Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Distributed Fast Fourier Transform for Heterogeneous GPU Systems

Simon Egger

Course of Study:	Informatik
Examiner:	Prof. Dr. Miriam Schulte
Supervisor:	Malte Brunn, M.Sc.
Commenced:	April 22, 2021
Completed:	October 22, 2021

Abstract

The Fast Fourier Transform (FFT) is a numerical method to convert the input data to a representation in the frequency domain. A wide range of applications requires the computation of three-dimensional FFTs, which makes the utilization of Graphics Processing Units (GPUs) on distributed systems particularly appealing. The most common approach for distributed computation is to partition the global input data, resulting in slab decomposition or pencil decomposition.

For large numbers of processes, it is well known that slab decomposition only provides limited scalability and is generally outperformed by pencil decomposition. This often leaves their performance comparison on fewer GPUs as a blind spot: We found that slab decomposition generally dominates for larger input sizes when utilizing fewer GPUs, which is compliant with simple theoretical models. An exception to this rule is when the processor grid of pencil decomposition is specifically aligned to fully utilize available NVLink interconnections.

Next to the default implementation of slab decomposition and pencil decomposition, we propose Realigned as a possible optimization for both decomposition methods by taking advantage of cuFFT's advanced data layout. Most notably, Realigned reduced the additional memory requirements of pencil decomposition and computes the 1D-FFTs in y-direction more efficiently.

Since both decomposition methods require global redistribution of the intermediate results, we further compare the performance of different Peer2Peer and All2All communication techniques. In particular, we introduce Peer2Peer-Streams, which avoids the need for additional synchronization and allows the complete overlap of communication and packing phase. Our performance benchmarks show that this approach generally performs best for large input sizes on test systems with a limited number of GPUs when considering MPI without CUDA-awareness. Furthermore, we utilize custom MPI datatypes and adopt MPI_Type for GPUs, which reduces the additional memory requirements dramatically and avoids the need for a packing and unpacking phase altogether. By identifying a redistributed partition as a batch of slices, where each slice consists of the maximum number of contiguous, complex-valued words, we found that MPI_Type often poses a worthwhile consideration when both sent and received partitions are not composed of one-dimensional slices.

Acknowledgements

The author acknowledges support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1134-1 FUGG.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
cuFFT	CUDA Fast Fourier Transform Library.
DFT	Discrete Fourier Transform.
FFT	Fast Fourier Transform.
GPU	Graphics Processing Unit.
IDFT	Inverse Discrete Fourier Transform.
MPI	Message Passage Interface.
PCIe	Peripheral Component Interconnect Express.
SMP	Symmetric Multiprocessing.

1 Introduction

The *Fast Fourier Transform (FFT)* is considered to be one of the top 10 algorithms of the 20th century [DS00]. Most remarkably, it is able to compute the *Discrete Fourier Transform (DFT)* of N data points in $O(N \log N)$. In its general form, the FFT algorithm was introduced and analyzed by James William Cooley and John Tukey in 1965 [CT65]. Later, it was even discovered that Carl Friedrich Gauss used an equivalent algorithm to efficiently calculate the orbits of the asteroids Juno and Pallas in 1805 [HJB84]. As of today, there are many realizations and optimizations of the FFT algorithm used in a wide range of applications, e.g., biomedical imaging [BHB+20], turbulence simulation [LL11], molecular dynamics [PPS97], N-body simulation [HMF+13], and solvers for acoustic scattering problems [BK01].

For the previously named applications, it is often required to compute the FFT of three-dimensional, real-valued input data. Due to the separability property of multidimensional DFTs, this is often done by computing batched 1D-FFTs in each direction. For distributed systems, there are consequently two fundamental approaches for computing 3D-FFTs: *1D-Decomposition (Slab Decomposition)* and *2D-Decomposition (Pencil Decomposition)*. When considering input data of size $N_x \times N_y \times N_z$ and P processes, slab decomposition assumes that the global input is split in x-direction such that each process starts with a local input of size $\frac{N_x}{P} \times N_y \times N_z$. Since the computation of 1D-FFTs requires all data points of the particular direction, each process begins by computing the 2D-FFTs in y- and z-direction. Thereafter, the processes globally exchange their intermediate results to complete the procedure by computing the remaining 1D-FFT in x-direction. It is apparent that this approach only allows limited scalability for fixed input sizes (i.e. because of $P \leq N_x$). Therefore, it is common for applications to consider pencil decomposition: Here, a process grid of size $P_1 \times P_2$ (with $P = P_1 \cdot P_2$ processes) is considered, and the input data is assumed to be split in x- and y-direction respectively, i.e., each process starts with a local input of size $\frac{N_x}{P_1} \times \frac{N_y}{P_2} \times N_z$. Thus, the processes compute the 1D-FFTs in z-, y-, and x-direction individually, between which the processes have to exchange their intermediate results globally.

Related Work. There are many available CPU and GPU libraries for computing the Fast Fourier Transform. On single-node systems, hardware vendors usually provide their own FFT algorithms that are highly optimized for their hardware, e.g., NVIDIA's cuFFT library [NVIj], AMD's clFFT library [Adv], Intel's MKL library [Inta], and IBM's ESSL library [Intb]. When reviewing the available CPU libraries, FFTW [FJ05] stands out for being one of the earliest and most prominent. Most notably, modern libraries are often built on top of FFTW or follow the same paradigm of strictly differentiating between planning and executing: During the planing phase, the user specifies the problem, e.g., number of dimensions, problem size, and strides. Out of many available plans that fit the specification, FFTW then selects the plan that executes the fastest on the given hardware [FJ05]. Thereafter, the selected plan can be executed an arbitrary number of times. FFTW also supports computation on distributed systems by using slab decomposition [FFT]. To overcome the previously described scalability problem of slab decomposition, P3DFFT [Pek12] is

the first to introduce an open-source CPU library for utilizing both slab decomposition and pencil decomposition. Further CPU libraries that are extensively used in practice are: 2DECOMP&FFT used by Xcompact3D [LL10], fftMPI used by LAMMPS [Ste], SWFFT used by HACC [RAC+18], and OpenFFT used by OpenMX [DO14]. The most prominent CPU-GPU libraries available are AccFFT [GHMB16] and heFFTe [ATHD20]: AccFFT utilizes FFTW and cuFFT for computing the FFTs on each node and supports both slab decomposition and pencil decomposition. By utilizing CUDA-aware MPI, AccFFT also provides multiple options to realize the global redistribution of intermediate results. heFFTe provides even more flexibility and is designed for future exascale supercomputers. Most notably, it additionally supports brick decomposition and implements its custom non-blocking collective MPI routine for fine-tuning, which proves to be effective for small numbers of participating nodes. A performance comparison of the CPU and GPU libraries can be found in the report of Ayala et al. [ATL+21].

We stress that this work does not aim to compete with the already available GPU libraries AccFFT or heFFTe. Instead, we focus on possible optimizations of the individual decomposition methods, along with the applicable global redistribution methods, and compare their performance on distributed GPU systems. In this context, Takahashi introduces a simplified theoretical comparison of the communication time of slab decomposition and pencil decomposition [Tak20]. Takahashi shows that pencil decomposition is expected to perform better than slab decomposition if the number processes (i.e. GPUs) is large enough. Although this commonly requires that hundreds or thousands of processes participate, many authors use similar calculations to justify that they only consider pencil decomposition in their performance benchmarks. Therefore, the performance comparison of slab decomposition and pencil decomposition for fewer processes often remains a blind spot on distributed GPU systems.

Contribution. In this work, we compare the performance of slab decomposition and pencil decomposition on multiple distributed GPU systems, where the scalability of slab decomposition imposes no restriction. Furthermore, we consider possible optimizations by taking advantage of cuFFT's advanced data layout and propose Realigned, which is applicable for each decomposition method. In particular, Realigned is beneficial when considering pencil decomposition, since it reduces the additional memory requirements and performs the computation of the 1D-FFTs in y-direction more efficiently.

Since both slab decomposition and pencil decomposition require global redistribution of the intermediate results, we further study the performance of available strategies when using MPI with and without CUDA-awareness. For this purpose, we propose Peer2Peer-Streams which utilizes CUDA-streams to notify a secondary thread to start sending via MPI's non-blocking point-to-point communication routine `MPI_Isend`. Furthermore, we adapt MPI_Type for GPUs from the previous work of Dalcin et al. [DMK19], which is applicable for both Peer2Peer and All2All communication. To the best of our knowledge, neither approach was realized for distributed GPU systems before.

For the performance evaluation of the decomposition method and the underlying global redistribution method, we perform benchmarks on five test systems. Instead of focusing on the scalability results, we analyze the performance for a fixed number of GPUs in dependency of the underlying hardware, the problem size $N_x \times N_y \times N_z$, and the processor grid if pencil decomposition is considered. We further differentiate between computing the forward (real-to-complex) or inverse (complex-to-real) transform and whether MPI is used with or without CUDA-awareness.

Limitations. We stress that this work is by no mean exhaustive and that there are several limitations: (1) We perform our benchmarks with OpenMPI on all test systems for consistency reasons, which might not always result in the best performance [ATL+21]. (2) We assume that the global input data is partitioned in a specific manner for slab decomposition and for pencil decomposition, which results in limited flexibility, as opposed to heFFTe's brick decomposition [ATHD20]. (3) There are several decomposition methods and global redistribution methods which are not considered in this work. For example, OpenFFT implements an alternative decomposition method to reduce the communication volume [DO14]. Furthermore, we solely focus on non-blocking routines of MPI's point-to-point communication mechanisms.

Outline of this work. We start in Chapter 2 by introducing the theoretical fundamentals. Chapter 3 provides an overview of the relevant aspects of CUDA, cuFFT, and MPI for our implementation. Afterwards, a description of our implementation is given in Chapter 4, where we focus on GPU specific details of the decomposition methods, introduce Realigned, and present the global redistribution methods. The results of the performance benchmarks are presented in Chapter 5. Finally, we conclude this work in Chapter 6. Additional benchmark results can also be found in Appendix A.

2 Fundamentals

In this chapter, we introduce the relevant theoretical fundamentals which are used throughout this work. For additional reading material on signal processing and the Fast Fourier Transform, we recommend [VKG14] and [RKH10].

We start with the fundamentals of trigonometric interpolation in Section 2.1, before deriving the Discrete Fourier Transform in Section 2.2. Afterwards, we present the Cooley-Tukey FFT algorithm in Section 2.3. Since we are interested in computing three-dimensional FFTs, we generalize the one-dimensional case to higher dimension in Section 2.4. The fundamentals of the distributed computation of 3D-FFTs are discussed in Section 2.5. Finally, we introduce a possible strategy for validating the computed results of 3D-FFTs in Section 2.6.

2.1 Trigonometric Interpolation

For a given natural number $N \in \mathbb{N}$, the set of *trigonometric polynomials* of degree N is defined as

$$T_N = \{t : \mathbb{C} \to \mathbb{C} \mid t(z) = \sum_{k=0}^{N} c_k e^{ikz}\}.$$

Trigonometric interpolation is an approximation using trigonometric polynomials with $N \in \mathbb{N}$ and data points $\{(x_j, y_j)\}_{j=0}^{N} \subseteq \mathbb{C}^2$ as input parameters. The positions $\{x_j\}_{j=0}^{N}$ are equidistant on the interval $[0, 2\pi)$, i.e., $x_j = \frac{2\pi}{N+1} j$ for each $j = 0, \ldots, N$. The interpolation problem is defined as the computation of fitting coefficients $c_0, \ldots, c_N \in \mathbb{C}$ of a trigonometric polynomial $p \in T_N$, such that $t(x_j) = y_j$ for each data point of $\{(x_j, y_j)\}_{j=0}^{N}$.

It is well known that there exists a unique solution for the *polynomial interpolation problem*. For trigonometric interpolation, this fact is shown in Theorem 1. In the following, it is often convenient to define $\omega_{N+1} = e^{\frac{2\pi i}{N+1}}$. Using this convention, the $(N+1)$th roots of unity are given by:

$$\omega_{N+1}^0, \omega_{N+1}^1, \omega_{N+1}^2, \ldots, \omega_{N+1}^N$$

Theorem 1
For a given natural number $N \in \mathbb{N}$ and data points $\{(x_j, y_j)\}_{j=0}^{N} \subseteq \mathbb{C}^2$, where the positions $\{x_j\}_{j=0}^{N}$ are equidistant on the interval $[0, 2\pi)$, there exists a unique solution $t \in T_N$ which solves the formulated interpolation problem. The coefficients $c_0, \ldots, c_N \in \mathbb{C}$ are specified by

$$c_k = \frac{1}{N+1} \sum_{j=0}^{N} y_j e^{-\frac{2\pi i j k}{N+1}} = \frac{1}{N+1} \sum_{j=0}^{N} y_j \omega_{N+1}^{-jk}, \quad \text{for } k = 0, \ldots, N.$$

Proof. We start by showing the uniqueness of the solution. Rewriting the trigonometric polynomials as

$$t(z) = \sum_{k=0}^{N} c_k (e^{iz})^k,$$

it is apparent that the trigonometric interpolation problem can be formulated as a polynomial interpolation problem. Thus, the data points are modified to $\{(\tilde{x}_j, y_j)\}_{j=0}^{N} \subseteq \mathbb{C}^2$, where $\tilde{x}_j = e^{ix_j}$ for each $j = 0, \ldots, N$. Since the solution of the polynomial interpolation problem is unique, there exist a unique polynomial $p : \mathbb{C} \to \mathbb{C}$ of degree N with coefficients $c_0, \ldots, c_N$ such that

$$p(z) = \sum_{k=0}^{N} c_k z^k \quad \text{and } p(\tilde{x}_j) = y_j \text{ for each } j = 0, \ldots, N.$$

By definition, it follows that

$$t(x_j) = \sum_{k=0}^{N} c_k (e^{ix_j})^k = \sum_{k=0}^{N} c_k \tilde{x}_j^k = p(\tilde{x}_j) = y_j,$$

where the trigonometric polynomial $t \in T_N$ is uniquely defined by the coefficients $c_0, \ldots, c_N$. Therefore, the solution of the trigonometric interpolation problem is unique as well.

In order to compute the coefficients $c_0, \ldots, c_N$, the following observation for arbitrary $0 \leq l, k \leq N$ is used:

$$\sum_{j=0}^{N} \omega_{N+1}^{j(l-k)} = \begin{cases} N+1 & \text{if } l = k, \\ 0 = \frac{1-(\omega_{N+1}^{l-k})^{N+1}}{1-\omega_{N+1}^{l-k}} & \text{else.} \end{cases} \tag{2.1}$$

Finally, it follows that

$$\begin{aligned} \sum_{j=0}^{N} y_j \omega_{N+1}^{-jk} &= \sum_{j=0}^{N} \left(\sum_{l=0}^{N} c_l e^{ilx_j} \right) \omega_{N+1}^{-jk} = \sum_{j=0}^{N} \left(\sum_{l=0}^{N} c_l \omega_{N+1}^{lj} \right) \omega_{N+1}^{-jk} \\ &= \sum_{j=0}^{N} \sum_{l=0}^{N} c_l \omega_{N+1}^{j(l-k)} = \sum_{l=0}^{N} c_l \left(\sum_{j=0}^{N} \omega_{N+1}^{j(l-k)} \right) \overset{2.1}{=} (N+1) c_k. \end{aligned}$$

□

2.2 The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) takes an input signal $\mathbf{f} = (f_j)_{j=0}^{N} \in \mathbb{C}^{N+1}$ and computes $\mathcal{F}[\mathbf{f}] = (\hat{f}_k)_{k=0}^{N}$, where

$$\hat{f}_k = \sum_{j=0}^{N} f_j \omega_{N+1}^{-jk}, \text{ for each } k = 0, \ldots, N. \tag{2.2}$$

When compared to Section 2.1, the computed result $\mathcal{F}[\mathbf{f}]$ can be viewed as the set of scaled coefficients from the unique solution of the trigonometric interpolation problem (cf. Theorem 1). It is therefore common to say that $(\hat{f}_k)_{k=0}^{N}$ is located in frequency space.

Symmetrically to the DFT, the *Inverse Discrete Fourier Transform (IDFT)* takes $\hat{\mathbf{f}} = (\hat{f}_k)_{k=0}^N$ as input and computes $\mathcal{F}^{-1}\left[\hat{\mathbf{f}}\right] = (g_j)_{j=0}^N$, where

$$g_j = \sum_{k=0}^{N} \hat{f}_k \omega_{N+1}^{jk}.$$

Note that it is common for other literature to scale either the elements of $\mathcal{F}[\mathbf{f}]$ or $\mathcal{F}^{-1}\left[\hat{\mathbf{f}}\right]$ by $\frac{1}{N+1}$. For this work, the definitions above are chosen in order to coincide with the behavior of the utilized CUDA library for computing FFTs: *cuFFT* [NVIj]. Using this convention, the relationship between DFT and IDFT is highlighted in Theorem 2.

Theorem 2
Let $\boldsymbol{f} = (f_j)_{j=0}^N$ be an arbitrary input signal. It holds true that

$$\mathcal{F}^{-1}[\mathcal{F}[\boldsymbol{f}]] = (N+1)\boldsymbol{f}.$$

Proof. Implicitly using equidistant positions $(x_j)_{j=0}^N$, i.e., $x_j = \frac{2\pi}{N+1} j$ for each $j = 0, \ldots, N$, Theorem 1 states that for input values $\mathbf{f}$ there exists a unique solution $t \in T_N$ for the trigonometric interpolation problem. It states further, that the coefficients of t are specified by

$$c_k = \frac{1}{N+1} \sum_{j=0}^{N} f_j \omega_{N+1}^{-jk} = \frac{1}{N+1} \hat{f}_k, \quad \text{for each } k = 0, \ldots, N.$$

It follows that:

$$\begin{aligned}
(N+1) f_j &= (N+1) t(x_j) = (N+1) \sum_{k=0}^{N} c_k e^{ikx_j} \\
&= (N+1) \sum_{k=0}^{N} c_k \omega_{N+1}^{jk} = \sum_{k=0}^{N} \hat{f}_k \omega_{N+1}^{jk} = g_j \qquad \square
\end{aligned}$$

Instead of considering arbitrary complex input signals, real valued signals $\mathbf{f} = (f_j)_{j=0}^N \in \mathbb{R}^{N+1}$ are often used in practice. For such signals, it is easy to see, that

$$\begin{aligned}
\hat{f}_{N+1-k} &= \sum_{j=0}^{N} f_j \omega_{N+1}^{-j(N+1-k)} = \sum_{j=0}^{N} f_j \omega_{N+1}^{-j(N+1)+jk} \\
&= \sum_{j=0}^{N} f_j \omega_{N+1}^{jk} = \overline{\left(\sum_{j=0}^{N} f_j \omega_{N+1}^{-jk} \right)} = \overline{\hat{f}_k},
\end{aligned}$$

where $\overline{z}$ denotes the complex conjugate of $z \in \mathbb{C}$. Therefore, it is sufficient to compute $(\hat{f}_k)_{k=0}^{\lfloor \frac{N+1}{2} \rfloor}$ for real valued signals.

2.3 The Fast Fourier Transform

As seen in Section 2.2, a naive implementation of the Discrete Fourier Transform required $O(N^2)$ arithmetic operations. *Fast Fourier Transforms (FFT)* provide a family of algorithms that reduce the number of required arithmetic operations to $O(N \log(N))$. While there are many algorithms available, this work solely focuses on the *Cooley-Tukey FFT Algorithm* [CT65] [HJB84], which is utilized by the cuFFT library. As stated in the documentation, the cuFFT library implements *Radix-2, Radix-3, Radix-5*, and *Radix-7* as its main building blocks [NVIh]. This section follows [CT65] and [RKH10], where an additional overview of different FFT algorithms is given. In Section 2.3.1, we therefore start with the simplest form of the Cooley-Tukey FFT algorithm: The *Radix-2 FFT Algorithm*. Afterwards, the algorithm is generalized in Section 2.3.3.

2.3.1 The Radix-2 FFT Algorithm

Let $\hat{\mathbf{f}} = (\hat{f}_k)_{k=0}^N$ and $\hat{\mathbf{f}}^*$ denote the complex conjugate of the vector. It is easy to see that

$$\mathcal{F}^{-1}\left[\hat{\mathbf{f}}\right] = \mathcal{F}\left[\hat{\mathbf{f}}^*\right]^* \quad \text{(cf. Section 2.2).}$$

It is therefore sufficient to provide a fast algorithm for the forward DFT. For simplicity, we consider the Radix-2 FFT algorithm solely for input sizes $N = 2^m - 1$ (where $m \in \mathbb{N}$), such that there are $N+1 = 2^m$ data points $\mathbf{f} = (f_j)_{j=0}^N$. Splitting the data points into even and odd indices, the following holds true for $0 \leq k \leq N$:

$$\begin{aligned}
\hat{f}_k &= \sum_{j=0}^{N} f_j\, \omega_{N+1}^{-jk} = \sum_{j=0}^{\frac{N-1}{2}} f_{2j}\, \omega_{N+1}^{-2jk} + \sum_{j=0}^{\frac{N-1}{2}} f_{2j+1}\, \omega_{N+1}^{-(2j+1)k} \\
&= \sum_{j=0}^{\frac{N-1}{2}} f_{2j}\, \omega_{N+1}^{-2jk} + \omega_{N+1}^{-k} \sum_{j=0}^{\frac{N-1}{2}} f_{2j+1}\, \omega_{N+1}^{-2jk} = (*)
\end{aligned}$$

Due to

$$\omega_{N+1}^2 = e^{\frac{2\cdot 2\pi i}{N+1}} = e^{\frac{2\pi i}{(N+1)/2}} = \omega_{\frac{N+1}{2}},$$

it follows that:

$$(*) = \underbrace{\sum_{j=0}^{\frac{N-1}{2}} f_{2j}\, \omega_{\frac{N+1}{2}}^{-jk}}_{=:\hat{g}_k} + \omega_{N+1}^{-k} \underbrace{\sum_{j=0}^{\frac{N-1}{2}} f_{2j+1}\, \omega_{\frac{N+1}{2}}^{-jk}}_{=:\hat{h}_k} = \hat{g}_k + \omega_{N+1}^{-k} \hat{h}_k$$

Here, the signals $(\hat{g}_k)_{k=0}^N$ and $(\hat{h}_k)_{k=0}^N$ are periodic with period $\frac{N+1}{2}$. Computing the signal $(\hat{f}_k)_{k=0}^N$ can therefore be realized as

$$\begin{aligned}
\hat{f}_k &= \hat{g}_k + \omega_{N+1}^{-k} \hat{h}_k, && \text{and} \\
\hat{f}_{k+\frac{N+1}{2}} &= \hat{g}_k + \omega_{N+1}^{-(k+\frac{N+1}{2})} \hat{h}_k = \hat{g}_k - \omega_{N+1}^{-k} \hat{h}_k, && \text{for } k = 0, \ldots, \frac{N-1}{2}.
\end{aligned} \tag{2.3}$$

Counting an addition and a multiplication as a single arithmetic operation, the number of performed arithmetic operations can be analyzed as follows: Let $c : \mathbb{N}\backslash\{0\} \to \mathbb{N}$, such that $c(n)$ denotes the number of arithmetic operations performed on input size n. In the following, we show that $c(n) = n \log_2(n)$.

For the base case $N + 1 = 2$, Equation 2.2 uses two arithmetic operations to compute $\hat{\mathbf{f}}$, i.e., $c(2) = 2$. For $N + 1 = 2^m$ where $m > 1$, Equation 2.3 provides the recursion

$$c(N+1) = 2 \cdot c\left(\frac{N+1}{2}\right) + (N+1) = (**),$$

where the induction hypothesis states, that the computations of $(\hat{g}_k)_{k=0}^{N}$ and $(\hat{h}_k)_{k=0}^{N}$ perform $c\left(\frac{N+1}{2}\right) = \frac{N+1}{2} \log_2\left(\frac{N+1}{2}\right)$ arithmetic operations each. Therefore, it follows that:

$$\begin{aligned}(**) &= 2 \cdot \left(\frac{N+1}{2} \log_2\left(\frac{N+1}{2}\right)\right) + (N+1) \\ &= (N+1) \cdot \left(\log_2\left(\frac{N+1}{2}\right) + 1\right) = (N+1)\log_2(N+1)\end{aligned}$$

2.3.2 Radix-2 Butterfly Diagrams

A common visualization method of a FFT algorithm is the *Butterfly Diagram*. An exemplary diagram is presented in Figure 2.1, which depicts the Radix-2 FFT algorithm for $N + 1 = 8$. Analogously to Equation 2.3, $\hat{f}_k$ is computed as follows:

$$\hat{f}_k = \hat{g}_k + \omega_8^{-k} \hat{h}_k, \qquad \text{and} \tag{2.4}$$

$$\hat{f}_{k+4} = \hat{g}_k - \omega_8^{-(k+4)} \hat{h}_k, \quad \text{for } k = 0, \ldots, 3. \tag{2.5}$$

This exact procedure is shown on the right side of Figure 2.1, where arrows annotated with "-1" indicate that Equation 2.5 is used instead of Equation 2.4. Furthermore, the butterfly diagram visualizes the partition of even and odd indices for the input data of $(\hat{g}_k)_{k=0}^{3}$ and $(\hat{h}_k)_{k=0}^{3}$.

In the next step, the same recursion is repeated for $(\hat{g}_k)_{k=0}^{3}$ and $(\hat{h}_k)_{k=0}^{3}$, i.e.,

$$\begin{aligned}\hat{g}_k &= \hat{g}_k^{(0)} + \omega_4^{-k} \hat{h}_k^{(0)}, \quad \hat{g}_{k+2} = \hat{g}_k^{(0)} - \omega_4^{-(k+2)} \hat{h}_k^{(0)} \quad \text{and} \\ \hat{h}_k &= \hat{g}_k^{(1)} + \omega_4^{-k} \hat{h}_k^{(1)}, \quad \hat{h}_{k+2} = \hat{g}_k^{(1)} - \omega_4^{-(k+2)} \hat{h}_k^{(1)} \quad \text{for } k = 0, 1.\end{aligned}$$

Focusing on $(\hat{g}_k)_{k=0}^{3}$, which can be seen as the FFT on input data $\mathbf{f}_{\text{even}} = (f_0, f_2, f_4, f_6)$, the signal $(\hat{g}_0^{(0)}, \hat{g}_1^{(0)})$ is again computed using the partition of $\mathbf{f}_{\text{even}}$ consisting of even indices (i.e. (f_0, f_4)), while $(\hat{h}_0^{(0)}, \hat{h}_1^{(0)})$ is computed using the partition consisting of odd indices (i.e. (f_2, f_6)). Similarly, the signal $(\hat{h}_k)_{k=0}^{3}$ is determined, completing the computation.

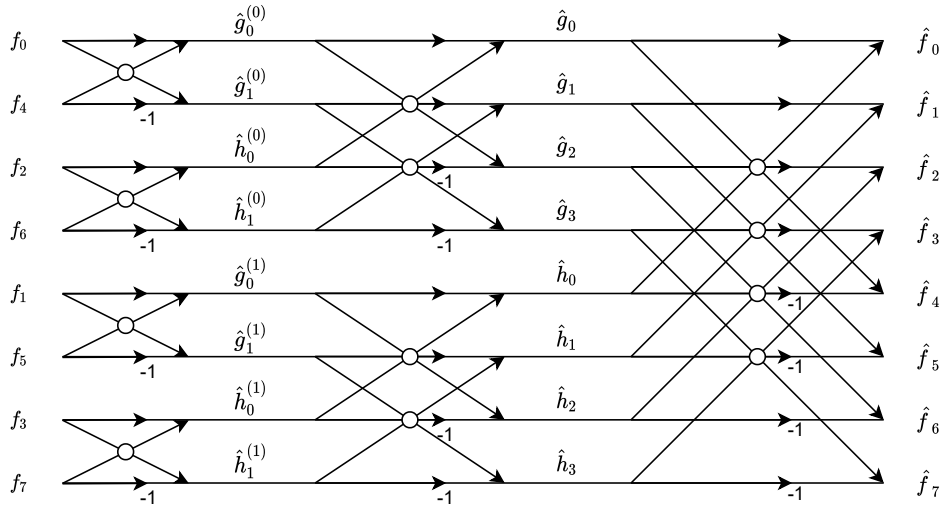


Figure 2.1: *Butterfly Diagram for the Radix-2 FFT Algorithm with $N+1=8$.* On the highest level (sc. the right-hand side of the diagram), a single butterfly represents the combined computation of $\hat{f}_k$ and $\hat{f}_{k+\frac{N+1}{2}}$, where latter is further emphasized by the annotation "-1". Analogously, the diagram visualizes the recursive calls for the computation of $\hat{g}_k$ and $\hat{h}_k$ ($0 \leq k < 4$).

2.3.3 The Cooley-Tukey FFT Algorithm

Following the work of J. Cooley and J. Tukey [CT65], the Radix-2 FFT algorithm (cf. Section 2.3.1) can be generalized as follows:

Let N be arbitrary and write $N+1=(N'+1)\cdot p$ for fitting $N', p \in \mathbb{N}$. Following the same construction of Section 2.3.1, we have for $0 \leq k \leq N$:

$$\begin{aligned}
\hat{f}_k &= \sum_{j=0}^{N} f_j \omega_{N+1}^{-jk} = \sum_{r=0}^{p-1}\sum_{j=0}^{N'} f_{pj+r}\omega_{N+1}^{-(pj+r)k} \\
&= \sum_{r=0}^{p-1} \omega_{N+1}^{-rk} \sum_{j=0}^{N'} f_{pj+r}\omega_{N+1}^{-pjk} = \sum_{r=0}^{p-1} \omega_{N+1}^{-rk} \underbrace{\sum_{j=0}^{N'} f_{pj+r}\omega_{N'+1}^{-jk}}_{=:\hat{g}_k^{(r)}} = \sum_{r=0}^{p-1} \hat{g}_k^{(r)} \omega_{N+1}^{-rk}
\end{aligned}$$

For $0 \leq r < p$, the values $\hat{g}_k^{(r)}$ are periodic with period $N'+1$. The signal $(\hat{f}_k)_{k=0}^{N}$ can therefore be computed in the following manner:

$$\hat{f}_{k+l(N'+1)} = \sum_{r=0}^{p-1} \hat{g}_k^{(r)} \omega_{N+1}^{-r(k+l(N'+1))}, \quad \text{for } 0 \leq k < N' \text{ and } 0 \leq l < p. \tag{2.6}$$

The computation starts by recursively determining the signals $(\hat{g}_k^{(r)})_{k=0}^{N'}$ for $0 \leq r < p$. Afterwards, the evaluation of $\hat{f}_{k+l(N'+1)}$ can be estimated to use less than p arithmetic operations (cf. Equation 2.6), where a single arithmetic operation is again counted as an addition combined with a

multiplication. This argument results in the following recursive formula for the number of performed arithmetic operations:

$$\begin{aligned} c(1) &= 0 \\ c(N+1) &= p \cdot c(N'+1) + p(N+1) \end{aligned} \tag{2.7}$$

Note that Equation 2.7 covers the special case $N' + 1 = 1$, where $c(p) = p^2$.

Before resolving the recursion in Theorem 3, the following lemma is introduced:

Lemma 1
Let $N + 1 = (N' + 1)p^s$ (for $s \geq 1$, $N' \geq 0$), such that p does not divide $(N' + 1)$ any further. It holds true that

$$c(N+1) = p^s c(N'+1) + sp(N+1).$$

Proof. We prove the statement by induction on $s \geq 1$. For the base case $s = 1$, the claim immediately follows from Equation 2.7. For $s > 1$, we have:

$$\begin{aligned} c(N+1) &= p \cdot c((N'+1)p^{s-1}) + p(N+1) && \text{(\textit{Equation} 2.7)} \\ &= p \cdot \left(p^{s-1}c(N'+1) + (s-1)p(N'+1)p^{s-1}\right) + p(N+1) && \text{(\textit{Induction Hypothesis})} \\ &= p \cdot \left(p^{s-1}c(N'+1) + (s-1)(N+1)\right) + p(N+1) \\ &= p^s c(N'+1) + (s-1)p(N+1) + p(N+1) \\ &= p^s c(N'+1) + sp(N+1) \end{aligned}$$

□

Again, Lemma 1 covers the special case $N' + 1 = 1$, resulting in

$$c(p^s) = p^s c(1) + sp \cdot (N+1) = p \cdot (N+1)\log_p(N+1). \tag{2.8}$$

Theorem 3
Let $N + 1 = p_1^{s_1} \cdots p_m^{s_m}$ be the prime factorization of $N + 1$, such that for each $1 \leq i \leq m$, the prime factor p_i does not divide $\frac{N+1}{p_i^{s_i}}$ any further. It holds true that

$$c(N+1) = (N+1)(s_1 p_1 + \ldots + s_m p_m).$$

Proof. We prove the statement by induction on $m \geq 1$. The claim of the base case $m = 1$ follows from Equation 2.8. For $m > 1$, we have:

$$\begin{aligned} c(N+1) &= p_1^{s_1} \cdot c(p_2^{s_2} \cdots p_m^{s_m}) + s_1 p_1 (N+1) && \text{(\textit{Lemma} 1)} \\ &= p_1^{s_1} \cdot \left(p_2^{s_2} \cdots p_m^{s_m}(s_2 p_2 + \ldots + s_m p_m)\right) + s_1 p_1 (N+1) && \text{(\textit{Induction Hypothesis})} \\ &= (N+1)(s_2 p_2 + \ldots + s_m p_m) + s_1 p_1 (N+1) \\ &= (N+1)(s_1 p_1 + \ldots + s_m p_m) \end{aligned}$$

□

Although it can be seen that any arbitrary factorization of $N + 1$ would suffice in Theorem 3, it appears obvious that the prime factorization results in the lowest runtime. Furthermore, input sizes solely consisting of small prime factors are theoretically preferable when using the Cooley-Tukey FFT algorithm.

For the sake of completeness, we state that while it is common for libraries like FFTW to provide highly optimized computation methods for input sizes consisting of prime factors 2, 3, 5 and 7, the library implementation might use significantly larger radices (e.g. *Radix-32*), depending on the underlying hardware [FJ05].

2.4 Expansion to Higher Dimensions

So far, we have seen that the DFT computes the scaled coefficients corresponding to the unique solution of the trigonometric interpolation problem (cf. Section 2.1 and Section 2.2). In this work, we are interested in computing the FFT of three-dimensional input data. Therefore, this section introduces the trigonometric interpolation problem for higher dimensions in Section 2.4.1, before exploring its coherence to multidimensional DFTs in Section 2.4.2.

2.4.1 Trigonometric Interpolation

In this section, trigonometric interpolation is introduced for sampling points in higher dimensions. Furthermore, we show that, analogously to the one-dimensional case (cf. Theorem 1), this interpolation problem has a unique solution as well. For the sake of simplicity, we only consider the two-dimensional extension and note that the expansion to even higher dimensions can be done in similar fashion.

Consider the input parameters $N_1, N_2 \in \mathbb{N}$, along with the data points:

$$\left\{ \left(x_{[j_1,j_2]}, y_{[j_1,j_2]}\right) \mid 0 \leq j_1 \leq N_1,\ 0 \leq j_2 \leq N_2,\ x_{[j_1,j_2]} = \left(\frac{2\pi j_1}{N_1+1}, \frac{2\pi j_2}{N_2+1}\right) \right\}$$

The interpolation problem is to find a fitting mapping $t : \mathbb{C}^2 \to \mathbb{C}$ of the form

$$t(z_1, z_2) = \sum_{k_1=0}^{N_1} \sum_{k_2=0}^{N_2} c_{[k_1,k_2]} e^{iz_1k_1} e^{iz_2k_2},$$

such that $t(x_{[j_1,j_2]}) = y_{[j_1,j_2]}$ for each $0 \leq j_1 \leq N_1$ and $0 \leq j_2 \leq N_2$. We start by showing that this interpolation problem has a unique solution.

Theorem 4
There exists a unique solution to the formulated two-dimensional trigonometric interpolation problem.

Proof. At first, let $0 \leq j_2 \leq N_2$ be fixed. Theorem 1 states that there exists a unique solution $u_{j_2} \in T_{N_1}$ for the one-dimensional trigonometric interpolation problem with data points

$$\left\{ \left(x_{j_1}, y_{[j_1,j_2]}\right) \mid 0 \leq j_1 \leq N_1 \text{ and } x_{j_1} = \frac{2\pi j_1}{N_1+1} \right\}. \tag{2.9}$$

Thus, there exist coefficients $\tilde{c}_{[0,j_2]}, \ldots, \tilde{c}_{[N_1,j_2]}$, such that

$$u_{j_2}(z) = \sum_{k_1=0}^{N_1} \tilde{c}_{[k_1,j_2]} e^{izk_1}, \quad \text{and} \tag{2.10}$$

$$y_{[j_1,j_2]} = u_{j_2}(x_{j_1}) = \sum_{k_1=0}^{N_1} \tilde{c}_{[k_1,j_2]} \omega_{N_1+1}^{j_1 k_1}, \quad \text{for } j_1 = 0, \ldots, N_1. \tag{2.11}$$

Secondly, let $0 \leq k_1 \leq N_1$ be fixed. As before, Theorem 1 states that the interpolation problem with data points

$$\left\{ \left(x_{j_2}, \tilde{c}_{[k_1,j_2]}\right) \mid 0 \leq j_2 \leq N_2 \text{ and } x_{j_2} = \frac{2\pi j_2}{N_2+1} \right\} \tag{2.12}$$

has a unique solution $v_{k_1} \in T_{N_2}$. Let $c_{[k_1,0]}, \ldots, c_{[k_1,N_2]}$ denote the coefficients, such that

$$v_{k_1}(z) = \sum_{k_2}^{N_2} c_{[k_1,k_2]} e^{izk_2}, \quad \text{and} \tag{2.13}$$

$$\tilde{c}_{[k_1,j_2]} = v_{k_1}(x_{j_2}) = \sum_{k_2}^{N_2} c_{[k_1,k_2]} \omega_{N_2+1}^{j_2 k_2}, \quad \text{for } j_2 = 0, \ldots, N_2. \tag{2.14}$$

Finally, we define $t : \mathbb{C}^2 \to \mathbb{C}$ with

$$t(z_1, z_2) = \sum_{k_1=0}^{N_1} \sum_{k_2=0}^{N_2} c_{[k_1,k_2]} e^{iz_1 k_1} e^{iz_2 k_2}.$$

Note that due to the uniqueness of the solutions for the one-dimensional trigonometric interpolation problem for the data points in 2.9 and 2.12, the coefficients $c_{[k_1,k_2]}$ are uniquely determined as well. Furthermore, the following holds true for arbitrary $0 \leq j_1 \leq N_1$ and $0 \leq j_2 \leq N_2$:

$$\begin{aligned}
t(x_{[j_1,j_2]}) &= \sum_{k_1=0}^{N_1} \sum_{k_2=0}^{N_2} c_{[k_1,k_2]} \omega_{N_1+1}^{j_1 k_1} \omega_{N_2+1}^{j_2 k_2} \\
&= \sum_{k_1=0}^{N_1} \omega_{N_1+1}^{j_1 k_1} \left(\sum_{k_2=0}^{N_2} c_{[k_1,k_2]} \omega_{N_2+1}^{j_2 k_2} \right) \\
&= \sum_{k_1=0}^{N_1} \omega_{N_1+1}^{j_1 k_1} \tilde{c}_{[k_1,j_2]} && (\textit{Equation } 2.14) \\
&= y_{[j_1,j_2]} && (\textit{Equation } 2.11) \qquad \square
\end{aligned}$$

There are two important aspects concerning the given proof of Theorem 4:

i) *First Observation:* The proof is constructive, i.e., it provides a scheme to compute the coefficients $(c_{[k_1,k_2]})_{k_1,k_2}$. In particular, we start by determining coefficients $\tilde{c}_{[0,j_2]}, \ldots, \tilde{c}_{[N_1,j_2]}$ for the data points $D_{j_2} = \{(x_{j_1}, y_{[j_1,j_2]})\}_{j_1=0}^{N_1}$ and fixed j_2. This corresponds to computing the one-dimensional DFT of D_{j_2}. Repeating the same procedure for each $0 \leq j_2 \leq N_2$ therefore

corresponds to the batched computation of one-dimensional DFTs in the first direction. Analogously, the coefficients $(\tilde{c}_{[k_1,j_2]})_{k_1,j_2}$ are used in the second iteration to compute the batched computation of one-dimensional DFTs in the second direction, which results in the coefficients $(c_{[k_1,k_2]})_{k_1,k_2}$.

ii) *Second Observation:* The proof can easily be extended to higher dimensions. For m dimensional sampling points of size $(N_1 + 1) \times \cdots \times (N_m + 1)$, we can start by determining the coefficients $\tilde{c}_{[0,j_2,\ldots,j_m]}, \ldots, \tilde{c}_{[N_1,j_2,\ldots,j_m]}$ for fixed $j_2, \ldots, j_m$ (cf. Equations 2.9 – 2.11). Afterwards, the collection of these coefficients $(\tilde{c}_{[j_1,j_2,\ldots,j_m]})_{j_1,\ldots,j_m}$ can be used by the induction hypothesis to uniquely determine the final coefficients $(c_{[j_1,j_2,\ldots,j_m]})_{j_1,\ldots,j_m}$.

2.4.2 Multidimensional Discrete Fourier Transforms

Following the structure of Section 2.4.1, we start by defining the DFT and IDFT for the two-dimensional case (short: *2D-DFT* and *2D-IDFT*).

Definition 2.4.1 (2D-DFT)
Let $\boldsymbol{f} = (f_{[j_1,j_2]})_{j_1,j_2}$ be an arbitrary input signal of size $(N_1 + 1) \times (N_2 + 1)$. Similarly to the one-dimensional case (cf. Section 2.2), we define the 2D-DFT to be $\mathcal{F}[\boldsymbol{f}] = (\hat{f}_{[k_1,k_2]})_{k_1,k_2}$, such that

$$\hat{f}_{[k_1,k_2]} = (N_1 + 1)(N_2 + 1) \cdot c_{[k_1,k_2]}$$

corresponds to the scaled coefficients $(c_{[k_1,k_2]})_{k_1,k_2}$ of the unique solution for the interpolation problem stated in Section 2.4.1.

In the first observation of the proof presented for Theorem 4, we already found that the 2D-DFT is *separable*, i.e., it can be expressed as a product of one-dimensional DFTs in each direction. Therefore, $\mathcal{F}[\mathbf{f}]$ can be computed in the following manner:

1. Split the input data into $N_2 + 1$ partitions $\mathbf{f}_{j_2} = (f_{[j_1,j_2]})_{j_1=0}^{N_1}$ and compute the one-dimensional DFT $\mathcal{F}\left[\mathbf{f}_{j_2}\right] = (\hat{f}_{k_1}^{(j_2)})_{k_1=0}^{N_1}$ for each $0 \leq j_2 \leq N_2$.
2. Afterwards, split the collected results $(\hat{f}_{k_1}^{(j_2)})_{k_1,j_2}$ into $N_1 + 1$ partitions $\mathbf{f}_{k_1} = (\hat{f}_{k_1}^{(j_2)})_{j_2=0}^{N_2}$ and compute the one-dimensional DFT $\mathcal{F}\left[\mathbf{f}_{k_1}\right] = (\hat{f}_{k_2}^{(k_1)})_{k_2=0}^{N_2}$ for each $0 \leq k_1 \leq N_1$.
3. The result is given by $\hat{f}_{[k_1,k_2]} = \hat{f}_{k_2}^{(k_1)}$ for each $0 \leq k_1 \leq N_1$ and $0 \leq k_2 \leq N_2$.

Definition 2.4.2 (2D-IDFT)
Let $\hat{\boldsymbol{f}} = (\hat{f}_{[k_1,k_2]})_{k_1,k_2}$ be an arbitrary input signal in frequency space of size $(N_1 + 1) \times (N_2 + 1)$. Similarly to the one-dimensional case (cf. Section 2.2), we define the 2D-IDFT to be $\mathcal{F}^{-1}\left[\hat{\boldsymbol{f}}\right] = (g_{[j_1,j_2]})_{j_1,j_2}$, where

$$g_{[j_1,j_2]} = \sum_{k_1=0}^{N_1} \sum_{k_2=0}^{N_2} \hat{f}_{[k_1,k_2]} \omega_{N_1+1}^{j_1 k_1} \omega_{N_2+1}^{j_2 k_2}, \quad \textit{for } 0 \leq j_1 \leq N_1 \textit{ and } 0 \leq j_2 \leq N_2.$$

To show the separability of the 2D-IDFT, we start by summarizing the computational steps:

1. Split the signal into $N_1 + 1$ partitions $\hat{\mathbf{f}}_{k_1} = (\hat{f}_{[k_1,k_2]})_{k_2=0}^{N_2}$ and compute the one-dimensional IDFT $\mathcal{F}^{-1}\left[\hat{\mathbf{f}}_{k_1}\right] = (g_{j_2}^{(k_1)})_{j_2=0}^{N_2}$ for each $0 \leq k_1 \leq N_1$.
2. Afterwards, split the collected result $(g_{j_2}^{(k_1)})_{k_1,j_2}$ into N_2 partitions $\hat{\mathbf{f}}_{j_2} = (g_{j_2}^{(k_1)})_{k_1=0}^{N_1}$ and compute the one-dimensional IDFT $\mathcal{F}^{-1}\left[\hat{\mathbf{f}}_{j_2}\right] = (g_{j_1}^{(j_2)})_{j_1=0}^{N_1}$ for each $0 \leq j_2 \leq N_2$.
3. The result is given by $g_{[j_1,j_2]} = g_{j_1}^{(j_2)}$ for each $0 \leq j_1 \leq N_1$ and $0 \leq j_2 \leq N_2$.

The correctness of above separability algorithm can be seen as follows:

$$\begin{aligned} g_{[j_1,j_2]} &= \sum_{k_1=0}^{N_1}\sum_{k_2=0}^{N_2} \hat{f}_{[k_1,k_2]}\omega_{N_1+1}^{j_1k_1}\omega_{N_2+1}^{j_2k_2} = \sum_{k_1=0}^{N_1}\omega_{N_1+1}^{j_1k_1}\left(\sum_{k_2=0}^{N_2}\hat{f}_{[k_1,k_2]}\omega_{N_2+1}^{j_2k_2}\right) \\ &= \sum_{k_1=0}^{N_1}\omega_{N_1+1}^{j_1k_1} g_{j_2}^{(k_1)} = g_{j_1}^{(j_2)} \end{aligned}$$

Using the generalization of the second observation in Section 2.4.1, it is easy to see how to extend the definitions of 2D-DFT and 2D-IDFT to higher dimensions. Analogously, the separability property is preserved for higher dimensions. The general computation of the m-dimensional DFT (IDFT) can therefore be considered as a product of m one-dimensional DFTs (IDFTs) in a similar manner as described above.

When considering a real-valued input signal $(f_{[j_1,\ldots,j_m]})_{j_1,\ldots,j_m}$ of size $(N_1 + 1) \times \cdots \times (N_m + 1)$, the separability property yields a similar symmetry of the computed signal $(\hat{f}_{[k_1,\ldots,k_m]})_{k_1,\ldots,k_m}$ as in the one-dimensional case: Without loss of generality we assume that the computation starts by calculating the batched one-dimensional DFTs in the first direction. Thus, we compute the 1D-DFTs for input data $(f_{[j_1,\ldots,j_m]})_{j_1=0}^{N_1}$ for each $l = 2, \ldots, m$ and $0 \leq j_l \leq N_l$. Using the symmetry property presented in Section 2.2, it is sufficient to compute and store the complex-valued data

$$(\hat{f}_{k_1}^{(j_2,\ldots,j_m)})_{k_1=0}^{\left\lfloor \frac{N_1+1}{2} \right\rfloor}, \quad \text{for } l = 2, \ldots, m \text{ and } 0 \leq j_l \leq N_l.$$

Therefore, the output size of a real-valued m-dimensional DFT is often considered to be

$$\left(\left\lfloor \frac{N_1 + 1}{2} \right\rfloor + 1\right) \times (N_2 + 1) \times \cdot \times (N_m + 1).$$

In particular, the signal $(\hat{f}_{[k_1,\ldots,k_m]})_{k_1,\ldots,k_m}$ satisfies *Hermitian Symmetry*, which is defined as

$$\hat{f}_{[N_1+1-k_1,\ldots,N_m+1-k_m]} = \overline{\hat{f}_{[k_1,\ldots,k_m]}}, \quad \text{for } l = 1, \ldots, m \text{ and } 0 \leq k_l \leq N_l,$$

where $\overline{z}$ denotes the complex conjugate of $z \in \mathbb{C}$.

To proof this property, we use induction on $m \geq 1$. For the base case $m = 1$, the statement was already shown in Section 2.2. For $m > 1$, we have:

$$\begin{aligned}
\hat{f}_{[N_1+1-k_1,\ldots,N_m+1-k_m]} &= \sum_{j_m=0}^{N_m} \hat{f}^{(j_m)}_{[N_1+1-k_1,\ldots,N_{m-1}+1-k_{m-1}]} \omega_{N_m+1}^{-j(N_m+1-k_m)} && \text{(Separability)} \\
&= \sum_{j_m=0}^{N_m} \overline{\left(\hat{f}^{(j_m)}_{[k_1,\ldots,k_{m-1}]}\right)} \omega_{N_m+1}^{+jk_m} && \text{(Induction Hypothesis)} \\
&= \overline{\left(\sum_{j_m=0}^{N_m} \hat{f}^{(j_m)}_{[k_1,\ldots,k_{m-1}]} \omega_{N_m+1}^{-jk_m}\right)} \\
&= \overline{\hat{f}_{[k_1,\ldots,k_m]}} && \text{(Separability)}
\end{aligned}$$

2.4.3 Multidimensional Fast Fourier Transforms

As previously introduced in Section 2.3.3, the Cooley-Tukey FFT algorithm performs

$$(N+1)(s_1 p_1 + \ldots + s_m p_m)$$

arithmetic operations for input sizes $N + 1$ and the prime factorization $N + 1 = p_1^{s_1} \cdots p_m^{s_m}$. In Section 2.4.2, we found that the computation of the m-dimensional DFT is separable. Analogously, the Cooley-Tukey FFT algorithm can easily be generalized for higher dimensions as a product of one-dimensional FFTs in each dimension.

To analyze the number of performed arithmetic operations, we exemplarily consider the two-dimensional and note, that the cost analysis can again easily be generalized for higher dimensions: Let $N_1, N_2 \in \mathbb{N}$ and $N_1 + 1 = p_1^{s_1} \cdots p_m^{s_m}$, $N_2 + 1 = q_1^{r_1} \cdots q_n^{r_n}$ be the prime factorizations. For the first computation step presented in Section 2.4.2, we compute $N_2 + 1$ FFTs of size $N_1 + 1$, resulting in a total of

$$(N_2+1)(N_1+1) \cdot (s_1 p_1 + \ldots + s_m p_m)$$

performed arithmetic operations. For the second computational step, we compute $N_1 + 1$ FFTs of size $N_2 + 1$, resulting in a total of

$$(N_1+1)(N_2+1) \cdot (r_1 q_1 + \ldots + r_n q_n)$$

performed arithmetic operations. Combining the costs of both steps, the 2D Cooley-Tukey FFT algorithm performs

$$(N_1+1)(N_2+1) \cdot (s_1 p_1 + \ldots + s_m p_m + r_1 q_1 + \ldots + r_n q_n)$$

arithmetic operations. The special case $N_1 + 1 = p^s$, $N_2 + 1 = p^r$ for $r, s \in \mathbb{N}$ and prime number p therefore results in costs

$$(N_1+1)(N_2+1) \cdot (sp + rp) = p \cdot (N_1+1)(N_2+1) \cdot \log_p((N_1+1)(N_2+1)).$$

2.5 Approaches for Distributed Computation

After introducing the necessary prerequisites for the computation of multidimensional FFTs in Section 2.4, we focus now on three-dimensional input data of size $(N_1 + 1) \times (N_2 + 1) \times (N_3 + 1)$. Furthermore, this section solely concentrates on the basic computation methods, while further details on the GPU implementation are provided in Chapter 4. For convenience, we denote the axes by x, y, and z and write $N_x \times N_y \times N_z$ instead of $(N_1 + 1) \times (N_2 + 1) \times (N_3 + 1)$. If not stated otherwise, we further assume that the data is aligned in z-direction, i.e., position (x, y, z) is accessed via

$$\text{data}[x \cdot N_y N_z + y \cdot N_z + z].$$

Since multidimensional DFTs are separable (cf. Section 2.4.2), the general approach for distributed computation is to partition the input data and sequentially compute the 1D-FFTs for each direction, while exchanging the intermediate results with other processes. Therefore, the fundamental approaches can be categorized into the 1D-Decomposition (*Slab Decomposition*) and the 2D-Decomposition (*Pencil Decomposition*) of the input data. We start by introducing slab decomposition in Section 2.5.1 and pencil decomposition in Section 2.5.2. Afterwards, Section 2.5.3 introduces some simple observations concerning the key differences between both approaches.

2.5.1 Slab Decomposition

Let P be the number of processes involved in the computation. Slab decomposition then splits the global input data of size $N_x \times N_y \times N_z$ along the x-axis and assigns each process a *slab*. More specifically, the input data is split into partitions of size $\frac{N_x}{P} \times N_y \times N_z$. Algorithm 2.1 provides an overview of the procedure for process p (with $0 \leq p < P$), where we assume for the sake of simplicity that P divides N_x and N_y.

Algorithm 2.1 Slab Decomposition for Process p

Input: Data of size $\frac{N_x}{P} \times N_y \times N_z$

Output: Data of size $N_x \times \frac{N_y}{P} \times N_z$

1: Compute the 2D-FFT in z- and y-direction

2: Split the computed results of step 1 into P partitions in y-direction (of size $\frac{N_x}{P} \times \frac{N_y}{P} \times N_z$) and transmit the kth partition to the process k

3: After receiving all partitions, compute the remaining 1D-FFT in x-direction

Step 2 of the algorithm is further visualized in Figure 2.2 and consists of the *Packing*, *Communication*, and *Unpacking Phase*: After computing the 2D-FFT in z- and y-direction, the data is split into equally sized partitions along the y-axis. Figure 2.2 exemplarily highlights the third partition for each process, which can be seen to reside dissected in memory. Therefore, the packing phase is used to copy the partition into a contiguous send buffer. Thereafter, in the communication phase process p sends the kth partition to process k (for each $0 \leq k < P$ with $k \neq p$) and symmetrically receives its pth partition. After receiving all incoming data, the data is usually copied into contiguous memory again, as part of the unpacking phase, before computing the remaining 1D-FFT in x-direction. Disregarding GPU implementation details (like host-to-device memory transfers), it can be seen in Figure 2.2 that the unpacking phase can be omitted for slab decomposition.

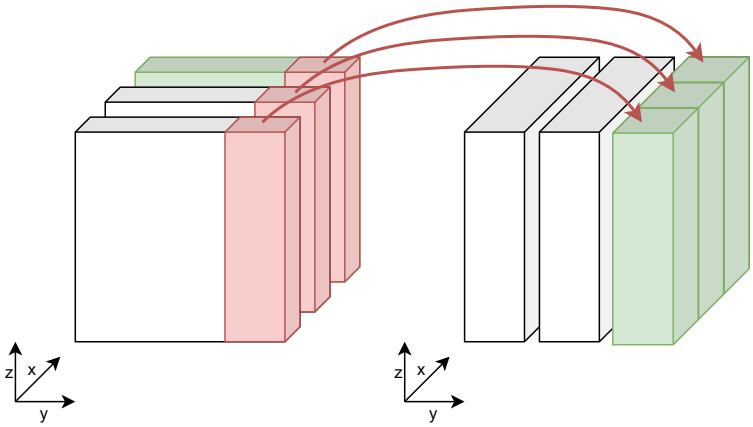


Figure 2.2: *Slab Decomposition for* 3 *Processes*. The green slab highlights the third process, which receives the red partitions from each process to compute the remaining 1D-FFT in x-direction.

2.5.2 Pencil Decomposition

Here, a process grid $P_1 \times P_2$ is considered. A single process is then identified as (p_i, p_j), for $0 \leq p_i < P_1$ and $0 \leq p_j < P_2$. Pencil decomposition then splits the global input of size $N_x \times N_y \times N_z$ along the x- and y-axis and assigns each process a *pencil*. More specifically, the input data is split into partitions of size $\frac{N_x}{P_1} \times \frac{N_y}{P_2} \times N_z$. Algorithm 2.2 provides an overview of the procedure for process (p_i, p_j). For the sake of simplicity, we assume again that P_1 and P_2 divide N_x, N_y, and N_z.

Algorithm 2.2 Pencil Decomposition for Process (p_i, p_j)

Input: Data of size $\frac{N_x}{P_1} \times \frac{N_y}{P_2} \times N_z$.

Output: Data of size $N_x \times \frac{N_y}{P_1} \times \frac{N_z}{P_2}$.

1: Compute the 1D-FFT in z-direction.
2: Split the computed results of step 1 into P_2 partitions in z-direction (of size $\frac{N_x}{P_1} \times \frac{N_y}{P_2} \times \frac{N_z}{P_2}$) and transmit the kth partition to process (p_i, k), for each $0 \leq k \leq P_2 - 1$.
3: After receiving all partitions, compute the 1D-FFT in y-direction.
4: Split the computed results of step 3 into P_1 partitions in y-direction (of size $\frac{N_x}{P_1} \times \frac{N_y}{P_1} \times \frac{N_z}{P_2}$) and transmit the kth partition to process (k, p_j), for each $0 \leq k \leq P_1 - 1$.
5: After receiving all partitions, compute the 1D-FFT in x-direction.

Steps 2 and 4 of Algorithm 2.2 are visualized in Figure 2.3 and consist of a packing, communication, and unpacking phase (cf. Section 2.5.1). Here, it can be seen that the first global redistribution (cf. step 2 in Algorithm 2.2) requires both packing and unpacking phase, while the second global redistribution (cf. step 4 in Algorithm 2.2) only requires a packing phase. Figure 2.3 shows further, that for each $0 \leq p_i < P_1$ only processes $(p_i, 0), \ldots, (p_i, P_2 - 1)$ have to communicate in the first global redistribution. Likewise, only the processes $(0, p_j), \ldots, (P_1 - 1, p_j)$ have to communicate in the second global redistribution, for each $0 \leq p_j < P_2$.

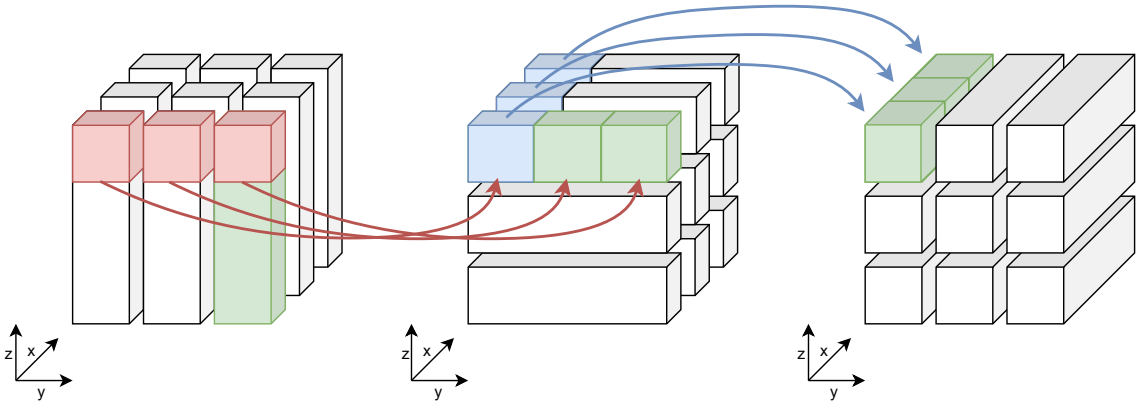


Figure 2.3: *Pencil Decomposition for a* 3×3 *Process Grid.* The highlighted axis marks the direction in which the 1D-FFTs are computed. Furthermore, the green pencil identifies process $(0, 2)$, which receives the red and blue partitions in the first and second global redistribution, respectively.

2.5.3 Theoretical Comparison of Slab and Pencil Decomposition

After the introduction of both slab and pencil decomposition, this section aims to summarize some simple key differences between both approaches. As before, let P denote the number of processes. For pencil decomposition, we consider the process grid $P_1 \times P_2$ (with $P = P_1 \cdot P_2$) such that P_1 and P_2 denote the number of input partitions in x-direction and y-direction, respectively. Finally, we consider arbitrary complex-valued data as input and assume $N = N_x = N_y = N_z$ for the sake of simplicity.

The first key difference of both approaches is that slab decomposition requires a single global redistribution, while pencil decomposition requires a second global redistribution. This is caused by the fact that while multidimensional DFTs can be considered as a product of 1D-DFTs in each direction, the computation of a single 1D-DFT (1D-FFT) still requires all data points of the corresponding direction (cf. Section 2.4.2). Additionally, it is easy to see that P processes are involved in the single communication phase of slab decomposition, whereas there are P_2 and P_1 processes involved in the first and second communication phase of pencil decomposition, respectively.

Secondly, it is obvious that slab decomposition only provides limited scalability. Since the input data is solely partitioned in x-direction, the number of processes is trivially limited by $N_x = N$. In contrast, pencil decomposition allows to scale the number of utilized processes up to $N_x \cdot N_y = N^2$.

Next, we can determine the total number of bytes which are transmitted during the computation. For slab decomposition, each process starts with a complex-valued input partition of size $\frac{N^3}{P}$. Depending on the used precision, this partition allocates $2s \cdot \frac{N^3}{P}$ bytes of memory with $s = 4$ for single- and $s = 8$ for double-precision. As we have seen in Section 2.5.1, each process transmits a partition of size $\frac{N^3}{P^2}$ to the other $P - 1$ processes. Therefore, each process transmits and receives a total of $(P - 1)\frac{2sN^3}{P^2}$ bytes. Repeating similar arguments for pencil decomposition, here, each

process transmits and receives $(P_2 - 1)\frac{2sN^3}{PP_2}$ bytes in the first and $(P_1 - 1)\frac{2sN^3}{PP_1}$ bytes in the second communication phase. Thus, the total number of bytes transmitted and received are

$$(P_2 - 1)\frac{2sN^3}{PP_2} + (P_1 - 1)\frac{2sN^3}{PP_1} = (P - 1)\frac{2sN^3}{P^2} + (P_1 - 1)(P_2 - 1)\frac{2sN^3}{P^2},$$

for each process.

Finally, we can introduce a model to estimate the duration of the communication phases, similar to [Tak20], [HGTT10]. For now, we consider simplified conditions and assume that the duration for two communicating processes can be modeled by $\frac{d}{B} + L$, where d denotes the number of bytes transmitted, B the bandwidth and L the latency caused by the connection establishment. Furthermore, the model presented in this section disregards possible influences of network congestion and topology.

Using the previous analysis of the number of transmitted bytes, the duration of the communication phase can be estimated in a straightforward manner: For slab decomposition, we have

$$T_C^S = (P - 1)\frac{2sN^3}{P^2B} + (P - 1)L.$$

Analogously, we have the following model for pencil decomposition:

$$\begin{aligned} T_C^P &= (P_2 - 1)\frac{2sN^3}{PP_2B} + (P_2 - 1)L + (P_1 - 1)\frac{2sN^3}{PP_1B} + (P_1 - 1)L \\ &= (P - 1)\frac{2sN^3}{P^2B} + (P_1 - 1)(P_2 - 1)\frac{2sN^3}{P^2B} + (P_1 + P_2 - 2)L \end{aligned}$$

To compare the communication duration of slab and pencil decomposition, we assume $P_1, P_2 > 1$ and consider a fixed latency L and bandwidth B. It then holds true that:

$$\begin{aligned} T_C^P < T_C^S &\iff (P_1 - 1)(P_2 - 1)\frac{2sN^3}{P^2B} < (P - P_1 - P_2 + 1)L \\ &\iff (P_1 - 1)(P_2 - 1)\frac{2sN^3}{P^2B} < (P_1 - 1)(P_2 - 1)L \\ &\iff \frac{P^2}{N^3} > \frac{2s}{LB} \end{aligned}$$

As an exemplarily setup (e.g. where multiple GPUs are located on the same node) we consider a bandwidth of $B = 25 \cdot 10^9$ bytes per second and a latency of $L = 2 \cdot 10^{-5}$ seconds. For $P = 16$ processes and double-precision, pencil decomposition is faster than slab decomposition for $N < 200$, according to the model above.

2.6 Validation Methods

After introducing the general approaches for the distributed computation of three-dimensional FFTs in Section 2.5, this section presents the method which will be later used in Chapter 5 to validate the computed results of our benchmarks.

Fundamentally, we compute the Laplacian of a known function with a forward and inverse transform for each implemented variant and compare the approximated results with the exact results. Since we focus on three-dimensional input data, we compute the Laplacian of 2π-periodic and differentiable functions $f : \mathbb{R}^3 \to \mathbb{R}$, i.e.,

$$\Delta f(x, y, z) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2}.$$

As introduced in Section 2.1, the discrete sampling points are $x_j = y_j = z_j = \frac{2\pi j}{N+1}$, when assuming $N_x = N_y = N_z = N + 1$ for the sake of simplicity. Therefore, it remains to show how the individual second order partial derivatives can be approximated using the Fast Fourier Transform. Furthermore, due to the separability of multidimensional DFTs (cf. Section 2.4.2), we solely introduce a method for the one-dimensional case. For this, we follow the work of S. G. Johnson in [Joh11].

In Section 2.1, we introduced the trigonometric interpolation problem for given data points $\{(x_j, y_j)\}_{j=0}^{N} \subseteq \mathbb{C}^2$ and $x_j = \frac{2\pi}{N+1} j$ to be the computation of fitting coefficients $c_0, \ldots, c_N$, such that the trigonometric polynomial

$$t(x) = \sum_{k=0}^{N} c_k e^{ikx}$$

satisfies $t(x_j) = y_j$ for each $0 \leq j \leq N$. The first approach for approximating the second order derivatives would be to simply compute $t''(x_j)$, for each x_j. Here, it can be seen that there is a certain kind of ambiguity. In particular, the approximation t could be replaced by

$$t(x) = \sum_{k=0}^{N} c_k e^{i(k+m_k(N+1))x},$$

and arbitrary choices of $\{m_k\}_{k=0}^{N}$, without violating the constraint $t(x_j) = y_j$ for each x_j. On the other hand, the choice of the parameters m_k has an enormous effect on the resulting derivative. Thus, Johnson describes the *unique minimal-oscillation trigonometric interpolation*, with

$$t(x) = c_0 + \sum_{k=1}^{\frac{N-1}{2}} \left(c_k e^{ikx} \right) + \sum_{k=1}^{\frac{N-1}{2}} \left(c_{N+1-k} e^{-ikx} \right) + c_{\frac{N+1}{2}} e^{i\frac{N+1}{2}x}.$$

Note that the coefficients c_k can still be computed by using the DFT. Furthermore, we implicitly assume that $N + 1$ is even and note that the adjustments for odd values of $N + 1$ are analogous. The newly formulated interpolation problem can therefore be seen as limiting the frequencies by $|k + m_k(N + 1)| \leq \frac{N+1}{2}$, which results in the fewest oscillations possible [Joh11]. The second order derivative can then be approximated by simply computing:

$$\begin{aligned} y_j'' := t''(x_j) &= \sum_{k=1}^{\frac{N-1}{2}} \left((-k^2) c_k e^{ikx_j} \right) + \sum_{k=1}^{\frac{N-1}{2}} \left((-k^2) c_{N+1-k} e^{-ikx_j} \right) - \left(\frac{N+1}{2} \right)^2 c_{\frac{N+1}{2}} e^{i\frac{N+1}{2}x_j} \\ &= \sum_{k=0}^{N} \alpha_k c_k e^{i\frac{2\pi jk}{N+1}}, \quad \text{where } \alpha_k = \begin{cases} -k^2 & \text{if } 0 \leq k \leq \frac{N+1}{2}, \\ -(N+1-k)^2 & \text{else.} \end{cases} \end{aligned}$$

In summary, the algorithm starts by computing the Fourier coefficients c_k for input $\{y_j\}_{j=0}^{N}$ with a forward FFT iteration. Afterwards, the coefficient c_k is multiplied with α_k, before computing the inverse FFT of the modified coefficients to obtain y_j''.

3 Utilized Software

Section 2.3 introduced how to compute a 1D-DFT with time complexity $\mathcal{O}(N \log N)$, using Fast Fourier Transforms. After generalizing DFTs for higher dimensions in Section 2.4, we have seen that multidimensional DFTs are separable and can therefore be computed efficiently by using a product of 1D-FFTs in each direction. Afterwards in Section 2.5, we presented the general approaches for the distributed computation of 3D-FFTs, where both slab decomposition and pencil decomposition required global redistribution of the intermediate results.

In preparation for an in-depth description of our GPU library for computing distributed FFTs on three-dimensional input data in Chapter 4, this chapter provides an overview of the utilized software. We start by introducing the basic functionalities of CUDA in Section 3.1, along with cuFFT in Section 3.2, which is an high-performance library for computing 1D-, 2D-, and 3D-FFTs on a single host. For process communication between GPUs, which are located on the same or on different nodes, we describe the relevant principles of MPI in Section 3.3.

3.1 The CUDA Toolkit

The CUDA Toolkit was introduced by the NVIDIA corporation and provides an environment for developing parallel applications on NVIDIA GPUs [NVIk]. While this work focuses on the CUDA extension of C++, additional support is provided for programming languages like FORTRAN, Java, and Python [NVIa].

This section aims to provide information about the fundamental concepts of CUDA, which is later required in Chapter 4. A more in-depth outline of the provided functionalities can be found in the CUDA programming guide [NVIa]. In particular, this section covers the relevant parts of device [NVIb], memory [NVId], and stream management [NVIe] and follows the corresponding documentation, for reference. We start by introducing a simple example C++/CUDA program, before addressing the more advanced functions in details.

General Structure of a C++/CUDA Program. CUDA generally considers a separate *device* (i.e. the GPU) for executing the CUDA threads and the *host* (i.e. the CPU) which is responsible for executing the C++ program [NVIa]. Furthermore, it is assumed that the device has its own physically separate *device memory*. A general structure of a simple C++/CUDA program can be found in Listing 3.1. We note that this program does not handle any errors for the sake of simplicity. The program starts by allocating host and device memory. For the allocation of device memory, CUDA provides the method `cudaError_t cudaMalloc(void **devPtr, size_t size)`, where parameter `size` is given in bytes. As an example, the return value of the method might indicate that the allocation was successful or that there is not enough available device memory available as requested. For convenience, the return type is often neglected in the following. Usually, the input data is gathered by the host, e.g., by

Listing 3.1 Exemplary structure of a simple C++/CUDA program.

```
int main() {
    size_t N = ...;
    double *h_data, *d_data;

    // host and device memory allocation
    h_data = new double[N];
    cudaMalloc((void **)&d_data, N*sizeof(double));

    // initialize data on host (h_data)
    ...

    // copy data from host to device memory
    cudaMemcpy(d_data, h_data, N*sizeof(double), cudaMemcpyHostToDevice);

    // perform parallel computation on GPU and store modifications in d_data
    ...

    // copy results from device to host memory
    cudaMemcpy(h_data, d_data, N*sizeof(double), cudaMemcpyDeviceToHost);

    // evaluate results on host
    ...

    // free allocated memory
    delete[] h_data;
    cudaFree(d_data);

    return 0;
}
```

user interaction or by reading from disc. After potentially preprocessing the input data, it is copied to device memory, where the data is accessible by the GPU. There is a large variety of methods that can be used form memory transfer in CUDA, which will be discussed later. For now, we consider simply `cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind)`, where source and destination pointer, along with the number of transferred bytes, are specified. For copying data from host to device, parameter `kind` can be set to `cudaMemcpyHostToDevice`. Afterwards, the data stored in `d_data` can be used for computation on the GPU. As an example, the CUDA library cuFFT (see Section 3.2 for details) could be used to compute the in-place 1D-FFT. Similar as before, the computed results can be copied back to host memory, where further postprocessing can occur. At the end of the computation, the device memory is freed by using `cudaFree(void *devPtr)`.

Device Management. In case there are multiple GPUs located on a single node, CUDA provides device management [NVIb] to select a specific GPU for the execution of the CUDA threads. This is required by our implementation, since each input partition (cf. Section 2.5) is assigned to exactly one GPU. When executing on a given node, the function `cudaGetDeviceCount(int *count)` can be used to determine the available number of GPUs of this node, where `count` is used as the output parameter. Afterwards, `cudaSetDevice(int device)` can be used for $0 \leq$ `device` $<$ `count` to specify the GPU on which future CUDA threads shall be executed.

Memory Management. Listing 3.1 shows how CUDA allocates and frees device memory. Furthermore, the example illustrates the use of `cudaMemcpy` for contiguous memory. Since we have already seen in Section 2.5 that the packing phase, for example, has to copy non-contiguous partitions into the send-buffer, `cudaMemcpy` is not always sufficient. Therefore, CUDA provides the copy-operation `cudaMemcpy2D`, whose specification can be found in Listing 3.2. Here, `width` specifies the column size in bytes, while `height` represents the number of rows. The parameters `dpitch` and `spitch` specifies the distance (in bytes) between the first byte of two consecutive rows. Therefore, the values of both parameters has to be greater than, or equal to the value of `widht`. To further illustrate the usage of `cudaMemcpy2D`, consider the packing phase visualized in Figure 2.2. For the sake of simplicity, we assume complex-valued input data in single-precision, such that the highlighted partitions in red are of size $\frac{N_x}{P} \times \frac{N_y}{P} \times N_z$ and a complex number is stored using 8 bytes. To copy this partition into a contiguous send-buffer, both `dpitch` and `width` are set to $8 \cdot N_z \frac{N_y}{P}$, while `spitch` and `height` are set to $8 \cdot N_z N_y$ and $\frac{N_x}{P}$, respectively.

Listing 3.2 Specification of cudaMemcpy2D.

```
cudaError_t cudaMemcpy2D(void *dst, size_t dpitch, const void *src, size_t spitch,
    size_t width, size_t height, cudaMemcpyKind kind)
```

Asynchronous Operations and Stream Management. Until now, we only considered methods that are synchronous, with respect to the host. For example, `cudaMemcpy` returns for most memory transfers only after the copy is complete. In contrast, CUDA also provides the asynchronous methods `cudaMemcpyAsync` and `cudaMemcpy2DAsync`, which allow the possibility of an overlap between computation and memory transfer. As there are some exceptions to this rule, we refer the interested reader to [NVIf], for more information. In addition, asynchronous operations can be assigned to a specific CUDA stream. While individual streams can be seen as a first in, first out queue for commands, two different streams may execute operations concurrently [NVIg]. With regard to the packing and unpacking phase of slab and pencil decomposition, this peculiarity allows to copy the individual partitions simultaneously. As a part of stream management [NVIe], streams can be created with `cudaStreamCreate(cudaStream_t *pStream)`. In order to wait for the completion of a stream's tasks, `cudaStreamSynchronize(cudaStream_t stream)` can be called. Alternatively, `cudaDeviceSynchronize()` can be used to wait for the entire device to finish its computation [NVIb]. In some cases it can also appear beneficial to use `cudaLaunchHostFunc(cudaStream_t stream, cudaHostfn_t fn, void *userData)`. As specified by CUDAs execution control [NVIc], this operation waits for the specified stream to finish, whereupon the given host function `fn` is called with parameter `userData`. As an example, this could be used by the packing phase, where `fn` starts the send operation via MPI, after the corresponding partition has been copied into the send-buffer.

As the last consideration, it might be favorable to use host memory that is *page-locked*, which is accessible by the device and tracked by the driver [NVId]. The main advantage of page-locked host memory is that the memory transfer bandwidth between host and device can be significantly increased, e.g., when using `cudaMemcpy` or `cudaMemcpy2D`. Page-locked host memory can be allocated with `cudaMallocHost(void **ptr, size_t size)` and is freed using `cudaFreeHost (void *ptr)` [NVId].

3.2 cuFFT: The CUDA FFT Library

The cuFFT library [NVIj] is part of the CUDA Toolkit [NVIk] and is specialized for NVIDIA GPUs. It implements the Cooley-Tukey FFT algorithm (cf. Section 2.3) and provides highly optimized Radix-2, Radix-3, Radix-5, and Radix-7 building blocks. Similar to FFTW [FJ05], cuFFT starts by configuring a *plan*, which specifies the details on how to compute the FFT, before *executing* it later on. The advantage of this approach is that the plan can be specified once upon initialization and thereafter be reused to compute the wanted FFT for different input data. Furthermore, plans can be assigned to a specific stream (cf. Section 3.1), such that multiple plans potentially can be executed in parallel.

The subsequent description follows the documentation of cuFFT [NVIh]. We start by describing the execution phase, whereupon the relevant details of the plan configuration are introduced. Finally, we motivate and explain the use of the *Advanced Data Layout*. As before, we stress that this section solely focuses on the relevant parts for our work.

Execution Phase. Starting with the execution phase, Listing 3.3 provides an overview of the available methods, provided by cuFFT. As we have seen in Section 2.4.2, the output size of 3D-FFTs for real-valued input data can be considered to be

$$N_x \times N_y \times \left(\left\lfloor \frac{N_z}{2} \right\rfloor + 1 \right),$$

when starting with the 1D-FFT in z-direction (cf. Section 2.5). Therefore, cuFFT allows differentiating between *Real-to-Complex* and *Complex-to-Complex* forward FFTs. Analogously, cuFFT distinguishes between *Complex-to-Real* and *Complex-to-Complex* inverse FFTs. To specify

Listing 3.3 Plan Execution for cuFFT.

```
// Real to Complex FFT (forward)
cufftResult cufftExecR2C(cufftHandle plan, cufftReal *idata, cufftComplex *odata);

// Complex to Real FFT (inverse)
cufftResult cufftExecC2R(cufftHandle plan, cufftComplex *idata, cufftReal *odata);

// Complex to Complex FFT
cufftResult cufftExecC2C(cufftHandle plan, cufftComplex *idata, cufftComplex *odata,
    int direction);
```

the direction, the methods `cufftExecR2C` and `cufftExecC2R` can be used to implicitly compute the forward and inverse transform respectively, whereas `cufftExecC2C` provides `direction` as an additional parameter. For the sake of completeness, we note that the methods presented in Listing 3.3 are solely used to compute the corresponding FFTs with single-precision. Except for the method names, the computation with double-precision is identical.

Plan Configuration. Following the description of the execution phase, we continue by explaining the plan configuration. While cuFFT provides multiple ways to configure plans, we solely focus on `cufftPlanMany`, which allows plan creation for the relevant 1D- and 2D-FFTs for slab and pencil decomposition in either forward or inverse direction. The available parameters are visualized in Listing 3.4. Here, parameter `rank` specifies the number of dimensions and the size of the ith dimension is given by $n[i-1]$, for $1 \leq i \leq$ `rank`. The parameter `type` is used to tell, which methods can be called in the execution phase (cf. Listing 3.3). Furthermore, `cufftPlanMany` allows the use of an *Advanced Data Layout*, which is specified by the remaining parameters and motivated in the following.

Listing 3.4 Plan Creation for cuFFT.

```
cufftResult cufftPlanMany(cufftHandle *plan, int rank, int *n, int *inembed,
    int istride, int idist, int *onembed, int ostride,
    int odist, cufftType type, int batch);
```

Advanced Data Layout. Considering the computation of the 1D-FFT in x-direction for three-dimensional input data as an example (cf. Figures 2.2, 2.3), it appears obvious that the corresponding data points in x-direction do not lie contiguous in memory. Therefore, it is necessary to provide an additional stride parameter. Continuing with this example, we are also interested in computing multiple 1D-FFTs at once, e.g. $N_z \cdot \frac{N_y}{P}$ many for complex-valued input data when using slab decomposition (cf. Section 2.5.1). Thus, the distance between the first elements of each batch has to be specified as well.

Generalizing the example presented above yields the memory accesses summarized in Table 3.1, when using the advance data layout. It can be seen that cuFFT distinguishes between input and output data alignment, which is why the output alignment can be different from the input alignment. As an example, this can be used such that the intermediate results are contiguous in y- or x-direction, while the input data layout is by default considered to be contiguous in z-direction. For the sake of completeness, we note that Table 3.1 shows that the second entries of `inembed` and `onembed` are used to specify plans for 2D-FFTs. Here, the first entries are neglected, since they correspond to the parameters `idist` and `odist`, respectively.

	1D-FFTs	2D-FFTs
Input	`input[b*idist+x*istride]`	`input[b*idist+(x*inembed[1]+y)*istride]`
Output	`output[b*odist+x*ostride]`	`output[b*odist+(x*onembed[1]+y)*ostride]`

Table 3.1: Memory Access when using an Advanced Data Layout. The entries describe, how positions x and (x, y) of batch b are accessed for 1D- and 2D-FFTs, respectively.

3.3 MPI: Message Passage Interface

Message Passage Interface (MPI) originated by the joint effort of various organizations in 1992 and provides a unified specification for message-passing programs. At the time of writing, MPI-4.0 [MPI21] is the newest approved standard and will be used as reference throughout this work. As stated by the standard, the goals of MPI include, but are not limited to designing an interface that allows for an efficient and thread safe process communication in an heterogeneous environment, where the user does not have to take care of possible communication failures [MPI21, p. 1-2]. To achieve this goal, MPI specifies a large variety of possible communication mechanisms, which can be generally categorized into *Point-to-Point Communication* and *Collective Communication*. As indicated by their names, point-to-point communication is used for message-passing between two processes, while collective communication covers a group of processes. Furthermore, new implementations (e.g. OpenMPI [GFB+04], [Sofa]) provide CUDA-aware MPI, where device pointers (cf. Section 3.1) can be directly passed to the communication routines.

This section starts by illustrating the general structure of a simple MPI application. Thereafter, we introduce MPIs non-blocking send and receive operations for point-to-point communication. Finally, we cover relevant aspects of collective communication, along with more advanced features. As before, we stress that this introduction to MPI is by no means exhaustive, since we solely focus on the necessary fundamentals for our work.

Listing 3.5 Exemplary usage of MPI in C++.

```
#include <iostream>
#include <mpi.h>

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    int comm_size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        char str[13] = "Hello World!";
        for (size_t p = 1; p < comm_size; p++)
            MPI_Send(str, 13, MPI_CHAR, p, p, MPI_COMM_WORLD);
    } else {
        char str[13];
        MPI_Recv(str, 13, MPI_CHAR, 0, rank, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::cout << str << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

General Structure of an MPI Application. Listing 3.5 shows the general structure of a simple MPI application in C++, which can be executed by P processes in parallel via `mpirun -n P <executable>`. MPI identifies each process by a unique ID called *rank*, with $0 \leq \text{rank} < P$. In order to describe the individual computational steps of the example application, we consider the view of a fixed process with rank p. First, `int MPI_Init(int *argc, char **argv)` is used to initialize the MPI environment [MPI21, p. 488-489], where the purpose of the parameters `argc` and `argv` depends on the specific MPI implementation. Afterwards, the functions `int MPI_Comm_size(MPI_Comm comm, int *size)` and `int MPI_Comm_rank(MPI_Comm comm, int *rank)` are used to obtain the values P and p, respectively. In our example, the default communicator `MPI_COMM_WORLD` is used for `comm`. In case p is equal to 0, the process sends the string `"Hello World!"` to each of the other $P - 1$ processes individually via the send operation `MPI_Send`. Otherwise, the process receives the string via the blocking receive operation `MPI_Recv`. The specification of both operations can be found in Listing 3.6. It can be seen that the `tag` parameter is used to identify the message and therefore has to equal for `MPI_Send` and `MPI_Recv`. Furthermore, the MPI specification predefines multiple datatypes, e.g., `MPI_CHAR`, `MPI_BYTE`, `MPI_INT`, and `MPI_DOUBLE` [MPI21, p. 34], such that parameter `count` is always set with respect to the underlying datatype. When calling `MPI_Recv`, an additional parameter `status` is provided that can be either ignored, by passing `MPI_STATUS_IGNORE` (cf. Listing 3.5), or can be used to get further information about possible errors. While MPI specifies the additional *buffered*, *synchronous*, and *ready* communication modes, we solely focus on the standard mode (cf. Listing 3.6) and refer the interested reader to [MPI21, p. 49-55], for more information. Finally, `MPI_Finalize()` has to be called by each computing process to clean up all MPI states, which acts as a collective routine ensuring that each process finishes all MPI calls before exiting [MPI21, p. 494-495].

Listing 3.6 Blocking send and receive routines of MPI.

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm)

int MPI_Recv(const void *buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Status *status)
```

Non-Blocking Communication. We have utilized the *blocking* send and receive operations in the previous example application, where `MPI_Send` returns only after the data is copied out of the send-buffer and `MPI_Recv` after the entire message is stored in the receive-buffer [MPI21, p. 60-61]. It is therefore apparent that the user especially has to take care of avoiding potential deadlocks during execution. As an alternative, MPI specifies additional *non-blocking* send and receive operation in standard mode, which are presented in Listing 3.7. Here, the main difference is that non-blocking operations return immediately and require detached calls to ensure the completion of either send or receive operation. For this purpose, the parameter `request` is added to both non-blocking operations, which can be later passed to `int MPI_Wait(MPI_Request *request, MPI_Status *status)` to ensure the completion of the corresponding operation [MPI21, p. 71]. For the sake of completeness, we note that our implementation makes additional use of `MPI_Waitany` and `MPI_Waitall`, where an array of requests is taken as a parameter. For more information on their specification, we refer the interested reader to [MPI21, p. 76-79].

Listing 3.7 Non-blocking send and receive routines of MPI.

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(const void *buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Request *request)
```

Custom MPI-Datatypes. Until now, we only considered predefined datatypes for send and receive operations. Although these datatypes can easily be used to send from and receive to contiguous buffers, MPI also allows the user to define custom datatypes for more complicated memory structures. Again, MPI provides a large variety of constructors for custom datatypes, whereas we solely focus on `MPI_Type_vector` [MPI21, p. 122-123] which is specified in Listing 3.8. For this constructor, an underlying and predefined datatype `oldtype` is grouped into multiple and individually contiguous blocks. The number of blocks is specified by `count` and the number of elements, each of type `oldtype`, is given by `blocklength`. The parameter `stride` sets the distance of the first elements of two blocks. By setting the value of `stride` greater than that of `blocklength`, the resulting datatype `newtype` can be used to send from and receive to non-contiguous buffers.

Listing 3.8 Construction of custom MPI datatypes.

```
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

All-to-All Communication. With regard to collective communication, MPI specifies routines which allow *All-to-All Communication* within a group of processes. Intuitively, all-to-all communication can be recreated using point-to-point communication, where each of the P processes sends to and receives from the $P-1$ other processes, individually. Listing 3.9 provides the specification of two such routines: `MPI_Alltoall` [MPI21, p. 217-218] and `MPI_Alltoallv` [MPI21, p. 219-220]. In the following, we will only explain the parameters that specify how the data is sent and note that the data is received in a symmetrical manner. Starting with the simpler routine, `MPI_Alltoall` assumes that `sendcount` specifies the number of elements sent to each of the other $P-1$ processes, where each element is of type `sendtype`. For $0 \leq k < P$, each process starts then sending with offset `sendbuf+k*sendcount*sizeof(sendtype)` to process k. In case the processes have to sent different amounts of data to the other processes, MPI provides `MPI_Alltoallv`. Here, arrays `sendcounts` and `sdispls` of size P can be passed to the routine, such that for $0 \leq k < P$, each process sends `sendcounts[k]` elements of type `sendtype` to process k and starts sending from memory location `sendbuf+sdispls[k]*sizeof(sendtype)`.

When using MPIs all-to-all communication in combination with custom datatypes, it is often required to use dissimilar datatypes for different receiving processes. To provide a remedy for this issue, MPI offers `MPI_Alltoallw`, which is identical to `MPI_Alltoallv` (cf. Listing 3.9) except that arrays `sendtypes` and `recvtypes` are passed to routine. A more details specification of `MPI_Alltoallw` can be found in [MPI21, p. 221-222].

Listing 3.9 Collective all-to-all communication in MPI.

```
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Alltoallv(const void *sendbuf, const int sendcounts[], const int sdispls[],
    MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[],
    MPI_Datatype recvtype, MPI_Comm comm)
```

Splitting Communicators. Instead of using the default communicator `MPI_COMM_WORLD`, it is also possible to create new communicators. In particular, MPI provides the function `MPI_Comm_split` to partition the set of processes and creates a new communicator for each partition [MPI21, p. 335-336]. The function's specification is presented in Listing 3.10, where parameter `color` specifies to which partition the process is assigned and `key` sets the process's rank for the new communicator.

Listing 3.10 Splitting the MPI communicator.

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

4 Implementation

In Section 2.5, we already introduced the general approaches for distributed computation of the FFT: Slab decomposition and pencil decomposition. There, we have seen the necessity of the global redistribution of intermediate results, which is composed of a packing, communication, and unpacking phase. For the GPU implementation described in the following, we utilize the CUDA library cuFFT for computing FFTs (cf. Section 3.2) along with MPI for communication between different processes (cf. Section 3.3).

Our Implementation[1] of slab and pencil decomposition offers a wide variety of possible optimizations. For instance, implementation strategies for slab decomposition can be subdivided even further into 2D-1D Slab Decomposition and 1D-2D Slab Decomposition: 2D-1D Slab Decomposition was already introduced in Section 2.5.1, where the procedure starts by computing the 2D-FFTs in z- and y-direction, before globally redistributing the intermediate results and computing the remaining 1D-FFTs in x-direction. Complementary to this strategy, 1D-2D Slab Decomposition starts with the 1D-FFTs in z-direction and finishes by computing the remaining 2D-FFTs in y- and x-direction. Furthermore, we have seen in Section 3.2, that the plan configuration allows differing input and output data alignments. Therefore, an alternative implementation using data realignment, can be considered for Pencil Decomposition, 2D-1D Slab Decomposition, and 1D-2D Slab Decomposition each. To differentiate between both variants, we denote Realigned as the corresponding variant using data realignment, whereas Default is used to refer to the variation without data realignment. The possible advantages and disadvantages of both variants depend on the underlying decomposition method and are therefore discussed in the corresponding section. Finally, CUDA and MPI provide different methods for realizing the global redistribution phase, i.e., the packing, communication, and unpacking phase for both Default and Realigned combined with each available decomposition method. Since the global data redistribution of the intermediate results is responsible for a large proportion of the total runtime, the right choice of such a method can result in a significant performance increase.

We start by presenting the underlying assumptions and notations in Section 4.1. Afterwards, we present the different decomposition methods: 2D-1D Slab Decomposition in Section 4.2, followed by 1D-2D Slab Decomposition in Section 4.3 and Pencil Decomposition in Section 4.4. For each of these decomposition methods, an abstract implementation strategy is described with and without data realignment. We note that these sections solely considers an abstract realization of the MPI communication, where data is sent from and received in contiguous memory. Thereafter, Section 4.5 summarizes the available methods for the global redistribution phase, where in contrast to the previous sections additional, more advanced MPI communication strategies are considered and discussed.

[1] https://github.com/eggersn/DistributedFFT

4.1 Assumptions and Notations

For our implementation, we solely focus on real-to-complex forward and complex-to-real inverse 3D-FFTs. The description of the different decomposition methods will often only consider the forward transform, since the inverse transform can be seen as reversing each step individually. We adopt the notations from Section 2.5, along with the assumption, that the input data is aligned contiguous in z-direction. For convenience, we further denote $\hat{N}_z = \left\lfloor \frac{N_z}{2} \right\rfloor + 1$ as the output size after computing real-valued forward 1D-FFTs of size N_z in z-direction.

With regard to the required memory workspace, we assume two distinct and previously allocated device memory spaces `in` and `out` for the input and output data. Depending on the decomposition method and its variant, along with additional special cases, each section will explain which additional memory buffers have to be used. These buffers are denoted as `send`, `recv`, and `temp` and are generally considered to be pairwise distinct if not stated otherwise.

When considering variants that utilize data realignment, there are three available data layouts which are uniquely determined by their innermost dimension. We therefore say, that data-buffer `data`, of size $S_x \times S_y \times S_z$, is *aligned contiguous in*

i) *z-direction* if position (x, y, z) is accessed via `data[(x * `S_y` + y) * `S_z` + z]`,

ii) *y-direction* if position (x, y, z) is accessed via `data[(z * `S_x` + x) * `S_y` + y]`, or

iii) *x-direction* if position (x, y, z) is accessed via `data[(y * `S_z` + z) * `S_x` + x]`.

Furthermore, we note that the data alignment only changes after a computational step with cuFFT.

As we have already seen in Section 2.5, the intermediate results are partitioned and sent to different processes afterwards. For the following descriptions and visualizations of the implemented decomposition methods, it is therefore often convenient to categorize the different positionings of these partitions. This is done by identifying each partition as a batch of slices, where each slice consists of the maximum number of contiguous, complex-valued words. The different partitions can then be categorized by whether the contained slices are one-, two-, or three-dimensional. Respectively, the partitions are therefore denoted as *1D-*, *2D-*, or *3D-partitions*. The visualization of this categorization can be found in Figure 4.1, which will implicitly be used throughout the following chapter.

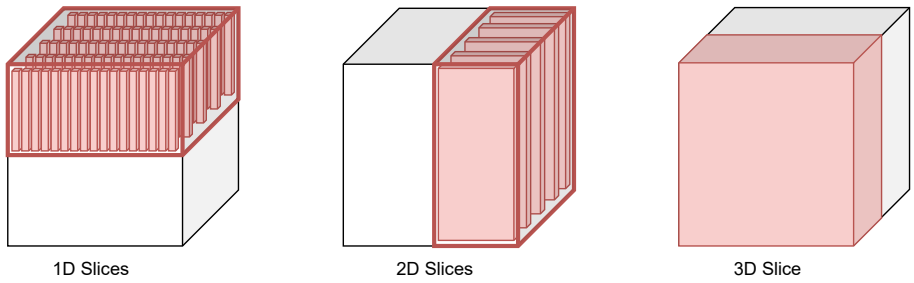


Figure 4.1: *Partition Categorization.* The figure visualizes three memory spaces, where each is split into two partitions. The red bounding box is used to highlight one partition for each memory space. Independent of the previously described data alignment, the visualizations throughout this work always assume that the data lies contiguous in the vertical direction.

4.2 2D-1D Slab Decomposition

The essential idea of the forward transform with 2D-1D Slab Decomposition was already introduced in Section 2.5.1. For both variants, i.e. Default and Realigned, the global data is partitioned as follows:

Input: In x-direction, i.e., each processor starts with input size $\frac{N_x}{P} \times N_y \times N_z$.

Output: In y-direction, i.e., each processor ends with output size $N_x \times \frac{N_y}{P} \times \hat{N}_z$.

In the following, we provide a description of Default and Realigned for 2D-1D Slab Decomposition, where we focus on the GPU specific implementation details. Afterwards, the general schematic of the inverse 3D-FFT is explained. Finally, the advantages and disadvantages of both variants are reviewed and other possible realizations of 2D-1D Slab Decomposition are discussed.

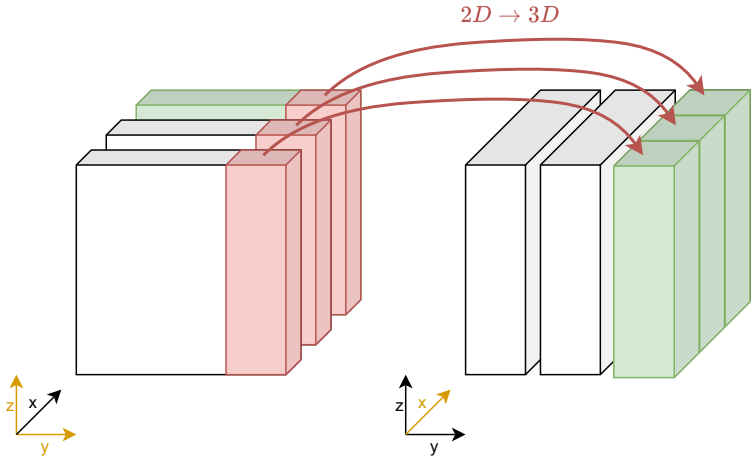


Figure 4.2: *Default 2D-1D Slab Decomposition.* The procedure is visualized for 3 processes, where the third process is highlighted in green. Beginning with the left decomposition, it starts by computing the 2D-FFTs in z- and y- direction, as indicated by the color of the coordinate axes. The communication phase is visualized by the red arrows, where the third process receives the red partitions. The arrow annotation $2D \rightarrow 3D$ further emphasizes that the sending process distributes 2D-partitions and the receiving process requires 3D-partitions for the subsequent computations (cf. Section 4.1). Therefore, solely a packing phase is required. Finally, the remaining 1D-FFTs in x-direction are computed.

Default Variant. Figure 4.2 visualizes the forward transform of 2D-1D Slab Decomposition without data realignment, i.e., the data is always aligned contiguous in z-direction. Thus, each process p starts by computing the batched 2D-FFTs in z- and y-direction, using the default input and output data layout, and stores the results in `out`. In the following packing phase, each process partitions the intermediate results along the y-axis and copies the 2D-partitions into the contiguous send-buffer `send`. Depending on whether CUDA-aware MPI is used, `send` is located on device or host memory. Afterwards, the k-th packed partition is sent to process k, for $0 \leq k < P$ if $k \neq p$, while the p-th partition is transposed locally and copied into `temp`. In case CUDA-aware MPI is used, it can be

seen that there is no need for an unpacking phase and an additional receive-buffer, since the required 3D-partitions are already received correctly. Therefore, the received data can be written directly into `temp`. If MPI without CUDA-awareness is used on the other hand, the receive-buffer `recv` is allocated on host memory and used to receive incoming messages, before copying the data via a host-to-device memory transfer to `temp`. Finally, the remaining 1D-FFTs in x-direction can be computed on the data stored in `temp`. Here, an input and output stride of $\hat{N}_z \cdot \frac{N_y}{P}$ is used.

Realigned Variant. As previously motivated, we can consider an alternative implementation with data realignment. Figure 4.3 visualizes the Realigned procedure for 3 processes and Table 4.1 summarizes the relevant cuFFT parameters for the advanced data layout. As before, the procedure starts by computing the 2D-FFTs in z- and y-direction. Using the parameters for the data alignment presented in Table 4.1a, it can be seen that two consecutive elements in x-direction have a distance

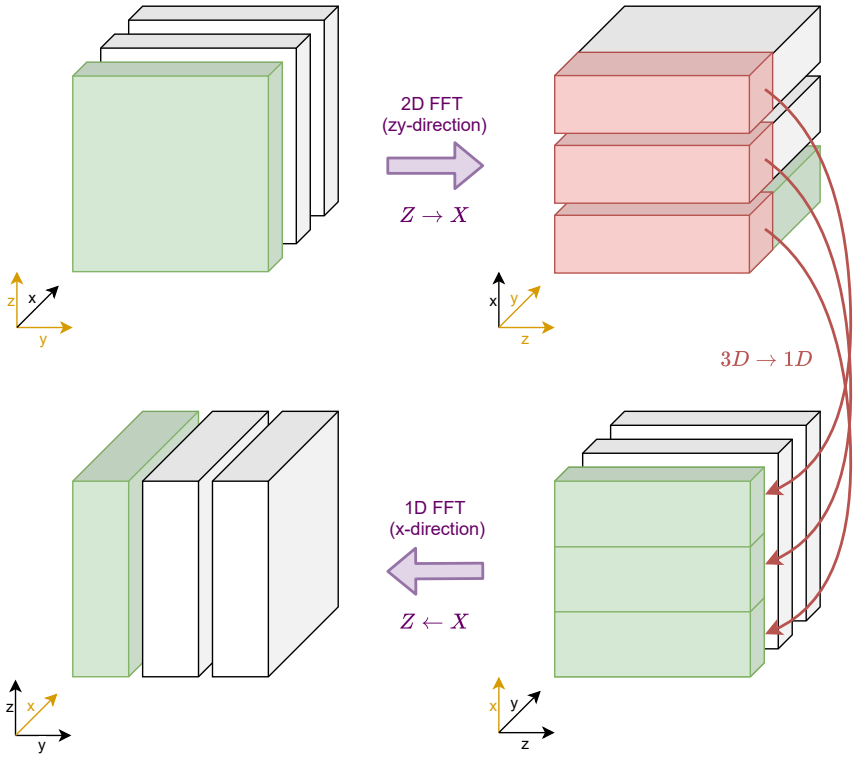


Figure 4.3: *Realigned 2D-1D Slab Decomposition.* The procedure is visualized for 3 processes, where the first process is highlighted in green. The first transition depicts the data realignment when computing the 2D-FFTs in z- and y-direction (as indicated by the color of the coordinate axes). The annotation $Z \rightarrow X$ emphasizes that the data alignment changes from being contiguous in z-direction to being contiguous in x-direction, which is further indicated by the change of the axis labels. In the following transition, the communication phase is visualized by the red arrows, where the first process receives the red partitions. The annotation $3D \rightarrow 1D$ highlights that the sending process distributes 3D-partitions, whereas the receiving process requires 1D-partitions in the subsequent computations. Therefore, an additional unpacking phase is required, before computing the remaining 1D-FFTs in x-direction. Using a second data realignment, as displayed by the last transition, the output data is aligned contiguous in z-direction again.

Variant	idist	inembed[1]	istride	odist	onembed[1]	ostride	batch
Default	$N_z \cdot N_y$	N_z	1	$\hat{N}_z \cdot N_y$	$\hat{N}_z$	1	$\frac{N_x}{P}$
Realigned	$N_z \cdot N_y$	N_z	1	1	$\hat{N}_z$	$\frac{N_x}{P}$	$\frac{N_x}{P}$

(a) *Summary of the Parameters Used by Default and Realigned for the 2D-FFTs in Z- and Y-Direction.*

Variant	idist	istride	odist	ostride	batch
Default	1	$\hat{N}_z \cdot \frac{N_y}{P}$	1	$\hat{N}_z \cdot \frac{N_y}{P}$	$\hat{N}_z \cdot \frac{N_y}{P}$
Realigned	N_x	1	1	$\hat{N}_z \cdot \frac{N_y}{P}$	$\hat{N}_z \cdot \frac{N_y}{P}$

(b) *Summary of the Parameters Used by Default and Realigned for the 1D-FFTs in X-Direction.*

Table 4.1: *2D-1D Slab Decomposition cuFFT Parameters for the Advanced Data Layout.* The tables are segmented by the plan configuration of the 2D-FFTs in z- and y-direction and the plan configuration of the 1D-FFTs in x-direction. Both of them summarize the different parameters used to configure a cuFFT plan for Default and Realigned. A detailed description of the plan configuration can be found in Section 3.2.

of $N_z \cdot N_y$ for the input data and a distance of 1 for the intermediate results, which are stored in `out`. Using Table 3.1 as reference, position (x, y, z) of the intermediate results can be accessed by

$$\begin{aligned} &\texttt{out[b * odist + (y * onembed[1] + z) * ostride]} \\ =\ &\texttt{out[x + (y *} \hat{N}_z \texttt{ + z) *} N_x/P\texttt{]}, \end{aligned} \tag{4.1}$$

where it can be seen that batch b corresponds to position x. Therefore, the data alignment of the intermediate results can be considered to be aligned contiguous in x-direction. As before, the data is partitioned in y-direction. Since the individual partitions are already contiguous memory regions, there is no need for a packing phase and an additional send-buffer, when CUDA-aware MPI is used. Otherwise, the intermediate results are simply transferred into `send`, which is located on host memory, via a single memory copy. Consequently, this procedure requires an unpacking phase such that the received partitions are aligned contiguous in x-direction again. This can be realized by using the receive-buffer `recv`, which is, depending on whether CUDA-aware MPI is used, located on either device or host memory. Since the partitions are received contiguously in `recv`, they are transformed and copied into `temp` to satisfy the alignment criteria of the 1D-partitions (e.g. by using `cudaMemcpy2D` [NVId]). Finally, the remaining 1D-FFTs in x-direction can be computed on the data stored in `temp`, using the same input data alignment as Equation 4.1. Applying the cuFFT parameters for the output layout presented in Table 4.1b, yields that position (x, y, z) of the final result can then be accessed by

$$\begin{aligned} &\texttt{out[b * odist + x * ostride]} \\ =\ &\texttt{out[(y *} \hat{N}_z \texttt{ + z) + x *} \hat{N}_z \cdot N_y/P\texttt{]} \\ =\ &\texttt{out[(x *} N_y/P \texttt{ + y) *} \hat{N}_z \texttt{ + z]}, \end{aligned}$$

where batch b can be uniquely identified by position (y, z). Thus, the resulting output data alignment is identical to the alignment of the input data. When using CUDA-aware MPI, it is important that the procedure waits until the send operation is complete, before computing the final 1D-FFTs, since `out` is directly used as the send-buffer.

Inverse Transform. Next, we explain how the inverse 3D-FFT can be computed. For this, we solely focus on the more interesting variant that utilizes data realignment. The procedure starts with an input partition of size $N_x \times \frac{N_y}{P} \times \hat{N}_z$ and essentially reverses each computational step visualized in Figure 4.3. It starts by computing the inverse 1D-FFTs in x-direction, where the values of the input and output parameters of the advanced data layout are swapped (cf. Table 4.1b). Thereafter, when performing the global redistribution of the intermediate results, it appears obvious that the packing and unpacking phase can be considered to be mirrored when computing the inverse transform: In particular, the inverse transform requires an packing (unpacking) phase if and only if the forward transform requires an unpacking (packing) phase. Finally, the inverse 2D-FFTs in y- and z-direction can be computed by using `cufftExecC2R` (cf. Section 3.2) and by switching the cuFFT parameters of Table 4.1a in the same manner as before. Another technical difficulty is the buffer utilization when computing the inverse transform. This will be discussed in the following (cf. Figure 4.4).

Key Differences. In summary, both variants (i.e. 2D-1D Slab Decomposition with and without data realignment) require two additional data buffers when performing the forward an inverse transform, in case CUDA-aware MPI is used. Otherwise, `send` and `recv` are allocated on host memory, while `temp` is allocated on device memory. Since memory allocation is generally considered to be expensive, we note that the allocation of these buffers is performed during the initialization phase, instead of the execution phase.

As the first disadvantage when using Realigned and CUDA-aware MPI, we found that the forward procedure has to wait idle until the send operation is complete, before computing the 1D-FFTs in x-direction. The necessity for this is visualized in Figure 4.4, where the buffer utilization is summarized for 2D-1D Slab Decomposition. In particular, it can be seen that one has to pay special attention that the send-buffer is not overwritten if there is no need for an additional packing phase. Interestingly for Default, this can be avoided for the inverse transform by reusing `in` (cf. Section 4.1). We stress, that this is only possible for the inverse transform, since `in` is only here a complex-valued buffer.

While the previous arguments depend on whether the forward or inverse transform is performed, there is also a second disadvantage of Realigned that becomes apparent when reviewing the performed packing and unpacking phase in detail: It can be seen in Figure 4.2 that the packing phase of Default's 2D-partitions can be realized by copying $\frac{N_x}{P}$ slices of size $\frac{N_y}{P} \times \hat{N}_z$ into the send-buffer (e.g. by using `cudaMemcpy2D`). In contrast, Figure 4.3 visualizes that the unpacking phase of Realigned's 1D-partitions has to be realized by copying $\hat{N}_z \cdot \frac{N_y}{P}$ slices of size $1 \times \frac{N_x}{P}$ from the receive-buffer into `temp`. Therefore, latter has to perform significantly more subroutines to copy smaller slices than the former, which can be expected to result in poorer performance.

Although the comparison above might convey the impression that Realigned generally does not yield significant advantages, we will see the contrary for 1D-2D Slab Decomposition.

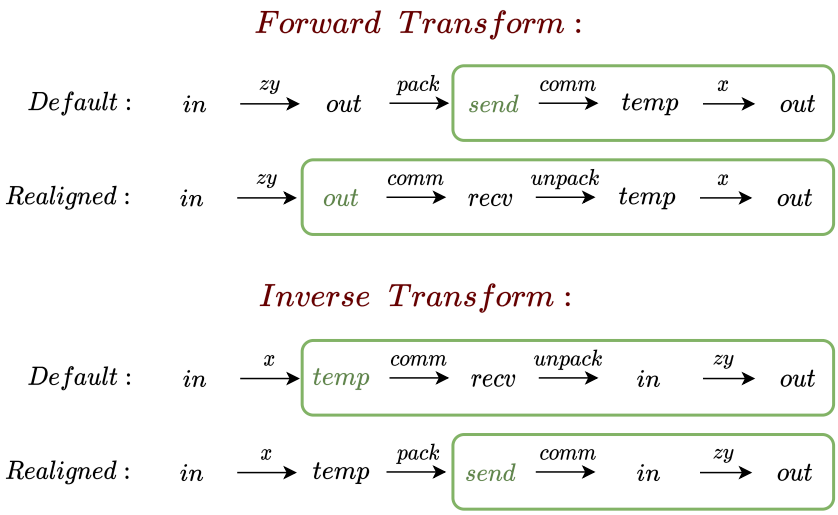


Figure 4.4: *2D-1D Slab Decomposition Buffer Usage.* The figure summarizes the utilization of the available buffers (cf. Section 4.1) for the forward and inverse transform, while differentiating between Default and Realigned. All scenarios are displayed with the assumption that CUDA-aware MPI is used. The input data is always stored in `in` (sc. first buffers on the left side), and the output data is always stored in `out` (sc. last buffer on the right side). The arrow annotations indicate which task is performed, i.e., FFT computation, packing, unpacking, or communication. Furthermore, the green boxes show regions where special care has to be taken when reusing the send-buffer (highlighted in green) to avoid overwriting its content.

Discussion. Finally, we want to review other possible realizations of 2D-1D Slab Decomposition. We note, that this discussion solely aims to highlight some design considerations and is by no means exhaustive. First, it is obvious that the computed 2D-FFTs depend on the given input partition, which in turn is specified by the utilized application. When possible, an input partition of size $\frac{N_x}{P} \times N_y \times N_z$ is beneficial since the computation of the individual 2D-FFTs are performed on contiguous batches, which is recommended by the cuFFT documentation [NVIh].

For the global redistribution of the intermediate results, we partitioned the data along the y-axis. A different possibility would be the partitioning along the z-direction, since only the 1D-FFTs in x-direction remain for computation. The advantage of this approach would be that the output partition would be identical to the one computed by 1D-2D Slab Decomposition (see Section 4.3 for details). Opposing this, there are different disadvantages depending on whether Realigned or Default is used. When Realigned is considered, it can be seen in Figure 4.3 that a partitioning in z-direction would require both packing and unpacking phase. For Default on the other hand, it is evident by Figure 4.2 that a partitioning in z-direction would require a packing phase for the 1D-partitions. As argued before, this can be generally expected to result in poorer performance. Therefore, our implementation only considers 2D-1D Slab Decomposition where the intermediate results are partitioned along the y-direction.

4.3 1D-2D Slab Decomposition

We have seen in the previous section that 2D-1D Slab Decomposition generally starts by computing the 2D-FFTs in z- and y-direction and finishes by computing the remaining 1D-FFTs in x-direction. In contrast to this, 1D-2D Slab Decomposition starts with the 1D-FFTs in z-direction and finishes with the 2D-FFTs in y- and x-direction. For both variants, i.e. Default and Realigned, the global data is partitioned as follows:

Input: In x-direction, i.e., each processor starts with input size $\frac{N_x}{P} \times N_y \times N_z$.

Output: In z-direction, i.e., each processor ends with output size $N_x \times N_y \times \frac{\hat{N}_z}{P}$.

This section starts by describing both variants and concentrates on the GPU specific implementation details. Thereafter, we discuss other possible realizations of 1D-2D Slab Decomposition and summarize the key differences between the two variants.

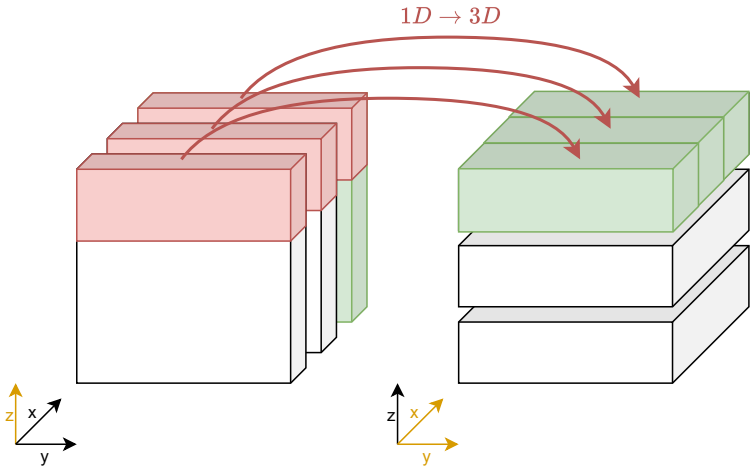


Figure 4.5: *Default 1D-2D Slab Decomposition.* The procedure is visualized for 3 processes, where the third process is highlighted in green. It starts on the left by computing the 1D-FFTs in z-direction, as indicated by the color of the coordinate axes. The communication phase is visualized by the red arrows, where the third process receives the red partitions. The arrow annotation $1D \rightarrow 3D$ further emphasizes that the sending process distributes 1D-partitions and the receiving process requires 3D-partitions for the subsequent computations. Finally, the remaining 2D-FFTs in y- and x-direction are computed.

Default Variant. Figure 4.5 depicts Default 1D-2D Slab Decomposition. Each process p starts by computing the 1D-FFTs in z-direction, using the default input and output data layout, and stores the results in `out`. In order to compute the 2D-FFTs in y- and x-direction later on, all data points in both directions have to be accessible by the computing process. Therefore, the intermediate results are partitioned in z-direction and each 1D-partition is copied into the contiguous send-buffer `send`, as part of the packing phase. During the following communication phase, the k-th packed partition is sent to process k, for each $0 \leq k < P$ with $k \neq p$. It can be seen that there is no need to transform the received 3D-partitions, which is why the unpacking phase can be skipped, and the received

data is directly stored in `temp` if CUDA-aware MPI is used. Otherwise, the receive-buffer `recv` is allocated on host memory and upon receiving the last partition, the data is copied to `temp` via a single host-to-device memory transfer. Finally, the remaining 2D-FFTs in y- and x-direction can be computed. As introduced in Section 3.2, this can be realized using the advanced data layout. The relevant parameters for the plan configuration can be found in Table 4.2b.

Realigned Variant. For 1D-2D Slab Decomposition, it is also possible to consider Realigned. This procedure is visualized in Figure 4.6. As before, the procedure starts by computing the 1D-FFTs in z-direction and storing the intermediate results in `out`. As summarized in Table 4.2a, the default data alignment is used for the input data. The output stride, which describes the distance between two consecutive elements of the corresponding 1D-FFT, is set to $N_y \cdot \frac{N_x}{P}$, while the output distance,

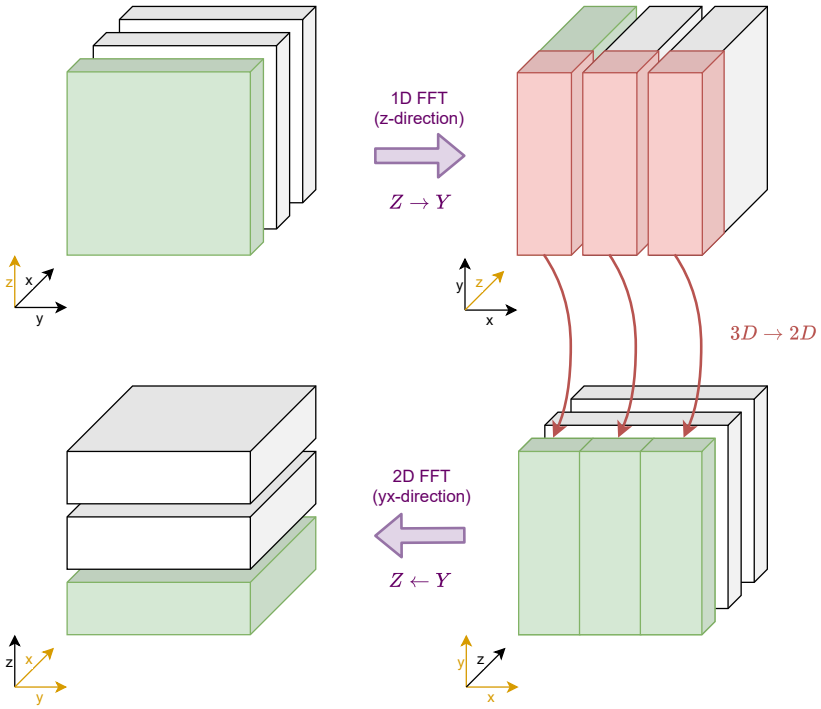


Figure 4.6: *Realigned 1D-2D Slab Decomposition.* The procedure is visualized for 3 processes, where the first process is highlighted in green. The first transition depicts the data realignment when computing the 1D-FFTs in z-direction (as indicated by the color of the coordinate axes). The annotation $Z \rightarrow Y$ emphasizes that the data alignment changes from being contiguous in z-direction to being contiguous in y-direction, which is further indicated by the change of the axis labels. In the following transition, the communication phase is indicated by the red arrows, where the first process receives the red partitions. The annotation $3D \rightarrow 2D$ highlights that the sending process distributes 3D-partitions, whereas the receiving process requires 2D-partitions for the subsequent computations. Therefore, an additional unpacking phase is required, before computing the remaining 2D-FFTs in y- and x-direction. Using a second data realignment, as displayed by the last transition, the output data is aligned contiguous in z-direction again.

which describes the distance of the first elements of two batches, is set to 1. Using Table 3.1 as reference, position (x, y, z) of the intermediate results can then be accessed by

$$\begin{aligned} & \texttt{out[b * odist + z * ostride]} \\ = \ & \texttt{out[(x *} N_y \texttt{+ y) * 1 + z *} N_y \cdot N_x/P \texttt{]} \\ = \ & \texttt{out[(z *} N_x/P \texttt{+ x) *} N_y \texttt{+ y]}, \end{aligned}$$

where batch b can be uniquely identified by position (x, y). Therefore, the data layout of `out` can be considered to be contiguous in y-direction. Since the intermediate results are partitioned along the z-axis, it can be seen that there is no need for a packing phase, if CUDA-aware MPI is used. Otherwise, the data is simply copied into `send` via a single device-to-host memory transfer. Analogously to the first variant, process p sends the k-th partition to process k during the communication phase, for each $0 \leq p, k < P$ if $p \neq k$. After unpacking the received data into the required 1D-partitions from `recv` to `temp`, the remaining 2D-FFTs in y- and x-directions can be computed. Here, the advanced data layout parameters from Table 4.2b are used, such that position (x, y, z) of the final result can be accessed via

$$\begin{aligned} & \texttt{out[b * odist + (x * onembed[1] + y) * ostride]} \\ = \ & \texttt{out[z * 1 + (x *} N_y \texttt{+ y) *} \hat{N}_z/P \texttt{]} \\ = \ & \texttt{out[(x *} N_y \texttt{+ y) *} \hat{N}_z/P \texttt{+ z]}, \end{aligned}$$

where batch b can be identified as z. Consequently, the data layout of the final result is identical to the data layout of the input data. Analogously to Section 4.2, the procedure has to wait until the send operation is complete, before computing the 2D-FFTs, since `out` is used directly as the send-buffer.

Variant	idist	istride	odist	ostride	batch
Default	N_z	1	$\hat{N}_z$	1	$N_y \cdot \frac{N_x}{P}$
Realigned	N_z	1	1	$N_y \cdot \frac{N_x}{P}$	$N_y \cdot \frac{N_x}{P}$

(a) *Summary of the Parameters Used by Default and Realigned for the 1D-FFTs in Z-Direction.*

Variant	idist	inembed[1]	istride	odist	onembed[1]	ostride	batch
Default	1	N_y	$\frac{\hat{N}_z}{P}$	1	N_y	$\frac{\hat{N}_z}{P}$	$\frac{\hat{N}_z}{P}$
Realigned	$N_y \cdot N_x$	N_y	1	1	N_y	$\frac{\hat{N}_z}{P}$	$\frac{\hat{N}_z}{P}$

(b) *Summary of the Parameters Used by Default and Realigned for the 2D-FFTs in Y- and X-Direction.*

Table 4.2: *1D-2D Slab Decomposition cuFFT parameters for the Advanced Data Layout.* The tables are segmented by the plan configuration of the 1D-FFTs in z-direction and the plan configuration of the 2D-FFTs in y- and x-direction. Both of them summarize the different parameters used to configure a cuFFT plan for Default and Realigned. A detailed description of the plan configuration can be found in Section 3.2.

Inverse Transform. Using above descriptions on how the forward 3D-FFT is computed using 1D-2D Slab Decomposition, it is easy to see that the computation of the inverse transform can be achieved by simply reversing each computational step individually. In particular, the plan configuration for the 2D- and 1D-FFTs simply swaps the input and output data layout parameter of Table 4.2. For more details, we refer the reader to Section 4.2. Again, the most notable difficulty is the memory utilization, which will be discussed in the following (cf. Figure 4.7).

Key Differences. Next, we summarize the key differences between both variants and review their possible advantages and disadvantages. For CUDA-aware MPI, we have seen that both variants require two additional data-buffers. Otherwise, both variants use `send` and `recv` on host memory, while also requiring `temp`, which is allocated on device memory and used to unpack the received data.

Analogous to 2D-1D Slab Decomposition, a general disadvantage when using Realigned and CUDA-aware MPI is that the forward procedure has to wait idle until the send operation is complete, before computing the 2D-FFTs. The necessity for this is visualized in Figure 4.7, where the buffer utilization is summarized for 1D-2D Slab Decomposition. Again, Default is able to avoid this problem for the inverse transform by reusing `in`.

There is also a significant advantage when using Realigned in this context: It can be seen in Figure 4.6 that the unpacking phase of Realigned's 2D-partitions can be realized by copying $\frac{\hat{N}_z}{P}$ slices of size $\frac{N_x}{P} \times N_y$ from the receive-buffer into `temp`. In contrast, the packing phase of Default's

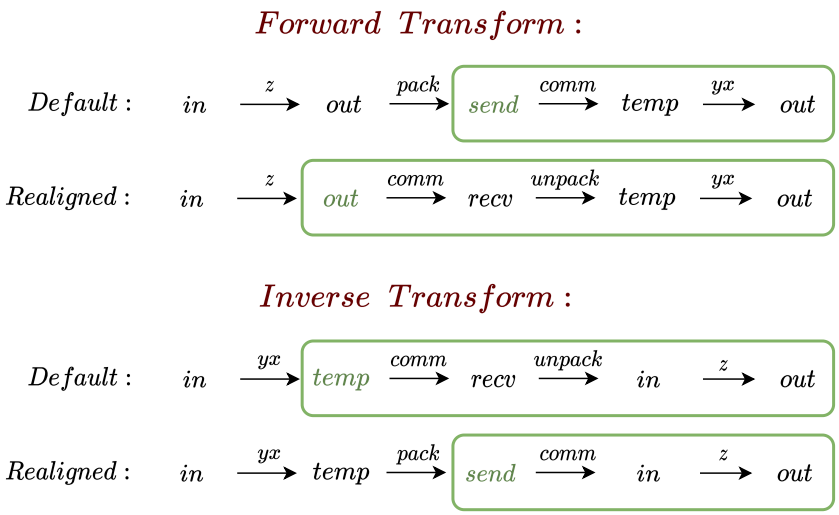


Figure 4.7: *1D-2D Slab Decomposition Buffer Usage.* The figure summarizes the utilization of the available buffers (cf. Section 4.1) for the forward and inverse transform, while differentiating between Default and Realigned. All scenarios are displayed with the assumption that CUDA-aware MPI is used. The input data is always stored in `in` (sc. first buffers on the left side), and the output data is always stored in `out` (sc. last buffer on the right side). The arrow annotations indicate which task is performed, i.e., FFT computation, packing, unpacking, or communication. Furthermore, the green boxes show regions where special care has to be taken when reusing the send-buffer (highlighted in green) to avoid overwriting its content.

1D-partitions has to be realized by copying $N_y \cdot \frac{N_x}{P}$ slices of size $1 \times \frac{\hat{N}_z}{P}$ from `out` into the send-buffer (cf. Figure 4.5). Therefore, former has to perform significantly fewer subroutines to copy larger slices than the latter, which can result in a considerable performance gain.

Discussion. Following the detailed description of both variants, it can be argued that there is another approach for computing 1D-2D Slab Decomposition. In particular, it is possible to start by computing the 1D-FFTs in y-direction and finishing with the 2D-FFTs in z- and x-direction. The advantage of this approach would be that for complex-valued input data, the resulting output partition of size $N_x \times \frac{N_y}{P} \times N_z$ would be identical to the output partition when using 2D-1D Slab Decomposition. This would allow an interchangeable computation of the forward and inverse transform of both decomposition methods, without an additional global data redistribution. As an example, consider a setup where it is known that Realigned 1D-2D Slab Decomposition performs best for the computation of forward 3D-FFTs and Default 2D-1D Slab Decomposition performs best for inverse 3D-FFTs. Then the following, optimized routine can be performed, without the need for an additional global data redistribution:

1. Compute the forward 3D-FFT using Realigned 1D-2D Slab Decomposition.
2. Modify the computed results locally, e.g., filter out high frequencies.
3. Perform the inverse 3D-FFT using Default 2D-1D Slab Decomposition.

On the contrary, it can be seen that a single cuFFT plan does not suffice for computing all 1D-FFTs in y-direction (cf. Listing 3.4). Particularly the parameter `idist` has to be set to either 1 or $N_z \cdot N_y$. Therefore, both cases result in the fact that the $\frac{N_x}{P} \times N_z$ grid of 1D-FFTs in y-direction is not fully covered by a single cuFFT plan. One way to overcome this problem is to introduce multiple cuFFT plans and assign them to individual CUDA streams. Depending on the utilized hardware, this allows for a potential overlap of the batched transforms [NVIi]. However, even when considering CUDA-streams, it appears obvious that above application example does not work for real-valued input data, where the resulting output partition of the first step would be of size $N_x \times \frac{\hat{N}_y}{P} \times N_z$ with $\hat{N}_y = \left\lfloor \frac{N_y}{2} \right\rfloor + 1$. Thus, the application would require an additional computation step, which transposes the results globally, using Hermitian symmetry (cf. Section 2.4.2). Since our work solely focuses on applications with real-valued input data, we therefore narrow the scope of the implementation down to the variants starting with the 1D-FFTs in z-direction.

4.4 Pencil Decomposition

A general overview of pencil decomposition was already presented in Section 2.5.2. We adopt the notations and consider a process grid of size $P_1 \times P_2$ such that each process can be uniquely identified as (p_i, p_j) for fitting $0 \leq p_i < P_1$ and $0 \leq p_j < P_2$. Furthermore, we assume that each process starts with a real-valued input partition of size $\frac{N_x}{P_1} \times \frac{N_y}{P_2} \times N_z$. Analogously to slab decomposition (cf. Sections 4.2, 4.3), pencil decomposition can be realized with and without data realignment. Thus, the global data is for both variants partitioned as follows:

Input: In $P_1 \times P_2$ partitions along the x- and y-direction respectively, i.e., each processor starts with input size $\frac{N_x}{P_1} \times \frac{N_y}{P_2} \times N_z$.

Output: In $P_1 \times P_2$ partitions along the y- and z-direction respectively, i.e., each processor ends with output size $N_x \times \frac{N_y}{P_1} \times \frac{\hat{N}_z}{P_2}$.

We start with the description of both variants, where we focus on the GPU specific implementation details. Afterwards, we review possible advantages and disadvantages when using either variant and discuss other possible realizations of pencil decomposition.

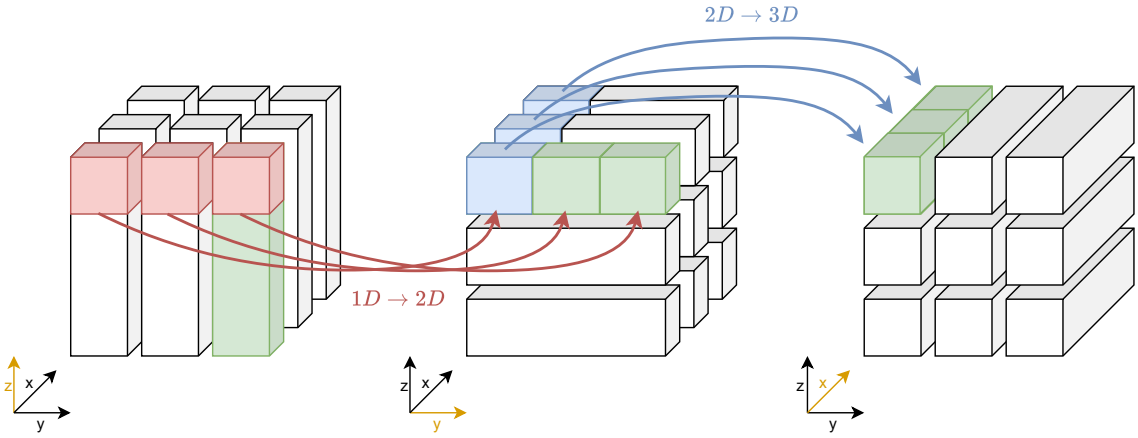


Figure 4.8: *Default Pencil Decomposition.* The procedure is visualized from left to right for a 3×3 process grid, where process $(0, 2)$ is highlighted in green. The marked axis indicated the direction in which the 1D-FFTs are computed. The first communication phase is indicated by the red arrows, where process $(0, 2)$ receives the red partitions. Analogously, the second communication phase is depicted by the blue arrows, where the process receives the blue partitions. The arrow annotation $1D \rightarrow 2D$ ($2D \rightarrow 3D$) further emphasizes that the sending process distributes 1D-partitions (2D-partitions) and the receiving process requires 2D-partitions (3D-partitions) for the subsequent computations.

Default Variant. Figure 4.8 visualizes Default Pencil Decomposition, which was already briefly introduced in Section 2.5.2. Each process (p_i, p_j) starts by computing the 1D-FFTs in z-direction, using the default input and output data layout, and storing the results in `out`. Afterwards, the intermediate results are partitioned in z-direction and the 1D-partitions are copied into the contiguous send-buffer `send`, which is located on device memory or host memory, depending on whether CUDA-aware MPI is used. For each $0 \leq k < P_2$ with $k \neq p_j$, the k-th packed partition is sent to process (p_i, k). Most notably, it can be seen that each process only has to communicate with $P_2 - 1$ other processes. After completing the communication phase, the data is stored contiguously in `recv`. Therefore, each process has to unpack the data into `temp` to satisfy the alignment criteria of the 2D-partitions, before computing the 1D-FFTs in y-direction. Since `send` was used as a send-buffer in the previous stage, the computed results can be stored in `out` even when the send operation is not fully complete. As we have seen in Section 4.3, there is an additional obstacle when computing the 1D-FFTs in y-direction on data aligned contiguous in z-direction: Since the cuFFT plan configuration has to set `idist` to either 1 or $\frac{\hat{N}_z}{P_2} \cdot N_y$, a single cuFFT plan cannot cover the full $\frac{N_x}{P_1} \times \frac{\hat{N}_z}{P_2}$ grid of batched 1D-FFTs. Therefore, a total of $\frac{\hat{N}_z}{P_2}$ cuFFT plans are introduced and configured during the initialization phase, where `istride` is set to $\frac{\hat{N}_z}{P_2} \cdot N_y$. An overview of the cuFFT

parameters can be found in Table 4.3b. While each plan is configured with the same parameters, they are assigned to different CUDA streams, which allows for a possible overlap in computation. During the execution phase, the j-th plan can then be used to perform the complex-to-complex transform on input pointer `&temp[j]` and on output pointer `&out[j]`. After completing the computation for each $0 \leq j < \frac{\hat{N}_z}{P_2}$, the partitioned results (i.e. 2D-partitions) can be packed into `send`, whereupon the k-th partition is sent to process (k, p_j), for each $0 \leq k < P_1$ with $k \neq p_i$. Analogous to the first global data redistribution, is can be seen that each process has to communicate only with $P_1 - 1$ other processes. Thereafter, the received data can be simply stored in `temp`, without requiring an additional unpacking phase. Finally, the remaining 1D-FFTs in x-direction can be computed, according to the summarized parameters in Table 4.3c.

Realigned Variant. A visualization of the alternative realization, using data realignment, can be found in Figure 4.9. The corresponding parameters for the cuFFT plan configuration are summarized in Table 4.3. Using the default input data layout, process (p_i, p_j) starts by computing the 1D-FFTs in z-direction and stores the results in `out`. Using the parameters of the output data layout from Table 4.3a, position (x, y, z) is accessed by

$$\begin{aligned} &\texttt{out[b * odist + z * ostride]} \\ =\ &\texttt{out[(x *} N_y/P_2 \texttt{ + y) * 1 + z *} N_y/P_2 \cdot N_x/P_1\texttt{]} \\ =\ &\texttt{out[(z *} N_x/P_1 \texttt{ + x) *} N_y/P_2 \texttt{ + y]}, \end{aligned}$$

where batch b can be identified as the position (x, y). Therefore, the intermediate results can be considered to be aligned contiguous in y-direction. In case CUDA-aware MPI is used, there is no need for a packing phase. Otherwise, the 3D-partitions of `out` are copied into `send` via a single device-to-host memory transfer. The following communication phase is then performed in the same manner as for Default, where the received data is stored in the contiguous receive-buffer `recv`. After unpacking the intermediate results into `temp` to satisfy the alignment criteria of the 2D-partitions, the 1D-FFTs in y-direction can be computed and stored in `out`. Since `out` is used as a send-buffer for the first communication phase, it has to be ensured that the send operation is complete, before performing this step. While we have seen that the first variant required multiple cuFFT plans to compute the 1D-FFTs in y-direction, a single plan is sufficient when using data realignment, because the input data is aligned contiguous in y-direction. Using the cuFFT parameters provided in Table 4.3b, position (x, y, z) of the intermediate results can be accessed by

$$\begin{aligned} &\texttt{out[b * odist + y * ostride]} \\ =\ &\texttt{out[(z *} N_x/P_1 \texttt{ + x) * 1 + y *} N_x/P_1 \cdot \hat{N}_z/P_2\texttt{]} \\ =\ &\texttt{out[(y *} \hat{N}_z/P_2 \texttt{ + z) *} N_x/P_1 \texttt{ + x]}. \end{aligned}$$

Again, batch b is uniquely identified by the position (x, z) (cf. Figure 4.9). Consequently, the data in `out` is aligned contiguous in x-direction. At this point, it can be seen that the second global packing, communication, and unpacking phase is identical to the first one. Finally, after waiting for the send operation to complete and computing the 1D-FFTs in x-direction, the data is aligned contiguous in z-direction again. This can be seen by the fact, that position (x, y, z) is accessed via

$$\begin{aligned} &\texttt{out[b * odist + x * ostride]} \\ =\ &\texttt{out[(y *} \hat{N}_z/P_2 \texttt{ + z) * 1 + x *} \hat{N}_z/P_2 \cdot N_y/P_1\texttt{]} \\ =\ &\texttt{out[(x *} N_y/P_1 \texttt{ + y) *} \hat{N}_z/P_2 \texttt{ + z]}, \end{aligned}$$

when using the cuFFT parameters provided in Table 4.3c. Similarly to the previous times, batch b is simply identified by the position (y, z).

Inverse Transform. After the detailed description of both variants, it is clear that the inverse transform can be computed by simply reversing the individual computational steps. As in the previous sections, the packing and unpacking phase can be considered to be mirrored and the plan

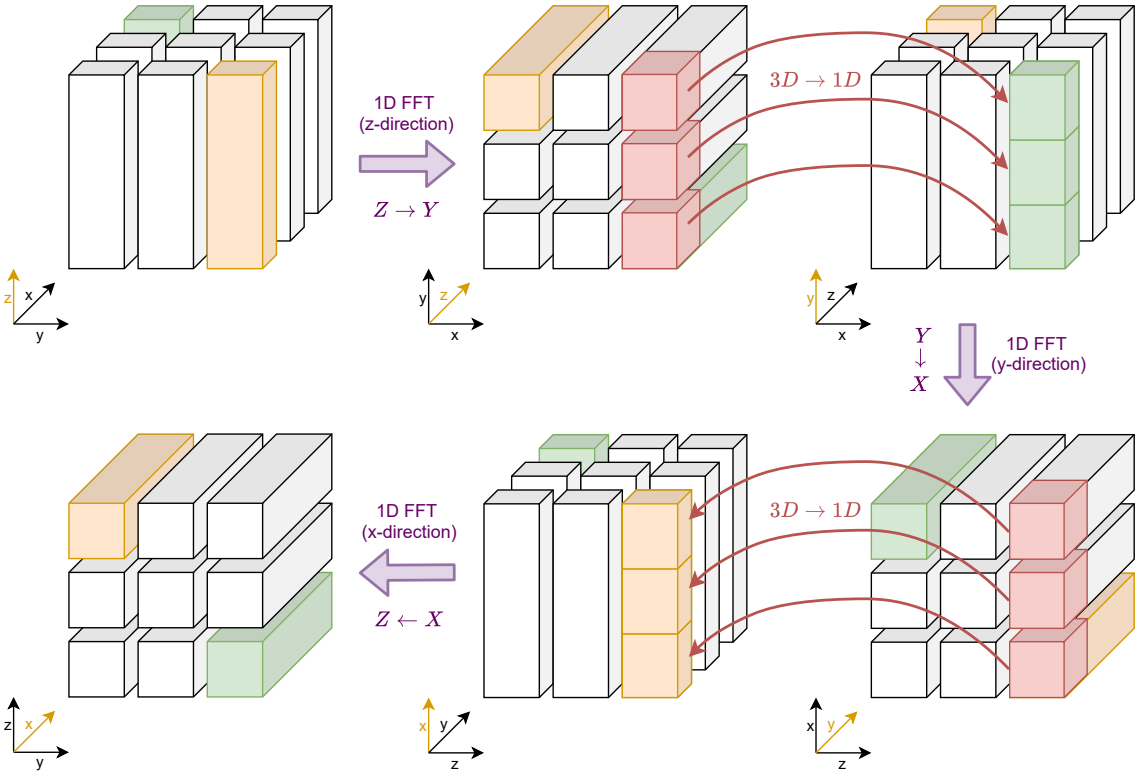


Figure 4.9: *Realigned Pencil Decomposition.* The procedure is visualized for a 3×3 process grid, where process $(2, 0)$ and $(0, 2)$ are highlighted in green and orange, respectively. Starting from the upper left corner, the first transition depicts the data realignment when computing the 1D-FFTs in z-direction (as indicated by the color of the coordinate axes). The annotation $Z \rightarrow Y$ emphasizes that the data alignment changes from being contiguous in z-direction to being contiguous in x-direction, which is further indicated by the change of the axis labels. The first communication phase corresponds to the second transition with the red arrows, where process $(2, 0)$ receives the red partitions. Similarly, the following transition visualizes the data realignment during the computation of the 1D-FFTs in y-direction, whereupon the data is aligned contiguous in x-direction (as highlighted by the annotation $Y \rightarrow X$). For the second communication phase, the partitions highlighted in red are received by process $(0, 2)$. For both communication phases, the arrow annotation $3D \rightarrow 1D$ emphasizes that the sending process distributes 3D-partitions, whereas the receiving process requires 1D-partitions in the subsequent computations. Finally, the last transition depicts the data redistribution when computing the 1D-FFTs in x-direction, after which the data is aligned contiguous in z-direction again (as highlighted by the annotation $X \rightarrow Z$).

Variant	idist	istride	odist	ostride	batch
Default	N_z	1	$\hat{N}_z$	1	$\frac{N_y}{P_2} \cdot \frac{N_x}{P_1}$
Realigned	N_z	1	1	$\frac{N_y}{P_2} \cdot \frac{N_x}{P_1}$	$\frac{N_y}{P_2} \cdot \frac{N_x}{P_1}$

(a) *Summary of the Parameters Used by Default and Realigned for the 1D-FFTs in Z-Direction.*

Variant	idist	istride	odist	ostride	batch	plans
Default	$\frac{\hat{N}_z}{P_2} \cdot N_y$	$\frac{\hat{N}_z}{P_2}$	$\frac{\hat{N}_z}{P_2} \cdot N_y$	$\frac{\hat{N}_z}{P_2}$	$\frac{N_x}{P_1}$	$\frac{\hat{N}_z}{P_2}$
Realigned	N_y	1	1	$\frac{N_x}{P_1} \cdot \frac{\hat{N}_z}{P_2}$	$\frac{N_x}{P_1} \cdot \frac{\hat{N}_z}{P_2}$	1

(b) *Summary of the Parameters Used by Default and Realigned for the 1D-FFTs in Y-Direction.*

Variant	idist	istride	odist	ostride	batch
Default	1	$\frac{\hat{N}_z}{P_2} \cdot \frac{N_y}{P_1}$	1	$\frac{\hat{N}_z}{P_2} \cdot \frac{N_y}{P_1}$	$\frac{\hat{N}_z}{P_2} \cdot \frac{N_y}{P_1}$
Realigned	N_x	1	1	$\frac{\hat{N}_z}{P_2} \cdot \frac{N_y}{P_1}$	$\frac{\hat{N}_z}{P_2} \cdot \frac{N_y}{P_1}$

(c) *Summary of the Parameters Used by Default and Realigned for the 1D-FFTs in X-Direction.*

Table 4.3: *Pencil Decomposition cuFFT Parameters for the Advanced Data Layout.* The tables are segmented by the plan configuration of the 1D-FFTs in z-direction, y-direction, and x-direction. All three of them summarize the different parameters used to configure a cuFFT plan for Default and Realigned. A detailed description of the plan configuration can be found in Section 3.2.

configuration of the 1D-FFTs is achieved by swapping the input and output data layout parameters (cf. Section 4.2). The only difficulty is the buffer utilization, which will be discussed in the following (cf. Figure 4.10).

Key Differences. When comparing both variants, the usage of data realignment has some salient advantages. Commencing with the first advantage, we have seen that `out` can be used directly as a send-buffer for both global data redistributions, when CUDA-aware MPI is used. Therefore, the Realigned only requires two additional data buffers, instead of three. In case MPI is considered without CUDA-awareness, both variants require `send` and `recv` to be allocated on host memory, while using `temp` on device memory as before.

For the second advantage, it is apparent that Default requires two packing and one unpacking phase, whereas Realigned solely requires two unpacking phases. Furthermore, we have seen that the contiguous alignment in y-direction of the intermediate results for Realigned entails that the computation of the 1D-FFTs in y-direction only require a single cuFFT plan, which can be executed more efficiently.

Finally, Realigned experiences the same disadvantage as for Slab Decomposition when computing the forward transform: Since `out` is used directly as a send-buffer for both global redistribution phases, the procedure has to wait idle until the send operations are complete, before the subsequent

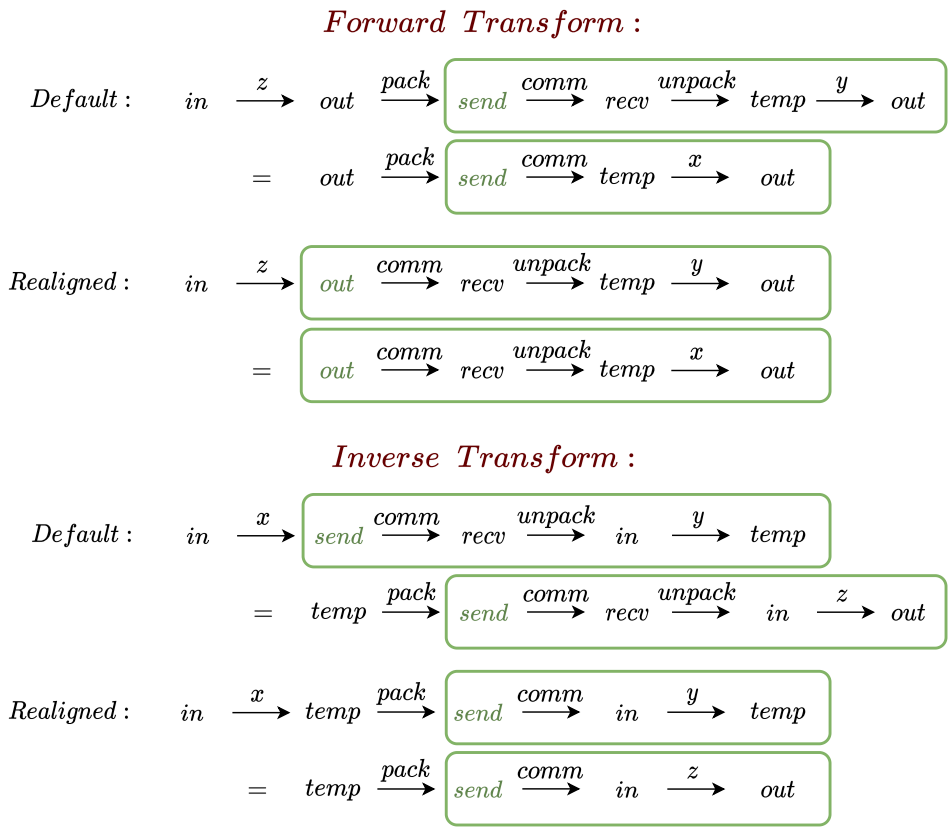


Figure 4.10: *Pencil Decomposition Buffer Usage.* The figure summarizes the utilization of the available buffers (cf. Section 4.1) for the forward and inverse transform, while differentiating between Default and Realigned. All scenarios are displayed with the assumption that CUDA-aware MPI is used. The input data is always stored in `in` (sc. first buffers on the left side), and the output data is always stored in `out` (sc. last buffer on the right side). The arrow annotations indicate which task is performed, i.e., FFT computation, packing, unpacking, or communication. Furthermore, the green boxes show regions where special care has to be taken when reusing the send-buffer (highlighted in green) to avoid overwriting its content. For better readability, there are line breaks as indicated by the equal signs.

1D-FFTs in either y- or x-direction can be computed. A summary for the utilized buffers is given in Figure 4.10. As for Slab Decomposition, Default is able to prevent this undesirable behavior by reusing `in` when computing the inverse transform. Again, we stress that this is exclusively possible for the inverse transform, since `in` is only here a complex-valued buffer.

Discussion. Finally, we discuss other possible realizations of Pencil Decomposition. As before, this discussion solely aims to provide further inside in some design considerations and is by no means meant to be exhaustive. The first interesting consideration is to further improve the performance of Realigned by introducing another temporary buffer. This buffer could be used to avoid that the procedure needs to wait idle before computing the 1D-FFTs in y- and x-direction. In this case, the

memory requirements would be identical to Default, which could be acceptable for some application. Since the additional buffer would not be used when computing the inverse transform, we decided against this approach and solely considered the presented realization for Realigned.

Depending on the given application, there are obviously other input partitions that could be considered, e.g., input partitions of size $N_x \times \frac{N_y}{P_1} \times \frac{N_z}{P_2}$. To accommodate for this, our variants could be extended by another global data redistribution, which closes the cycle in Figures 4.8 and 4.9. We stress that this would result in different output partitions when computing real-to-complex forward transforms.

When assuming our input partition of size $\frac{N_x}{P_1} \times \frac{N_y}{P_2} \times N_z$, another alternative could be to compute the 1D-FFTs in x-direction, before computing the 1D-FFTs in y-direction. When utilizing Realigned (cf. Figure 4.9) it is obvious that, next to other disadvantages, the output data layout would not be identical to the input data layout, which is generally undesirable. In contrast, this alternative could indeed result in performance improvement, when considering Default (cf. Figure 4.8). This is caused by the fact, that this approach would not require a packing phase for the second global data redistribution. Since our implementation already measures the duration and therefore the impact of the individual packing, communication, and unpacking phases, we do not incorporate this approach. Nonetheless, we note that this alternative could provide an interesting trade-off for computing the forward transform, since it would enjoy both fewer packing phases and no need to wait for a completion of the send operation.

4.5 Global Redistribution Methods

Until now, we solely focused on 2D-1D Slab Decomposition, 1D-2D Slab Decomposition, and Pencil Decomposition and introduced the variants Default and Realigned for each of them. While we have seen that there are already considerable advantages and disadvantages between the different approaches, we only regarded an abstract global redistribution of the intermediate results. To realize the global redistribution, which consists of packing, communication, and unpacking phase, CUDA and MPI provide a large variety of tools that were introduced in Section 3.1 and Section 3.3, respectively. As before, we stress that the right choice of the utilized tools is vital to achieve the best performance, since the global redistribution of the intermediate results is responsible for a significant proportion of the total runtime.

Generally, the different approaches can be categorized by their underlying MPI communication method, which we denote by *Peer2Peer* and *All2All*: Peer2Peer utilizes MPI's non-blocking send and receive operations in standard mode, which are part of its point-to-point communication mechanism. In contrast to this, All2All uses parts of MPI's collective communication mechanism. Depending on the choice of the communication method, there are further optimizations that focus on the packing and unpacking phase. In particular, we implement *Sync*, *Streams*, and *MPI_Type* as available options for Peer2Peer communication, along with Sync and MPI_Type for All2All communication. While the subsequent sections provide a detailed insight into these packing and unpacking options, w.r.t. the corresponding communication method, the general concept is as follows: For Peer2Peer communication, Sync copies each individual partition into the contiguous send-buffer and waits for the copy to complete, before the data is sent to the corresponding process. On the contrary, Streams does not wait for the copy to complete. Instead, each copy operation is assigned to a specific CUDA-stream, which allows a second thread to be notified and to start

sending, as soon as the operation is complete. If the data has to be unpacked, both Sync and Streams copy the partitions individually to `temp`. As a third option, MPI_Type is inspired by the work of Dalcin et al. [DMK19] and adapted for GPUs. The key property of MPI_Type is to avoid the packing and unpacking phase altogether by using custom MPI data-types, which allows sending from and receiving into non-contiguous memory. The same principles are applicable for All2All communication, with the difference that Sync and Stream are identical, since a single synchronization before calling the All2All communication routine is sufficient.

Following the structure from above, we start by introducing the packing and unpacking methods for Peer2Peer in Section 4.5.1. Afterwards the packing / unpacking methods for All2All are presented in Section 4.5.2.

4.5.1 Packing / Unpacking Methods for Peer2Peer Communication

As already described above, Peer2Peer utilizes MPI's non-blocking send and receive operations in standard mode, namely, `MPI_Isend` and `MPI_Irecv` (cf. Section 3.3). This section aims to provide further insight into the available packing and unpacking methods: Sync, Streams, and MPI_Type. It is apparent, that the specific implementation of the particular method highly depends on the underlying decomposition method (i.e. 2D-1D Slab Decomposition, 1D-2D Slab Decomposition or Pencil Decomposition) and its variant (i.e. Default or Realigned). In particular, we have already seen that the packing or unpacking phase can be skipped sometimes (cf. Sections 4.2-4.4). Therefore, we present the individual methods for the most general case, where both packing and unpacking phase are required and state the necessary accommodations for the different special cases.

We start by introducing the relevant notations and assumptions for this section, whereupon a detailed description of Sync and Streams, together with their identical unpacking method is provided. Afterwards, we present the general structure of MPI_Type and describe how the local transpose of the intermediate results can be realized for the partition that is kept by the process.

Notations and Assumptions. We denote `pidx` as the unique identifier of the running process, w.r.t. the MPI communicator `comm`. We let `P_c` be the number of processes which are covered by the same communicator. In particular, the value of `P_c` is equal to P if slab decomposition is used, or P_i (for either $i \in \{1, 2\}$) if pencil decomposition is used. In case pencil decomposition is considered, we assume that `comm` is the result of splitting the global communicator via `MPI_Comm_split` (cf. Listing 3.10). For the sake of better readability, we further assume for this section that the value of `P_c` always divides N_x, N_y, and $\hat{N}_z$ without remainder. Due to this assumption, the same amount of bytes is sent to and received from all other processes. It is therefore sufficient to solely introduces the variables `width` and `height` to describe the potentially non-contiguous partitions of `out`: `width` is used to refer to the maximum number of words that lie contiguous in memory, which is also considered as a slice, whereas `height` denotes the total number of slices in the partition. Analogously, variables `n_width` and `n_height` are used to describe the partition `temp`. Therefore, the invariant `width*height = n_width*n_height` holds true. Finally, the variable `s` denotes the number of bytes per real-valued word, i.e., 4 bytes for single-precision and 8 bytes for double precision, and `kind` is introduced as a placeholder to describe either host-to-device, device-to-host, or device-to-device memory transfers.

Listing 4.1 *Peer2Peer-Sync: Packing Method.*

```
for (size_t i = 1; i < P_c; i++) {
    p = (pidx + i) % P_c;
    // start non-blocking receive operation for process p
    MPI_Irecv(&recv[p*width*height], 2*s*width*height, MPI_BYTE,
        p, p, comm, &recv_req[p]);
    // copy intermediate results to send-buffer and synchronize
    cudaMemcpy2DAsync(&send[p*width*height], 2*s*width,
        &out[p*width], 2*s*width*P_c, 2*s*width, height, kind);
    cudaDeviceSynchronize();
    // start non-blocking send operation for process p
    MPI_Isend(&send[p*width*height], 2*s*width*height, MPI_BYTE,
        p, pidx, comm, &send_req[p]);
}
```

Peer2Peer-Sync. Listing 4.1 provides an overview of the implementation of packing and communication phase when using Peer2Peer-Sync. The process iterates over the identifiers of the other processes and starts each iteration by calling the non-blocking receive operation `MPI_Irecv` (cf. Listing 3.7). As each sent and received partition consists of `width*height` complex-valued words, the offset for process p in the receive-buffer is equal to `p*width*height` and a total of `2*s*width*height` bytes are received. The corresponding receive-request is stored in `recv_req[p]`. Process `pidx` immediately continues by copying its intermediate results from `out` into the contiguous send-buffer `send`. The essential specification of `cudaMemcpy2DAsync` can be found in Listing 3.2. Most notably, setting the pitch of the send-buffer to the size of a single slice (i.e. `2*s*width`) results in the fact that the data is contiguously aligned in `send`. To ensure that the process only starts sending after the partition has been copied to the send-buffer, the process is synchronized via `cudaDeviceSynchronize()`. Finally, process `pidx` starts transmitting the partition to process p by using `MPI_Isend`. It is therefore apparent that the use of non-blocking send and receive operations allow for an overlap of the packing and communication phase.

When considering CUDA-aware MPI, `send` and `recv` are located on device memory and placeholder `kind` is set to `cudaMemcpyDeviceToDevice`. Otherwise, both buffers are located on page-locked host memory (cf. Section 3.1) and `kind` is substituted for `cudaMemcpyDeviceToHost`. In case Realigned is used (for the forward transform), we have seen that the partitions already lie contiguous in `out`. Therefore, when using CUDA-aware MPI, `out` can be used directly as the send-buffer and `cudaMemcpy2DAsync` can be omitted in Listing 4.1. Note, that in this case it is important to call `MPI_Waitall(P_c, send_req, MPI_STATUSES_IGNORE)` before computing the next FFTs, in order to avoid overwriting the send-buffer. If on the other hand MPI is used without CUDA-awareness, the intermediate results stored in `out` are copied to `send` via a single contiguous copy-operation, which is executed before the loop of Listing 4.1.

Peer2Peer-Streams. While Streams is similarly structured to Sync, the essential difference is that Streams does not wait for the individual copy operations to complete. The general structure of Streams can be found in Listing 4.2. In contrast to Sync, `cudaMemcpy2DAsync` is assigned to a particular CUDA-stream. We have seen in Section 3.1 that the operation `cudaLaunchHostFunc`

Listing 4.2 *Peer2Peer-Streams: Packing Method.*

```
// create and start thread which is responsible for sending the intermediate results
mpisend_thread = std::thread(MPIsend_Thread, args);
for (size_t i = 1; i < P_c; i++) {
    p = (pidx + i) % P_c;
    // start non-blocking receive operation for process p
    MPI_Irecv(&recv[p*width*height], 2*s*width*height, MPI_BYTE,
        p, p, comm, &recv_req[p]);
    // copy intermediate results to send-buffer without synchronizing
    cudaMemcpy2DAsync(&send[p*width*height], 2*s*width,
        &out[p*width], 2*s*width*P_c, 2*s*width, height, kind, streams[p]);
    // enqueue callback function for notifying mpisend_thread to start sending
    cudaLaunchHostFunc(streams[p], this->MPIsend_Callback, (void *)&params[p]);
}
```

can be used to start the execution of an host function after all previous tasks of the stream are completed. This can be used by Streams, by implementing a thread-safe synchronous channel between `MPIsend_Callback` and the second thread `mpisend_thread` to notify the thread to start sending to a particular process. Therefore, `params[p]` simply contains the necessary tools to create the channel along with the value of `p`. Similarly, parameter `args` contains the same tools, along with the required information to call `MPI_Isend`. For the sake of completeness, we note that Streams requires MPI to be thread-safe. This can be ensured by using `MPI_Init_Thread` to initialize the MPI environment and requiring `MPI_THREAD_MULTIPLE` as the required level for thread support [MPI21, p. 491].

The same adjustments of Sync, depending on whether CUDA-aware MPI is used, can be adopted for Streams as well. Analogously, `cudaMemcpy2DAsync` can be omitted in Listing 4.2 if Realigned is used in combination with CUDA-aware MPI (for the forward transform). Therefore, Streams and Sync are identical in this special case. If on the other hand MPI without CUDA-awareness is used together with Realigned, Streams simply replaces `cudaMemcpy2DAsync` with `cudaMemcpyAsync` and copies the individual contiguous partitions to the send-buffer. Note, that this approach slightly differs from Sync in this case, since Sync utilizes a single copy-operation.

Unpacking Method for Peer2Peer-Sync and Peer2Peer-Streams. When using Sync or Streams, we have seen that there has to be some sort of synchronization after copying the intermediate results to the send-buffer. Essentially, the manner of synchronization can be considered to be the sole difference between Sync and Streams. When reviewing different options for the unpacking phase it is apparent that the individual received partitions can be unpacked independently. Thus, there is no need for an additional synchronization, which is why we realize the unpacking method for Sync and Streams identically. Using the same notations and assumptions as before, an overview of this unpacking method is given in Listing 4.3. Since both Sync and Streams store their receive requests in `recv_req`, `MPI_Waitany` can be used to wait for individual receive requests to complete (cf. Section 3.3). Importantly, the value of `p` is set to the index of the completed request, which is by definition of Listings 4.1, 4.2 identical to the identifier of the sending process. Furthermore, `recv_req[p]` is marked as inactive after returning from the function call. In case there are no more

Listing 4.3 *Unpacking Method for Peer2Peer-Sync and Peer2Peer-Streams.*

```
int p;
do {
    // wait for a receive operation to complete
    MPI_Waitany(P_c, recv_req, &p, MPI_STATUSES_IGNORE);
    if (p == MPI_UNDEFINED) // if there are no more active handles
        break;
    // copy received data from contiguous recv to temp
    cudaMemcpy2DAsync(&temp[p*n_width], 2*s*n_width*P_c,
        &recv[p*n_width*n_height], 2*s*n_width,
        2*s*n_width, n_height, kind1);
} while (p != MPI_UNDEFINED);
cudaDeviceSynchronize();
```

active requests, i.e., all partitions were received, the function immediately returns and sets `p` to `MPI_UNDEFINED`. When reviewing the parameters of `cudaMemcpy2DAsync`, placeholder `kind1` is set to `cudaMemcpyDeviceToDevice` if CUDA-aware MPI is used and to `cudaMemcpyHostToDevice`, otherwise. Finally, `cudaDeviceSynchronize` is used to ensure that all memory copies are complete and the subsequent batched FFTs can be computed on input `temp`.

Next, we cover the special cases that might arise depending on the underlying decomposition method and its variant: Since `temp` is always allocated on device memory, placeholder kind is set to `cudaMemcpyDeviceToDevice` or `cudaMemcpyHostToDevice`, depending on whether CUDA-aware MPI is utilized. Furthermore, we have seen that 2D-1D Slab Decomposition, 1D-2D Slab Decomposition and the second global redistribution of Pencil Decomposition do not require an unpacking phase when using Default. Therefore, it is sufficient to use `MPI_Waitall(P_c, recv_req, MPI_STATUSES_IGNORE)` and set `temp` to `recv` if CUDA-aware MPI is used. Otherwise, `recv` is located on page-locked host memory and `cudaMemcpy2DAsync` is replaced in Listing 4.3 with `cudaMemcpyAsync` to copy the individual contiguous partitions to `temp`.

Listing 4.4 *Peer2Peer-MPI_Type*

```
if (!cuda_aware) // copy intermediate results to host
    cudaMemcpy(send, out, P_c*width*height, cudaMemcpyDeviceToHost);

for (size_t i = 1; i < P_c; i++) {
    p = (pidx + i) % P_c;
    // start non-blocking receive operation for process p
    MPI_Irecv(&recv[p*n_width], 1, MPI_R_TYPE, p, p, comm, &recv_req[p]);
    // start non-blocking send operation for process p
    MPI_Isend(&send[p*width], 1, MPI_S_TYPE, p, pidx, comm, &send_req[p]);
}
```

Peer2Peer-MPI_Type. In contrast to the previous two methods, MPI_Type aims to remove the demand for a packing and unpacking phase completely. This is realized by using custom MPI datatypes, which were introduced in Section 3.3. Listing 4.4 shows the general structure of the communication phase, when MPI_Type is used. Similarly to the combination of Sync and Realigned, all intermediate results are copied to `send` in case MPI without CUDA-awareness is used. Afterwards, the process iterates over the identifiers of the other processes and immediately starts the corresponding non-blocking receive and send operations. For this purpose, the two custom MPI datatypes `MPI_R_TYPE` and `MPI_S_TYPE` are utilized. For the sake of completeness, review Listing 4.5 for their initialization. Most notably, when using Listings 3.2, 3.8 as reference, parameters `count`, `blocklength`, and `stride` of `MPI_Type_vector` correspond exactly to the parameters `height`, `width`, and `spitch` of `cudaMemcpy2DAsync`, respectively (cf. Listings 4.1, 4.2). Finally, we note that while each iteration of Listing 4.4 uses the same MPI datatype, the passed pointer in the first argument of `MPI_Isend` and `MPI_Irecv` determines the corresponding offset in the send-buffer and receive-buffer.

Listing 4.5 *Peer2Peer-MPI_Type: Initialization of the Custom Datatypes.*

```
MPI_Datatype MPI_S_TYPE, MPI_R_TYPE;
MPI_Type_vector(height, 2*s*width, 2*s*width*P_c, MPI_BYTE, &MPI_S_TYPE);
MPI_Type_vector(n_height, 2*s*n_width, 2*s*n_width*P_c, MPI_BYTE, &MPI_R_TYPE);
```

As with Sync and Streams, there are a few special cases to consider when realizing MPI_Type: First, there is a major advantage when using this option together with CUDA-aware MPI, since both send- and receive-buffer can be omitted. To further illustrate this point, MPI_Type solely requires `temp` as an additional buffer, whereas Sync and Stream require either `send` or `recv` for 2D-1D / 1D-2D Slab Decomposition, or both for Pencil Decomposition. Similarly to using Realigned with CUDA-aware MPI, it is therefore necessary to ensure that the send-buffer is not overwritten when computing the subsequent FFTs. Secondly, the send and receive operations in Listing 4.4 can easily be replaced with the corresponding ones of Listing 4.1, in case either the packing or unpacking phase is not required. Finally, MPI_Type can wait for the completion of its receive requests by using `MPI_Waitall`, whereupon the data can be copied to `temp` via a single host-to-device memory transfer if MPI without CUDA-awareness is used.

Local Transpose. We have seen in Sections 4.2-4.4 that, for each decomposition method, the k-th partition of the intermediate results is sent to process k (w.r.t the communicator if Pencil Decomposition is considered). Until now, we have solely covered the case $k \neq$ `pidx` with Sync, Streams, and MPI_Type. For the remaining case $k =$ `pidx`, an easy solution would be to use MPI as well, by simply letting the loop variable `i` start from value 0 in Listings 4.1, 4.2, and 4.4. Alternatively, the additional overhead of using MPI can be avoided by using a device-to-device memory transfer. Since this operation is independent of the other partitions, it can be executed right before unpacking the received partitions, e.g., after Listing 4.1 and before Listing 4.3, without the need for an additional synchronization. Therefore, the local transpose can be hidden behind the packing, communication, and unpacking phase. For the sake of completeness, we note that one has to pay special attention if MPI without CUDA-awareness is used in combination with `MPI_Waitall`, followed by a single host-to-device memory transfer, since this could potentially overwrite the

copied data of the local transpose. To accommodate for this, the local transpose can either be realized by copying the intermediate results to `recv`, or by splitting the single host-to-device memory transfer in two.

4.5.2 Packing Methods for All2All Communication

As an alternative to Peer2Peer, MPI also provides All2All which is part of its collective communication mechanism. For this purpose, the relevant operations are `MPI_Alltoall`, `MPI_Alltoallv`, and `MPI_Alltoallw`, which were introduced in Section 3.3. We have seen in Section 4.5.1, that Peer2Peer allows Sync, Streams, and MPI_Type as available packing / unpacking methods. For All2All, there is no need to differentiate between Sync and Streams, since a single synchronization before the communication routine is sufficient. Therefore, we solely consider Sync and MPI_Type for All2All.

We start presenting the notations and assumptions for this section. Thereafter, the details of Sync and MPI_Type are presented for All2All.

Notations and Assumptions. We adapt the notations and assumptions from the previous section. In particular, we emphasize the particular assumption that the value of `P_c` always divides N_x, N_y, and $\hat{N}_z$ without remainder. This assumption allows the following description be simplified, since `MPI_Alltoall` can be used instead of `MPI_Alltoallv` (cf. Listing 3.9). As before, we start by presenting the individual options in the most general setting, where both packing and unpacking phase are required. The necessary accommodations for the different special cases of the underlying decomposition methods and their variants are discussed thereafter.

Listing 4.6 *All2All-Sync*

```
// copy intermediate results to send-buffer
for (size_t p = 0; p < P_c; p++) {
    cudaMemcpy2DAsync(&send[p*width*height], 2*s*width,
        &out[p*width], 2*s*width*P_c, 2*s*width, height, kind);
}
cudaDeviceSynchronize();

MPI_Alltoall(send, 2*s*width*height, MPI_BYTE,
    recv, 2*s*width*height, MPI_BYTE, comm);

// copy received data to temp-buffer
for (size_t p = 0; p < P_c; p++) {
    cudaMemcpy2DAsync(&temp[p*n_width], 2*s*n_width*P,
        &recv[p*n_width*n_height], 2*s*n_width*n_height,
        2*s*n_width, n_height, kind1);
}
cudaDeviceSynchronize();
```

All2All-Sync. The general structure of All2All-Sync can be found in Listing 4.6, where packing, communication, and unpacking phase are realized. Starting with the packing phase, the individual copy operations are identical to Peer2Peer-Sync. Before calling the blocking All2All communication routine, host and device are synchronized to ensure that all intermediate results are copied to the contiguous send-buffer. Due to the above assumptions, each process sends and receives exactly `2*s*widht*height` bytes. In particular, note that the local transpose, which was computed separately for Peer2Peer (cf. Section 4.5.1), is covered by `MPI_Alltoall` in this case. After completing the communication with each process, the received data can be unpacked into `temp`. Again, the individual copy operations of the unpacking phase can be realized in the same manner as already described in Listing 4.3.

To consider the arising special cases for the different decomposition methods, the corresponding values of placeholders `kind` and `kind1` are the same as for Peer2Peer. In case the packing or unpacking phase is not required, the corresponding instructions of Listing 4.6 can simply be omitted if CUDA-aware MPI is used. Otherwise, the corresponding loop is simply replaced by a single `cudaMemcpy` operation that performs a single device-to-host memory transfer for the packing phase or a single host-to-device memory transfer for the unpacking phase.

All2All-MPI_Type. Similarly to Peer2Peer-MPI_Type, this approach aims to remove the demand of the packing and unpacking phase completely. Listing 4.7 provides an overview of All2All-MPI_Type, where the additional variable `cuda_aware` is used to indicate whether CUDA-aware MPI is used. In contrast to `MPI_Alltoall`, this method uses `MPI_Alltoallw`, where each of the parameters `counts`, `sdispls`, `rdispls`, `MPI_R_TYPES`, `MPI_S_TYPES` is an array of size `P_c`. Starting with the custom MPI datatypes, `MPI_R_TYPES` and `MPI_S_TYPES` simply consist of `P_c` identical elements `MPI_S_TYPE` and `MPI_R_TYPE` (cf. Listing 4.5), respectively. Again, we note that without the assumptions from above, the individual elements would need to consider the remainders of `P_c` dividing N_x, N_y, or $\hat{N}_z$. When considering the remaining parameters `counts`, `sdispls`, and `rdispls`, their p-th elements are set respectively to 1, `p*width`, and `p*n_width`. This corresponds exactly to the individual pointer offsets of Peer2Peer-MPI_Type in Listing 4.4.

In case the packing phase can be skipped during execution, there are two considerable approaches: Either `height` is set to 1, when configuring `MPI_S_TYPE` during the initialization phase, or `MPI_S_TYPE` is set to `MPI_BYTE`. For the latter, a new array `sendcounts` has to be introduced which contains `P_c` elements with value `2*s*width` and is passed as the second parameter, instead of `count`. Symmetric adjustments can be made if the unpacking phase can be omitted.

Listing 4.7 *All2All-MPI_Type*

```
if (!cuda_aware) // copy intermediate results to host
    cudaMemcpy(send, out, P_c*width*height, cudaMemcpyDeviceToHost);

MPI_Alltoallw(send, counts, sdispls, MPI_S_TYPES,
    recv, counts, rdispls, MPI_R_TYPES, comm);

if (!cuda_aware) // copy received data to device
    cudaMemcpy(temp, recv, P_c*n_width*n_height, cudaMemcpyHostToDevice);
```

5 Evaluation

In Chapter 4, we presented the details for 2D-1D Slab Decomposition, 1D-2D Slab Decomposition, and Pencil Decomposition (cf. Sections 4.2-4.4). In particular, we proposed the two variants Default and Realigned for each underlying decomposition method and discussed potential advantages and disadvantages, e.g., additional data buffers and the risk of overwriting the send-buffer by computing the subsequent FFTs. For these six variants, we presented possible methods for the global data redistribution phases in Section 4.5: Peer2Peer-Sync, Peer2Peer-Streams, Peer2Peer-MPI_Type, All2All-Sync, and All2All-MPI_Type. Furthermore, all names decomposition methods, variants, and global redistribution methods were implemented for both forward and inverse transforms, with additional support for both CUDA-aware and default MPI.

While we have seen a simplified theoretical comparison of slab decomposition and pencil decomposition in Section 2.5.3, a model for modern computing environments would require additional fine-tuning for the specific systems. In particular, the assumption that the latencies and bandwidths are identical for every combination of two processes is far from applicable in practice. Instead, even when solely considering a single computing node with multiple GPUs, these GPUs are likely to use different communication links (e.g. NVLink or PCI Express) and are therefore required to be modeled by different parameters. For analyzing the performance of the presented decomposition methods, their variants, and all global redistribution methods, we are therefore interested in different test systems. In this work, we utilize five different test systems: *PCSGS*, *Argon*, *Krypton*, *BW-GPU4*, and *BW-GPU8*. PCSGS, Argon, and Krypton are small computing pools, with four GPUs each, and consist of four, two, and one computing nodes, respectively. To evaluate the performance on a larger number of GPUs, along with the weak and strong scalability of the individual methods, we use BW-GPU4 and BW-GPU8. Both test systems are part of the bwUniCluster 2.0 [bwHa] and are maintained by bwHPC [bwHb]. BW-GPU4 consists of 14 computing nodes with four NVIDIA Tesla V100 GPUs each, whereas BW-GPU8 consists of 10 nodes with eight NVIDIA Tesla V100 GPUs each. For this work, we utilize up to 32 GPUs of BW-GPU4 and up to 64 GPUs of BW-GPU8.

In this chapter, we start by presenting the underlying methods for conducting and evaluating the performance tests in Section 5.1. Afterwards, the different test systems are presented in Section 5.2. The performance results are discussed in Section 5.3.

5.1 Methods

As stated before, we are interested in evaluating the performance of 2D-1D Slab Decomposition, 1D-2D Slab Decomposition, and Pencil Decomposition, along with their variants Default and Realigned for each particular test system. For Slab Decomposition, all five global redistribution methods are tested. Since Pencil Decomposition requires two global redistribution phases, the implemented methods provide a total of 25 combinations (when neglecting that Peer2Peer-Streams is identically

to Peer2Peer-Sync for Realigned if CUDA-aware MPI is used). Instead of testing each of them, the two global redistribution phases are evaluated separately by choosing Peer2Peer-Sync as the fixed method for the other phase. Since both phases are independent, the remaining combinations can easily be approximated with high accuracy. As described in Chapter 4, this work focuses on real-to-complex forward transforms and complex-to-real inverse transforms. Both directions are tested, and we utilize OpenMPI 4.1 [Sofa] where the tests are conducted with and without CUDA-awareness. All benchmarks are performed using double precision and different input sizes are considered: Starting from input size 128^3, the sizes are increased by a factor of 2 in each iteration, e.g., $128^2 \times 256$ for the second iteration (as before, we use the notation $N_x \times N_y \times N_z$). The maximum input size depends on the underlying test system and is therefore presented in Section 5.2 for each particular system. Each individual test is performed 20 times with additional 10 warm-up rounds, which are excluded from the performance measurements. In total, over 550,000 test runs are performed.

To validate that each variant and each global redistribution methods behaves correctly on the particular test system, we perform additional validation tests. They follow the structure presented in Section 2.6 and are performed in a separate test iteration, which is excluded from the performance measurements as well. For this purpose, we coarsen the input sizes and conduct the validation tests for the input sizes 128^3, 256^3, 512^3, and 1024^3.

Finally, we note that our implementation provides an easily extendable framework for conducting benchmarks, where the tests can be defined in JSON [Bra] and exemplary SLURM [Sch] scripts are provided for the job-submission.[1] Furthermore, our framework ensures GPU affinity by automatically generating rankfiles for OpenMPI [Sofc].

5.2 Test Systems

Next, we introduce a general overview of the different test systems. We focus on their hardware specification and refine relevant details on how the performance benchmarks are conducted.

PCSGS. Starting with the simplest test system, PCSGS is composed of four computing nodes which are connected by Gigabit Ethernet. Each node is equipped with a single 18-core 3.00GHz Intel i9-10980XE processor, 64GiB of DDR4 memory, and one NVIDIA Geforce RTX 3080 GPU with 10GiB of GDDR6X video memory. The test system is able to compute distributed 3D-FFTs up to input size 512×1024^2.

Argon. This test system is composed of two computing nodes which are connected by 10Gbit/s Ethernet. Each node is equipped with two 12-core 2.10GHz Intel Xeon Silver 4116 processors and 191GiB of DDR4 memory. Furthermore, the nodes are equipped with similar GPUs: The first node with two NVIDIA Tesla P100 GPUs and the second node with two NVIDIA Quadro GP100 GPUs, each GPU having 16GiB of HBM2 memory. Each 12-core processor of a given node is connected to a single GPU via PCIe, such that processes that perform operations on GPU0 or GPU1 should be

[1] https://github.com/eggersn/DistributedFFT/tree/master/jobs

bound to the first or second processor respectively. In particular, the communication between the two GPUs traverses the SMP interconnect. The test system is able to compute distributed 3D-FFTs up to input size 1024^3.

Krypton. An overview of Krypton can be found in Figure 5.1. It consists of a single computing node, which is equipped with two 20-core 2.30GHz IBM Power9 processors, 382GiB of DDR4 memory, and four NVIDIA Tesla V100 GPUs with 32GiB of HBM2 memory each. Most notably, when compared to the previous two systems, Krypton utilizes NVLink 2.0 interconnections, where a single interconnect provides a unidirectional bandwidth of 25GB/s. By using pairs of three NVLinks, the bandwidth can be effectively tripled. It can be seen in Figure 5.1 that the four GPUs are grouped into pairs of two, such that the corresponding processor and the two GPUs are individually connected via NVLink. Therefore, when considering the exemplary data exchange of GPU0 and GPU2, the connection has to traverse the SMP interconnect, similar to Argon.

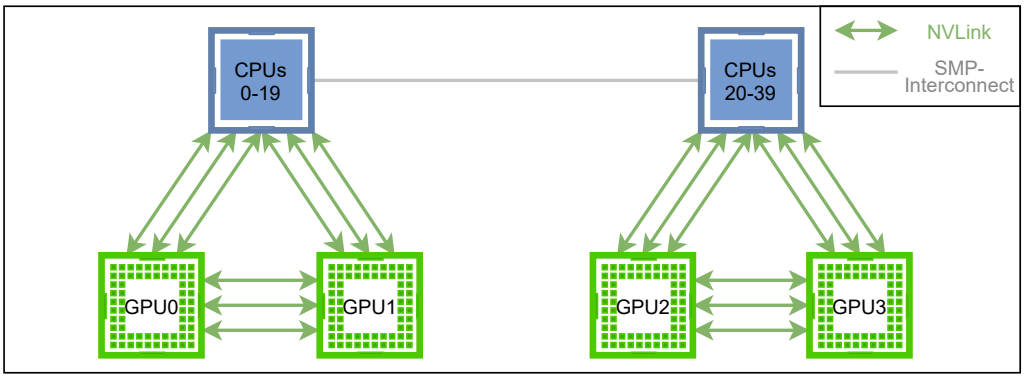


Figure 5.1: *Krypton: Overview of the Test System.* The schematic visualizes the interconnections of the available GPUs.

BW-GPU4. An overview of BW-GPU4 can be found in Figure 5.2. It is part of the bwUniCluster 2.0 and consists of 14 computing nodes, where each one is equipped with two 20-core 2.1GHz Intel Xeon Gold 6230 processors, 382GiB of DDR4 memory, and four NVIDIA Tesla V100 GPUs with 32GiB of HBM2 memory each. The four GPUs are grouped into pairs of two, such that the two GPUs are connected by two NVLinks, resulting in a unidirectional bandwidth of 50GB/s. In contrast to Krypton, the CPUs and GPUs are connected via PCIe. Furthermore, additional NVLinks provide high bandwidth between GPU0 (GPU1) and GPU2 (GPU3). Finally, the different nodes of BW-GPU4 are arranged in an Infiniband network (IB HDR 100), according to the fat-tree topology [Lei85], with a tree height of two. To ensure low communication latencies, each node is equipped with two Mellanox ConnectX-6 HDR100 Infiniband adapter cards (cf. mlx5_0 and mlx5_1 in Figure 5.2). On BW-GPU4, we perform our benchmarks on 4, 8, 16, 24, and 32 GPUs. For each number of GPUs, input sizes of up to 2048^3 are tested. For smaller numbers of utilized GPUs, the input size is increased until the node runs out of memory. The performance of Slab Decomposition is measured for MPI with and without CUDA-awareness. On the other hand, the performance of Pencil Decomposition is solely measured for CUDA-aware MPI.

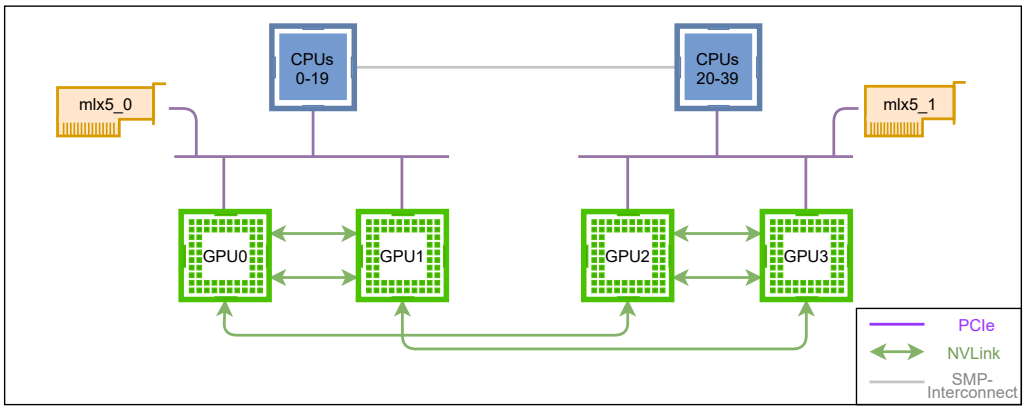


Figure 5.2: *BW-GPU4: Overview of the Test System.* The schematic visualizes the interconnections of the available GPUs and the Infiniband adapter cards (*mlx5_0* and *mlx5_1*) on a single node.

BW-GPU8. An overview of a single BW-GPU8 node can be found in Figure 5.3. It is part of the bwUniCluster 2.0 and consists of 10 computing nodes, where each one is equipped with two 20-core 2.1GHz Intel Xeon Gold 6248 processors, 764GiB of DDR4 memory, and eight NVIDIA Tesla V100 GPUs with 32GiB of HVM2 memory each. The GPUs are grouped in pairs of two, where each pair is connected to a PCIe switch and a Mellanox ConnectX-6 HDR100 Infiniband adapter card. All nodes are connected with Infiniband via a single switch. On BW-GPU8, we perform our benchmarks on 8, 16, 32, 48, and 64 GPUs. For each number of GPUs, input sizes of up to 2048^3 are tested. For smaller numbers of utilized GPUs, the input size is increased until the node runs out of memory. The performance of Slab Decomposition is measured for MPI with and without CUDA-awareness. On the other hand, the performance of Pencil Decomposition is solely measured for CUDA-aware MPI.

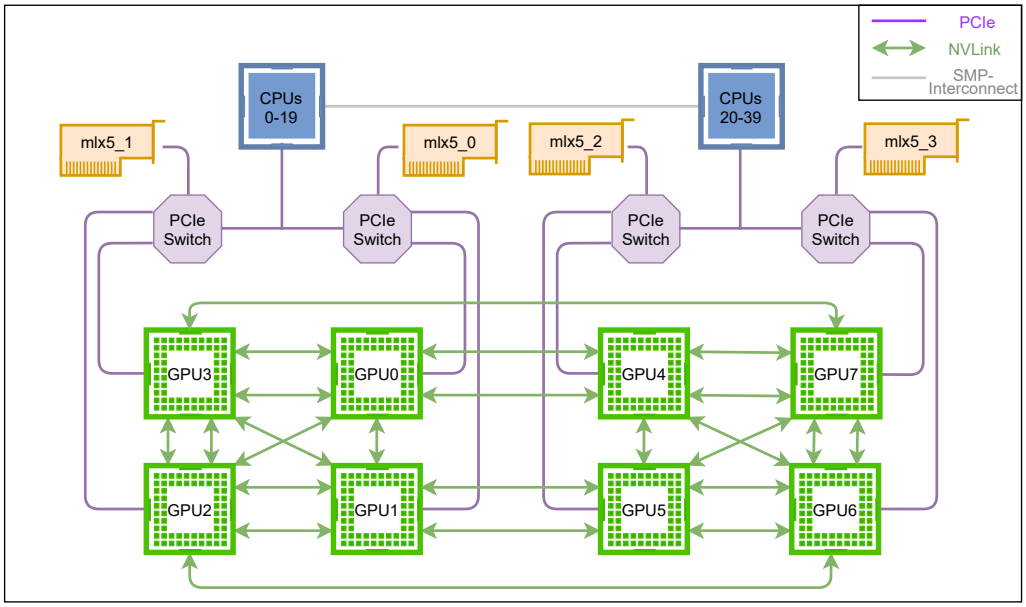


Figure 5.3: *BW-GPU8: Overview of the Test System.* The schematic visualizes the interconnections of the available GPUs and the Infiniband adapter cards (*mlx5_0*, . . ., *mlx5_3*) on a single node.

5.3 Results

After providing a general overview of the performed tests (cf. Section 5.1) and the utilized test systems (cf. Section 5.2), we present our results. The raw data of the benchmarks is also publicly available, along with the resulting graphs for all performed benchmarks.[2] We report the absolute and relative runtime results, where the relative result is obtained by dividing the respective (total) runtime by the minimal (total) runtime result of the given input size. Thus, the best performing option for a given input size has the relative runtime value 1.0. The advantage of this representation is that, for example, a relative runtime value of 2.5 for Peer2Peer-Sync and input size 512^3 immediately states that the best performing option for the same input size experiences a performance increase of 150%, when compared to Peer2Peer-Sync.

We start in Section 5.3.1 by presenting the performance of the different decomposition methods and their variants on the different test systems. On a more fine-grained level, the different methods for the global redistributions are analyzed in Section 5.3.2, with respect to the underlying decomposition method and the test system. Finally, the strong and weak scaling results are reviewed in Section 5.3.3.

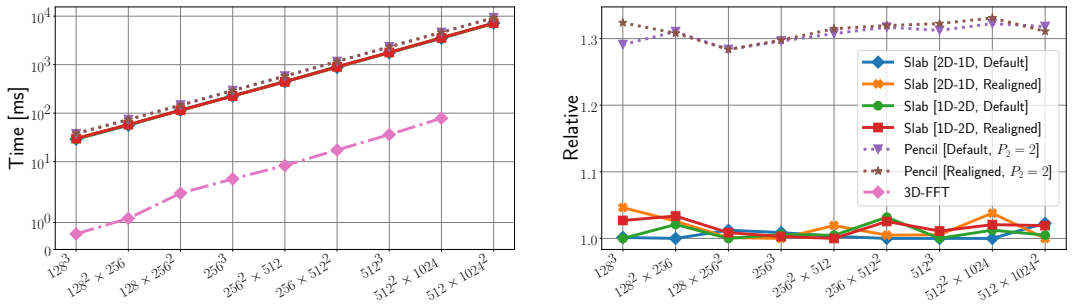
5.3.1 Comparison of the Decomposition Methods

This section reviews the performance of Default and Realigned for each of the available decomposition methods, i.e., 2D-1D Slab Decomposition, 1D-2D Slab Decomposition, and Pencil Decomposition (cf. Chapter 4). The methods for conducting the performance tests are described in Section 5.1. For solely evaluating the performance of the decomposition methods and their variants, the runtime results of 20 iterations are averaged and the best performing global redistribution method is selected for each considered input size. Thus, the presented results are generally grouped according to the computed direction (i.e. forward or inverse transform), by whether MPI is used with or without CUDA-awareness, and by the number of utilized GPUs.

PCSGS. Figure 5.4 visualizes the runtime results of PCSGS when computing the forward transform with CUDA-aware MPI. In particular, the best performing decomposition method and variant for a given input size can be found in Figure 5.4b having value 1.0. For Pencil Decomposition, a processor grid of size 2×2 is considered. For PCSGS, global data redistribution is responsible for more than 99% of the total runtime. Furthermore, the performance difference of computing the forward or inverse transform, with or without CUDA-aware MPI is negligible. Therefore, we solely consider a single case here and provide the full data in Appendix A.

It can be seen that Slab Decomposition dominates Pencil Decomposition in this setting and that the individual variants do not have much influence on the runtime. Most notably, the runtime of both Pencil Decomposition variants is for each input size roughly 1.3 times the runtime of the best performing variant of Slab Decomposition. Since each GPU is located on a different node and the point-to-point bandwidth is approximately the same for each pair of nodes, the simplified model of

[2] https://github.com/eggersn/DistributedFFT



(a) Total Runtime

(b) Relative Runtime

Figure 5.4: *PCSGS: Total and Relative Runtime of the Decomposition Methods.* The graphs visualize the results of computing the forward transform with CUDA-aware MPI. Figure **(a)** shows the total runtime results of the decomposition methods and their variants. Additionally, *3D-FFT* denotes the runtime of performing the full three-dimensional FFT on a single GPU without a distributed approach. Figure **(b)** visualizes the relative runtime results, where the result of each variant and each input size is divided by the minimal result of the given input size (without considering *3D-FFT*).

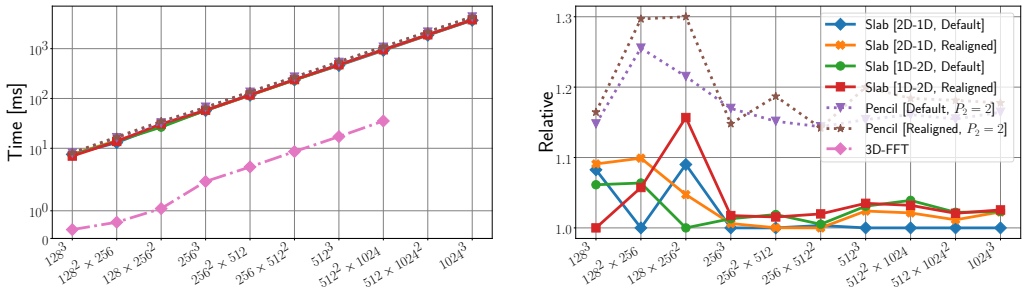
Section 2.5.3 can be applied. In particular, when neglecting the latencies for large input sizes, the runtime of pencil decomposition can be approximated by

$$T_C^P = (P_2 - 1)\frac{2sN_xN_y\left(\left\lfloor\frac{N_z}{2}\right\rfloor + 1\right)}{PP_2B} + (P_1 - 1)\frac{2sN_xN_y\left(\left\lfloor\frac{N_z}{2}\right\rfloor + 1\right)}{PP_1B}$$
$$= 2\frac{2sN_xN_y\left(\left\lfloor\frac{N_z}{2}\right\rfloor + 1\right)}{8B} = 4\frac{2sN_xN_y\left(\left\lfloor\frac{N_z}{2}\right\rfloor + 1\right)}{P^2B} = \frac{4}{3}T_C^S,$$

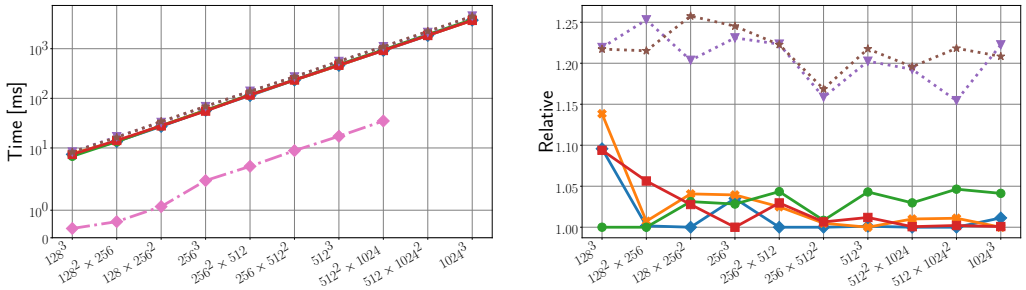
where T_C^S denotes the runtime of slab decomposition for the fixed input size.

Argon. Figure 5.5 shows the runtime results of Argon for the forward and inverse transform and for MPI with and without CUDA-awareness. For Slab Decomposition, it can be seen that the variants perform similar for the forward and inverse transform. An exception to this rule can be found when considering Realigned 1D-2D Slab Decomposition, which experiences a noticeable performance increase for the inverse transform (e.g. compare Figure 5.5c and Figure 5.5d). This is mainly caused by the fact that the forward and inverse transform of Default 2D-1D Slab Decomposition has approximately the same runtime (i.e. ±0.2%), whereas the computation of the 2D-FFT in yx-direction for Realigned 1D-2D Slab Decomposition is significantly faster for the inverse transform, e.g., for input size 1024^3 the average runtime the 2D-FFT in yx-direction for the forward and inverse transform are ~177ms and ~40ms respectively. Interestingly, it seems that this is not caused by the idle wait time of Realigned for the forward transform (cf. Sections 4.2-4.4), since the same phenomenon is also determined for MPI without CUDA-awareness.

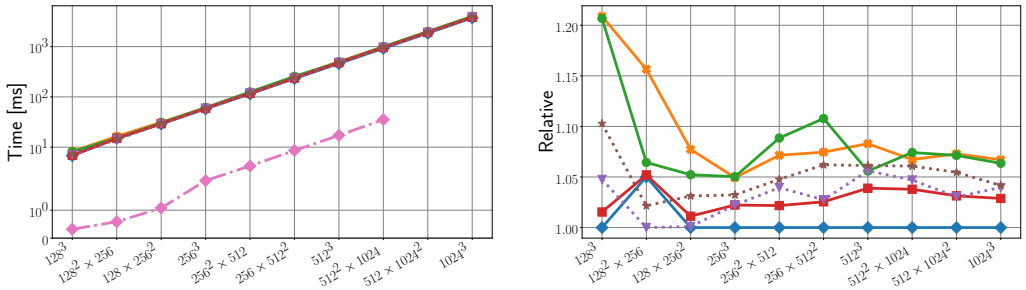
In to following, we provide further insight why Pencil Decomposition achieves significantly better results when using CUDA-aware MPI (cf. Figure 5.5a and Figure 5.5c): When reviewing the performance of Pencil Decomposition in this setting, the essential difference to PCSGS is that Argon



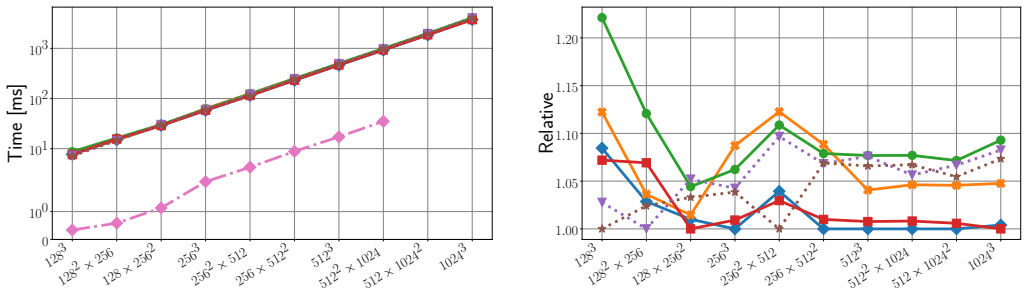
(a) *Forward Transforms for MPI without CUDA-Awareness.*



(b) *Inverse Transforms for MPI without CUDA-Awareness.*



(c) *Forward Transforms for CUDA-Aware MPI.*



(d) *Inverse Transforms for CUDA-Aware MPI.*

Figure 5.5: *Argon: Total and Relative Runtime of the Decomposition Methods.* Figures **(a)**-**(d)** show the results for the forward and inverse transforms, with and without CUDA-aware MPI. For the runtime, *3D-FFT* is used to denote the performance of computing the full three-dimensional FFT on a single GPU without a distributed approach. The graphs on the right side visualize the relative runtime results, where the result of each variant and each input size is divided by the minimal result for the given input size (without considering *3D-FFT*).

has two GPUs per computing node. Thus, the first global redistribution of Pencil Decomposition (cf. Section 4.4) can be executed on the same node, whereas only the second one requires communication between both nodes. Therefore, Pencil Decomposition with CUDA-aware MPI is able to avoid unnecessary device-to-host and host-to-device memory transfers in this scenario. For our benchmarks, this results in a performance increase of ~130% for large input sizes in the first global redistribution phase, e.g., for Default Pencil Decomposition and input size 1024^3 the first global redistribution phase endures 180ms and 424ms for MPI with and without CUDA-awareness respectively. When considering the impact on the total runtime, this approximately results in a performance increase of 10%, e.g., 3884ms and 4316ms for the same example. For the second global redistribution phase of pencil decomposition or the single global redistribution of Slab Decomposition, CUDA-aware MPI does not provide any benefits since the main bottleneck is the connection via Ethernet and the intermediate results have to be buffered on the host either way.

Finally, it is interesting to note that the performance of some Slab Decomposition variants decreases when using CUDA-aware MPI: While the runtime of Default 2D-1D Slab Decomposition and Realigned 1D-2D Slab Decomposition is approximately the same (i.e. ±1%), the runtime of the other two variants decreases by up to 8%. This is caused by the fact that MPI_Type is used in this scenario, whose performance will be discussed in Section 5.3.2.

Krypton. The total and relative runtime results are displayed in Figure 5.6. As before the results are categorized by whether the forward or inverse transform is computed and whether MPI is used with or without CUDA-awareness. Similar to Argon, the runtime of the forward and inverse transform are similar for most decomposition methods. The most notable exception to this is Realigned for both 2D-1D / 1D-2D Slab Decomposition: For CUDA-aware MPI, the inverse transform of Realigned 2D-1D Slab Decomposition experiences a performance increase of between 36% and 44% for input sizes above $512^2 \times 1024$ (cf. Figure 5.6c and Figure 5.6d). For Realigned 1D-2D Slab Decomposition and CUDA-aware MPI, the performance increase is even more outstanding and is greater than 50%, 80%, and 100% for input sizes above 256^3, 256×512^2, and 512×1024^2 respectively. The reason for the performance increase is, analogously to Argon, that the runtime of the inverse transform for Realigned is reduced by a significant factor. For input size 1024^3, the comparison of forward and inverse transform in computing the 1D- and 2D-FFTs is displayed in Table 5.1. Although the durations for computing the 1D-FFTs for the Realigned Pencil Decomposition are also reduced for the inverse transform, e.g., from 41.5ms to 21.2ms for the 1D-FFT in x-direction (or y-direction) and input size 1024^3, the impact on the total runtime substantially smaller with up to 7%.

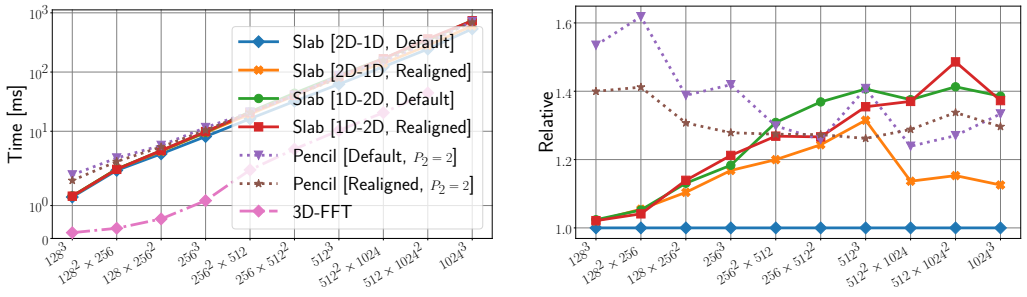
	Default		Realigned	
	Forward	Inverse	Forward	Inverse
zy-direction	16.0ms	16.3ms	90.8ms	46.9ms
x-direction	45.1ms	45.0ms	36.0ms	20.0ms

(a) *2D-1D Slab Decomposition.*

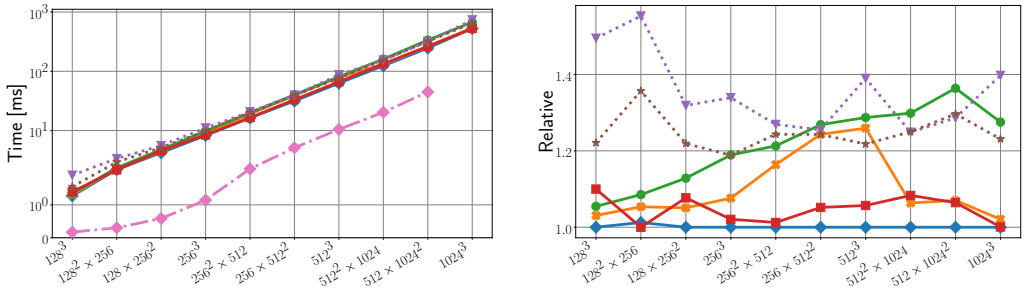
	Default		Realigned	
	Forward	Inverse	Forward	Inverse
z-direction	5.4ms	5.3ms	29.2ms	19.7ms
yx-direction	201.6ms	203.0ms	178.7ms	39.6ms

(b) *1D-2D Slab Decomposition.*

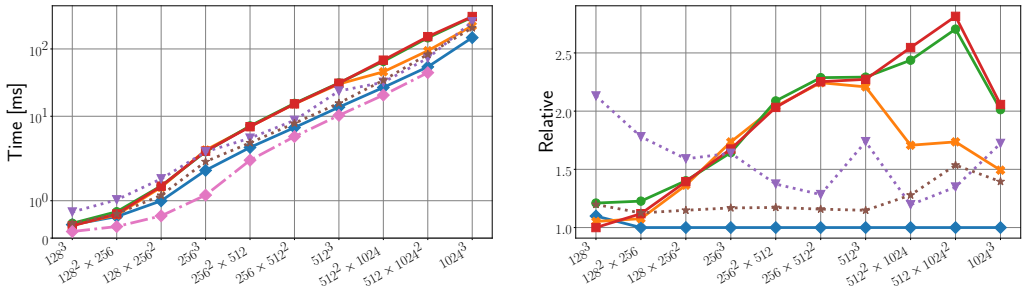
Table 5.1: *Krypton: Duration of Computing the FFTs for Slab Decomposition.* The results are grouped by 2D-1D Slab Decomposition (cf. Table 5.1a) and 1D-2D Slab Decomposition (cf. Table 5.1b) and show the runtime for the individual 1D- and 2D-FFTs for the forward and inverse transform. The runtime results are averaged for input size 1024^3 and are identical for MPI with and without CUDA-awareness.



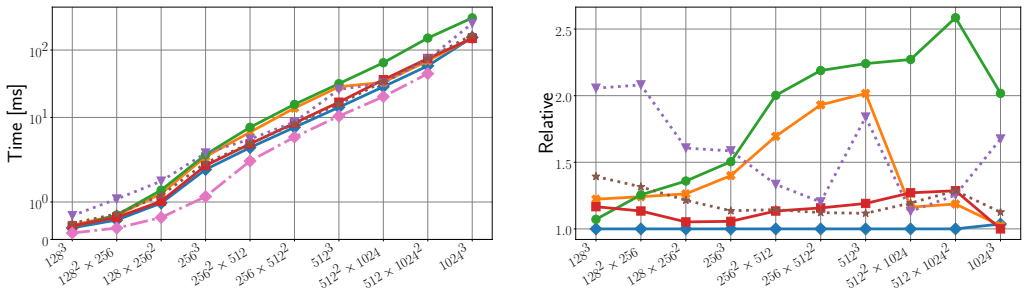
(a) *Forward Transforms for MPI without CUDA-Awareness.*



(b) *Inverse Transforms for MPI without CUDA-Awareness.*



(c) *Forward Transforms for CUDA-Aware MPI.*



(d) *Inverse Transforms for CUDA-Aware MPI.*

Figure 5.6: *Krypton: Total and Relative Runtime of the Decomposition Methods.* Figures **(a)**-**(d)** show the results for the forward and inverse transforms, with and without CUDA-aware MPI. For the runtime, *3D-FFT* is used to denote the performance of computing the full three-dimensional FFT on a single GPU without a distributed approach. The graphs on the right side visualize the relative runtime results, where the result of each variant and each input size is divided by the minimal result for the given input size (without considering *3D-FFT*).

For the second observation, it can be seen that the relative difference between Default 2D-1D Slab Decomposition and the other variants of Slab Decomposition increases for the forward transform when utilizing CUDA-aware MPI (cf. Figure 5.6a and Figure 5.6c). This is caused by the fact that, due to the system architecture of Krypton (cf. Figure 5.1), the duration of the global redistribution phase decreases considerable by a factor of more than five, whereas the duration of computing the 1D- and 2D-FFTs is not influenced by using CUDA-aware MPI. In particular for input size 1024^3, the duration of the global redistribution for Default 2D-1D Slab Decomposition is 86ms and 474.1ms when using MPI with and without CUDA-awareness respectively. Therefore, a similar reduction for the remaining Slab Decomposition variants results in a higher impact of the duration for computing the 1D- and 2D-FFTs (cf. Table 5.1).

When reviewing the performance of Pencil Decomposition in this setting, it can be seen that Realigned generally performs better than Default for small input sizes. This is mostly caused by the impact of the computational overhead of the 1D-FFTs in y-direction for Default. As we have seen in Section 4.4, Default requires $\hat{N}_z/P_2$ plans compared to one for Realigned. Furthermore, it can be seen throughout Figure 5.6 that Default Pencil Decomposition experiences significant performance hits when the input size is increased in x-direction. Whereas it is normal for the duration of computing the 1D-FFTs in x-direction to increase, e.g. 4.2ms and 51.3ms for input sizes 512×1024^2 and 1024^3 respectively, the duration of computing the 1D-FFTs in y-direction also experience a significant performance decrease, e.g., 13.0ms and 87.4ms for the same input sizes. A likely cause for this is that the individual plans of the 1D-FFTs in y-direction lie non-contiguous in memory with batch size N_x/P_1.

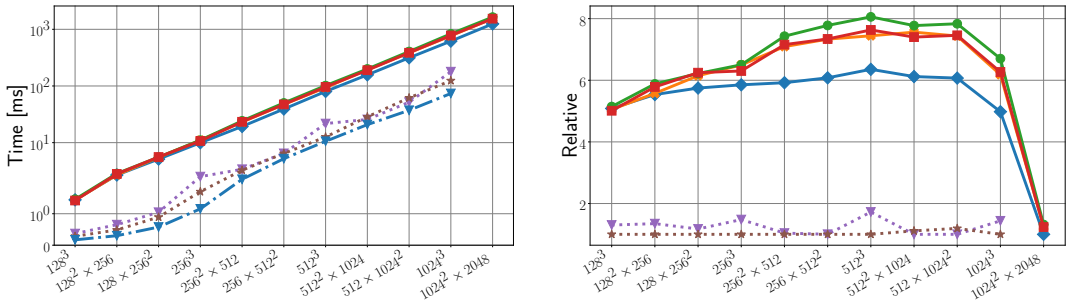
Finally, it can be seen that 2D-1D Slab Decomposition generally outperforms both Pencil Decomposition variants. In particular, when reviewing the influences on the total runtime, the duration for the global redistribution phases of Pencil Decomposition are consistently larger than the global redistribution of 2D-1D Slab Decomposition by a factor of approximately 1.2. When using CUDA-aware MPI, the impact of this effect decreases. This is beneficial for Realigned Pencil Decomposition, when considering small input sizes and CUDA-aware MPI, since its individual FFTs are computed similarly efficient as for 2D-1D Slab Decomposition (cf. Figure 5.6c). Due to the same reasons we have discussed earlier, this does not apply for Default Pencil Decomposition.

BW-GPU4. For this test system, the results are grouped by the number of utilized GPUs. All benchmarks are performed using CUDA-aware MPI. Figure 5.7 shows the total and relative runtime results for the forward transform. For the sake of readability, the results for the inverse transform can be found in Appendix A. In contrast to the previous test systems, we utilize up to 32 GPUs for BW-GPU4. This allows to a higher flexibility when choosing the processor grid for pencil decomposition. Using the same notation as before, $P_1 \times P_2$ describes the processor grid for pencil decomposition. In general, we select the processor grid size for the benchmarks of BW-GPU4 such that $P_2 \in \{2, 4, 8\}$: For $P_2 = 2$, we take advantage of the paired NVLinks between GPU0 and GPU1 (cf. Figure 5.2) and perform the first global redistribution phase locally on the same processor. For $P_2 = 4$, all GPUs of a single node are used in the first global redistribution phase. Analogously for $P_2 = 8$, all GPUs of two nodes are used. Finally, the benchmarks are performed for input sizes ranging from 128^3 to 2048^3.

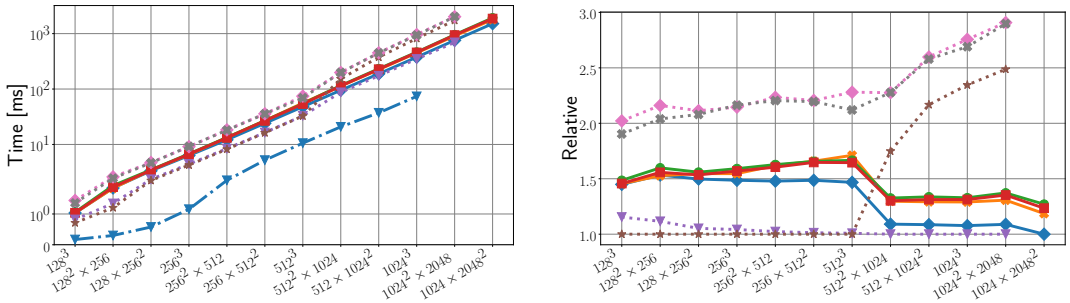
The first obvious observation is that, due to the additional memory requirements of Pencil Decomposition (cf. Section 4.4), it can be seen throughout the results, that Slab Decomposition is sometimes able to compute larger input sizes (e.g. $1024^2 \times 2048$ in Figure 5.7a).

The second notable observation is that the difference between the distributed decomposition methods and 3D-FFT, which denotes the total runtime of the full three-dimensional FFT on a single GPU, diminishes for increasing numbers of utilized GPUs. Furthermore, it can be seen in Figure 5.7a that both variants of Pencil Decomposition perform similar to 3D-FFT. This is caused by the fact, that the 2×2 processor grid for $P = 4$ is chosen such that GPU0 communicates with GPU1 and GPU2 in the first and second global redistribution phase respectively. Therefore, the NVLink interconnections of these GPUs can be fully utilized (cf. Figure 5.2). When considering $P > 4$, this advantage vanishes for the second global redistribution phase, since the processes have to communicate with GPUs of physically different nodes.

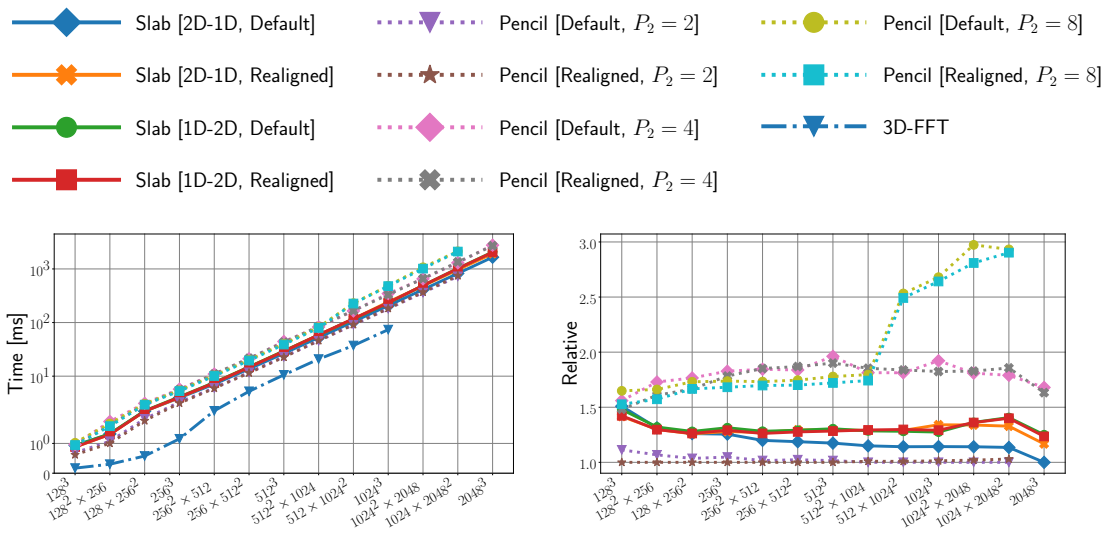
For the third observation, it can be seen throughout Figure 5.7 that a processor grid of size $P/2 \times 2$ provides the best performance for Pencil Decomposition. The reason for this is simply that the duration of the first global redistribution phase is only responsible for a small fraction of the total runtime, e.g., ~3% for $P = 32$ and Default Pencil Decomposition. Therefore, Pencil Decomposition only has to communicate with $P/2$ processes in the second global redistribution phase. This is significantly faster for smaller input sizes, e.g., a performance increase of >23% for $P = 32$ and input sizes smaller than 256×512^2, when compared to Default 2D-1D Slab Decomposition. For larger input sizes, we have already seen in the theoretical comparison of slab and pencil decomposition (cf. Section 2.5.3) that this difference decreases. Furthermore, the theoretical comparison can be used to explain why all processor grids of Pencil Decomposition perform better for small input sizes and large numbers of GPUs (cf. Figures 5.7d, 5.7e).



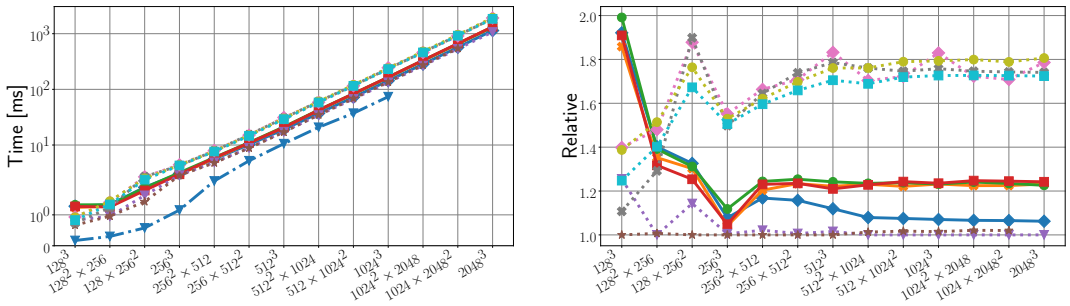
(a) *Absolute and Relative Results for* $P = 4$.



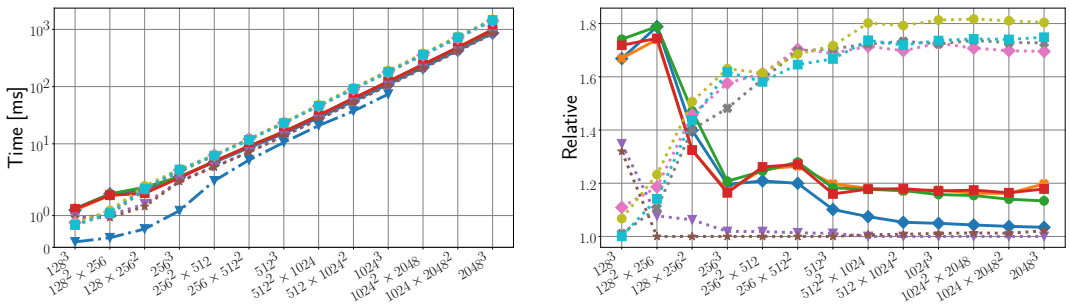
(b) *Absolute and Relative Results for* $P = 8$.



(c) *Absolute and Relative Results for* $P = 16$.



(d) *Absolute and Relative Results for* $P = 24$.



(e) *Absolute and Relative Results for* $P = 32$.

Figure 5.7: *BW-GPU4: Total and Relative Runtime of the Decomposition Methods.* Figures **(a)**-**(e)** show the results for the 4, 8, 16, 24, and 32 processes. Solely the forward transform with CUDA-aware MPI is considered. For the runtime, *3D-FFT* is used to denote the performance of computing the full three-dimensional FFT on a single GPU without a distributed approach. The graphs on the right side visualize the relative runtime results, where the result of each variant and each input size is divided by the minimal result for the given input size (without considering *3D-FFT*). For the sake of readability, only one legend is provided where P_2 unambiguously identifies the processor grid for Pencil Decomposition (i.e. with $P_1 = P/P_2$).

When comparing the performance of the individual decomposition methods for the forward and inverse transform, we found that Realigned performs significantly better for both inverse Slab Decomposition methods. Since this observation was already made for Krypton, we will not go into detail any further. Again, we refer the interested reader to Appendix A for the runtime results for the inverse transforms.

Finally, we note that for reasons unclear to us, some processor grids of Pencil Decomposition experience a significant performance hit for $P = 8$ with input sizes larger than $512^2 \times 1024$ (cf. Figure 5.7b) and $P = 16$ with input sizes larger than 512×1024^2 (cf. Figure 5.7c). In both cases, network congestion in the second global redistribution seems responsible for the runtime increase, e.g., for Realigned Pencil Decomposition and processor grid 4×2 its duration is 24.8ms and 144ms for input size 512^3 and $512^2 \times 1024$ respectively. It seems that this mainly affects processor grids $P_1 \times P_2$ with small values for P_1. Interestingly, Default Pencil Decomposition with processor grid 4×2 is not severely afflicted by this performance hit and still outperforms Slab Decomposition (cf. Figure 5.7b). Here, we found that this is caused by the utilized communication method Peer2Peer-MPI_Type, whose performance will be discussed further in Section 5.3.2.

BW-GPU8. The relative runtime results for this test system are similar to BW-GPU4. Therefore, we solely consider the results of the forward transform with CUDA-aware MPI for 64 utilized GPUs in Figure 5.8. The remaining results of the forward and inverse transform for 8, 16, 32, 48, and 64 GPUs can be found in Appendix A. For Pencil Decomposition, we consider the processor grids 8×8 and 16×4. Analogous to BW-GPU4, the processor grid 16×4 is aligned such that the GPUs communicating in the first global redistribution phase are assigned to the same processor, e.g.,

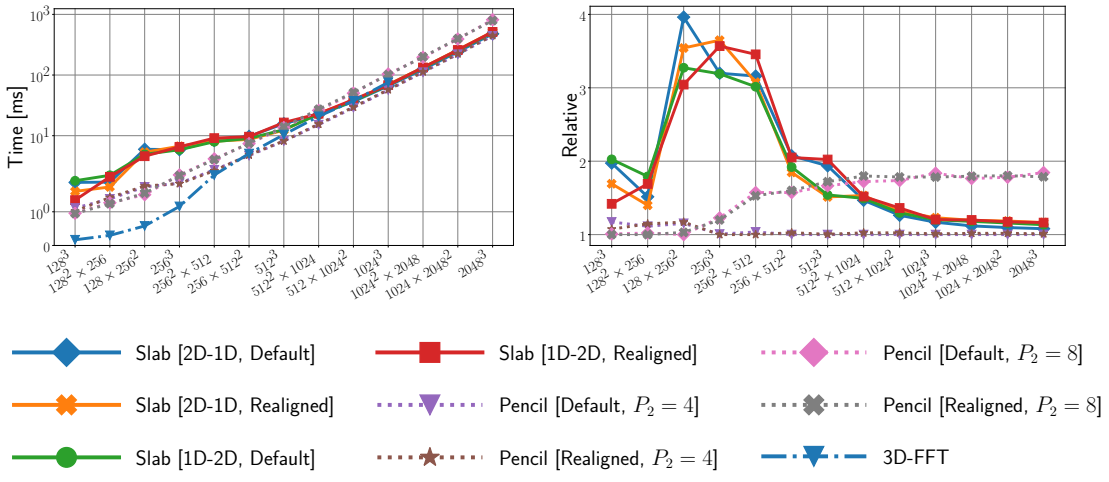


Figure 5.8: *BW-GPU8 Forward $P = 64$: Total and Relative Runtime of the Decomposition Methods.* The figure shows the results for 64 GPUs. Solely the forward transform with CUDA-aware MPI is considered. For the runtime, *3D-FFT* is used to denote the performance of computing the full three-dimensional FFT on a single GPU without a distributed approach. The graph on the right side visualize the relative runtime results, where the result of each variant and each input size is divided by the minimal result for the given input size (without considering *3D-FFT*).

GPU0 - GPU3 of Figure 5.3. This is beneficial since all involved GPUs are pair-wise connected via NVLink. For the processor grid 8×8, the processor grid is aligned such that all GPUs of a node are involved in the first global redistribution phase.

In Figure 5.8, there are two additional observations that stand out: The first one is that for small input sizes (i.e. smaller than 128×256^2) it can be seen that the more evenly spread processor grid 8×8 slightly outperforms the processor grid 16×4. When reviewing the durations of both global redistribution phases, we find a duration of 0.07ms and 0.53ms for the first phase with input size 128×256^2 and the processor grid 16×4 and 8×8, respectively. However, for the second phase we have respective durations of 1.49ms and 0.79ms, where the latencies of the connection establishment with 16 GPUs is non-negligible.

For the second observation, we explore a potential cause for the spike of the relative results of Slab Decomposition at input size 128×256^2: For message sizes smaller than 12KB, OpenMPI makes use of the *Eager Protocol*, where the message is directly sent to the endpoint [Sofb]. For larger message sizes, OpenMPI uses the *RDMA Pipeline Protocol*, where only a small fragment of the data is sent before waiting for the corresponding MPI receive [WSB+06]. Applying this to our use-case, the eager protocol is solely used for Slab Decomposition with $P = 64$ and input sizes smaller than $128^2 \times 256$ or with $P = 48$ and input size 128^3. Therefore, Slab Decomposition could experience increased latencies for input sizes larger than 128×256^2 in Figure 5.8.

5.3.2 Comparison of the Global Redistribution Methods

In Section 5.3.1, we compared the total and relative runtime of the decomposition methods 2D-1D Slab Decomposition, 1D-2D Slab Decomposition, and Pencil Decomposition, along with their variants Default and Realigned (cf. Sections 4.2-4.4). For the comparison of their runtime, we selected the best performing global redistribution method for each particular input size. Therefore, it remains open to discuss the performance of the implemented global redistribution method (cf. Section 4.5), with respect to the underlying variant, the decomposition method, and the utilized test system. For this, we use Peer2Peer-Sync as a baseline and solely consider cases where different global redistribution methods perform significantly better. Instead of categorizing the observations by the utilized test system, we list our observations and aim to generalize their requirements.

For the sake of full transparency, we note that the results of Peer2Peer-MPI_Type are excluded for the forward transform of Default 2D-1D Slab Decomposition on BW-GPU4 and BW-GPU8, due to an implementation fault which causes numerical inconsistencies. Since this scenario shows great similarities to the inverse transform of Realigned 1D-2D Slab Decomposition, the performance of Peer2Peer-MPI_Type can easily be approximated where necessary.

1. Observation (Peer2Peer-Streams). Starting with the most general observation, we find that Peer2Peer-Streams often outperforms the other global redistribution methods if, at least, the following three requirements are satisfied:

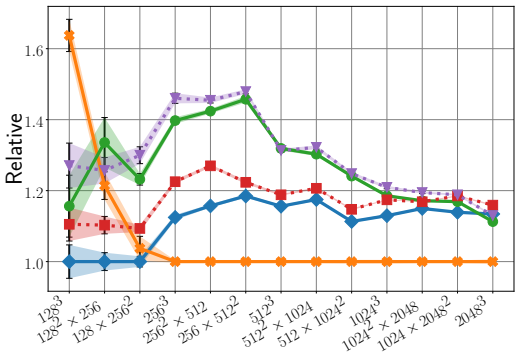
i) thread-safe MPI without CUDA-awareness is used,

ii) the input sizes are larger than a particular threshold, and

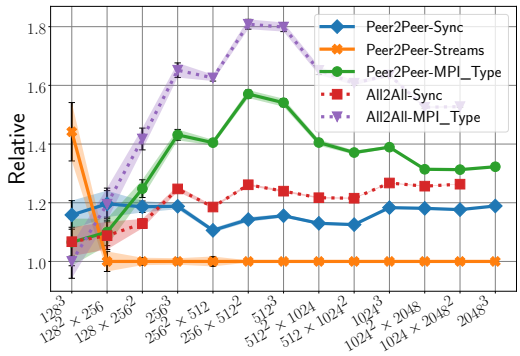
iii) the number of communicating GPUs is limited.

For us, the threshold of requirement *ii)* is often between 128×256^2 and 256^3. Furthermore, the limit of utilized GPUs for requirement *iii)* is used for BW-GPU8, where the limit is set to 32 GPUs (for 48 and 64 GPUs, All2All-Sync is more preferable). When considering CUDA-aware MPI, we have seen in Section 4.5 that Peer2Peer-Streams is in some cases identical to Peer2Peer-Sync, e.g., for forward transforms when using Realigned. In cases where Peer2Peer-Streams is applicable, the following observations apply for CUDA-aware MPI as well, only that the performance differences are often less preeminent.

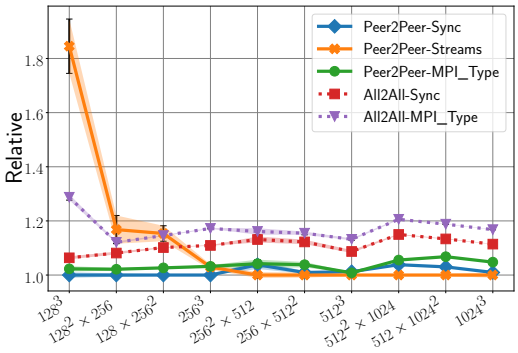
Figures 5.9a, 5.9b provide exemplary scenarios in which the use of Peer2Peer-Streams significantly increases the performance of the underlying decomposition method. In contrast to this, Figures 5.9c, 5.9d visualize two scenarios, in which Peer2Peer-Streams either performs significantly worse or does not provide a worthwhile performance increase. When reviewing the performance measurements in detail, we find that Peer2Peer-Streams starts waiting for incoming messages remarkably fast, e.g., for the scenario depicted in Figure 5.9a and input size 2048^3, Peer2Peer-Streams starts waiting after 159.0ms whereas Peer2Peer-Sync only starts waiting after 485.4ms. This is caused by the



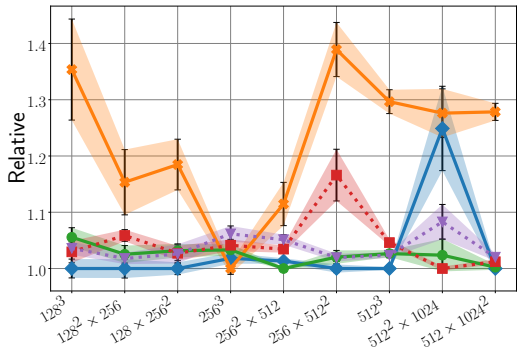
(a) *BW-GPU8: Forward Realigned 2D-1D Slab Decomposition for 32 GPUs*



(b) *BW-GPU4: Forward Default 1D-2D Slab Decomposition for 24 GPUs*



(c) *Krypton: Forward Realigned 2D-1D Slab Decomposition*

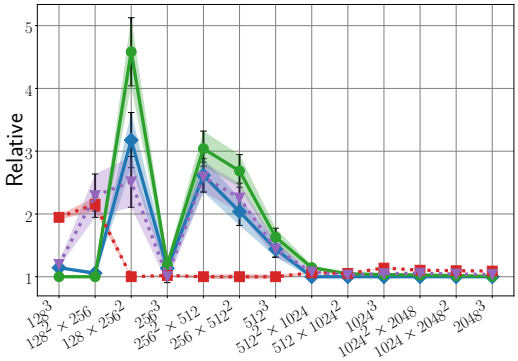


(d) *PCSGS: Forward Default 1D-2D Slab Decomposition*

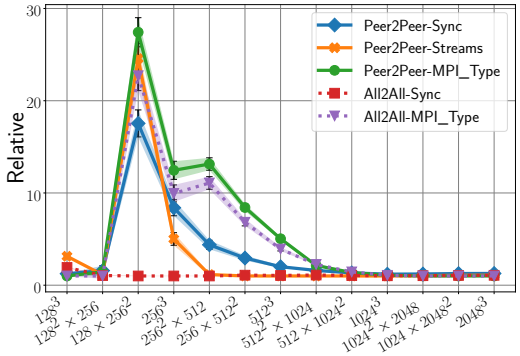
Figure 5.9: *1. Observation: Performance of Peer2Peer-Streams.* Figures **(a)**-**(d)** show the relative runtime results (cf. Section 5.3.1) for four exemplary decomposition methods. In each scenario, MPI without CUDA-awareness is considered. Each graph visualizes the averaged results (20 iterations and P processes), along with the 99% confidence interval of the mean value (using Student's t-distribution).

fact, that Peer2Peer-Streams does not synchronize with the device until all intermediate results are received (cf. Section 4.5). While this characteristic is also present for PCSGS in Figure 5.9d, here, Peer2Peer-Streams also shows a high deviation between minimum and maximum runtime, e.g., when considering the input size 512×1024^2 Peer2Peer-Streams shows a deviation of 517.6ms, compared to 200.7ms for Peer2Peer-Sync. Although this is not a problem for Krypton (cf. Figure 5.9c), it can be seen that, while Peer2Peer-Streams offers a good performance for input sizes larger than $256^2 \times 512$, the difference between Peer2Peer-Streams and Peer2Peer-Sync is negligible.

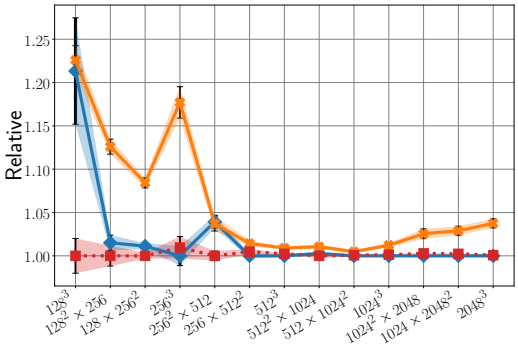
2. Observation (All2All-Sync). We found that All2All-Sync generally performs well for large numbers of utilized GPUs. Figure 5.10 highlights scenarios in which this is the case. The most notable difference was found for BW-GPU8 when utilizing 64 GPUs (cf. Figures 5.10a, 5.10b). Here, the other global redistribution methods show a significant performance increase for input sizes between 128×256^2 and 512^3. This is caused by a high deviation between minimum and maximum duration of the other global redistribution, e.g., for Figure 5.10a and input size 128×256^2, the minimum and maximum duration for Peer2Peer-Sync is 14.6ms and 71.0ms respectively, compared to the respective durations of 4.58ms and 4.67ms for All2All-Sync. When using MPI without CUDA-awareness, the margin between All2All-Sync and the remaining global redistribution methods is often even more amplified (cf. Figure 5.10b).



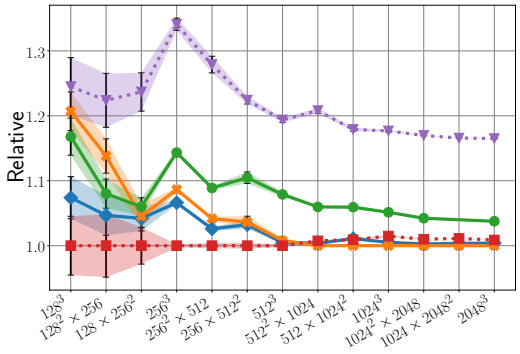
(a) *BW-GPU8: Forward Realigned 1D-2D Slab Decomposition for 64 GPUs and MPI with CUDA-awareness*



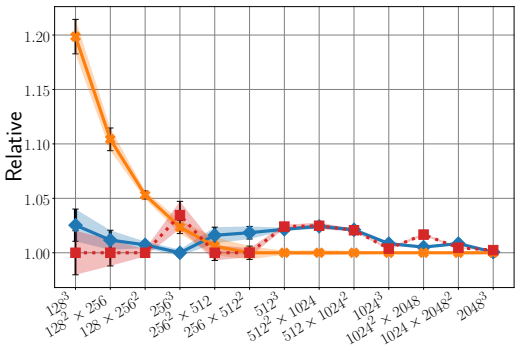
(b) *BW-GPU8: Forward Realigned 1D-2D Slab Decomposition for 64 GPUs and MPI without CUDA-awareness*



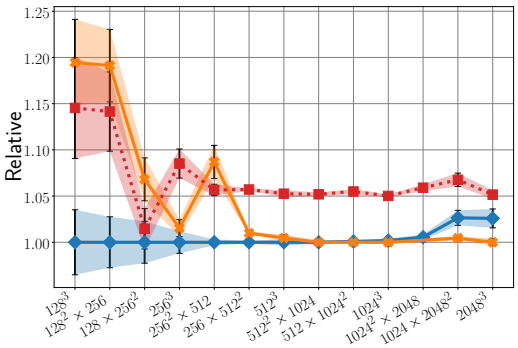
(c) *BW-GPU4: Forward Default Pencil Decomposition First Global Redistribution for* 4×8 *GPUs and MPI with CUDA-awareness*



(d) *BW-GPU4: Forward Default Pencil Decomposition Second Global Redistribution for* 16×2 *GPUs and MPI with CUDA-awareness*



(e) *BW-GPU4: Forward Default Pencil Decomposition First Global Redistribution for* 8×4 *GPUs and MPI with CUDA-awareness*



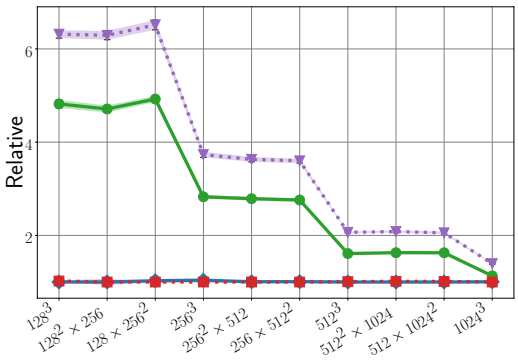
(f) *BW-GPU4: Forward Default Pencil Decomposition Second Global Redistribution for* 3×8 *GPUs and MPI with CUDA-awareness*

Figure 5.10: *2. Observation: Performance of All2All-Sync.* Figures **(a)**-**(f)** show the relative runtime results (cf. Section 5.3.1) for six exemplary decomposition methods. Each graph visualizes the averaged results (20 iterations and P processes), along with the 99% confidence interval of the mean value (using Student's t-distribution). For Figures 5.10c, 5.10e, and 5.10f, MPI_Type is excluded since it performs significantly worse (see 3. Observation for details).

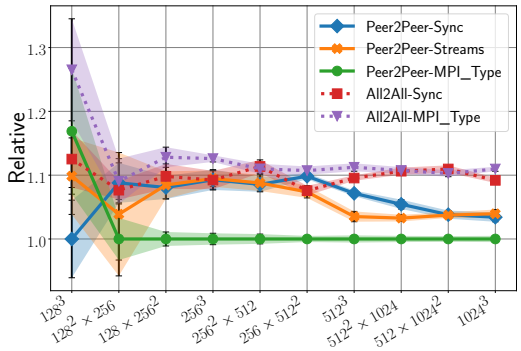
Since our previous examples solely reviewed the performance of Slab Decomposition, we note that All2All-Sync also provides good performance results for Pencil Decomposition for the first and second global redistribution if the respective parameters P_2 and P_1 are sufficiently large. The relative results of exemplary scenarios where this observation applies are depicted in Figure 5.10c and Figure 5.10d. Here, it can be seen that All2All-Sync provides the most notable performance difference for small input sizes. In contrast to this, Figure 5.10e and Figure 5.10f visualize the relative results of exemplary scenarios where P_2 and P_1 are rather small and Peer2Peer-Streams or Peer2Peer-Sync are more preferable methods.

3. Observation (Peer2Peer-MPI_Type). When using MPI without CUDA-awarness, we found that Peer2Peer-MPI_Type rarely provides significant advantages over Peer2Peer-Sync. In particular, when invoking the more expensive CUDA operations for device-to-host or host-to-device memory transfers, the difference between `cudaMemcpy` and `cudaMemcpy2D` often seem negligible for the packing and unpacking phase.

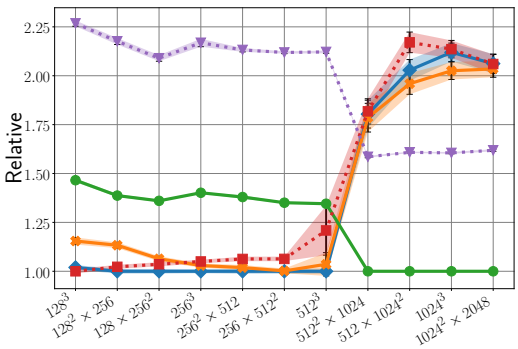
When reviewing the performance of Peer2Peer-MPI_Type for CUDA-aware MPI, we observe that the positioning of the partition in memory has an enormous impact on the runtime. By identifying an individual partition as a batch of slices and categorizing them as 1D-partitions, 2D-partitions, and 3D-partitions (cf. Section 4.1), we found that Peer2Peer-MPI_Type, and analogously All2All-MPI_Type, generally does not perform well if either the sent or the received partitions are one-dimensional. In particular, this is the case for the forward and inverse transforms when using MPI_Type for Realigned 2D-1D Slab Decomposition, Default 1D-2D Slab Decomposition, the first global redistribution phase of Default Pencil Decomposition, or for either global redistribution phase of Realigned Pencil Decomposition. As an example, Figure 5.11a visualizes the performance of MPI_Type for Realigned 2D-1D Slab Decomposition on Argon. To further emphasize this point, the relative distance for each input size is amplified by a factor of ~10, when reviewing the same



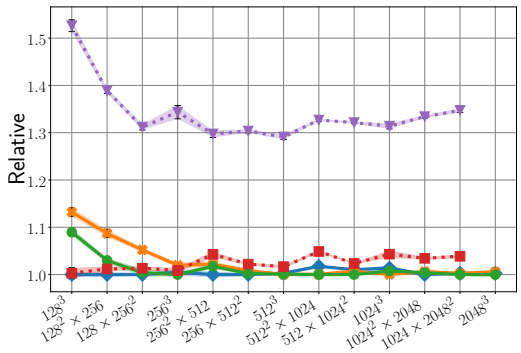
(a) *Argon: Forward Realigned 2D-1D Slab Decomposition*



(b) *Argon: Forward Default 2D-1D Slab Decomposition*



(c) *BW-GPU4: Forward Default Pencil Decomposition Second Global Redistribution for* 4×2 *GPUs*



(d) *BW-GPU4: Forward Default Pencil Decomposition Second Global Redistribution for* 4×4 *GPUs*

Figure 5.11: *3. Observation: Performance of Peer2Peer-MPI_Type.* Figures **(a)**-**(d)** show the relative runtime results (cf. Section 5.3.1) for four exemplary decomposition methods. Each graph visualizes the averaged results (20 iterations and P processes) when using CUDA-aware MPI, along with the 99% confidence interval of the mean value (using Student's t-distribution).

decomposition method on BW-GPU8 for 16 GPUs. Interestingly, it can also be seen in Figure 5.11a, that the relative distance between MPI_Type and Peer2Peer-Sync decreases dramatically when N_x is increased. This is simply caused by the fact, that the received 1D-partitions are composed of one-dimensional slices that lie contiguous in x-direction (cf. Section 4.2).

In case both sent and received partitions are two- or three-dimensional, Peer2Peer-MPI_Type is often able to provide competing performance to Peer2Peer-Sync. Exemplary scenarios for this can be found in Figures 5.11b-5.11d. Most notably, Peer2Peer-MPI_Type outperforms the remaining global redistribution methods on Argon (cf. Figure 5.11b), where both nodes are connected via 10Gbit/s Ethernet, and on BW-GPU4 for the second global redistribution of Default Pencil Decomposition on a 4×2 processor grid (cf. Figure 5.11c). For Argon, we found that the time difference between the first received message for Peer2Peer-MPI_Type and Peer2Peer-Sync is the sole cause for the performance difference, e.g., for input size $512^2 \times 1024$ the first partition is received after 952.15ms and 906.68ms on average when using Peer2Peer-Sync and Peer2Peer-MPI_Type respectively, which has to most significant impact on the respective total runtime values 974.56ms and 924.32ms. For BW-GPU4, the global redistribution methods without MPI_Type experience

a significant performance hit, e.g., the averaged total runtime of Peer2Peer-Sync is 32.89ms and 157.38ms for input size 512^3 and $512^2 \times 1024$ respectively. In contrast to this, Peer2Peer-MPI_Type shows an expected increase in its total runtime by a factor of ~2, i.e., respective runtimes of 44.26ms and 87.25ms. Similar to Section 5.3.1, these observations could indicate network congestion.

Finally, the correlation between the performance of MPI_Type and the size of the batched slices can be further explored by analyzing the duration of the second global redistribution phase of Figure 5.11c and Figure 5.11d. In particular, we have already seen that the transmitted partitions are of size

$$\frac{N_x}{P_1} \times \left(\frac{N_y}{P_1} \times \frac{\hat{N}_z}{P_2} \right),$$

where the individual slices are two-dimensional (as indicated by the brackets). Thus, it is easy to see that the size of the transmitted data during the second global redistribution is identical for processor grid 4×2 with input size $N_x \times N_y \times N_z$ (cf. Figure 5.11c) and processor grid 4×4 with input size $2N_x \times N_y \times N_z$ (cf. Figure 5.11d). The sole difference of both processor grids is therefore that grid 4×2 has to send fewer and larger two-dimensional slices to the other processes. In particular, it is important to note that P_1 is identical for both processor grids, which is essential when comparing the duration of the second global redistribution since each process has to communicate with $P_1 - 1$ other processes. The sampling rate of our performed benchmarks therefore allows the comparison for $2N_x = N_y = N_z = 512$ and $2N_x = N_y = N_z = 1024$: For the former, the duration is 20.49ms and 21.65ms for the processor grid 4×2 and 4×4 respectively. For the latter, the respective durations are 160.74ms and 166.51ms. Therefore, it is slightly beneficial to prefer 2D-partitions that contain fewer and larger two-dimensional slices.

4. Observation (All2All-MPI_Type). For the final observation, we found that All2All-MPI_Type shares little of the key properties of All2All-Sync (cf. Figure 5.10) and generally performs worse than Peer2Peer-MPI_Type for our test systems (cf. Figure 5.11).

5.3.3 Strong and Weak Scaling Results

Separately from the main results presented in Section 5.3.1 and Section 5.3.2, this section provides the strong and weak scaling results for Default 2D-1D Slab Decomposition and Default Pencil Decomposition. Since the number of utilizes GPUs was limited to 64 on BW-GPU8, these scaling results should only be used to obtain an additional overview of the behavior of both decomposition methods.

The results are visualized in Table 5.2. For Default Pencil Decomposition, the best performing processor grid $P/4 \times 4$ is selected. It can be seen that for input sizes larger than 512×1024^2, Default 2D-1D Slab Decomposition outperforms 3D-FFT, where the full forward transform is computed on a single GPU. Furthermore, the use of multiple GPUs obviously allows the computation of three-dimensional FFTs with larger input sizes. For Default Pencil Decomposition, this effect arises even earlier, where 3D-FFT is outperformed for 16×4 GPUs for input sizes larger than 256×512^2. Additionally, it can be seen that its performance with 12×4 GPUs is nearly identical to 3D-FFT.

		Utilized GPUs				
	3D-FFT	8	16	32	64	48
128^3	0.17	1.00	1.24	1.77	1.87	1.49
$128^2 \times 256$	0.30	1.83	1.47	2.29	1.88	2.38
128×256^2	0.58	3.31	2.30	2.86	5.97	3.62
256^3	1.16	6.23	4.51	3.15	5.87	4.50
$256^2 \times 512$	2.24	11.77	7.89	4.88	8.34	5.03
256×512^2	5.07	23.21	14.81	9.35	9.74	6.98
512^3	10.41	45.75	28.36	16.70	15.80	12.57
$512^2 \times 1024$	20.84	90.78	54.70	31.08	22.24	22.61
512×1024^2	37.55	177.99	112.27	59.58	36.52	41.12
1024^3	75.62	356.53	214.78	116.18	66.20	78.39
$1024^2 \times 2048$	—	715.68	440.59	230.96	124.36	153.85
1024×2048^2	—	1400.58	951.19	458.80	241.11	305.67
2048^3	—	—	1939.08	943.60	477.89	611.80

(a) *Default 2D-1D Slab Decomposition*

		Processor Grid				
	3D-FFT	2×4	4×4	8×4	16×4	12×4
128^3	0.17	0.29	0.58	0.88	1.11	1.10
$128^2 \times 256$	0.30	0.44	0.97	0.89	1.38	1.37
128×256^2	0.58	0.66	1.53	1.27	1.74	1.45
256^3	1.16	1.27	2.62	2.33	1.85	1.97
$256^2 \times 512$	2.24	2.78	4.94	4.09	2.73	3.08
256×512^2	5.07	5.71	9.39	7.34	4.69	5.42
512^3	10.41	11.61	20.07	13.61	8.15	10.20
$512^2 \times 1024$	20.84	23.38	35.50	26.18	15.15	19.78
512×1024^2	37.55	47.92	71.07	51.63	28.92	38.57
1024^3	75.62	95.54	205.70	102.37	56.66	76.12
$1024^2 \times 2048$	—	190.84	368.09	203.25	111.17	151.41
1024×2048^2	—	—	742.99	406.20	220.02	300.58
2048^3	—	—	1637.69	814.87	442.34	603.36

(b) *Default Pencil Decomposition for Processor Grids $P/4 \times 4$.*

Table 5.2: *Strong and Weak Scaling Results on BW-GPU8.* The runtime results (in milliseconds) are presented for the forward transform of Default 2D-1D Slab Decomposition (cf. Table 5.2a) and Default Pencil Decomposition (cf. Table 5.2b), when utilizing CUDA-aware MPI. As before, the best performing global redistribution method is chosen for each entry (cf. Section 5.3.1). The strong scaling results are displayed in the rows and the weak scaling results in the diagonals. For reference, *3D-FFT* denotes the results for performing the complete three-dimensional FFT on a single GPU, without using any distributed approach. Furthermore, the results for 48 GPUs are also included for the sake of completeness. To avoid confusion when assessing the weak scaling results, the runtime results of 48 GPUs are displayed separately.

Both decomposition methods generally indicate good strong and weak scaling results for large input sizes and $P \geq 16$. Since each node of BW-GPU8 is connected via Infiniband, Default 2D-1D Slab Decomposition even shows improvements when utilizing two computing nodes (i.e. 16 GPUs). In contrast to this, Default Pencil Decomposition shows performance degradation when switching from 2×4 to 4×4 as the underlying processor grid. As discussed before, this is caused by the fact that the processor grid 2×4 is able to fully utilize the NVLink interconnections on a single computing node. Furthermore, Table 5.2b highlights an additional performance hit for input size 1024^3 and the 4×4 processor grid. The reason for this seems to be twofold: On the one hand, there are indications for a possible network congestion, similar to our observations of Pencil Decomposition on BW-GPU4 with $P \in \{8, 16\}$ (cf. Section 5.3.1). On the other hand, Default Pencil Decomposition also shows significant performance hits if the input size is increased in x-direction. This phenomenon was already explored for Krypton in Section 5.3.1.

6 Conclusion

In this work, we present and benchmark multiple realizations for computing the Fast Fourier Transform of real-valued, three-dimensional input data on distributed GPU systems. For this purpose, we use slab decomposition and pencil decomposition of the global input data. Since both approaches require global communication between the processes, which has a significant impact on the total runtime, this work also focuses on the available communication strategies and their performance on different test systems. Our GPU library utilizes cuFFT for computing the individual FFTs, along with MPI for process communication.

The fundamental decomposition methods of our implementation are 2D-1D Slab Decomposition, 1D-2D Slab Decomposition, and Pencil Decomposition, whose details are presented and discussed in the respective Sections 4.2-4.4. Furthermore, we differentiate between the variants Default and Realigned, where we propose the latter to take advantage of the fact that cuFFT allows differing input and output data alignments. Both variants are applicable for each underlying decomposition method and are therefore presented in the corresponding Sections 4.2-4.4. Most notably, we found that Realigned allows a more efficient computation of the 1D-FFTs in y-direction for Pencil Decomposition and requires less memory when utilizing CUDA-aware MPI. Furthermore, Realigned provides additional advantages for 1D-2D Slab Decomposition, by avoiding the need of packing 1D-partitions in the global redistribution phase. On the other hand, we saw that Realigned requires additional synchronization when computing the forward transform to avoid overwriting the content of the send-buffer.

In Section 4.5, we presented multiple options to realize the global communication between the participating processes: Peer2Peer-Sync, Peer2Peer-Streams, Peer2Peer-MPI_Type, All2All-Sync, and All2All-MPI_Type. Here, we differentiated between Peer2Peer communication that utilizes non-blocking point-to-point communication mechanisms of MPI, and All2All which uses parts of MPIs collective communication mechanisms. Furthermore, we propose Peer2Peer-Streams which uses CUDA-streams to notify a second thread to start the non-blocking send operation, and adopt MPI_Type for GPUs from the previous work of Dalcin et al. [DMK19]. Most notably, the use of MPI_Type with CUDA-aware MPI entails significant benefits in that additional send- and receive-buffers are no longer required, which reduces the additional memory requirements by a factor of 2 for Slab Decomposition and by a factor of 3 for Pencil Decomposition, when used for both global redistribution phases.

We present our performance results in Chapter 5. To provide insight in the behavior of the decomposition methods, their variants, and the utilized global redistribution method in different computing environments, we perform the benchmarks on PCSGS, Argon, Krypton, BW-GPU4, and BW-GPU8. For this purpose, our implementation provides an easily extendable framework for defining multiple testcases in JSON.

Starting with the comparison of the decomposition methods and their variants (cf. Section 5.3.1), we found that Default 2D-1D Slab Decomposition generally performs best among the Slab Decomposition variants. The sole exception to this rule seems to be Realigned 1D-2D Slab Decomposition, which provides competing runtime results when the inverse transform is computed. With regard to the performance of Pencil Decomposition, we found that the underlying processor grid entails a significant performance increase for small numbers of GPUs, when the grid is specifically set up to utilize NVLink interconnections. Since our study is limited to 64 GPUs, the optimal processor grid for large numbers of GPUs remains uncertain. Here, the theoretical model indicates that a more evenly spread processor grid with $P_1 \approx P_2$ could provide better performance results for small input sizes. When comparing the overall performance of Slab Decomposition and Pencil Decomposition we found similar results. In particular, Pencil Decomposition either dominates or provides competing performance in our benchmarks if the duration of one global redistribution phase is negligible. For us, this was achieved when Pencil Decomposition fully utilizes the NVLink interconnections in the first global redistribution phase on BW-GPU4 and BW-GPU8.

Finally, we conduct extensive performance measurements of the global redistribution methods on each test system. Using Peer2Peer-Sync as the baseline, we found multiple scenarios in which other methods perform significantly better (cf. Section 5.3.2): For the first observation, we found that Peer2Peer-Streams often outperforms the remaining methods when MPI without CUDA-awareness is considered for input sizes above a certain threshold (for us between 128×256^2 and 256^3). On BW-GPU4 and BW-GPU8 the performance increase, when using Peer2Peer-Streams, is often between 10% and 20%. For the second observation, our benchmarks show that All2All-Sync should be considered if many processes are involved in the particular global redistribution phase. The most notable difference was measured when utilizing 64 GPUs of BW-GPU8. Finally, for the third observation, we found that MPI_Type generally does not perform well if 1D-partitions are sent or received. Otherwise, Peer2Peer-MPI_Type is often able to provide competing performance and even outperforms the remaining global redistribution methods in some scenarios where indications of network congestion are determined.

As before, we emphasize that this work is by no means exhaustive: For example, it would be interesting to include the utilized decomposition method of OpenFFT [DO14] in the comparison of slab decomposition and pencil decomposition. To provide further insight, it would be also interesting to analyze the impact of the utilized MPI implementation on the performance of the global redistribution methods. There are also many additional optimizations for computing the distributed 3D-FFTs, that are not covered by this work: For example, further refinements for Peer2Peer communication could be considered that include additional information about the network topology. Another possible optimization could be to further subdivide the input partition, such that parts of the FFT computation can be performed on CPU, allowing for an additional overlap of process communication and computation.

Bibliography

[Adv] Advanced Micro Devices, Inc. (AMD. *Accelerating Performance: The ACL clFFT Library*. Last accessed on 10/16/2021. URL: https://developer.amd.com/accelerating-performance-acl-clfft-library/ (cit. on p. 15).

[ATHD20] A. Ayala, S. Tomov, A. Haidar, J. Dongarra. "heFFTe: Highly Efficient FFT for Exascale". In: *Computational Science – ICCS 2020*. Ed. by V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, J. Teixeira. Cham: Springer International Publishing, 2020, pp. 262–275. ISBN: 978-3-030-50371-0 (cit. on pp. 16, 17).

[ATL+21] A. Ayala, S. Tomov, P. Luszczek, S. Cayrols, G. Ragghianti, J. Dongarra. *Interim Report on Benchmarking FFT Libraries on High Performance Systems*. ICL Tech Report ICL-UT-21-03. 2021-07 2021 (cit. on pp. 16, 17).

[BHB+20] M. Brunn, N. Himthani, G. Biros, M. Mehl, A. Mang. "Multi-Node Multi-GPU Diffeomorphic Image Registration for Large-Scale Imaging Problems". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov. 2020). DOI: 10.1109/sc41405.2020.00042. URL: http://dx.doi.org/10.1109/SC41405.2020.00042 (cit. on p. 15).

[BK01] O. P. Bruno, L. A. Kunyansky. "A fast, high-order algorithm for the solution of surface scattering problems: basic implementation, tests, and applications". In: *Journal of Computational Physics* 169 (2001), pp. 80–110 (cit. on p. 15).

[Bra] E. Bray T. *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90, RFC 8259, December 2017. Last accessed on 09/26/2021. URL: https://www.rfc-editor.org/info/std90 (cit. on p. 74).

[bwHa] bwHPC. *BwUniCluster 2.0*. Last accessed on 09/24/2021. URL: https://wiki.bwhpc.de/e/Category:BwUniCluster_2.0 (cit. on p. 73).

[bwHb] bwHPC. *High Performance Computing, Data Intensive Computing and Large Scale Scientific Data Management in Baden-Württemberg*. Last accessed on 09/24/2021. URL: https://www.bwhpc.de/ (cit. on p. 73).

[CT65] J. Cooley, J. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301 (cit. on pp. 15, 22, 24).

[DMK19] L. Dalcin, M. Mortensen, D. E. Keyes. "Fast parallel multidimensional FFT using advanced MPI". In: *Journal of Parallel and Distributed Computing* 128 (2019), pp. 137–150. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.02.006. URL: https://www.sciencedirect.com/science/article/pii/S074373151830306X (cit. on pp. 16, 65, 95).

[DO14] T. V. T. Duy, T. Ozaki. “A decomposition method with minimum communication amount for parallelization of multi-dimensional FFTs”. In: *ArXiv* abs/1302.6189 (2014) (cit. on pp. 16, 17, 96).

[DS00] J. Dongarra, F. Sullivan. “Guest Editors Introduction to the top 10 algorithms”. In: *Computing in Science & Engineering* 2 (Feb. 2000), pp. 22–23. DOI: 10.1109/MCISE.2000.814652 (cit. on p. 15).

[FFT] FFTW. *Distributed-memory FFTW with MPI*. Last accessed on 10/16/2021. URL: https://www.fftw.org/fftw3_doc/MPI-Data-Distribution.html (cit. on p. 15).

[FJ05] M. Frigo, S. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231. DOI: 10.1109/JPROC.2004.840301 (cit. on pp. 15, 26, 40).

[GFB+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Proceedings, 11th European PVM/MPI Users’ Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104 (cit. on p. 42).

[GHMB16] A. Gholami, J. Hill, D. Malhotra, G. Biros. *AccFFT: A library for distributed-memory FFT on CPU and GPU architectures*. 2016. arXiv: 1506.07933 [cs.DC] (cit. on p. 16).

[HGTT10] T. Hoefler, W. Gropp, R. Thakur, J. L. Träff. “Toward Performance Models of MPI Implementations for Understanding Application Scaling Issues”. In: *Recent Advances in the Message Passing Interface*. Ed. by R. Keller, E. Gabriel, M. Resch, J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 21–30. ISBN: 978-3-642-15646-5 (cit. on p. 34).

[HJB84] M. Heideman, D. Johnson, C. Burrus. “Gauss and the history of the fast fourier transform”. In: *IEEE ASSP Magazine* 1.4 (1984), pp. 14–21. DOI: 10.1109/MASSP.1984.1162257 (cit. on pp. 15, 22).

[HMF+13] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, K. Heitmann. “HACC: Extreme scaling and performance across diverse architectures”. In: *SC ’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–10. DOI: 10.1145/2503210.2504566 (cit. on p. 15).

[Inta] Intel Corporation. *Intel® oneAPI Math Kernel Library*. Last accessed on 10/16/2021. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html (cit. on p. 15).

[Intb] International Business Machines Corportation (IBM). *IBM Engineering and Scientific Subroutine Library documentation*. Last accessed on 10/17/2021. URL: https://www.ibm.com/docs/en/essl (cit. on p. 15).

[Joh11] S. G. Johnson. “Notes on FFT-based differentiation”. In: MIT Applied Mathematics. 2011 (cit. on p. 35).

[Lei85] C. E. Leiserson. “Fat-trees: Universal networks for hardware-efficient supercomputing”. In: *IEEE Transactions on Computers* C-34.10 (1985), pp. 892–901. DOI: 10.1109/TC.1985.6312192 (cit. on p. 75).

[LL10] N. Li, S. Laizet. “2DECOMP&FFT - A Highly Scalable 2D Decomposition Library and FFT Interface”. In: 2010 (cit. on p. 16).

[LL11] S. Laizet, N. Li. “Incompact3d: A powerful tool to tackle turbulence problems with up to O(105) computational cores”. In: *International Journal for Numerical Methods in Fluids* 67.11 (2011), pp. 1735–1757. DOI: https://doi.org/10.1002/fld.2480. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.2480. URL: https://www.incompact3d.com/ (cit. on p. 15).

[MPI21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (cit. on pp. 42–45, 67).

[NVIa] NVIDIA Corporation. *CUDA C++ Programming Guide*. Last accessed on 09/02/2021. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide (cit. on p. 37).

[NVIb] NVIDIA Corporation. *CUDA Documentation: Device Management*. Last accessed on 09/02/2021. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html (cit. on pp. 37, 39).

[NVIc] NVIDIA Corporation. *CUDA Documentation: Execution Control*. Last accessed on 09/02/2021. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXECUTION.html (cit. on p. 39).

[NVId] NVIDIA Corporation. *CUDA Documentation: Memory Management*. Last accessed on 08/30/2021. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (cit. on pp. 37, 40, 51).

[NVIe] NVIDIA Corporation. *CUDA Documentation: Stream Management*. Last accessed on 09/02/2021. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html (cit. on pp. 37, 39).

[NVIf] NVIDIA Corporation. *CUDA Documentation: Synchronization Behaviour*. Last accessed on 09/02/2021. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html (cit. on p. 39).

[NVIg] NVIDIA Corporation. *CUDA Programming Guide: Streams*. Last accessed on 08/31/2021. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams (cit. on p. 39).

[NVIh] NVIDIA Corporation. *cuFFT Documentation*. Last accessed on 08/21/2021. URL: https://docs.nvidia.com/cuda/cufft (cit. on pp. 22, 40, 53).

[NVIi] NVIDIA Corporation. *cuFFT Documentation: Streams*. Last accessed on 08/31/2021. URL: https://docs.nvidia.com/cuda/cufft/index.html#streamed-cufft-transforms (cit. on p. 58).

[NVIj] NVIDIA Corporation. *cuFFT: Fast Fourier Transforms for NVIDIA GPU*. Last accessed on 08/20/2021. URL: https://developer.nvidia.com/cufft (cit. on pp. 15, 21, 40).

[NVIk] NVIDIA Corporation. *The NVIDIA CUDA Toolkit*. Last accessed on 08/28/2021. URL: https://developer.nvidia.com/cuda-toolkit (cit. on pp. 37, 40).

[Pek12] D. Pekurovsky. “P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions”. In: *SIAM Journal on Scientific Computing* 34.4 (Jan. 2012), pp. C192–C209. ISSN: 1095-7197. DOI: 10.1137/11082748x. URL: http://dx.doi.org/10.1137/11082748X (cit. on p. 15).

[PPS97] S. Plimpton, R. Pollock, M. Stevens. “Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations”. In: *In Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. 1997. URL: https://www.lammps.org (cit. on p. 15).

[RAC+18] D. F. Richards, O. Aziz, J. Cook, H. Finkel, B. Homerding, T. Judeman, P. McCorquodale, T. Mintz, S. Moore. “Quantitative Performance Assessment of Proxy Apps and Parents”. In: (Apr. 2018). DOI: 10.2172/1438753. URL: https://www.osti.gov/biblio/1438753 (cit. on p. 16).

[RKH10] K. R. Rao, D. N. Kim, J.-J. Hwang. *Fast Fourier Transform - Algorithms and Applications*. 1st. Chapter 3: Fast Algorithms. Springer Publishing Company, Incorporated, 2010. ISBN: 1402066287 (cit. on pp. 19, 22).

[Sch] SchedMD®. *SLURM*. Last accessed on 09/26/2021. URL: https://www.schedmd.com/ (cit. on p. 74).

[Sofa] Software in the Public Interest. *Open MPI: Open Source High Performance Computing*. Last accessed on 09/05/2021. URL: https://www-lb.open-mpi.org/ (cit. on pp. 42, 74).

[Sofb] Software in the Public Interest. *Open MPI: Protocol for Small Messages*. Last accessed on 10/02/2021. URL: https://www.open-mpi.org/faq/?category=openfabrics#ib-small-message-rdma (cit. on p. 86).

[Sofc] Software in the Public Interest. *Open MPI: Rankfiles*. Last accessed on 09/26/2021. URL: https://www.open-mpi.org/doc/v4.1/man1/mpirun.1.php#sect13 (cit. on p. 74).

[Ste] Steve Plimpton. *fftMPI, a distributed-memory parallel FFT library*. Last accessed on 10/17/2021. URL: http://fftmpi.sandia.gov (cit. on p. 16).

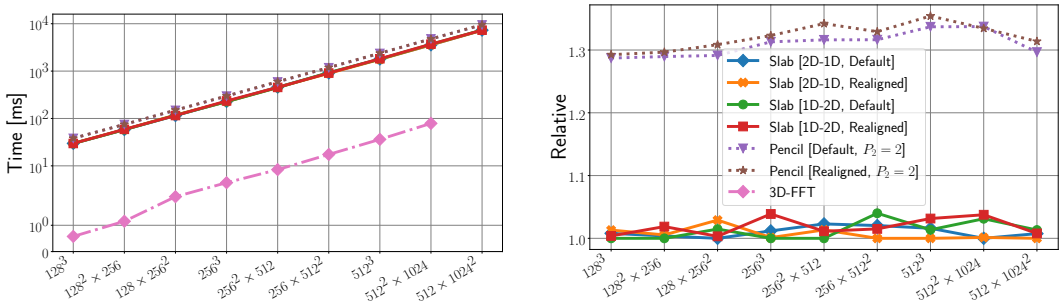
[Tak20] D. Takahashi. “Implementation of Parallel 3-D Real FFT with 2-D Decomposition on Intel Xeon Phi Clusters”. In: Mar. 2020, pp. 151–161. ISBN: 978-3-030-43228-7. DOI: 10.1007/978-3-030-43229-4_14 (cit. on pp. 16, 34).

[VKG14] M. Vetterli, J. Kovačević, V. K. Goyal. *Foundations of Signal Processing*. Cambridge University Press, 2014. DOI: 10.1017/CBO9781139839099 (cit. on p. 19).

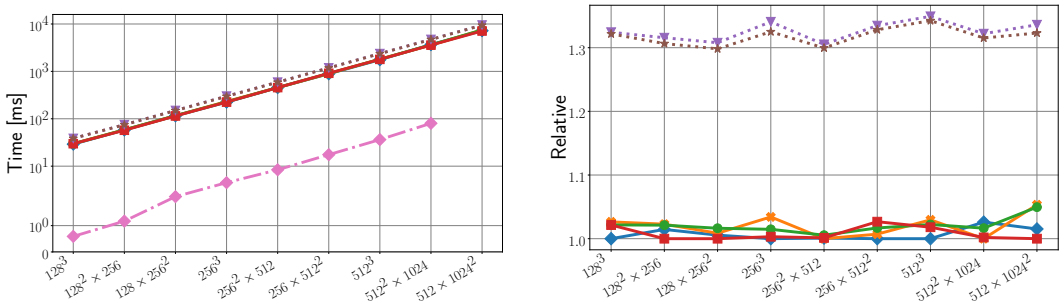
[WSB+06] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, A. B. Maccabe. “High Performance RDMA Protocols in HPC”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by B. Mohr, J. L. Träff, J. Worringen, J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 76–85. ISBN: 978-3-540-39112-8 (cit. on p. 86).

A Appendix

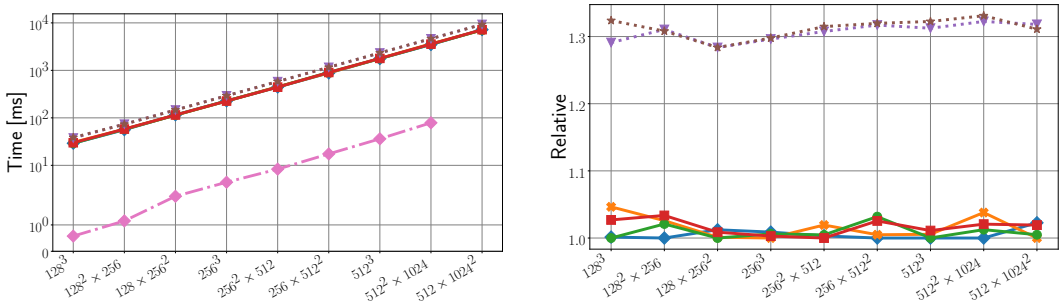
PCSGS: Decomposition Methods. Figure A.1 shows the comparison between the different decomposition methods when using MPI without CUDA-awareness.



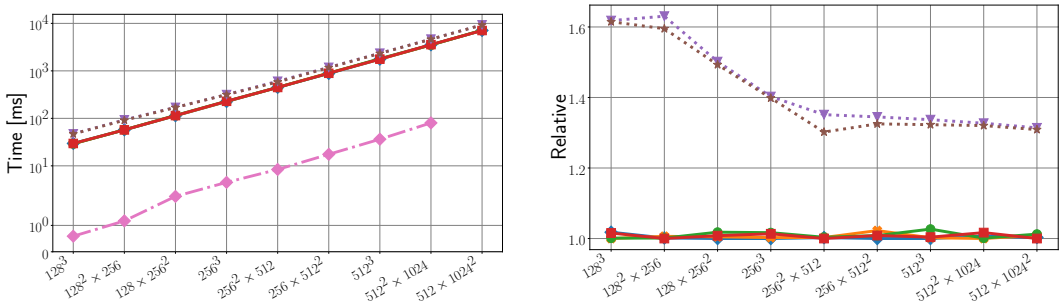
(a) *Forward Transforms for MPI without CUDA-Awareness.*



(b) *Inverse Transforms for MPI without CUDA-Awareness.*



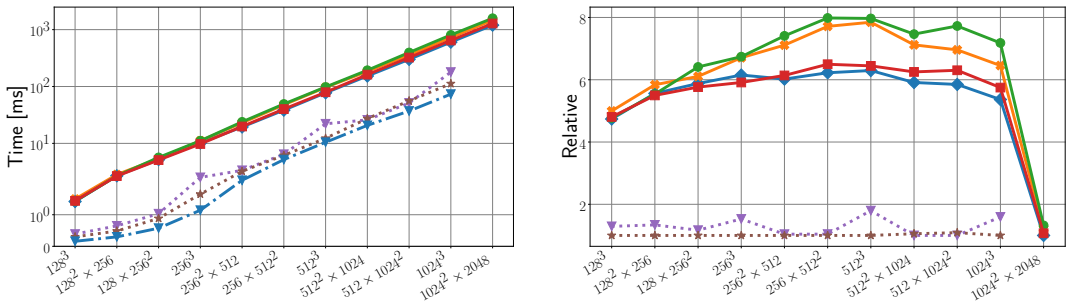
(c) *Forward Transforms for CUDA-Aware MPI.*



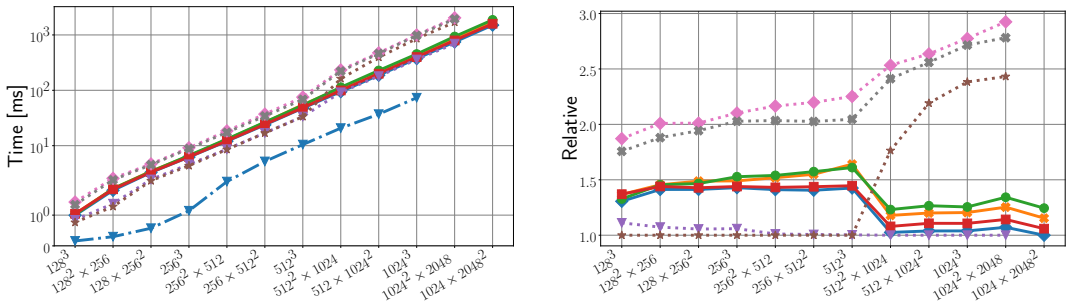
(d) *Inverse Transforms for CUDA-Aware MPI.*

Figure A.1: *PCSGS: Total and Relative Runtime of the Decomposition Methods.* Figures **(a)**-**(d)** show the results for the forward and inverse transforms, with and without CUDA-aware MPI. For the runtime, *Reference* is used to denote the performance of computing the full 3D-FFT on a single GPU without a distributed approach. The graphs on the right side visualize the relative runtime results, where the runtime of each variant and each input size is divided by the minimal result of the given input size (without considering *Reference*).

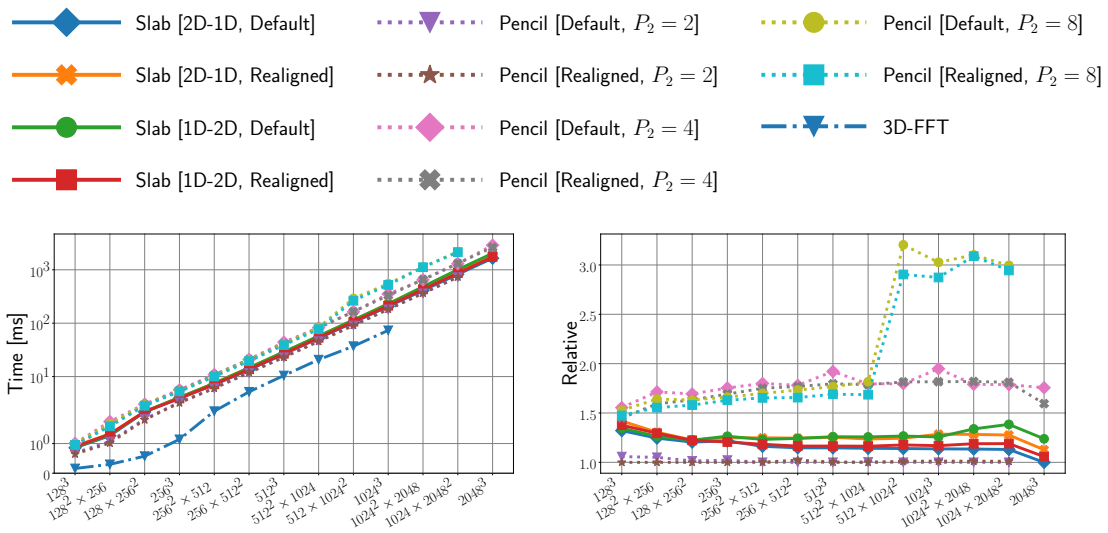
BW-GPU4: Decomposition Methods. Figure A.2 shows the comparison between the different decomposition methods for the inverse transform. The results are grouped by the number of utilized GPUs, and each benchmark is performed with CUDA-aware MPI. We note that the results are incomplete for pencil decomposition with a processor grid of 12×2.



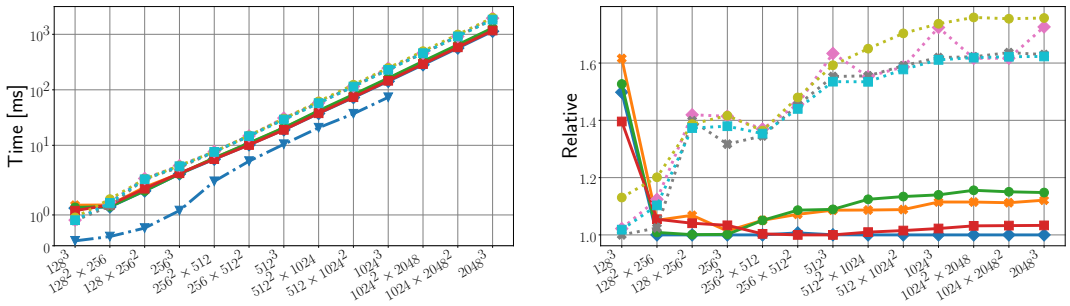
(a) *Total and Relative Results for* $P = 4$.



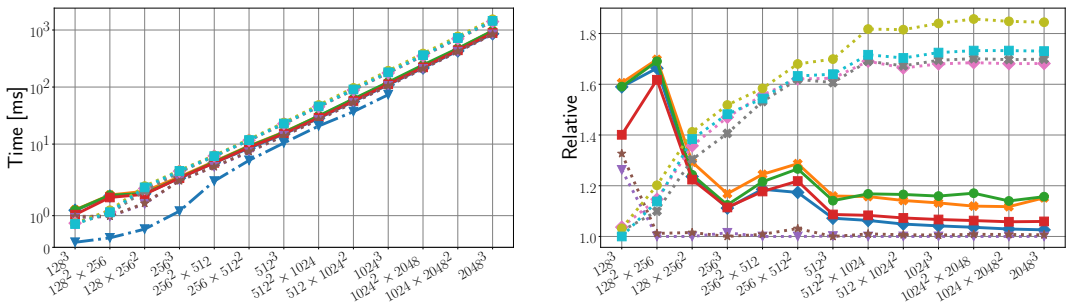
(b) *Total and Relative Results for* $P = 8$.



(c) *Total and Relative Results for* $P = 16$.



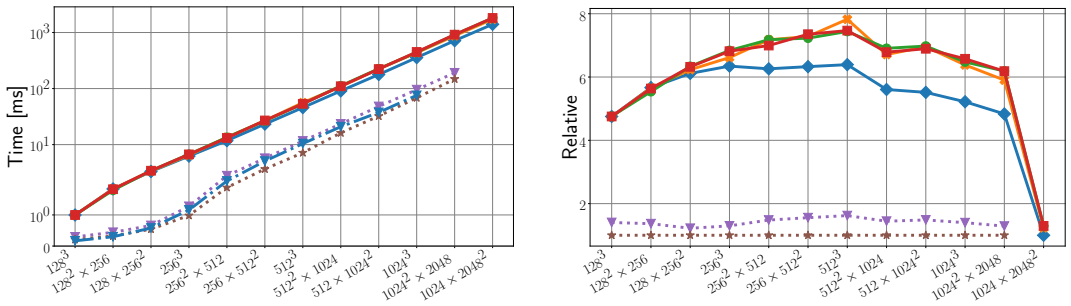
(d) *Total and Relative Results for* $P = 24$.



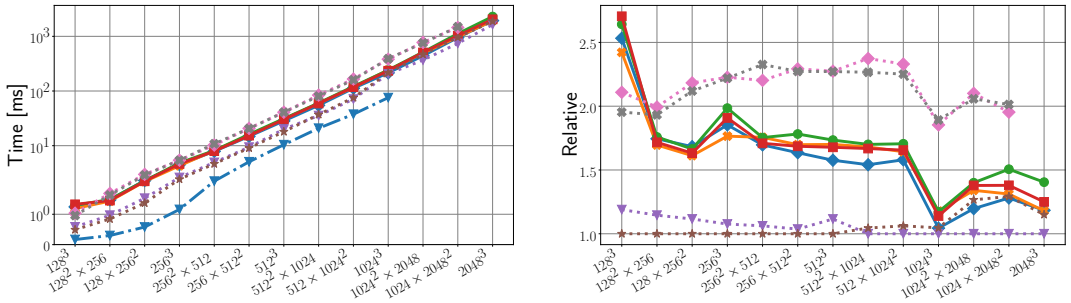
(e) *Total and Relative Results for* $P = 32$.

Figure A.2: *BW-GPU4 Inverse: Total and Relative Runtime of the Decomposition Methods.* Figures **(a)**-**(e)** show the results for the 4, 8, 16, 24, and 32 GPUs. Solely the inverse transform with CUDA-aware MPI is considered. For the runtime, *Reference* is used to denote the performance of computing the full 3D-FFT on a single GPU without a distributed approach. The graphs on the right side visualize the relative runtime results, where the result of each variant and each input size is divided by the minimal result for the given input size (without considering *Reference*).

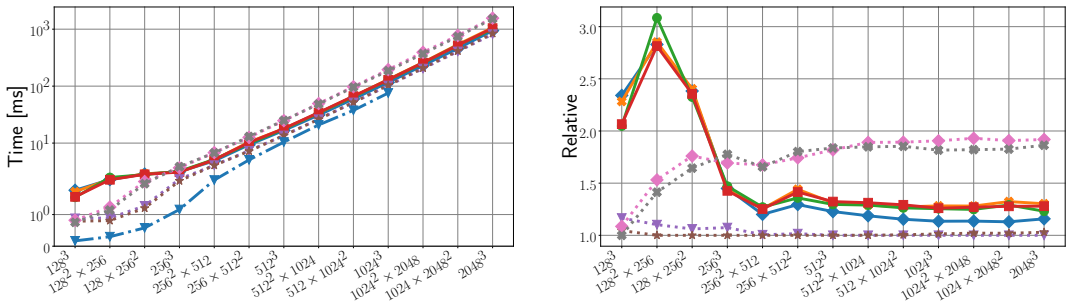
BW-GPU8: Decomposition Methods. Figure A.3 and Figure A.4 show the comparison between the different decomposition methods for the forward and inverse transform respectively. The results are grouped by the number of utilized GPUs, and each benchmark is performed with CUDA-aware MPI.



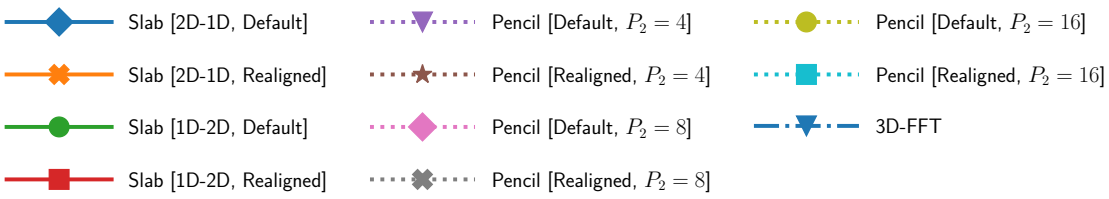
(a) *Total and Relative Results for* $P = 8$.

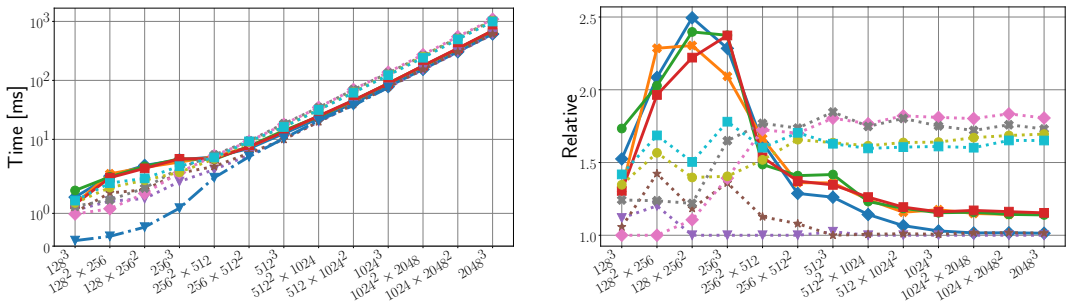


(b) *Total and Relative Results for* $P = 16$.

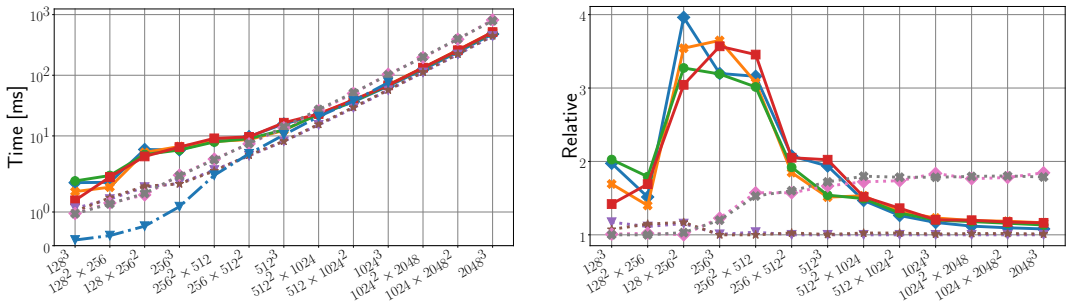


(c) *Total and Relative Results for* $P = 32$.



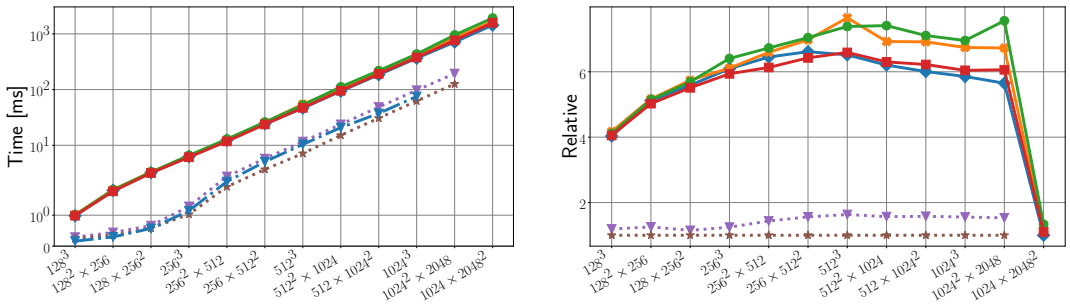


(d) *Total and Relative Results for* $P = 48$.

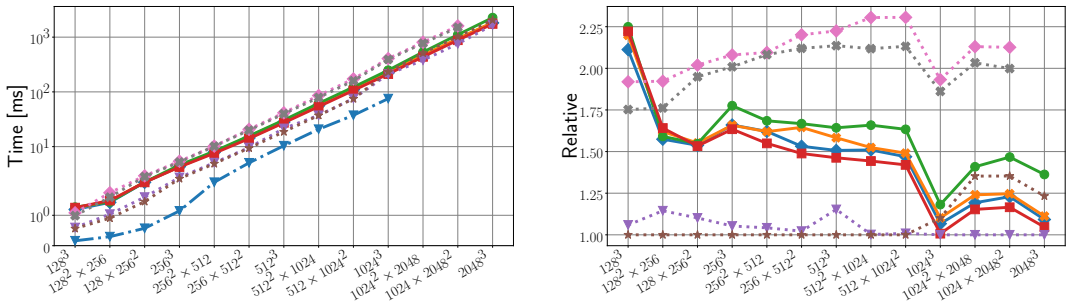


(e) *Total and Relative Results for* $P = 64$.

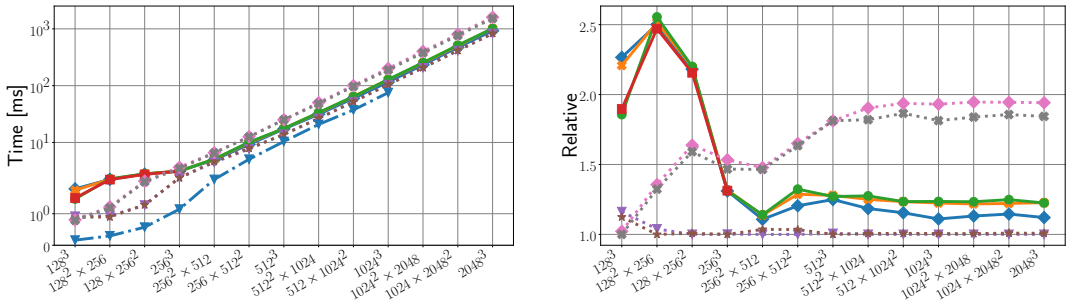
Figure A.3: *BW-GPU8 Forward: Total and Relative Runtime of the Decomposition Methods.* Figures **(a)**-**(e)** show the results for the 8, 16, 32, 48, and 64 GPUs. Solely the forward transform with CUDA-aware MPI is considered. For the runtime, *Reference* is used to denote the performance of computing the full 3D-FFT on a single GPU without a distributed approach. The graphs on the right side visualize the relative runtime results, where the result of each variant and each input size is divided by the minimal result for the given input size (without considering *Reference*).



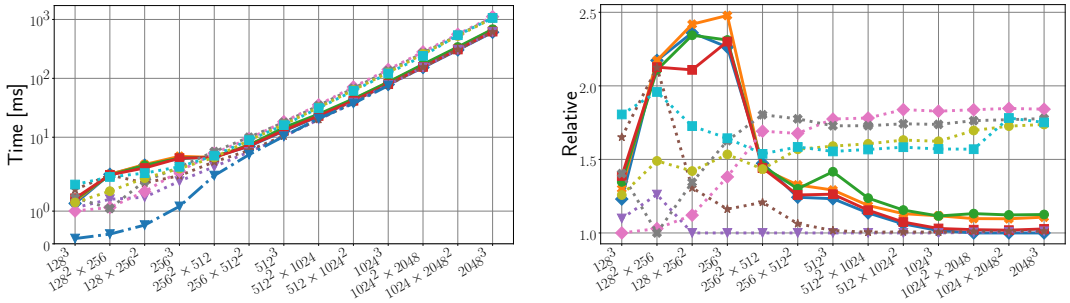
(a) *Total and Relative Results for* $P = 8$.



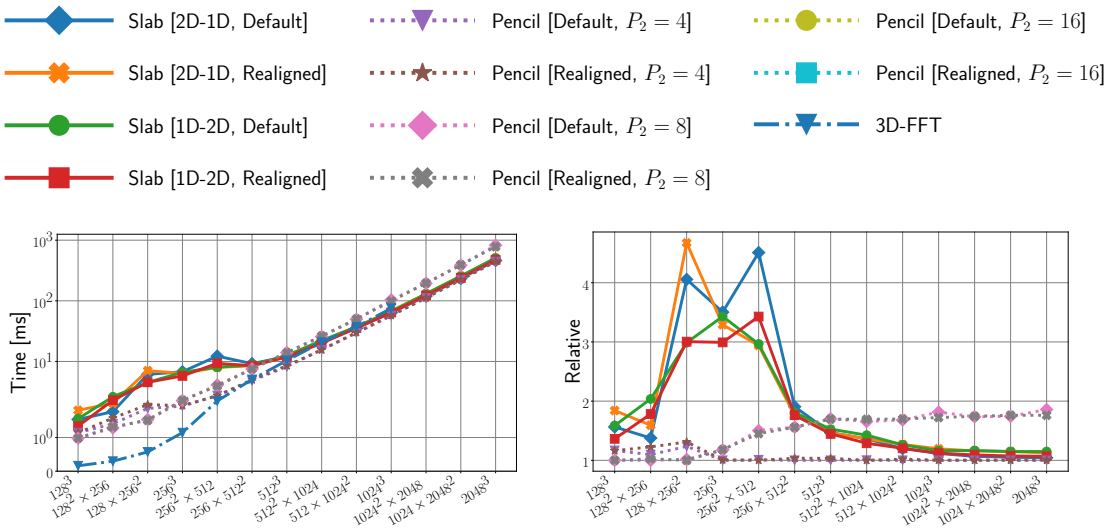
(b) *Total and Relative Results for* $P = 16$.



(c) *Total and Relative Results for* $P = 32$.



(d) *Total and Relative Results for* $P = 48$.



(e) *Total and Relative Results for* $P = 64$.

Figure A.4: *BW-GPU8 Inverse: Total and Relative Runtime of the Decomposition Methods.* Figures **(a)**-**(e)** show the results for the 8, 16, 32, 48, and 64 GPUs. Solely the inverse transform with CUDA-aware MPI is considered. For the runtime, *Reference* is used to denote the performance of computing the full 3D-FFT on a single GPU without a distributed approach. The graphs on the right side visualize the relative runtime results, where the result of each variant and each input size is divided by the minimal result for the given input size (without considering *Reference*).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature