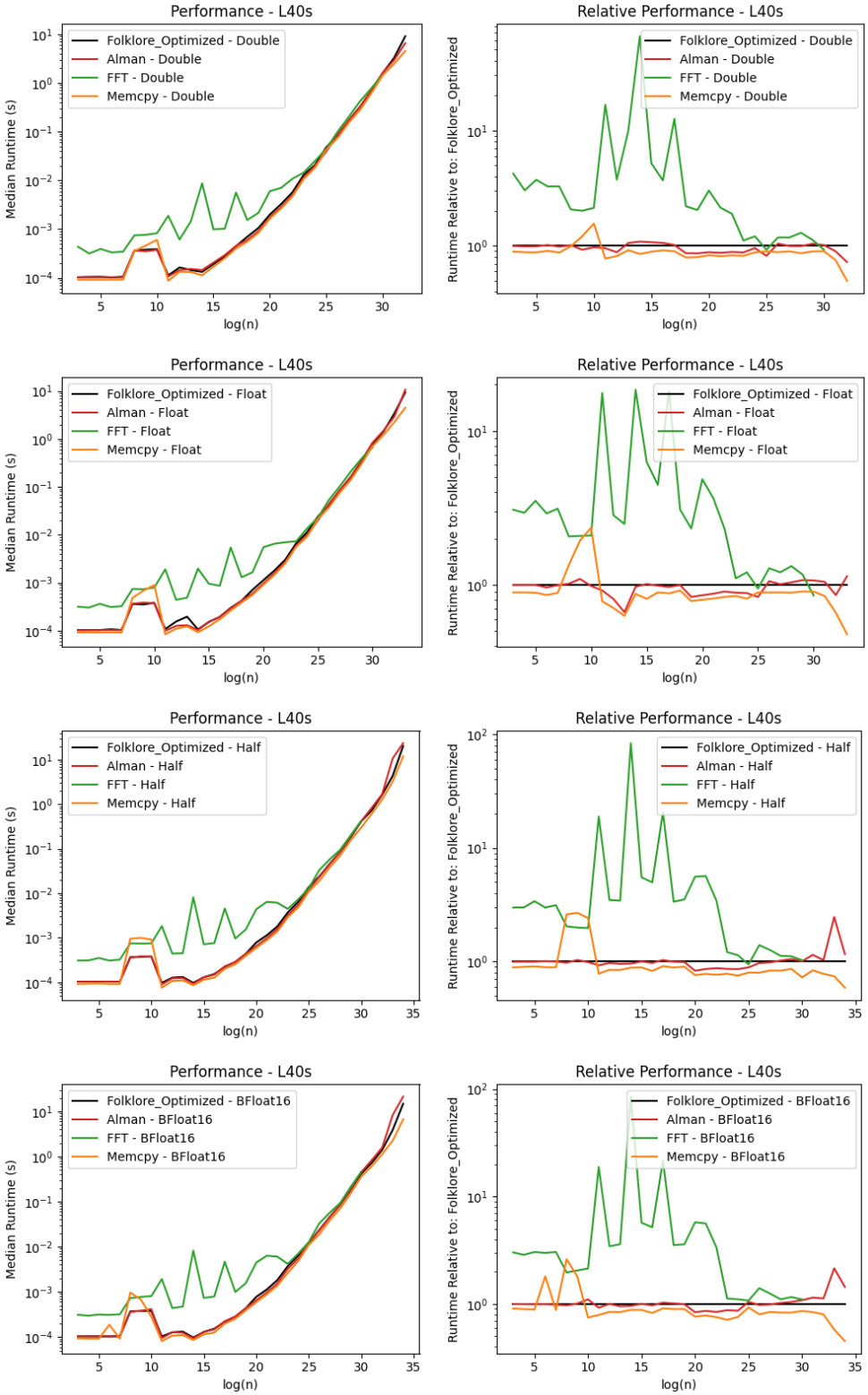

Figure 4.6: Performance measurements using Nvidia L40s for Double, Float, Half and BFloat16. The time presented is the median runtime and includes Memcpy time

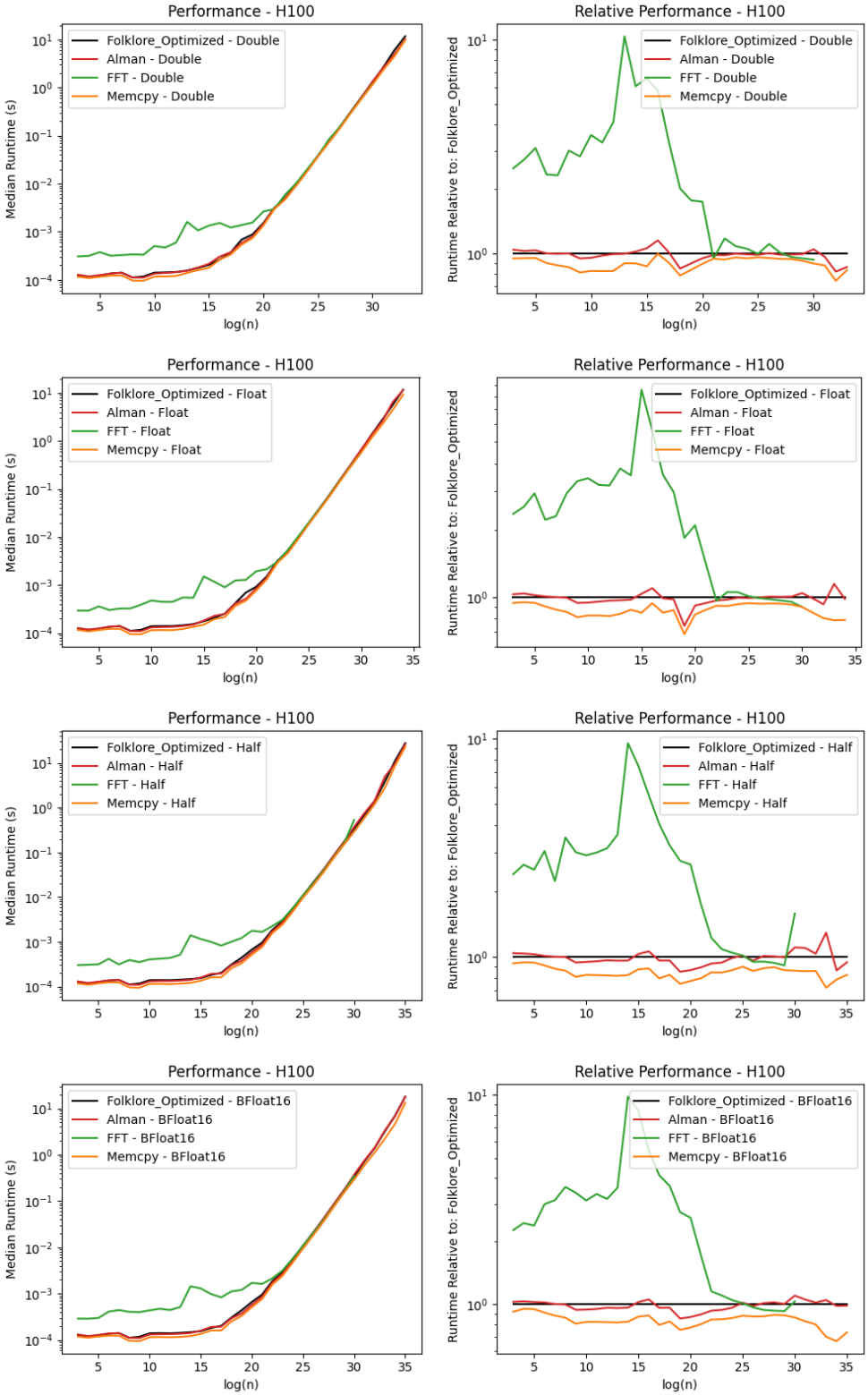

Figure 4.7: Performance measurements using Nvidia H100 NVL for Double, Float, Half and BFloat16. The time presented is the median runtime and includes Memcpy time

When comparing *Folklore* against *cuFFT* using smaller sizes ($\log n \leq 20$), *cuFFT* runs considerably slower. This is mainly due to *cuFFT* performing "planning" before starting execution. As the sizes grow and the planning step consumes less time relative to the actual transform, this advantage disappears. It is worth noting that on L4, *cuFFT* only handles up to $n = 2^{29}$ elements and on L40s/H100 NVL, it handles up to $n = 2^{30}$ elements. This is smaller than what we are able to evaluate for the FWHT implementations, thus the graph for *cuFFT* stops earlier.

It is clear that almost all time is spent on moving data between host and GPU. However, as the FWHT is an operation used as a part of other GPU implementation, this is often not necessary. Thus we also present the running time for the GPU kernels of *Folklore*, *Alman & Rao* and *cuFFT* in Figures 4.8, 4.9 and 4.10. This corresponds to a scenario in which the data already resides on the GPU and can be transformed immediately, this also means we do not include planning time for *cuFFT*, as this only has to be completed once, with the possibility of running many transforms with the same planning data. Once again, *Alman & Rao* and *Folklore* perform very similarly, but it is clear that *cuFFT* is faster than our implementations when using large sizes, especially true when using the H100 GPU. The two exceptions are when using Doubles on the L4/L40s GPUs, here *cuFFT* performs similarly or slightly worse, depending on the input size.

4.2.1 Performance of Stabilized Implementations

In addition to comparing our implementations to *cuFFT*, we also show the median runtime of the stabilized implementations compared to a baseline Folklore implementation in Figures 4.11, 4.12 and 4.13. They show that the difference in running time between *Kahan* and *Neumaier* is minimal, only really measurable when using Doubles. They also show both stabilized versions require $> 2\times$ as much running time compared to the baseline for larger sizes.

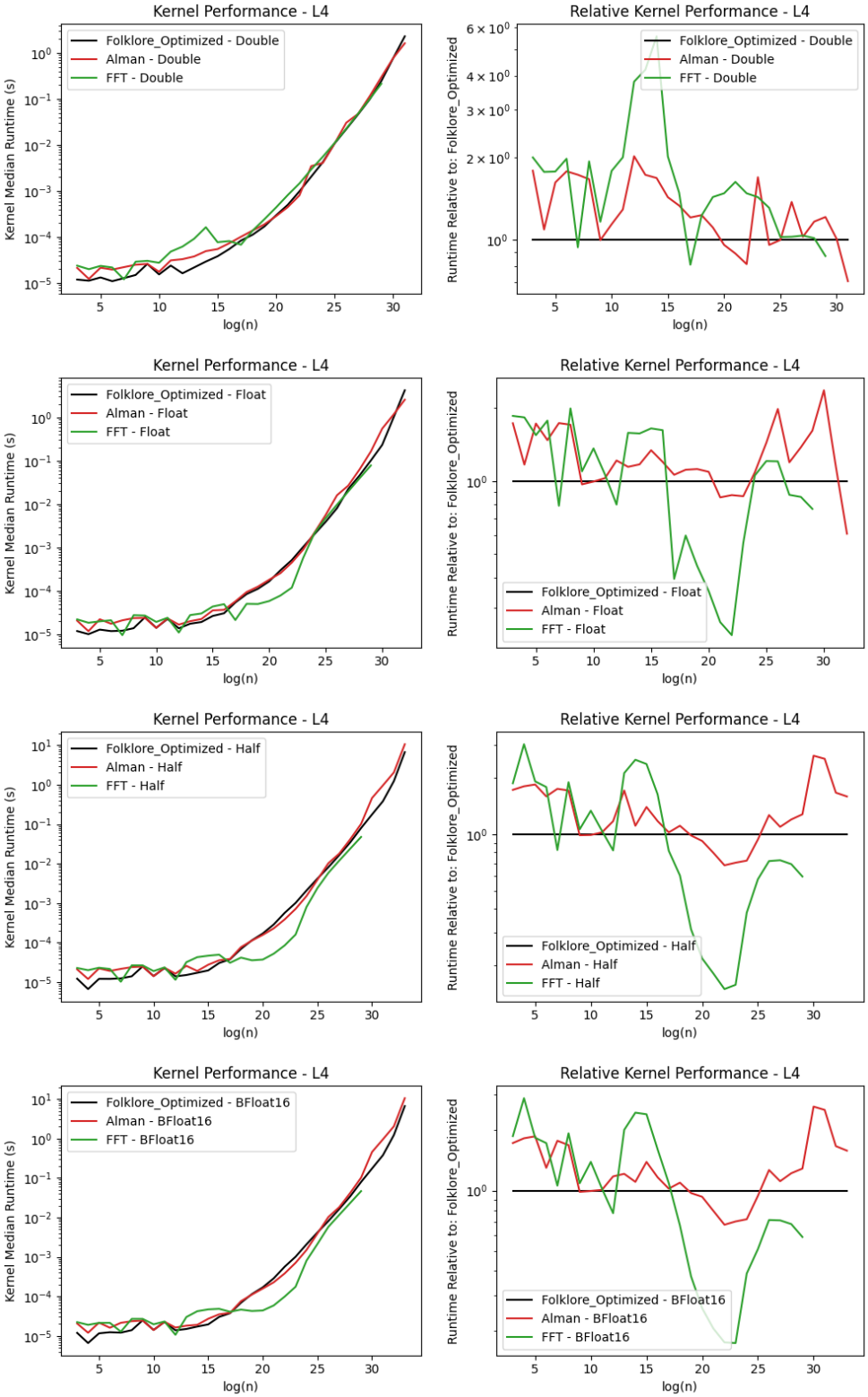

Figure 4.8: Kernel Performance measurements using Nvidia L4 for Double, Float, Half and BFloat16. The time presented is the median runtime and includes only the execution of the GPU kernel.

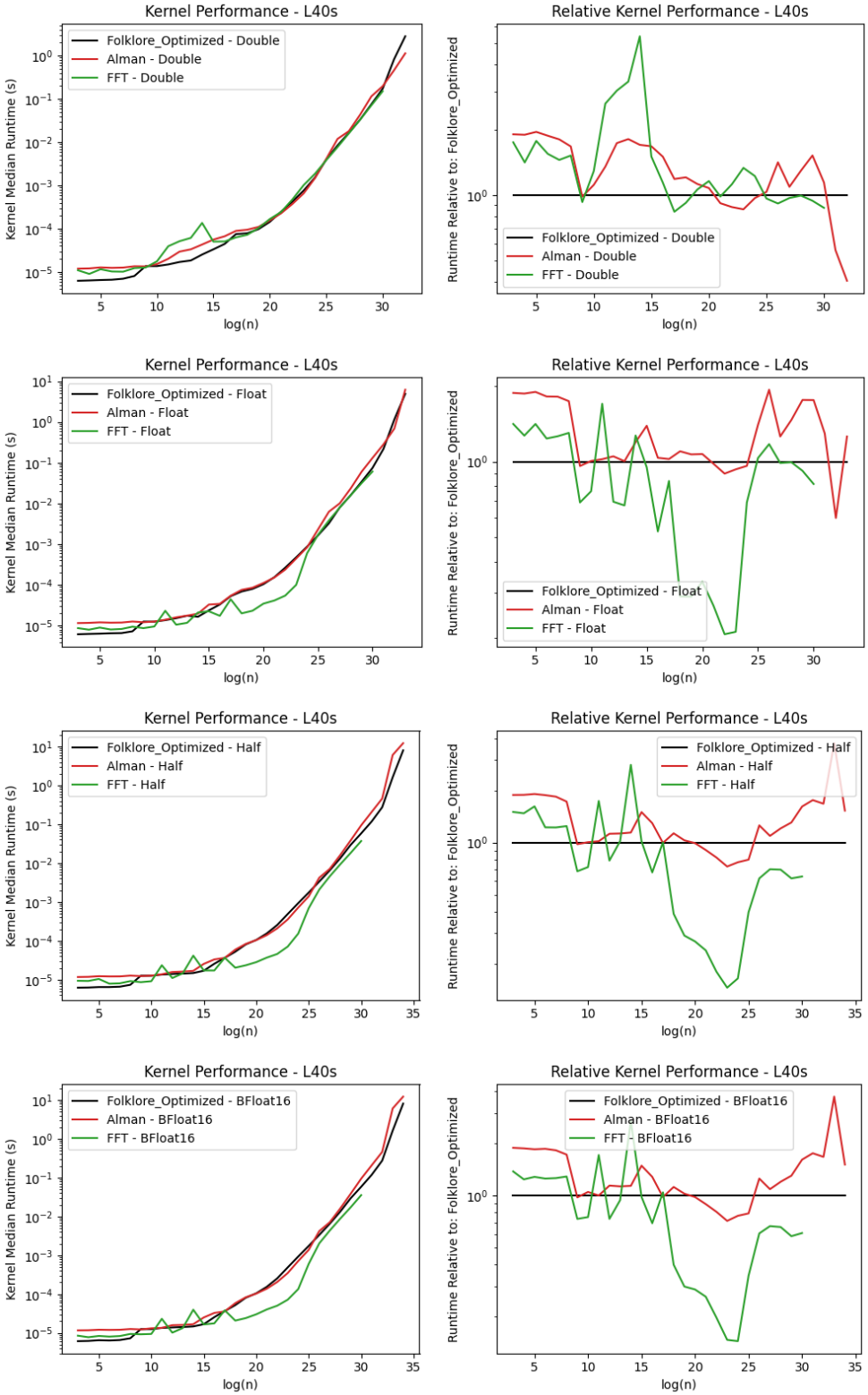

Figure 4.9: Kernel Performance measurements using Nvidia L40s for Double, Float, Half and BFloat16. The time presented is the median runtime and includes only the execution of the GPU kernel.

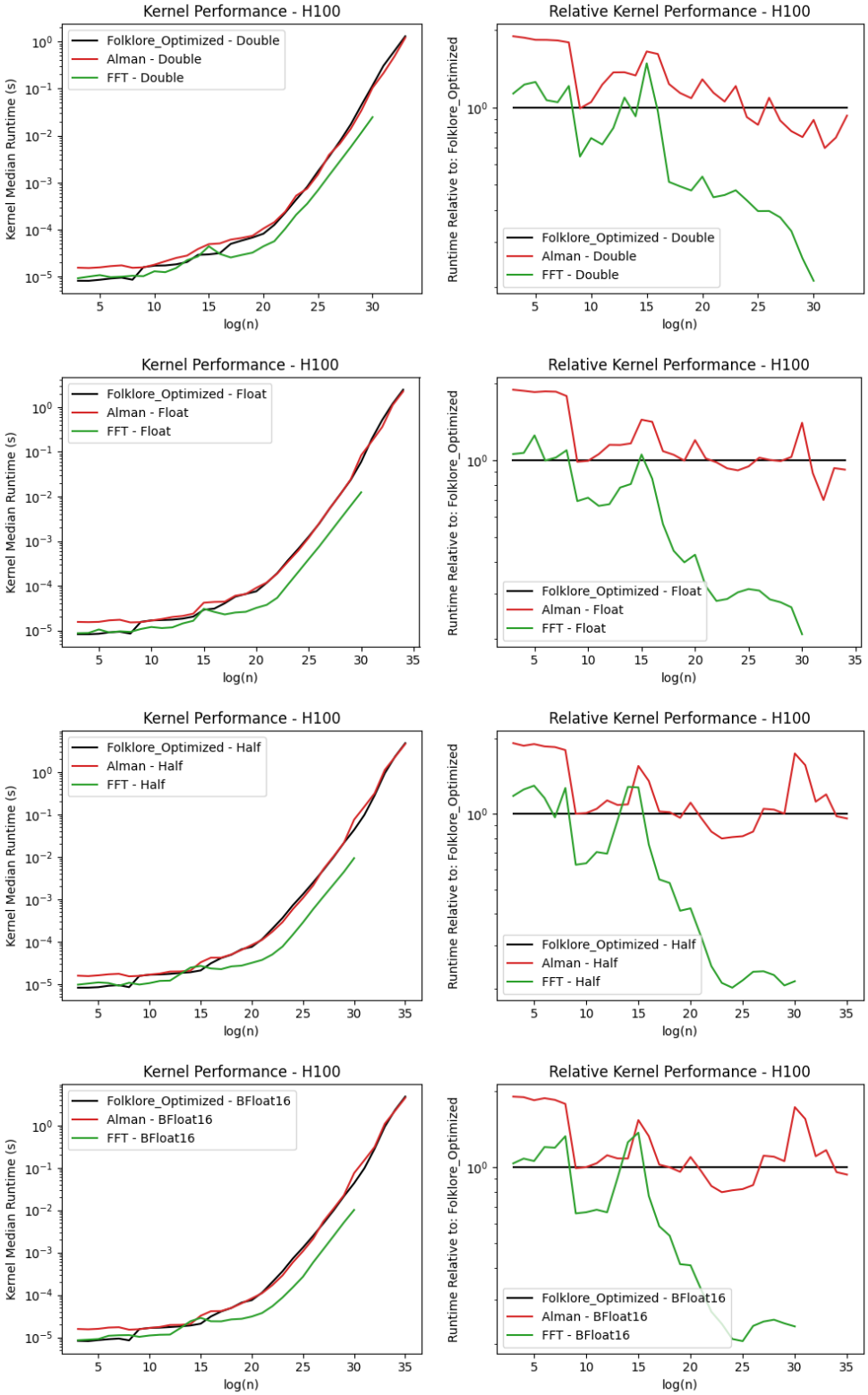

Figure 4.10: Kernel Performance measurements using Nvidia H100 for Double, Float, Half and BFloat16. The time presented is the median runtime and includes only the execution of the GPU kernel.

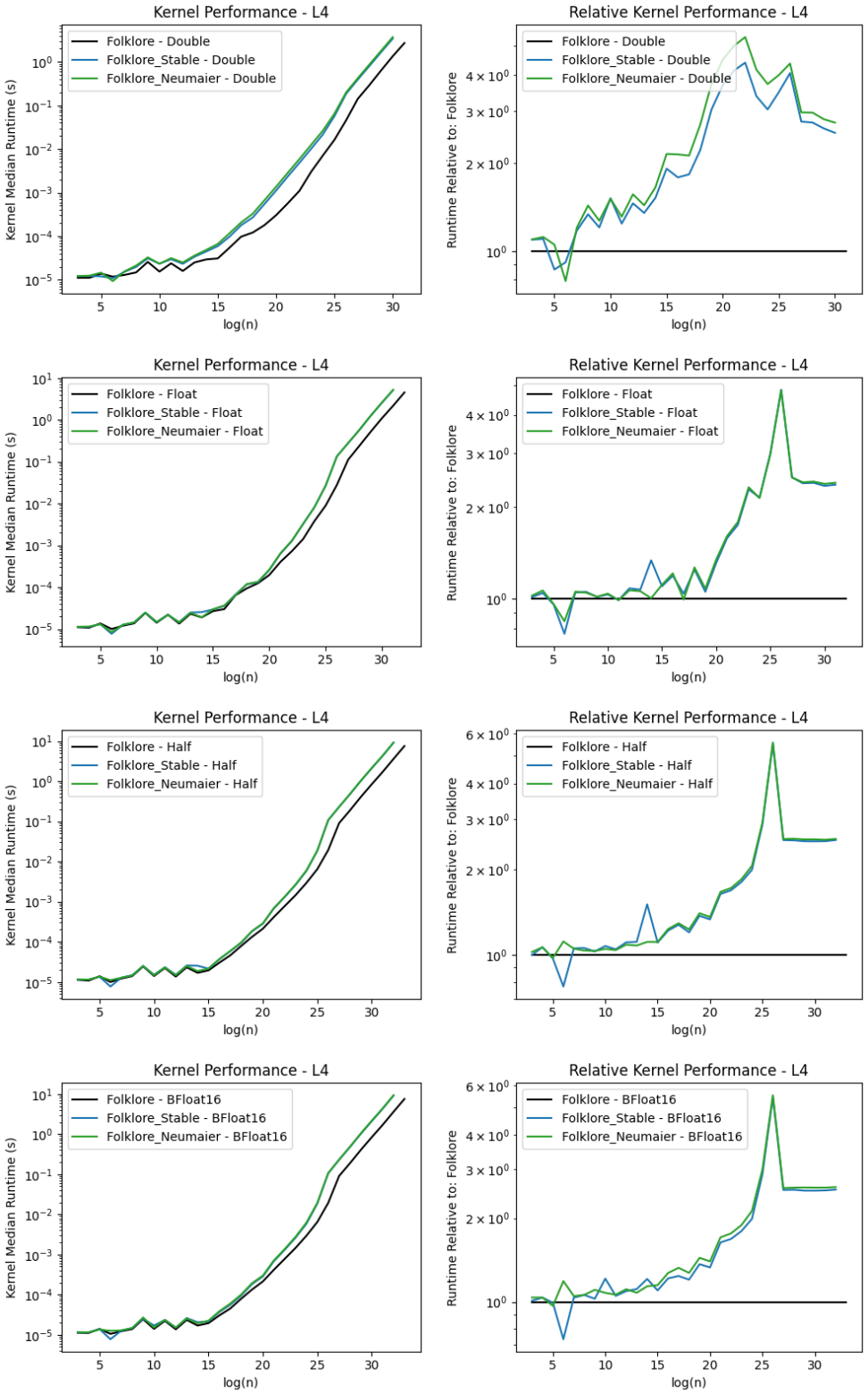

Figure 4.11: Kernel performance comparison of the three versions of the folklore algorithm using Nvidia L4

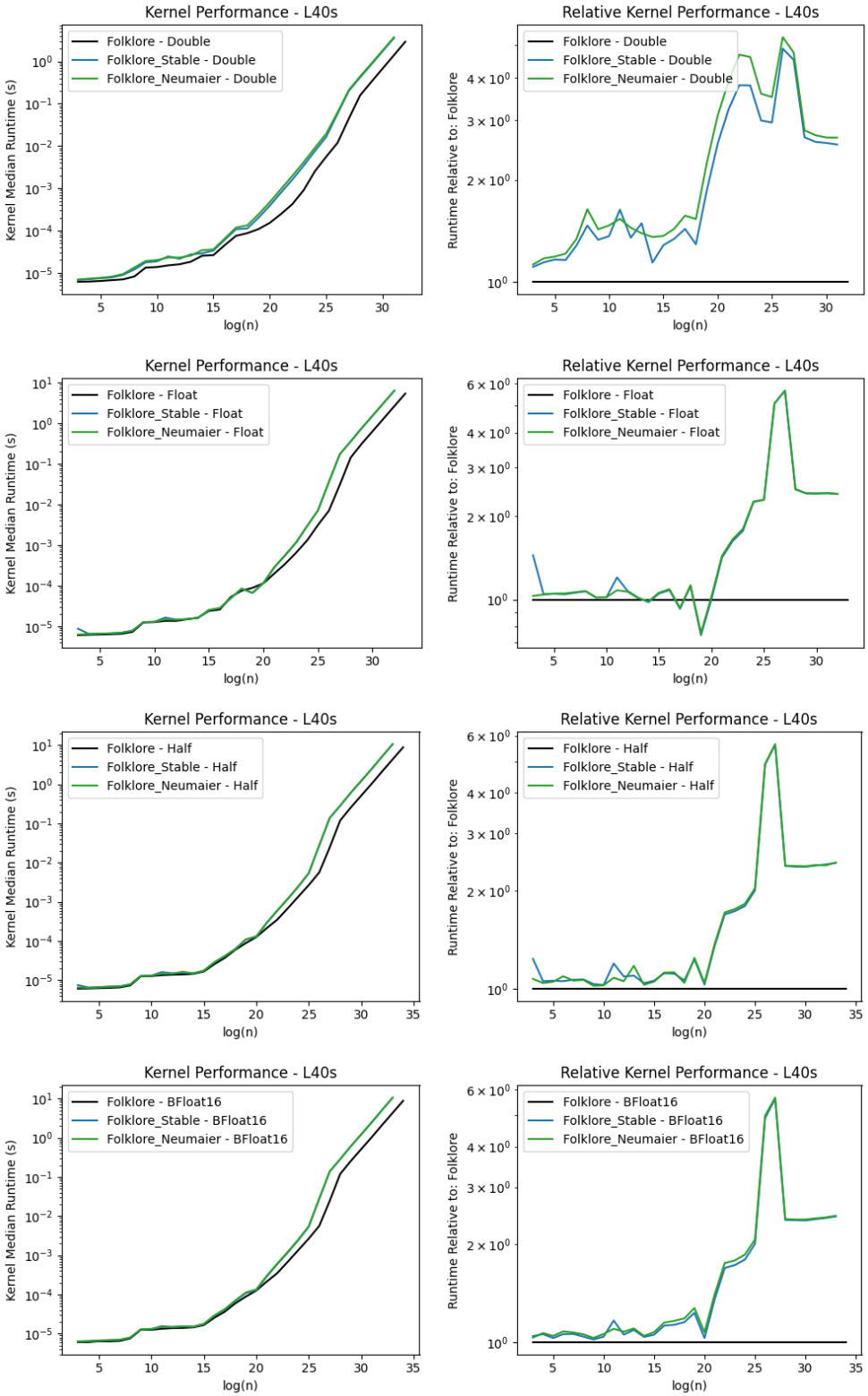

Figure 4.12: Kernel performance comparison of the three versions of the folklore algorithm using Nvidia L40s

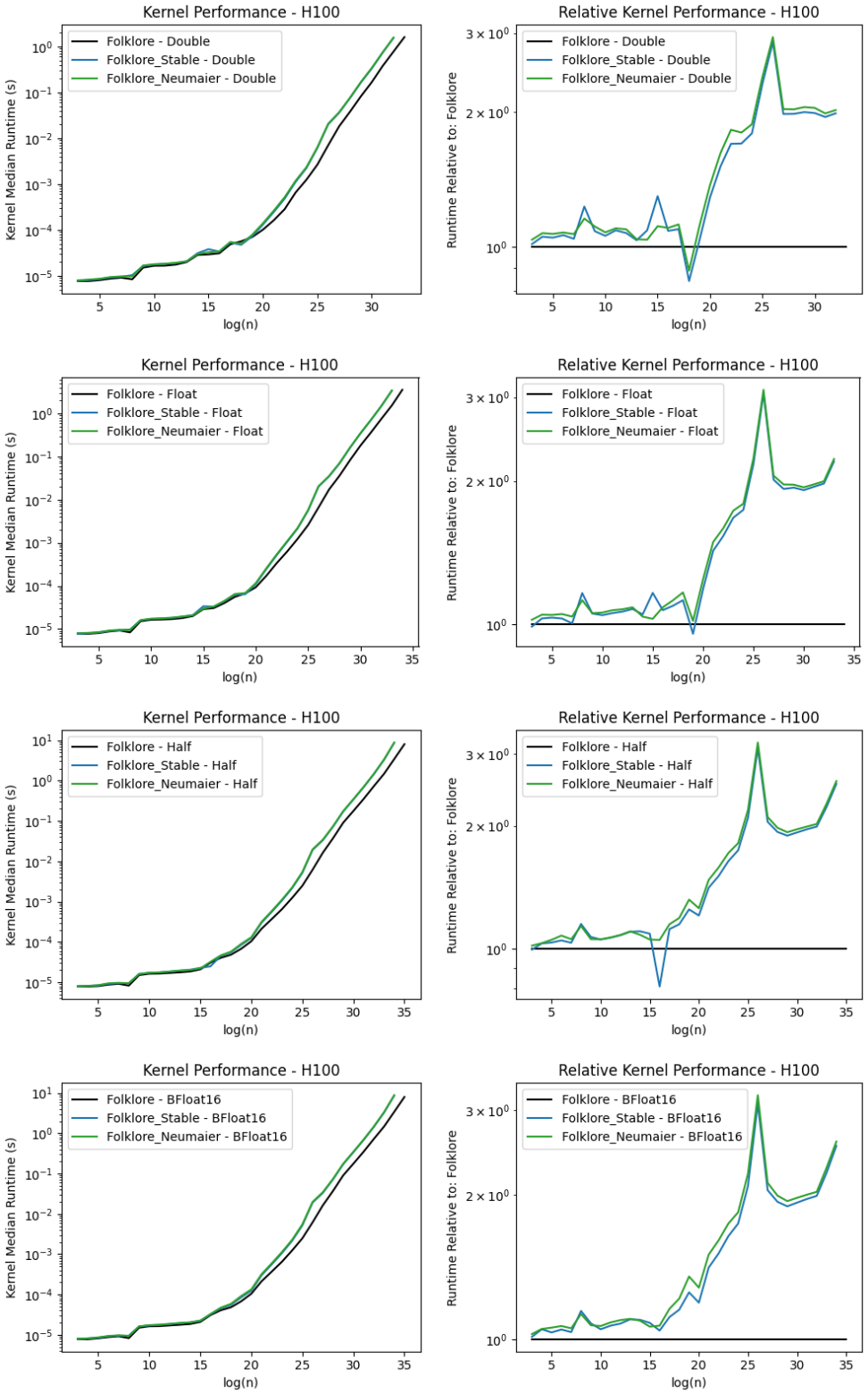

Figure 4.13: Kernel performance comparison of the three versions of the folklore algorithm using Nvidia H100

5

Discussion

There are several interesting findings in the results regarding numerical stability and performance, this chapter intends to highlight the most interesting of these. Additionally, it is used to discuss the quality of the implementations.

5.1 Numerical Stability

The main question regarding numerical stability was to evaluate how well the Kahan based and Neumaier based approaches to stabilization perform over the real world scenarios we experiment with. Additionally, we find in the results that switching to Alman and Rao's algorithm has a significant effect on numerical stability.

5.1.1 Stabilization Techniques

As the results show, we achieve roughly a 30%/75% reduction of the mean relative error using *Kahan* or *Neumaier* respectively. This is in some sense quite remarkable as we are utilizing these techniques on problems where the conditioning number $\kappa(x)$ is in expectation infinitely large, and thus the theoretical properties do not provide any bound at all. On the other hand, applying Kahan or Neumaier summation on the original problem of a linear sum can show much greater results, as the baseline algorithm can be very weak in certain real-world scenarios. We find that the FWHT is already quite well stabilized due to implicitly utilizing pairwise summation, and thus works very well over the evaluated experiments. This is especially noticeable with the PMONE and PAGH_PMONE distributions, where the baseline already computes the result without error for most numerical formats and sizes. Even for many larger sizes, the error is smaller than 1 machine epsilon, making any improvements quite difficult as even with a perfect algorithm you can not avoid an error of 0.5 machine epsilon in the worst case.

In cases where numerical stability is of great concern it is likely that the results from this implementation will be beneficial, and they show empirically that Kahan and Neumaier summation based techniques hold up well even in scenarios where their theoretical guarantees do not apply. However, we do not expect a broad adoption of these techniques for the use case of FWHT specifically as they consume twice as much memory and $> 2\times$ as much running time. With these properties using a transform with a numerical representation with twice as many bits is more beneficial

than using stabilization techniques. Thus the only benefit is when using the most precise floating point format provided on a given platform.

5.1.2 Numerical Stability of Alman

As the results showed, *Alman & Rao* performs worse than *Folklore* when it comes to numerical stability, increasing the relative error by roughly 50%. This is not too surprising, the practice of scaling the input values before performing summation is very likely to exacerbate round-off errors, especially when summing two elements with differing scaling coefficients.

5.2 Performance

We discuss performance by comparison among two groups of transforms, first we compare the unstabilized versions of FWHT to Memcpy and *cuFFT*, to better understand how well the implementations have been made and what limiting factors exist. Afterwards we discuss how much performance is affected by adding operations to increase numerical stability. As a general point, the performance measurement is very unstable for small $\leq 2^{10}$ sizes, which leads to somewhat unexpected results where transforms outperform memcpy or stabilized implementations outperform the non-stabilized implementation. Due to the running time being so low for these scenarios, and mostly being interested in large transform sizes anyway, we disregard these results as measurement anomalies.

5.2.1 Host Memory Situation

In the situation where memory resides on the host, and we wish to only compute a singular transform, the results are very positive, the running time is only about 10% longer than when simply measuring the data movement between host and GPU. Additionally, we outperform *cuFFT* with $2 - 10\times$ less running time on all three GPUs when $n \leq 2^{20}$. However, this advantage disappears for very large transform sizes. As already mentioned, these results are in large part due to *cuFFT* performing planning before starting the transform. As this is required to perform a singular transform, the result is certainty valid, but as we expect the FWHT to be part of another GPU implementation (where it is used multiple times) it is not very representative of real world scenarios.

5.2.2 GPU Memory Situation

The scenario measuring only the kernel runtime shows less positive results. While we do still outperform *cuFFT* when using Doubles on L4 and L40s, and even remain close performance-wise with the other numerical formats, we would have hoped to see an improvement, as we are computing a "Simpler" algorithm. However, this is not too surprising, with the massive amounts of computation offered by modern

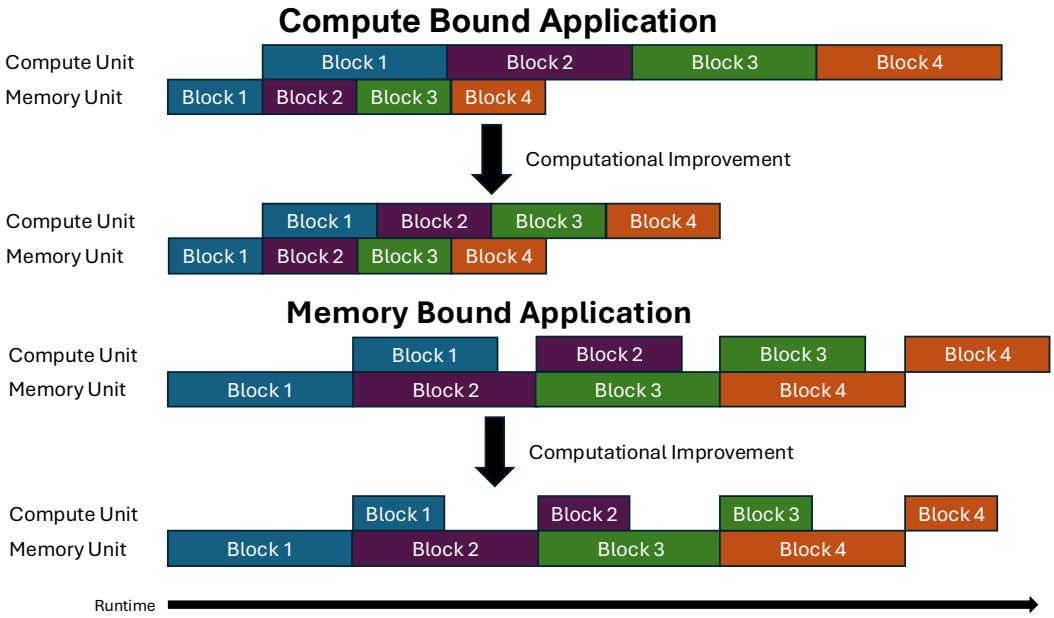

Figure 5.1: A simple example of a computational improvement in a computationally bound and a memory bound situation. The example includes 4 blocks of computation on one memory and one compute unit and highlights that in a memory bound application, a computational improvement has negligible impact on the full running time.

GPUs, the FWHT (and the FFT[1]) becomes very memory-bound, spending almost all the time moving data from GPU global memory to GPU shared memory. As this can be done concurrently with computation by switching what warp is active in the SM, the computational improvement would only be relevant if we were spending more time computing than moving data. A simplified illustration of the problem is provided as Figure 5.1. The results indicate this is only the case when using Doubles, as these present a somewhat more complex computational challenge to the L4 and L40s GPUs, which do not have native support for FP64 computation, and must emulate it using FP32 [25].

5.2.2.1 H100 Kernel Performance

Using the H100 GPU, the performance gap is even wider, *cuFFT* considerably outperforms the FWHT implementations, even using FP64. The fact that FP64 performs well is most likely an effect of the H100 having hardware support for FP64. The most likely reason for the widened performance gap overall is the introduction of a new unit to move data on the H100 called a Tensor Memory Accelerator [26]. This unit is not implicitly used by CUDA code, but must instead be managed by the programmer. Due to this issue being found out quite late during the project as access to the H100 was granted later, this is not utilized in the FWHT implementations, but it is highly likely that *cuFFT* uses the Tensor Memory Accelerator.

[1] *cuFFT* does not use the Tensor cores, as it is deemed too memory bound [23]. However, there are some recent implementation that show performance increases using Tensor Cores [24]

5.2.2.2 FWHT and FFT as Parts of Other Implementations

It is worth noting that even in the scenarios where *cuFFT* outperforms the FWHT implementations, the availability of a efficient GPU implementation of the FWHT for larger sizes than previously supported could still be meaningful when they are used as parts of other GPU implementations as it enables the remaining operations to be performed over real numbers instead of complex numbers. As an example, in Pagh's algorithm the two transformed vectors need to be multiplied element wise, doing this with real numbers instead of complex numbers is computationally easier.

5.2.3 Alman Implementation

We find no improvement in performance by utilizing *Alman & Rao* over *Folklore*, this might not be too surprising considering the context of other results, and our method. Firstly, we have already ruined the theoretical advantage by calculating scaling in $O(n \log n)$ time, while we have no proof that this is required, it seems difficult to improve on this. Then, in a practical implementation, there are a lot of operations required to do data indexing such that we access the correct data, this reduces the impact of having 1 less "data" operation per H_8. Finally, the FWHT is as seen very memory bound, and thus a computational performance increase might not even reflect on the final running time, which is more heavily impacted by data movement.

5.2.4 Performance Loss From Stabilization

Looking at the performance of the stabilized implementations *Kahan* and *Neumaier*, it is remarkable that the extra operations introduced to increase the stability with *Neumaier* does not impact the running time at all (except for when using Doubles). This is once again most likely due to the FWHT beeing heavily memory-bound, and thus the addition of a more complex stabilizing technique does not impact the final runtime. However, since we are adding extra memory by keeping track of the error vector we lose performance by increasing the necessary memory movements. It is possible that a less aggressive approach, which for example only keeps track of the error within a CUDA warp, or even a CUDA block, and can stabilize partial transforms, could run with the same performance as the non-stabilized transform.

5.3 Large Transform Sizes

In terms of increasing the supported sizes for transforms, we greatly improve on current state of the art implementations from HadaCore and Dao AI Lab, increasing the maximum size from 2^{15} to 2^{35} experimentally and 2^{63} theoretically. This means we allow sizes $2^{20}\times$ larger experimentally and 2^{48} theoretically. Even considering academic papers, such as the one from Bikov and Bouyukliev [6], which achieves an input size of 2^{18} experimentally and 2^{20} theoretically is significantly improved upon. The evaluated sizes are also $64\times$ larger than those of *cuFFT*, which stops at 2^{30}.

6

Conclusion

The FWHT implements well on GPU hardware, both using the standard *Folklore* algorithm, as well as using Alman and Rao's new algorithm [7]. This includes implementations that support far larger sizes than previously available, evaluating a FWHT with input size $2^{17}\times$ larger than the previous best from Bikov and Bouyukliev [6] as well as theoretically supporting sizes up to 2^{63}, larger than any GPU memory will allow for the foreseeable future. However, both implementations are heavily bounded by memory, ultimately only outperforming the target of *cuFFT* on L4 and L40s GPUs when using Doubles, and not at all when using an H100 GPU.

The addition of numerical stabilization to the implementation, drawing heavy inspiration from the two summation techniques Kahan and Neumaier summation, gives empirically very good results. This is true despite the fact that we apply the techniques to sums where the conditioning number $\kappa(X)$ is infinite in expectation. Over real-world use cases of the FWHT the stabilization techniques reduce the relative error by roughly 30% or 75% respectively, and in many cases reduce the relative error to below 1 machine epsilon. On the other hand Alman and Rao's algorithm shows larger errors than the baseline, roughly increasing the relative error by 50%.

6.1 Further Work

The project leaves a number of open doors for further work, most pressingly investigating the usage of the Tensor Memory Accelerator introduced with the H100. Considering the memory bound nature of the FWHT, it is highly likely that putting effort into this new memory movement technology could bear good results. In a similar vein, exploring the possibility to utilize Tensor Cores would be interesting, HadaCore already showed promising results on small transform sizes [4]. With improved memory transfers, it is possible it could help for larger transforms as well.

Exploring semi-stabilized implementations, which require no additional memory movements, would also be interesting, as we see from the comparisons of the *Kahan* and *Neumaier* implementations that introducing extra operations barely affects the running time. Potentially there is a possibility to stabilize the transform slightly at no performance cost.

Finally, to utilize the implementations made in this project, it would be beneficial to provide them as python functions, for example by using pyBind11 [27].

Bibliography

[1] H. Pan, D. Badawi, and A. E. Cetin, "Block walsh-hadamard transform based binary layers in deep neural networks," 2022. [Online]. Available: https://arxiv.org/abs/2201.02711

[2] J. Andersson and M. Karppa, "Practical Implementation of Compressed Matrix Multiplication," 2024, manuscript.

[3] R. Pagh, "Compressed Matrix Multiplication," 2011. [Online]. Available: https://arxiv.org/abs/1108.1320

[4] K. Agarwal, R. Astra, A. Hoque, M. Srivatsa, R. Ganti, L. Wright, and S. Chen, "HadaCore: Tensor Core Accelerated Hadamard Transform Kernel," 2024. [Online]. Available: https://arxiv.org/abs/2412.08832

[5] T. Dao, "Fast-Hadamard-Transform," https://github.com/Dao-AILab/fast-hadamard-transform, 2023.

[6] D. Bikov and I. Bouyukliev, "Parallel Fast Walsh Transform Algorithm and Its Implementation with CUDA on GPUs," *Cybern. Inf. Technol.*, vol. 18, no. 5, p. 21–43, May 2018. [Online]. Available: https://doi.org/10.2478/cait-2018-0018

[7] J. Alman and K. Rao, "Faster Walsh-Hadamard and Discrete Fourier Transforms From Matrix Non-Rigidity," 2023. [Online]. Available: https://arxiv.org/abs/2211.06459

[8] W. Kahan, "Pracniques: further remarks on reducing truncation errors," *Commun. ACM*, vol. 8, no. 1, pp. 40–41, Jan. 1965. [Online]. Available: https://doi.org/10.1145/363707.363723

[9] A. Neumaier, "Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen," *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 54, no. 1, pp. 39–51, 1974. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.19740540106

[10] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2023, version 12.2, Accessed January 28, 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[11] Khronos Group, *OpenCL Specification*, 2021, version 3.0, Accessed January 28, 2025. [Online]. Available: https://www.khronos.org/registry/OpenCL/

[12] AMD, *ROCm Documentation*, 2025, accessed January 28, 2025. [Online]. Available: https://rocm.docs.amd.com/

[13] M. Taboga, "Kronecker product," https://www.statlect.com/matrix-algebra/Kronecker-product, 2021, lectures on matrix algebra.

[14] ——, "Properties of the Kronecker product," https://www.statlect.com/matrix-algebra/Kronecker-product-properties, 2021, lectures on matrix algebra.

[15] R. K. R. Yarlagadda and J. E. Hershey, *Hadamard Matrix Analysis and Synthesis: With Applications to Communications and Signal/Image Processing*. Boston: Kluwer Academic Publishers, 1997.

[16] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. [Online]. Available: https://www.jstor.org/stable/2003354

[17] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[18] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A Study of BFLOAT16 for Deep Learning Training," *arXiv preprint arXiv:1905.12322*, 2019, arXiv:1905.12322 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1905.12322

[19] N. J. Higham, "The Accuracy of Floating Point Summation," *SIAM Journal on Scientific Computing*, vol. 14, no. 4, pp. 783–799, 1993. [Online]. Available: https://doi.org/10.1137/0914050

[20] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Reading, Massachusetts: Addison-Wesley, 1997.

[21] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520. [Online]. Available: https://arxiv.org/abs/1801.04381

[22] NVIDIA Corporation, *cuFFT Library User's Guide*, 2022, accessed: 2025-01-27. [Online]. Available: https://docs.nvidia.com/cuda/cufft/

[23] NVIDIA Developer Forums, "Does cuFFT optimized by the tensor cores?" 2019, accessed: 2025-05-23. [Online]. Available: https://forums.developer.nvidia.com/t/does-cufft-optimized-by-the-tensor-cores/84105

[24] B. Li, S. Cheng, and J. Lin, "tcFFT: Accelerating Half-Precision FFT through Tensor Cores," 2021. [Online]. Available: https://arxiv.org/abs/2104.11471

[25] NVIDIA Corporation, *NVIDIA Ada GPU Architecture Tuning Guide*, 2025, version 12.9. [Online]. Available: https://docs.nvidia.com/cuda/ada-tuning-guide/index.html

[26] ——, *NVIDIA Hopper GPU Architecture Tuning Guide*, 2023, cUDA Toolkit Version 12.3. [Online]. Available: https://docs.nvidia.com/cuda/archive/12.3.0/hopper-tuning-guide/index.html

[27] W. Jakob, J. Rhinelander, D. Moldovan *et al.*, "pybind11 – Seamless operability between C++11 and Python," https://github.com/pybind/pybind11, 2016, version 2.x.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY