A Parallel Implementation of the Fast Fourier Transform Algorithm

Advanced Methods for Scientific Computing

Carrà Edoardo
edoardo.carra@mail.polimi.it

Gentile Lorenzo
lorenzo3.gentile@mail.polimi.it

Ferrario Daniele
daniele6.ferrario@mail.polimi.it

1 Introduction

The Fast Fourier Transform (FFT) is a powerful tool used in various fields, from pure mathematics to audio engineering and even finance. It's a method for expressing a function as a sum of periodic components, and for recovering the signal from those components. When both the function and its Fourier transform are replaced with discretized counterparts, it is called the discrete Fourier transform (DFT).

In this project, we explore various implementations for calculating the FFT, based on well-known algorithms such as Cooley-Tukey and Stockham. The project extends to parallelization techniques, covering not only the traditional one-dimensional FFT but also the two and three-dimensionsal case. This report contains:

1. *A brief analysis of the Cooley-Tukey and Stockham algorithms for computing the FFT.*
2. *How to parallelize Cooley-Tukey algorithm: partitioning, analysis of the communication, aggregation and mapping.*
3. *Multidimensional FFT.*
4. *Project code structure and project design.*
5. *Some results on the numerical tests obtained.*

The project structure is designed as a scientific library, drawing inspiration from well-known C++ libraries design like the one of Eigen and DealII. This approach facilitates the effective application of design principles, enabling from the beginning the construction of a robust and flexible project. It also provides an opportunity to explore various techniques for computing the FFT, while keeping the code structure as simple as possible. To obtain this generality the code strongly exploit the new features of C++, such as smart pointers and templates for implementing the strategy pattern and encapsulation to wrap other libraries features.

This philosophy takes inspiration by the famous FFTW3 library for computing the FFT: instead of focusing on finding the best implementation for computing the FFT, the idea is to build the smallest collection of simple algorithmic fragments whose composition spans the optimal algorithm on as many computer architectures as possible. [4]

***Please note**: this report does not contain the code documentation. You can find it at the official repository of the project: `https://github.com/AMSC22-23/FFT-Carra-Ferrario-Gentile`.*

2 The Discrete Fourier Transform

Given a finite sequence of N equally-spaced complex numbers, the *Discrete Fourier Transform (DFT)* computes a corresponding same-length sequence of complex numbers, revealing the signal's frequency components. Mathematically, the DFT of a sequence x_n is given by:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{2\pi i}{N} kn}$$

where X_k represents the frequency component at index k and i is the imaginary unit. The inverse formula, formally named *Inverse Discrete Fourier Transform (IDFT)*, is given by:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{\frac{2\pi i}{N} kn}$$

If we apply the definition, the time complexity of both the DFT and IDFT is $O(N^2)$. In order to reduce this complexity, the DFT computation can be recursively breaking down into smaller subproblems. This approach is known as Fast Fourier Transform (FFT), and will be discussed extensively in the following sections.

2.1 Multidimensional DFT

Given an array $x_(n_1, n_2, ..., n_d)$ with a d-dimensional vector of indices, the multidimensional DFT is composed by d nested summations. It reads:

$$X_{K_1,..,K_d} = \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} x_{n_1,...,n_d} \cdot e^{-\frac{2\pi i}{N_1} n_1 K_1 - ... - \frac{2\pi i}{N_d} n_d K_d}$$

where $X_()$ is an equally sized d-dimensional vector which represents the frequency component at index k and i is the imaginary unit. Analogously the multidimensional IDFT reads:

$$x_{n_1,...,n_d} = \frac{1}{N_1 \cdot N_2 \cdot ... N_d} \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} X_{K_1,..,K_d} \cdot e^{\frac{2\pi i}{N_1} n_1 K_1 + ... + \frac{2\pi i}{N_d} n_d K_d}$$

Fromt the definition, the multidimensional DFT requires to compute the one-dimensional discrete transform along each direction of the tensor. For instance, in the two dimensional case, the first stage of the algorithm requires to transform the rows, followed by the transformation of the columns. This algorithm is known as the **row-column algorithm**. In this way the tensor is transformed along n_1 dimension and then n_2 and so on. This can be easily generalized to higher dimension, computing the one dimensional transform along each dimension.

Henceforth, it is assumed that all data dimensions are a **power of two** for FFT computation. Although various methods, such as **padding**, exist for applying the FFT to datasets that are not a power of two, these techniques are not explored in this project.

3 Cooley-Tukey algorithm

Algorithm 1 Cooley-Tukey Algorithm [2]

```
1: function FFT(a)
2:     bit_reverse(a)
3:     n ← length(a) for s = 1 to log(n) do
4:     m ← 2^s
5:     ω_m ← exp(−2πi/m) for k = 0 to n − 1 by m do
6:         ω ← 1 for j = 0 to m/2 − 1 do
7:             t ← ω · a[k + j + m/2]
8:             u ← a[k + j]
9:             a[k + j] ← u + t
10:            a[k + j + m/2] ← u − t
11:            ω ← ω · ω_m
               end
           end
           end
12:    return a  =0
```

The Cooley-Tukey algorithm has a time complexity of $O(NlogN)$. This is because the algorithm recursively breaks down a DFT of any composite size into many smaller DFTs, and this process is repeated log N times. At each stage, the algorithm performs N multiplications, leading to a total of N log N multiplications. Note that, in the beginning of the algorithm it is required to perform a bit-reversal operation which requires linear time to be performed. However, this is dominated by the $O(NlogN)$ time complexity of the FFT part of the algorithm.

A simple example of data access pattern of the Cooley-Tukey algorithm is:

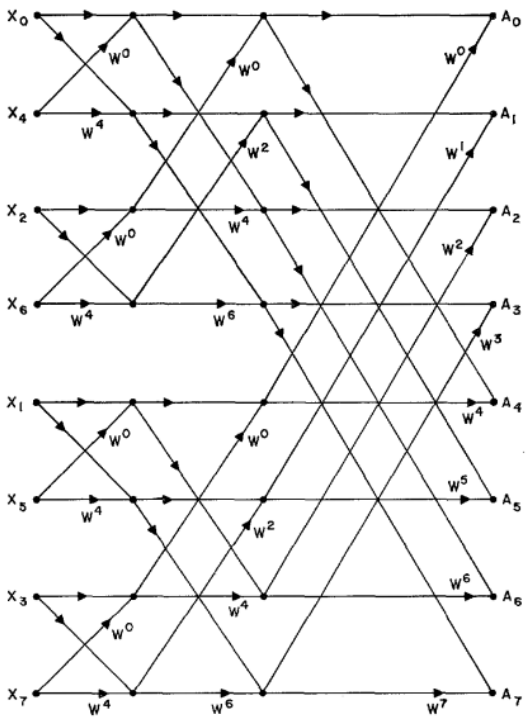


Figure 1: Cooley-Tukey algorithm on a size 8 collection [3] (after bit reversal)

3.1 Analysis of Cooley-Tukey algorithm

In the following section the problem: how to parallelize the Cooley-Tukey algorithm is addressed. In order to obtain a parallel version of the procedure, the following steps are necessary:

1. *Partitioning*: dependencies between instructions and access data pattern are identified and divided into tasks.
2. *Communication*: identify the amount of communication needed between the tasks of the previous step.
3. *Aggregation*: partition the dependency graph into multiple tasks that depend on each other. These tasks are aggregated into a unique block.
4. *Mapping*: map the previous blocks to processing elements.

While doing these operations, the objective is to reduce communication and synchronization to zero. Obviously, there is not a unique way to produce the parallel version of the sequential program. Here it is reported the parallelization addressed in the project for OMP and MPI technologies.

3.1.1 Partitioning and aggregation

By examining the Cooley-Tukey algorithm, we can construct a dependency graph to identify potential task partitions for parallelization.

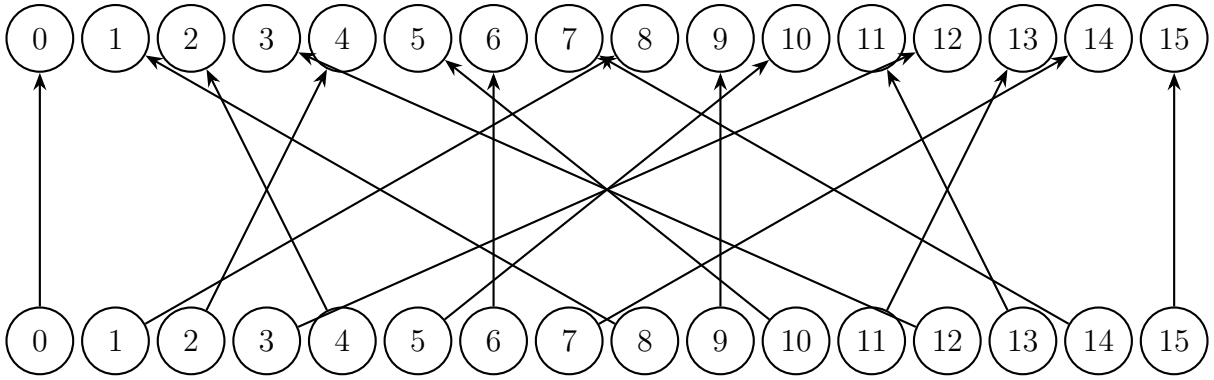


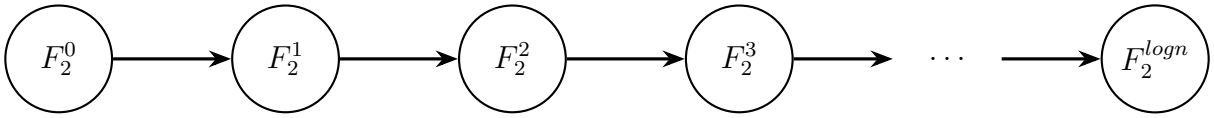
Figure 2: Example of bit reversal access with n=16

The initial stage of the algorithm involves reordering the data through a bit-reversal procedure. This procedure can be viewed as a mapping with exclusive read and write access to the vector. Please note that this part of the execution is highly parallelizable, but it's not cache-friendly due to its non-contiguous memory access pattern.

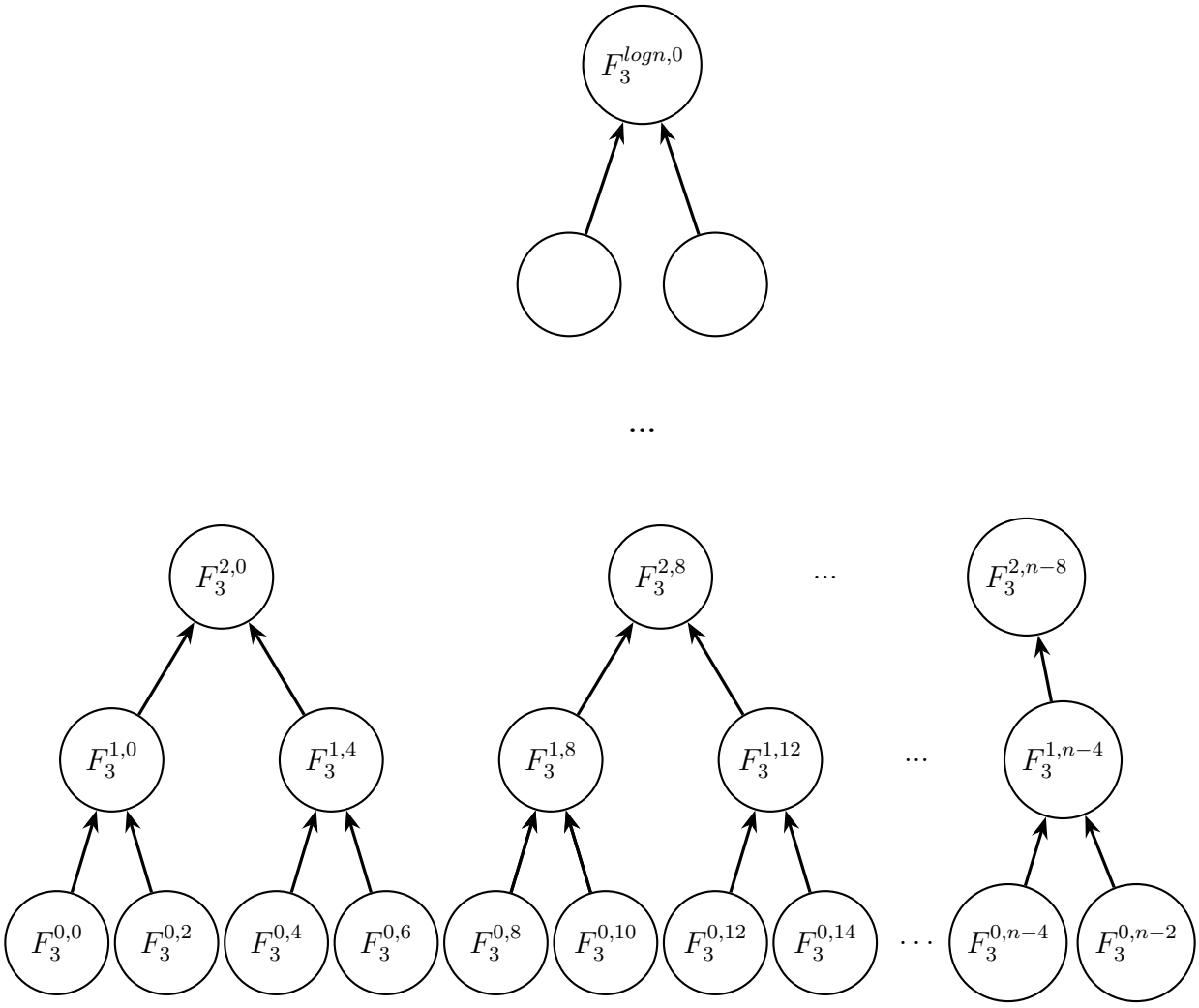
After the bit reversal part, the algorithm is composed of three nested for-loops. To address them in the following paragraph, the following notation will be used:

1. F_1: the outer loop that executes $log(n)$ iterations. It's responsible for the main iteration over the data set and each iteration is called **stage**.
2. F_2^s: the first loop nested inside F_1 at the iteration s. It executes $n/2^s$ iterations, with k that is increased by 2^s at each iteration. This loop creates partitions of the input vector.
3. $F_2^{s,k}$: the inner loop at the iteration k of the loop F_2^s. It executes 2^{s-1} iterations. This loop performs the actual computation on each partition with a **butterfly** access to the input vector.

Upon initial inspection, it appears that a *lexically forward flow dependence* exists between each F_2^s loop in the Cooley-Tukey algorithm. This means that the execution of each F_2^s loop depends on the completion of the previous F2s loop.
To address this flow dependence in the algorithm, we can construct a dependency graph:



At each stage, $F_3^{s,k}$ are all independent. Furthermore, when we examine the dependency graph of these operations, we discover that a partition of the dependencies exists between each stage. This means that the tasks can be grouped in a way that allows for parallel execution until a certain stage is reached:



The dependency graph follows a tree structure, where each $F_3^{s,k}$ solely depends on the subtree it generates. The depth of the subtree depends on the stage, for instance $F_3^{2,0}$ depends on $F_3^{1,0}$, $F_3^{1,4}$ which depends respectively $F_3^{0,0}$ and $F_3^{0,2}$, $F_3^{0,4}$ and $F_3^{0,6}$.

Each $F_3^{s,k}$ performs a butterfly access of a partition of the vector, of length 2^s, as it shown in the following example:

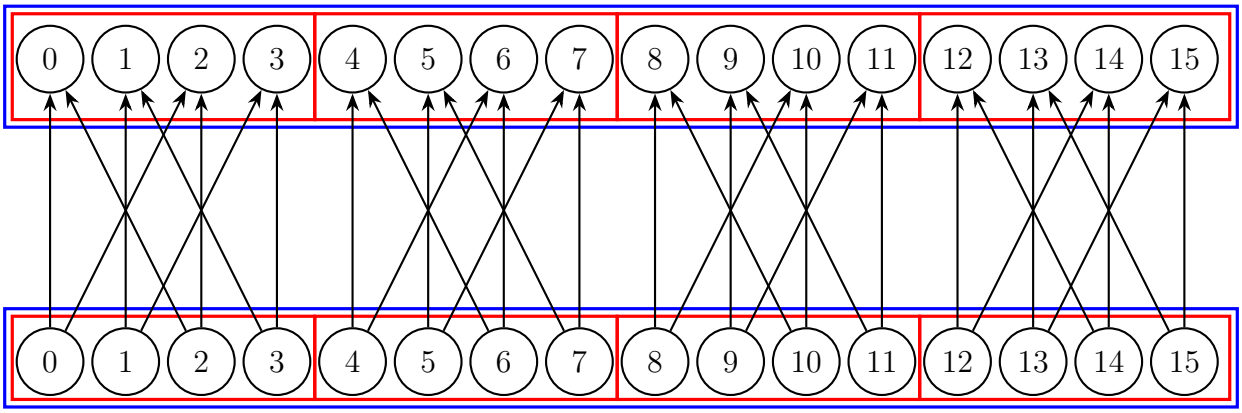


Figure 3: Example with n=16 of butterfly access at the second stage of the Cooley-Tukey algorithm

These partitions of length 2^s, where s is the stage, is the key of the parallelization carried out with the use of OpenMP and MPI. In fact, at each stage all the partitions can be resolved independently computing the same algorithm recursively. For OMP, this data partitioning prevents also *false-sharing*.

The former analysis reveals that within the same stage, there is no **communication** between each $F_3^{s,k}$. Consequently, the communication graph closely reflects the structure of the dependency graph. Specifically, at any given stage, each element relies on the entirety of data within its associated subtree.

3.1.2 Aggregation and Mapping

To minimize communication overhead, the decision was made to aggregate distinct subtrees. Consequently, tasks remain independent until the stage $log(n) - log(num_{task})$ is reached. Beyond this point, the number of tasks is successively halved at each stage until the $log(n)$ stage is attained.

4 Stockham algorithm

Algorithm 2 Stockham Algorithm [3]

```
1: function FFT(a)  for s = 1 to log_2(n) do
2:     m ← 2^s
3:     r ← n/m
4:     for k = 0 to m/2 − 1 do
5:         ω ← exp(−2 · π · k/m)
6:         for j = 0 to r − 1 do
7:             u ← a[k · 2 ∗ r + j]
8:             t ← ω · a[k · 2 ∗ r + j + r]
9:             buffer[k · r + j] ← u + t
10:            buffer[k · r + n/2 + j] ← u − t
           end
       end
11:    Swap a and buffer
       end
12:    return a  =0
```

The Stockham algorithm is obtained by reordering the butterflies at each stage, in a way such that the input sequence does not need to be in bit reversed order. This is often an advantage in practice, since the bit reversal operation is not cache-friendly, but it comes at the costs of a more complex indexing scheme and memory usage. In fact, the Stockham algorithm is not an in-place algorithm, but it requires a buffer of size n to store the intermediate results, so the space complexity is $O(2n)$. This algorithm is a good choice for a GPU implementation since it avoids the bit reversal step, which can be a bottleneck in the computation.
A simple example of data access pattern of the Stockham algorithm is the following:

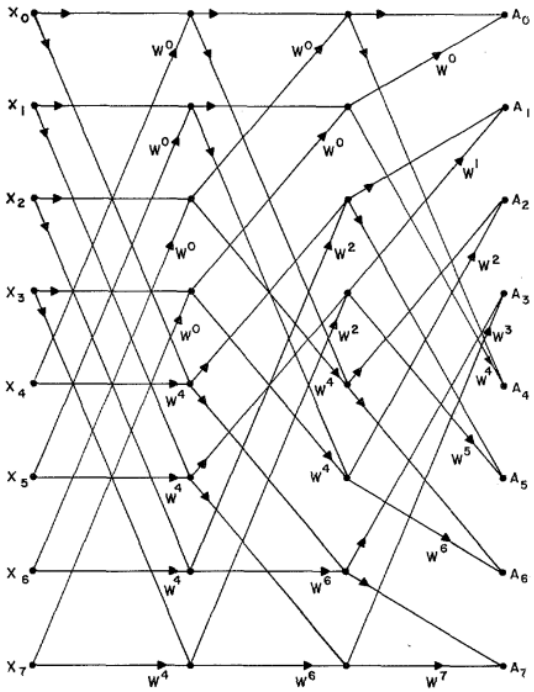


Figure 4: Stockham algorithm on a size 8 collection [3]

5 Multidimensional FFT

The multidimensional FFT requires to compute the one dimensional discrete transform along each direction of the tensor. The most trivial and surely not the most efficient strategy, is the row-column algorithm: first you transform along the first dimension, then along the second one, and so on.[5] Even if not efficient, the row-column algorithm can be extended without effort.

The data-dependency of the tensor follows an **extended stencil parallel pattern**: each tensor element depends on all the elements along the same line in one dimension. For example, in the two-dimensional case, every element depends on all the elements within the same column and row. Furthermore, only one dimension can be processed at a time, without any specific order restriction.

A numerical example in the case of a two-dimensional dataset is the following:

$$\begin{pmatrix} 2 & -3 & 1 & -2 & 4 & 0 & -5 & 0 \\ -1 & 1 & -3 & -4 & 0 & 5 & -2 & 0 \\ -2 & -3 & 2 & 0 & 1 & -1 & 3 & 2 \\ 1 & 0 & 3 & 2 & -1 & 4 & -2 & 0 \\ 0 & -2 & -1 & -1 & 3 & -2 & 0 & 0 \\ 4 & -1 & -4 & 0 & -3 & 1 & -1 & 6 \\ -3 & 2 & 1 & -8 & 2 & 2 & 0 & 4 \\ -3 & 2 & 0 & -1 & 2 & 3 & 0 & 0 \end{pmatrix}$$

For example, the element -3 at position (2,3) depends on all the elements in the second row and the third column.

This data-dependency does not present any problem in a **shared memory model**, but in a **message passing model**, the amounts of communication can become significant.

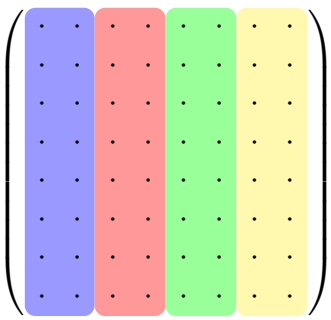
5.1 Two-dimensional FFT

In the library, two distinct parallelization approaches are implemented:

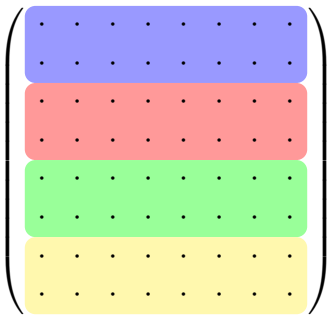
1. The first strategy parallelizes the one-dimensional FFT using the approach discussed in previous sections.

2. The second strategy exploits the independence of each one-dimensional FFT along a dimension, allowing for concurrent execution.

As an example of the second approach, consider a two-dimensional dataset with 8 rows and 8 columns. In this case, each of the 4 threads/processes is responsible for a specific partition of the columns, allowing for independent transformations of the columns.



Only after all the columns are transformed, the rows are partitioned analogously, and then transformed:



The second approach becomes non-trivial when MPI is employed, as it necessitates a substantial volume of communication during each dimension transformation. In this scenario, each process must partition the transformed data and send it to the corresponding process, resulting in significant communication overhead. However, this approach facilitates working with massive datasets since the work is split among multiple nodes.

The library implements both approaches: certain MPI implementations distribute the lines among the nodes and exchange partial results using **MPI_Alltoallv** as described previously, while others parallelize the one-dimensional FFT.

5.2 Three-dimensional FFT

The approach for the three-dimensional FFT is analogous to the two-dimensional case: first, the dataset is transformed along the first dimension, then along the second dimension, and finally along the third dimension. Similar to the two-dimensional case, every one-dimensional FFT is independent of the others.

6 Implementation

In this section will be discussed the project architectural features and design. To maintain clarity and conciseness, we have omitted some details from this report. For a more comprehensive understanding, look at the Doxygen documentation, which provides complete information on class structures and method functionalities.

6.1 Overview

The project was set up as a scientific library, taking inspiration from other famous library such as DealII, FFTW and Eigen. The project is composed by two main components:

1. The **FFFT** library itself. It is made up of 2 modules: the **fftcore** module and the **Spectrogram** module. The former contains a collection of various FFT algorithm implementations, while the latter enables the creation of a visual representation of the spectrum of frequencies in a signal as it varies over time.

   The library depends on two modules: **Eigen v3.3.9** and **fftw3**. The latter is utilized primarily for testing the correctness and performance of the different implementations.

2. **Zazam**: a music identification application based on FFFT Spectrogram module. It provides an example on how to use the library.

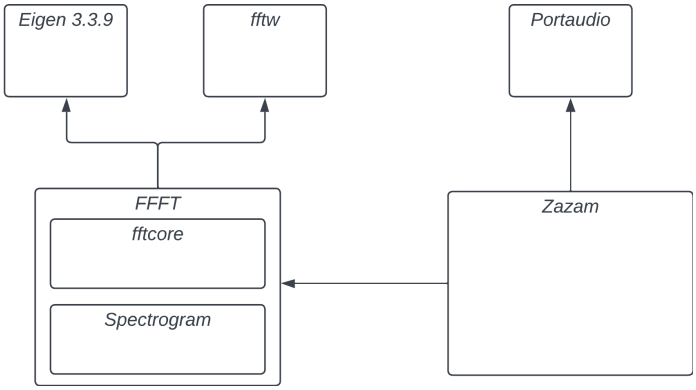


Figure 5: Package diagram

6.2 Design

The library was set up with a focus on software engineering principles, aiming to find a good balance between performance, abstraction and usability. The main philosophy behind the design was the **zero-overhead abstraction** principle. Citing Bjarne Stroustrup, the inventor of C++: "what you don't use, you don't pay for" and "what you use can be implemented without overhead compared with hand coding" (*Bjarne Stroustrup, The Design of C++0x*).

The library is predominantly a template-only library, making extensive use of various C++ features. Each FFT implementation is designed to be generic, supporting both single and double precision for optimization purposes. Moreover, each implementation has the flexibility to extend the dimension marker classes (FFT1D, FFT2D, and FFT3D) it plans to realize. This not only facilitates optimization based on data type but also provides the customization of optimizations according to the dimension.

6.2.1 Data structure

Among all the possible choices that were available for representing the data structure, it was decided to employ Eigen Tensors for representing the multidimensional input and output of the FFT. Compared

to std::vector, Eigen Tensors already handling multidimensional cases, and their performance is often comparable, if not superior in some cases. The adoption of a unified data structure for all the dimensions, significantly simplify the overall design of the library.

6.2.2 Design pattern

Two main design patterns were used:[1]

1. **Strategy Pattern**: This allows the user of the library to choose between different parallel and sequential implementations. This enabled us to divide the work independently, aiming to produce a sequential version, and try different parallel technologie, such as CUDA, OMP and MPI.

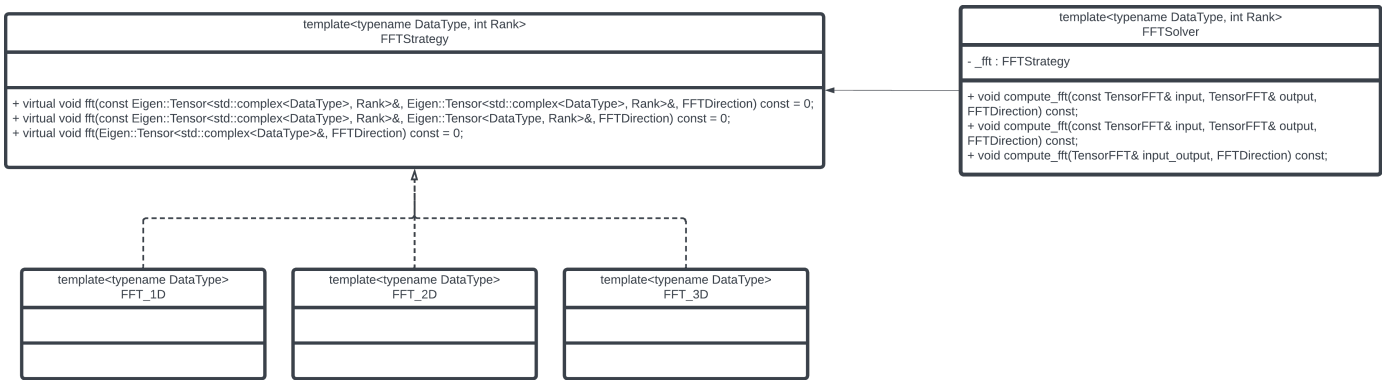


Figure 6: Strategy pattern

Note that FFTStrategy is intentionally kept as simple as possible to avoid a **fat interface**, preventing the imposition on concrete classes to implement methods that they do not use.

2. **Facade Pattern**: this pattern encapsulates the use of Eigen tensors, providing flexibility for the application to define custom cleaning and manipulation methods for input/output. Following the open/closed design principle, modifications specific to each application are facilitated through the extension of the TensorFFTBase class. This ensures that adjustments can be made without directly modifying the base class, promoting a modular and maintainable codebase.

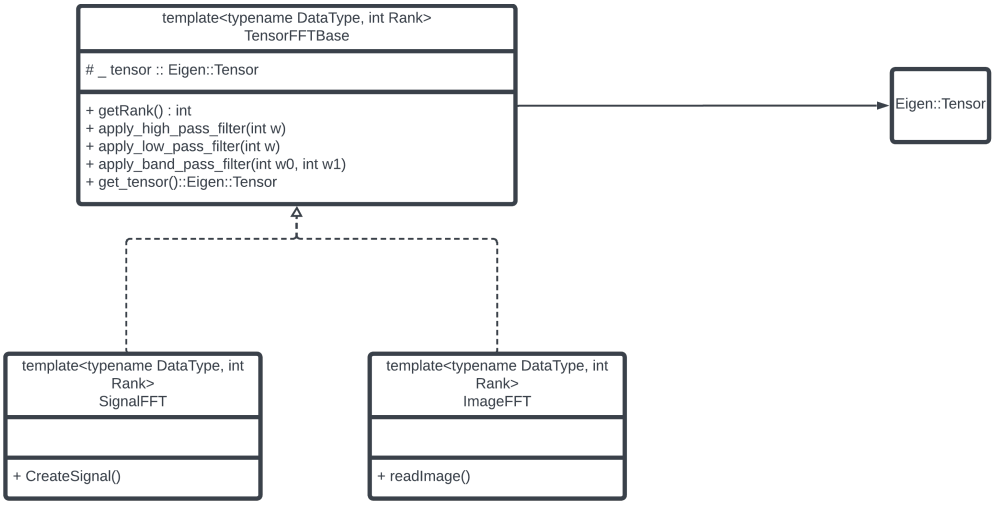


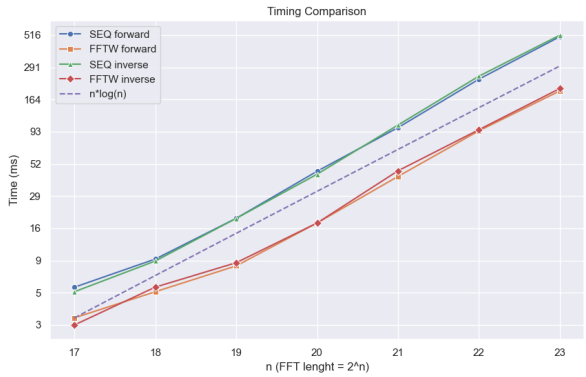
Figure 7: Facade pattern

7 Tests

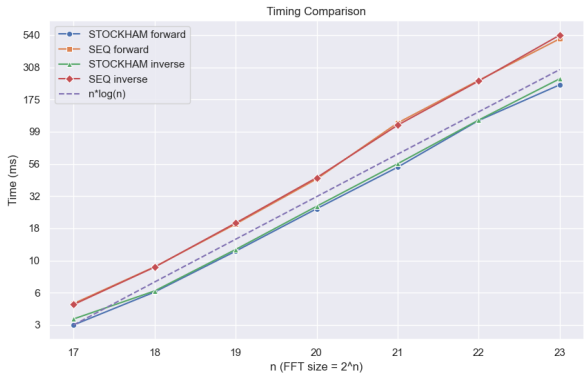
In order to test the library, multiple tests were built in order to assess the performance of a specified strategy against a baseline strategy. The evaluation is conducted on a dataset of complex numbers of size 2^n, populated with random values. The strategies are evaluated in both the forward and inverse directions.

The tests were conducted on a machine with an **Intel i5 7600k** with **16GB** of RAM. Here we report only the most significant results. To run your own tests, please refer to the repository.

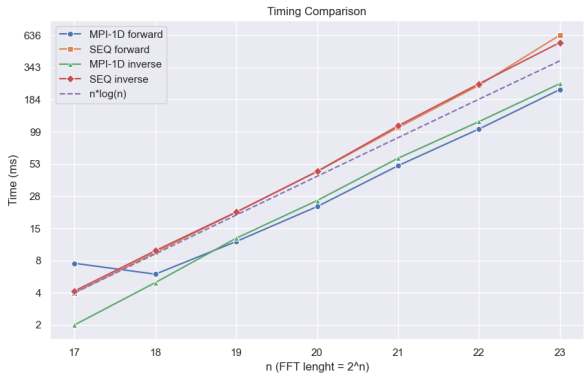
Just for semplicity, CT stands for sequential iterative Cooley-Tuckey implementation.



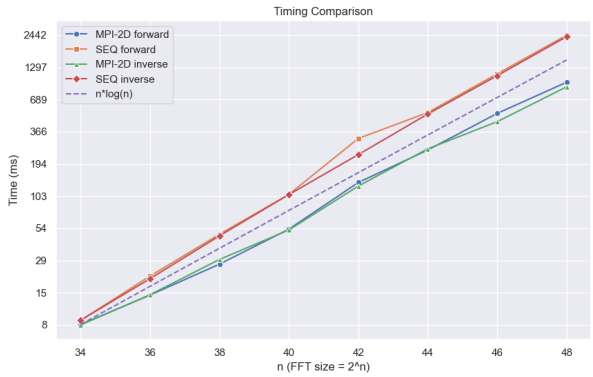
(a) CT 1D against FFTW



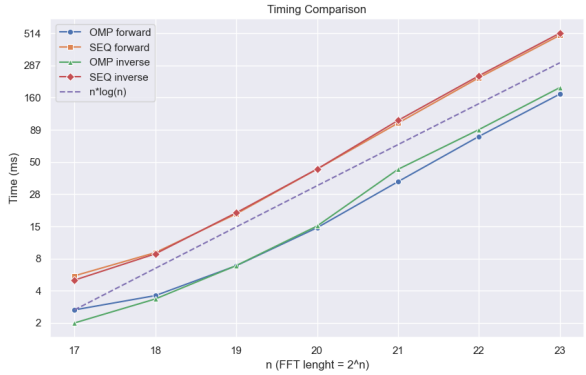
(b) Stockham 1D against CT 1D



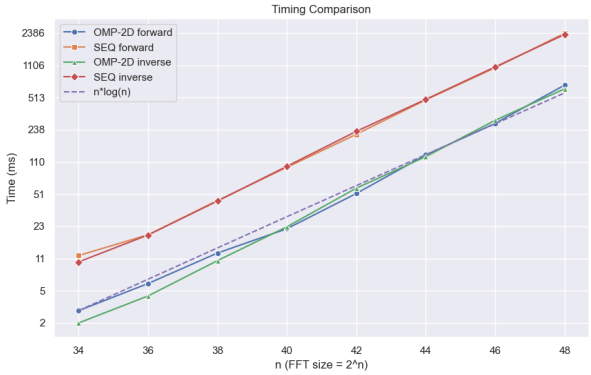
(a) MPI 1D against CT 1D



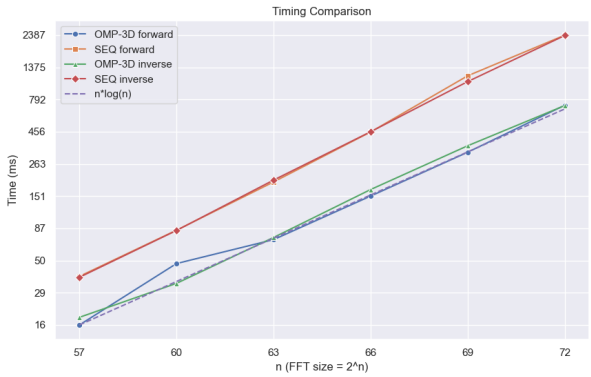
(b) MPI 2D against CT 2D



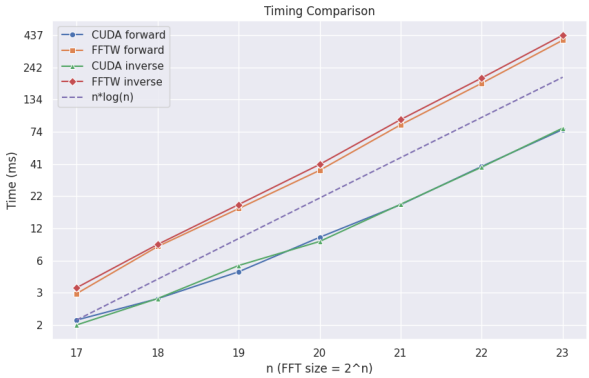
(a) OMP 1D against CT 1D



(b) OMP 2D against CT 2D



(a) OMP 3D against CT 3D



(b) CUDA 1D against FFTW

In summary, the tests indicate better performance for the parallelized implementation, notably with a clear advantage for CUDA, which manages to surpass even FFTW.. However, it's important to note that the baseline of OMP and MPI strategies, represented by the CT, exhibits slower performance compared to the FFTW.

We also present the results of the one-dimensional Stockham implementation templated on float datatype compared to the one templated on double. The findings reveal a slightly better performance in favor of the implementation templated on float:

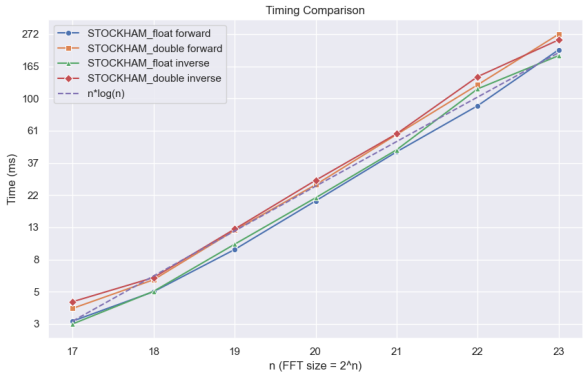


Figure 12: Stockham 1D float against Stockham 1D double

7.1 Error analysis

For all the test, it was calculated the relative error in norm-2, to assess the accuracy of the calculation:

$$e_r = \frac{\|T_s - T_b\|_2}{\|T_b\|_2}$$

where T_s is the result of the tested strategy and T_b is the result of the baseline. The norm-2 is calculated as the square root of the sum of the squares of the elements of the tensor.

The results reveal that the relative error of the Cooley-Tukey implementation, both sequential and parallel, compared to FFTW is on the order of 10^{-10}. In contrast, MPI and OMP implementations yield exactly the same result as the Cooley-Tukey. This consistency arises because they **execute the same operations in the same order**. The higher error shows up when comparing two strategies templated on different floating-points types. In this case the relative error reaches 10^{-7}.

7.2 Speedup and efficiency

To evaluate the **scalability** of the various implementations, the same test was executed multiple times on a fixed dataset size of 2^{24} elements, incrementing the number of processors/threads at each

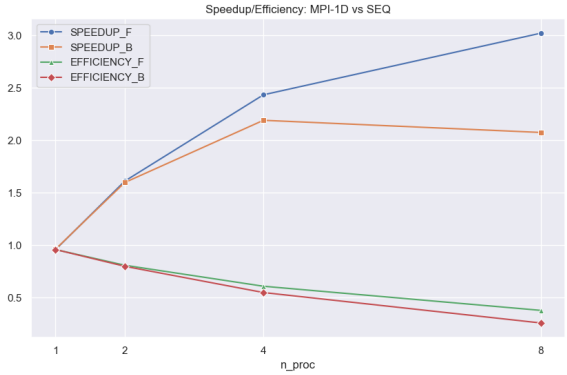
iteration. **Scalability** is linked to how parallel **efficiency** behaves as the size of the problem and the number of parallel processes vary. The parallel efficiency is defined as the ration between speedup over the number of processor/thread

$$E = \frac{S}{p}$$

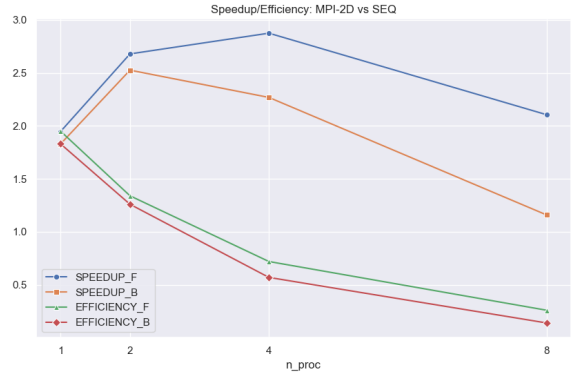
where the **speedup** is defined as the ratio between the computing time on a single process/thread and on the parallel implementation:

$$S = \frac{T_s}{T_p}$$

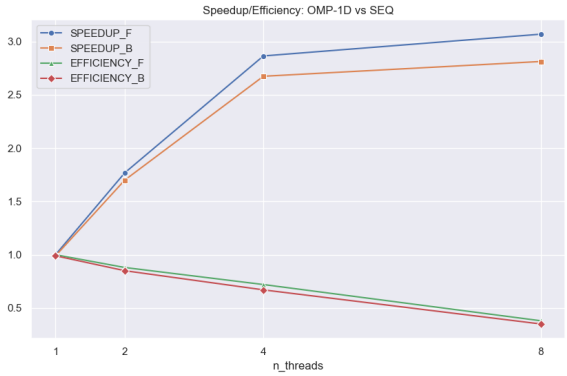
In addition to the efficiency, each plot represents the speedup as a function of the number of threads/processors. The results illustrate that the speedup does not increase linearly and it saturates with the growing number of threads/processors. Consequently, the efficiency declines as the number of threads/processors increases. This behavior arises from the fact that the problem is not embarrassingly parallel, indicating that the solution is not **strongly scalable**.



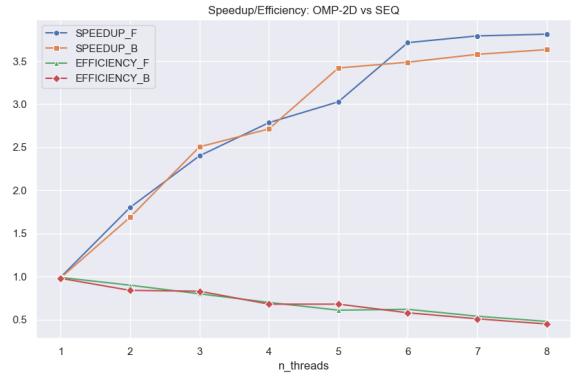
(a) MPI 1D against CT 1D



(b) MPI 2D against CT 2D



(a) OMP 1D against CT 1D



(b) OMP 2D against CT 2D

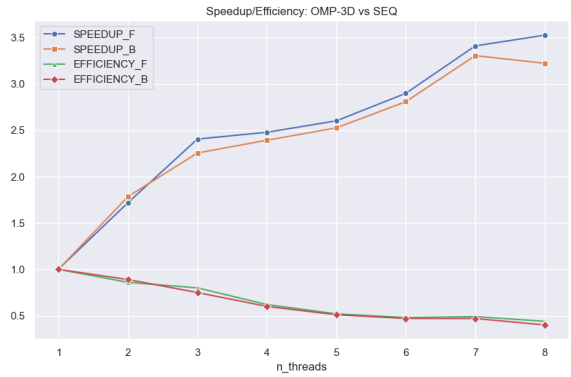


Figure 15: OMP 3D against CT 3D

7.3 MPI+OMP

To conduct a comprehensive MPI and OMP test, we opted to set up a basic **cluster** using our personal computers, simulating a distributed environment. It's essential to clarify that this was merely a preliminary test, and the cluster included two nodes with distinct characteristics, although they both are x64.

The cluster is composed by the following components:

1. **ThinkPad T490s**: CPU I7 8565U, quad-core, eight threads, memory 16GB.
2. **ThinkPad T450s**: CPU I5-5300U, dual-core, four threads, memory 8GB.
3. **D-Link DVA-5592** and **Cat5e** cables.

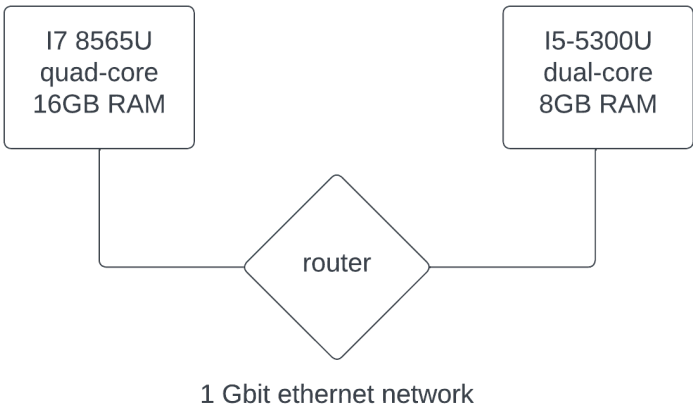


Figure 16: Cluster configuration

The MPI+OMP strategy constraints the number of MPI nodes to be a power of 2. Consequently, the only viable distributed configurations are as follows: one process on the ThinkPad T450s and one or three processes on the ThinkPad T490, or two processes on each node. The results presented here correspond to the latter configuration, which output of –**report-bindings** of **mpirun** is:

[T490:16844] MCW rank 0 bound to socket 0[core 0[hwt 0-1]]: [BB/../../..]
[T490:16844] MCW rank 1 bound to socket 0[core 1[hwt 0-1]]: [../BB/../..]
[T450s:30193] MCW rank 3 bound to socket 0[core 1[hwt 0-1]]: [../BB]
[T450s:30193] MCW rank 2 bound to socket 0[core 0[hwt 0-1]]: [BB/..]

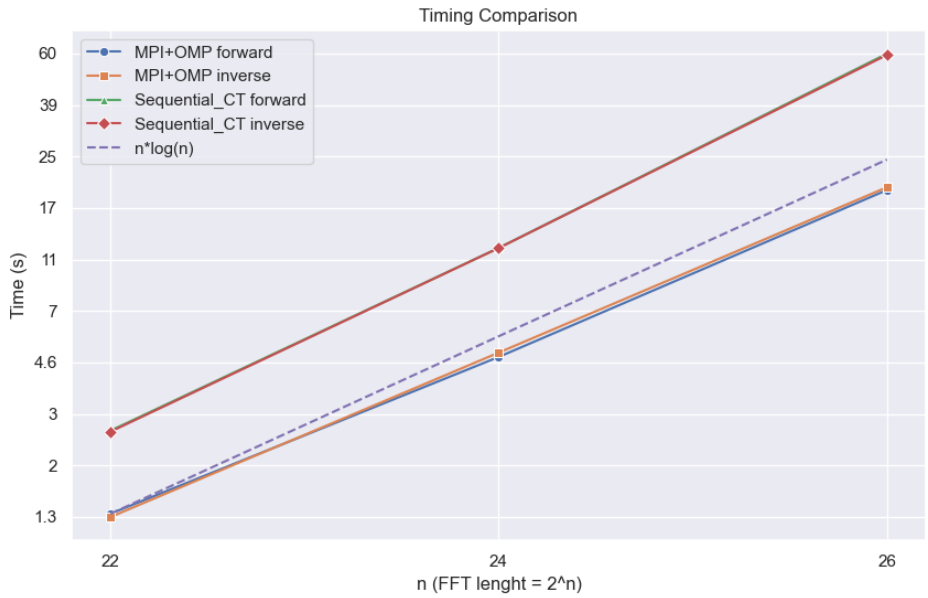


Figure 17: MPI+OMP 2D against CT 2D

7.4 CUDA

Both CUDA implementations (Cooley-Tukey and Stockham) have a speedup of over 5x compared to FFTW for large inputs. We also compared the performance of the two CUDA implementations. The timing of a single FFT is the same for both implementations, but if we analyze the kernel execution times with **nvprof**, we can see that Stockham kernel and Cooley-Tukey kernel have approximately the same execution times (260 ms), but the bit reversal kernel (which is only used in the Cooley-Tukey implementation) takes an additional 50 ms. On a single FFT, this cannot be noticed because of the overhead of the CUDA memory operations, since Stockham algorithm needs twice the CudaMalloc and CudaMemcpy calls. However, if one wants to implement a batched FFT on a GPU, where the memory operations are amortized, the Stockham algorithm is the best choice.

Time(%)	Time	Calls	Avg	Min	Max	Name
25.37%	303.67ms	4	75.917ms	73.138ms	78.982ms	[CUDA memcpy DtoH]
24.64%	295.01ms	12	24.584ms	160ns	76.261ms	[CUDA memcpy HtoD]
22.52%	269.55ms	50	5.3910ms	5.2506ms	6.0481ms	void fftcore::cudakernels::d_butterfly_kernel_cooleytukey<
22.26%	266.45ms	50	5.3290ms	5.2478ms	5.7105ms	void fftcore::cudakernels::d_butterfly_kernel_stockham<dou
4.33%	51.795ms	2	25.897ms	25.885ms	25.910ms	void fftcore::cudakernels::d_bit_reversal_permutation<doub
0.89%	10.653ms	2	5.3264ms	5.3259ms	5.3268ms	void fftcore::cudakernels::d_scale<double>(cuda::std::__4:

Figure 18: Comparison between the two CUDA implementations with Nvidia's profiler

8 Conclusions and further developments

While the performance results obtained may not be directly comparable to fftw's, the design of the library enables the exploration of various technologies and optimization strategies. For example, optimizing the real-to-complex transformation by leveraging the transform symmetry properties can be achieved without any alterations to the library's design. Additionally, optimizations in the choice of floating-point types can also be implemented thanks to templating.

Another potential extension of the library could involve adding support for computing the FFT for input data of any size. Various techniques exist for computing FFT on input sizes that are not powers of two, for instance zero-padding. This capability could be coupled with the introduction of a **planner**, a library component that enables the selection of the most efficient strategy, among all available strategies in the library, on a case-by-case basis for the given input.

References

[1] design patterns - strategy and facade patterns (https://refactoring.guru/design-patterns).

[2] Hands-on material - quinn chap 15.

[3] W.T. Cochran, J.W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, G.C. Maling, D.E. Nelson, C.M. Rader, and P.D. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, Oct 1967.

[4] M. Frigo and S.G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[5] Wikipedia contributors. Fast fourier transform — Wikipedia, the free encyclopedia, 2024. [Online; accessed 27-January-2024].