# An Approach to Specifying and Automatically Optimizing Fourier Transform Based Operations

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

**Doru Adrian Thom Popovici**

B.S., Computer Science Department, Politehnica University Timisoara

M.S., Engineering Department, Carnegie Mellon University Pittsburgh

Carnegie Mellon University

Pittsburgh, PA

December 2018

# Acknowledgments

A couple of pages seems insufficient to thank all the people who encouraged and supported me to bring this work to a closure. As such, I will try to keep it brief. First and foremost, I would like to thank my committee members – Dr. Kathy Yelick, Dr. Paul Kelly, Dr. James Hoe, Dr. Tze Meng Low and of course my adviser Dr. Franz Franchetti – for being part of this process. Your questions and feedback helped me improve the overall quality of the work, for which I am grateful. Franz, thank you for choosing my application and giving me the opportunity to join your group and CMU. Your dedication and guidance made me the sarcastic researcher I am today. Tze Meng, I would also like to thank you for always putting the way I tackle problems into check with questions like "what" and "why". Your perspective on things made me question everything that's out there.

Second, I would like to thank the people in the SPIRAL group and in the A-level, past and present. You are amazing friends, who made my stay in Pittsburgh a lot more pleasant. I would like to extend a warm thank you to Richard Veras, who always found the time to listen to my half baked idea and debate its importance, whether it was in the office, over the internet or out drinking several beers. Special shout-out to Milda Zizyte, who proved to be a true and wonderful friend. I would like to thank you for always finding the time to chat about various things or to go to all sorts of concerts and events around Pittsburgh. I would also like to thank Daniele Spampinato, Anuva Kulkarni, Joe Melber, and Upasana Sridhar for going through the dissertation and providing feedback about grammar mistakes, spelling errors and incoherent story flow. I am really grateful for the effort put into that and I am sorry for causing you all a headache. There are a lot more people I would like to thank, however the sheer amount of people requires an appendix.

Lastly, I would like to thank my parents, Mariana and Doru Popovici. Thank you! You have always been there for me, whether to chat about some of daily problems I have encountered during the PhD program or whether to simply say hello and see whether I have eaten. You both supported me since I was six years old. You gave me the chance of trying multiple things from painting, sculpture, music to math, physics and finally computers. You opened my mind to what is important. For that I am truly grateful.

The work in this dissertation could not have happened without of the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-16-2-0033, and HR0011-13-2-0007, the United States Department of Defense (DoD) under Contract No. FA8702-15-D-0002 and the National Science Foundation (NSF) under Contract No. 1116802.

Doru Adrian Thom Popovici

*Carnegie Mellon University*

*September 2018*

# Abstract

Ideally, computational libraries and frameworks should offer developers two key benefits. First, they should provide interfaces to capture complicated algorithms that are otherwise difficult to write by hand. Second, they should automatically map algorithms to efficient implementations that perform at least as well as expert hand-optimized code. In the spectral methods domain there are frameworks that address the first benefit by allowing users to express applications that rely on building blocks like the discrete Fourier transform (DFT). However, most current frameworks fall short on the second requirement because most opt for optimizing the discrete Fourier transform in isolation and do not attempt to integrate the DFT stages with the surrounding computation. Integrating the DFT computation with the surrounding computation requires one to expose the implementation of the DFT algorithm. However, due to the complexity of writing efficient DFT code, users typically resort to implementations for the DFT stages, in the form of black box library calls to high performance libraries such as MKL and FFTW. The cost of this approach is that neither a compiler nor an expert can optimize across the various DFT and non-DFT stages.

This dissertation provides a systematic approach for obtaining efficient code for DFT-based applications like convolutions, correlations, interpolations and partial differential equation solvers, through the use of cross-stage optimizations. Most of the applications follow a pattern: they permute the input data, apply a multi-dimensional discrete Fourier transform, perform some computation on the Fourier transform result, apply another multi-dimensional discrete Fourier transform and possibly another data permutation. Applying optimizations across the multiple stages is enabled by the ability to represent the DFT and the additional computation with the same high level mathematical representation. Capturing the compute stages of the entire algorithm with a high level representation allows one to apply high level algorithmic transformations like fusion and low level optimizations across the stages, optimizations that otherwise would not have been possible with the black box approach.

The first contribution of this work is a high level API for describing most common DFT-related problems. The API translates the problem specification into a mathematical representation that is readable by the SPIRAL code generator. The second contribution of this work is the extension to the SPIRAL framework to allow for cross-stage optimizations. The current work extends SPIRAL's capabilities to automatically apply fusion and other low level architecture dependent optimizations to the DFT and non-DFT stages, before generating the code for the most common CPUs. We show that the generated code, that adopts the proposed approach, achieves 1.2x to 2.2x performance improvements over implementations that use DFT library calls to MKL and FFTW.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The thesis of this dissertation is that for most Fourier-based algorithms there is a systematic way of achieving efficient code through cross-stage optimizations. Most Fourier-based algorithms like convolutions, correlations, interpolations or partial differential equation (PDE) solvers typically follow the same pattern, where discrete Fourier transform (DFT) stages are interleaved with other compute stages. This insight suggests that the focus of optimizations should be on the DFT-based algorithms rather than the DFT itself. Performance gains can be achieved by having a wider view of the entire algorithm, since high level loop transformations and low level optimizations can be applied across compute stages.

## 1.1 Motivation

In the spectral methods [4] for numerically solving certain differential equations, the discrete Fourier transform (DFT) has proven to be an ubiquitous mathematical tool. Various applications like LAMMPS [5, 6, 7],HACC [8, 9], XGC [10], WarpX [11], NWChemEx [12, 13, 14], AMReX [15] from different scientific domains as outlined

| Category | DFT Requirements |
|---|---|
| Field Calculations for Particle-In-Cell | large 3D DFTs of $10^9$ - $10^{12}$ points |
| Molecular Dynamics | relative small 3D DFTS of $10^6$ - $10^7$ points |
| First-Principles Calculations or Chemistry and Materials | relative medium size but many 3D DFTs of $256^3$ data points |
| Domain-Decomposed PDE solvers | relative medium size 3D DFTs of at most $256^3$ |

Table 1.1: Category of applications where the DFT is being used.

in Table 1.1, spend up to 40% of their total execution in the computation of the DFT on supercomputers. The importance of the DFT comes from the fact that it simplifies expensive computations by performing a change of basis from time/space domain to the frequency domain. For example, solving partial differential equations or computing long-range contributions to forces between particles in time/space domain become point-wise computation in the frequency domain. Hence, most applications that rely on the DFT to simplify computation follow a specific pattern as seen in Algorithm 1, where a forward DFT is applied on the input, the result is point-wise multiplied with pre-computed values and finally an inverse DFT is applied to convert the sequence back to the original domain.

Depending on the application, the computation pattern may slightly differ. Different applications require different shapes, sizes and types of DFTs. For example, some applications require medium size 3D DFTs of at most 1024 elements in each dimension, while some applications require large 3D DFTs of up to 10240 points in each dimension. Some applications require the DFT to be applied on complex data points, while some require the DFT to be applied on real data points. In addition to the DFT requirements, the point-wise computation may also vary. The point-wise computation may exhibit symmetry properties. For example, some

**Algorithm 1** DFT-based convolution operation

```
1: function CONV($x, h, y, N$)      ▷ Where x, h - input arrays, y - output array, N - length
2:     $X = \text{dft}(x)$
3:     $H = \text{dft}(h)$, can be pre-computed or computed on the fly
4:     for $i = 0$ to $N - 1$ do
5:         $Y[i] = X[i]H[i]$
6:     end for
7:     $y = \text{idft}(Y)$
8: end function
```

applications require multi-dimensional DFT-based operations where the point-wise computation is symmetric and thus can be separated on the dimensions of the data-set. Under this scenario, instead of computing the point-wise operation after the full forward multi-dimensional DFT, the point-wise computation can be decomposed and done locally in each dimension immediately after each 1D DFT computation. Given all these degrees of freedom and also given the complexity of writing efficient DFT code, most users resort to hand-written implementations tailored for the specific problem, where the DFT computation is done by invoking high performance libraries like MKL [16], FFTW [17], cuFFT [18] or armPL [19].

**Main Problem.** *Implementing any DFT-based operation using library calls comes at the cost that neither a compiler nor an expert can optimize across the various DFT and non-DFT compute stages.* Library calls are typically viewed as black boxes and inter-procedural optimizations [20] cannot be applied. Hence, performance is still left on the table. For example, Figure 1.1 shows the performance difference between three different implementations for a 3D DFT-based operation with separable point-wise operation. The point-wise operation is separable into smaller point-wise operations that are applied in each dimensions. Moreover, the 3D DFT is also a separable operation. The forward and inverse 3D DFTs can be decomposed in multiple forward and inverse 1D DFTs, that are applied in each di-

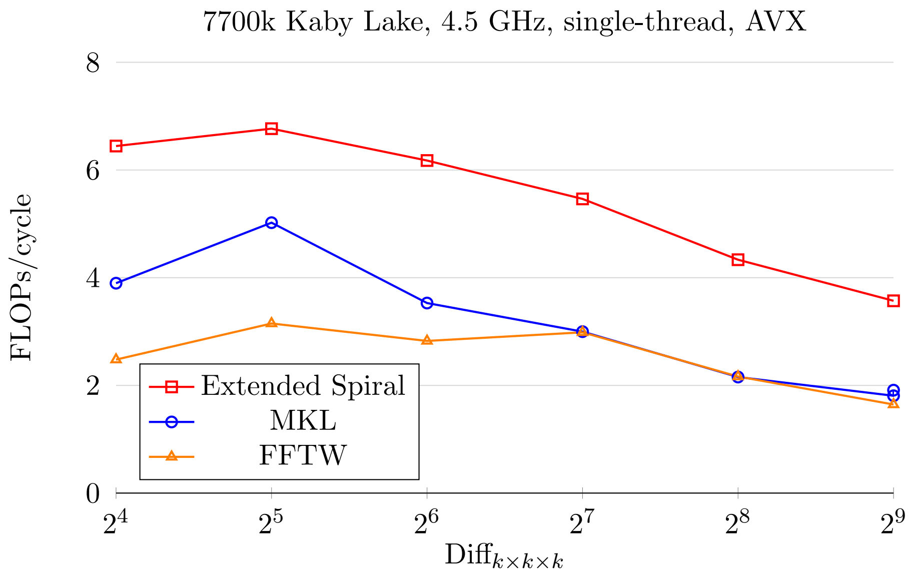


Figure 1.1: Performance plot for the 3D DFT-based differentiation operation on Intel Kaby Lake 7700K. The first implementation (red line) represents the performance achieved by the implementation using the proposed approach. The next two implementations use MKL and FFTW for the DFT stages and hand-written code for the point-wise multiplication.

mension. This suggests that the DFT and non-DFT computation can be efficiently grouped to reduce the number of stages. In addition, the forward 1D DFT, local point-wise operation and the inverse 1D DFT can further be merged. More details about merging the compute stages for multi-dimensional DFT-based operations with separable point-wise can be found in Chapter 3. Overall, the implementation that efficiently fuses the DFT and non-DFT stages (red line) outperforms the other two implementations (blue and brown line) by almost 2x for problem sizes that fit within the cache hierarchy and problem sizes that reside in main memory. It is well known that the DFT and point-wise computation are memory bound. As the problem size increases, the computation stalls waiting for data to be brought from the lower levels of the memory hierarchy. This affects the implementations that do not fully merge the DFT and non-DFT stages (blue and brown line). However, fusing the computation alleviates this problem by reducing the number of round-trips to the lower levels of the memory hierarchy. Once data is brought into the upper levels of the memory hierarchy, the local 1D DFTs and point-wise computation are immediately applied one after the other before writing the data back to memory. Chapter 3 goes into more details about fusing the computation and also about applying other low level optimizations across the stages, optimizations that would not be possible if the DFT-based operations are implemented using black box library calls.

## 1.2 State-of-the-Art Approaches

Over the past years the focus has mostly been on delivering efficient implementations for the DFT or other linear transforms that can replace the DFT computation. Users typically resort to writing code around library calls in order to implement more complicated algorithms, which comes with two drawbacks. On one side, users must understand the underlying system to efficiently write code and on the other side performance can still be gained since optimizations cannot be applied across stages. Therefore, frameworks like AccFFT [21, 22], Halide [23] and Indigo [24] have attempted to integrate the DFT within the larger picture of the entire application. They recognize the importance of having a wider view of the application to reason about cross stage optimizations, however even in these cases the DFT is not fully merged with the surrounding computation. In the following we discuss the two categories and outlines the drawbacks that are addressed by this work.

### 1.2.1 Frameworks for the DFT

In this section, we present frameworks that deal with *generating efficient DFT code*. The main drawback of using such frameworks is that *neither a compiler nor an experts can apply optimizations across the stages*.

**SPIRAL.** SPIRAL [25] is a framework built upon the Signal Processing Language (SPL) [26, 27] used to describe various linear transforms such as the discrete Fourier transform [28], the discrete Cosine transform [29] and many more. The framework is composed of several stages, where each stage performs a specific operation. The first stage takes in a problem described in the SPL notation and implements that problem using one of the algorithms stored in its database. For example, power of two DFTs are typically implemented using the Cooley-Tukey [30] algorithm. The second stage converts the SPL algorithm into the so-called $\sum$-SPL, where the notion of loops is introduced [31]. This stage applies optimizations to the $\sum$-SPL representation. Finally, the optimized $\sum$-SPL is converted to code for a given architecture. The generator can target architectures with features like Single Instruction Multiple Data (SIMD) instructions [32] and multiple threads [33].

The SPIRAL framework was built for generating high performance linear transforms such as the DFT. However, in recent years there has been work extending the framework to other domains. We extend the SPIRAL framework to capture DFT-based operations using the SPL representation and to perform cross stage optimizations. Having the high level mathematical representation enabled us to perform algorithmic transformation such as fusion and low level optimizations specific for the underlying architecture.

**FFTW.** FFTW [34, 35] is a framework to generate and auto-tune DFT code for different CPU architectures, ranging from Intel and AMD to ARM and

```
#include <fftw3.h>

int main(int argc, char ** argv) {
  fftw_complex *x, *X, *H, *y, *Y; // H is precomputed
  fftw_plan pF, pI; // fftw plans for the forward and inverse DFT

  // create the DFTs
  plan pF = fftw_plan_dft_1d(N, x, X, FFTW_FORWARD, FFTW_ESTIMATE);
  plan pI = fftw_plan_dft_1d(N, Y, y, FFTW_BACKWARD, FFTW_ESTIMATE);

  // DFT based convolution
  fftw_execute(pF); // forward DFT
  for(int i = 0; i != N; ++i) {
    Y[i] = X[i] * H[i]; // point-wise multiply
  }
  fftw_execute(pI); // inverse DFT

  // plan destruction and memory deallocation
}
```

Figure 1.2: C Code implementation for the DFT-based convolution operation. The code uses the FFTW interface for the forward and inverse DFT, while the point-wise multiplication is hand-written.

IBM Power. FFTW is viewed as the gold standard when it comes to DFT code generation, since it targets most features like (SIMD) and thread parallelism. The framework offers various functions to create and execute DFTs. The user can request for multi-dimensional DFTs, he/she can specify the data type or even the directions in which the DFTs are applied. For example, Figure 1.2 shows an example of implementing a 1D DFT-based convolution, where first the forward and inverse DFTs are created and then they are executed around the hand-written code that performs the point-wise complex multiplication.

Implementing any of the DFT-based algorithms using the FFTW interface as shown in Figure 1.2 does not allow optimizations to be applied across compute stages. Compilers cannot identify how the DFT code is implemented and thus cannot optimize the code. An alternative is to extend the framework. FFTW provides a language to describe the operations and thus the framework can be extended to support DFT-based operations. However, the performance will lack since most of the assumptions for optimizing the DFT cannot easily be extended to DFT-based operations.

**MKL.** MKL [16] is the Mathematical Kernel Library from Intel that provides implementations for a multitude of mathematical kernels. The DFT is among the offered kernels. MKL's DFT implementation is highly efficient and highly optimized on most Intel architectures. They support multi-dimensional DFTs, real and complex DFTs. However, they do not provide a rich interface similar to FFTW to allow the description of the dimensions in which the DFT can be applied. This complicates the life of the user. Data must be orchestrated to follow the requirements of the Intel API. Therefore, the user write and optimize the code that performs data rotations and movement.

Similar to FFTW, implementing DFT-based algorithms using MKL can be done following the same recipe as shown in Figure 1.2. First the forward and inverse DFTs must be constructed and glue code around the DFT compute functions must be written. Again the compiler treats the library calls as black boxes and does not attempt to optimize or fuse the various stages. MKL is closed source, therefore extending the framework to applications that build upon DFTs is not possible.

**FFTE.** The FFTE [36] package offers Fortran implementations for the Fourier transform for complex and real inputs. The package targets problem sizes of the form $2^p 3^q 5^r$ and provides support for CPUs and GPUs. The code is hand optimized and it follows the details presented in the papers from Daisuke Takahashi. In the papers, the author describes various methods for blocking the memory access of the DFT for the cache hierarchy [37] and shared-memory processors [38].

He also discusses the extension of the 1D and 3D DFTs for clusters of CPUs and GPUs [38, 39].

Similar to the previous packages, FFTE also imposes that each stage be written separately. In contrast to FFTW and MKL, the FFTE code needs to be compiled simultaneously with the main code that implements the DFT-based algorithms. Therefore, compilers can attempt to perform inter-procedural optimizations [20] across the compute stages. However, due to the sheer complexity of writing the DFT code, the compiler may or may not actually identify the possibilities of merging the stages or even apply other optimizations. An alternative is to extend the existing code to allow for the DFT-based operations to be computed, task that may prove to be quite labor intensive. In addition, such approach is not maintainable in the long run.

### 1.2.2 Frameworks for DFT-based Algorithms

In this section, we present some frameworks that identify that *the DFT stages need to be merged with the surrounding computation* to reduce the effects of long latencies to main memory, however due to *the complexity of decomposing and implementing the DFT* most of them still treat the computation in isolation and resort to MKL or FFTW for the DFT computation.

**Halide.** Halide [23, 40, 41] is a framework for describing, optimizing and generating code for image processing pipelines. The framework decouples the algorithm or the stages of the algorithm from the optimizations. The users can specify the order in which the optimizations are applied. In addition, there is an autotuner that tries out the different variants of the optimizations and chooses one that gives good performance. They provide a high level language to express the problems and a code generator that interprets the language and generates the appropriate nested loops. Since the framework was used to describe image processing pipelines, in the work by Ragan-Kelly [23] they implemented a 2D DFT computation that is applied on multiple images, or batches of data. They have shown that the performance of their implementation of the batch 2D DFT outperformed FFTW's implementation by almost 4x on ARM based systems.

The framework allows the description of applications composed of different pipelines. In addition it offers the benefit of allowing users to describe a list of optimizations and most important the order in which the optimizations must be applied. However, the 2D DFT is still optimized in isolation. The framework does not decompose the DFT and attempt to merge the stages with the computation that surrounds it. The focus has mostly been on optimizing the entire image processing pipeline, and therefore there has not been any work using the Halide language to express and decompose the DFT and DFT-based operations.

**Indigo.** Indigo [24] is a domain specific language used to describe image reconstruction algorithms. The framework proposes expression trees to represent linear operators. The leaves within the trees are represented either by matrix driven operators backed by explicit sparse or dense matrices and implemented using BLAS routines [42, 43, 44] or matrix-free operators such as the DFT where the matrix representing the DFT is implicit and never actually stored. The non-leaf nodes represent basic operations such as summations and products. The idea is that the expression tree gives better insight on how to apply high level loop transforms like loop reordering and loop fusion to improve performance and reduce memory footprint. It is shown that for applications like MRI, ptychography, magnetic particle imaging and fluorescent microscopy, the approach provides significant improvements.

The framework allows the description of algorithms that use the Fourier transform as a building block. Expressing the entire algorithm as an expression tree allows one to optimize the entire pipeline. However, once again the DFT computation is left as is and implemented using library calls to high performance libraries like MKL, FFTW and cuFFT. As mentioned in the paper, the implementation of the DFT algorithms leverage the structure of the DFT matrix in order to get performance, however since the decomposition of the DFT algorithm is not exposed to the pipeline, performance is still left for grabs.

**AccFFT.** AccFFT [21, 22] is a parallel framework used to compute 3D DFTs on distributed CPU/GPU systems. It provides an API to describe 3D DFTs and under the hood it creates the transforms by calling FFTW and/or cuFFT. The FFTW and cuFFT invocations are required for the creation of the local computation. The framework provides various implementations for the 3D DFT to improve scalability of the algorithm when increasing the number of nodes (the slab-pencil and pencil-pencil decompositions). In addition, the framework uses a $k$-way all-to-all communication [45] to improve overall data traffic. While previously built only for large scale 3D DFTs, the framework was extended for DFT-based algorithms. The API currently offers functions to create the necessary code for computing parallel Poisson Solvers and other spectral operators such as gradient, divergence, Laplace and biharmonic operators.

The framework offers functionalities for applications that use the DFT as a main building block. However, as with all other state-of-the-art frameworks, AccFFT does not attempt to expose the DFT computation and integrate it with the surrounding computation. The framework uses FFTW and/or cuFFT for performing the local DFT computation and uses glue code around the library calls for implementing the DFT-based operations covered by the API. Compilers cannot optimize across the stages and leave the code untouched. Although users do not have to deal with implementing the DFT-based operations, extending the framework for other computations may prove to be difficult.

## 1.3 The Approach in this Dissertation

In this dissertation, we focus on DFT-based algorithms like convolutions, correlations, interpolations and partial differential equation solvers and provide an approach for specifying and optimizing the entire algorithm rather than just optimizing the DFT computation in isolation. We make two observations:

- Most DFT-based operations follow a specific pattern where typically a point-wise computation is performed between a forward and inverse DFT. Depending on the DFT-based operation the point-wise computation may exhibit symmetry properties.

- Knowing the decomposition of the forward and inverse DFT using algorithms like the Cooley-Tukey algorithm permits for a systematic way of applying optimizations across the DFT and non-DFT stages.

The first observation is based on the properties of the Fourier transform, which are covered in more detail in Chapter 2. Most DFT-based operations follow the same computational pattern, where a forward DFT is followed by a point-wise and an inverse DFT. Hence, specifying and optimizing the entire DFT-based operation seems more logical. Instead of specifying only the DFT and writing glue code around the DFT calls as seen in Figure 1.2, one can specify the entire DFT-based operation using a simple API. For example, Figure 1.3 shows the steps required to

```
#include <spiral.h>

// ...

int main(int argc, char ** argv) {
  spiral_complex *x, *H, *y; // H is precomputed
  spiral_plan pConv; // convolution plan

  // plan creation
  pConv = spiral_plan_conv_1d(N, x, y, H, SPIRAL_FORWARD);

  // DFT based convolution
  spiral_execute(pConv);

  // plan destruction and memory deallocation
}
```

Figure 1.3: C Code implementation of the convolution operation using the in house API. The user creates the convolution operation by specifying a pointer to an array that stores the pre-computed values required for the point-wise multiplication.

create a 1D DFT-based with the proposed in-house API. The user is required to use a create function which is similar to the FFTW DFT create function. The only difference is that the `spiral_plan_conv_1d` takes an additional argument, namely a pointer to an array or to a function meant to compute the point-wise multiplication. In this example, the pointer is to an array that stores the pre-computed values of `H`. More details about the API and its capabilities are provided in Chapter 4.

The second observation suggests that knowing the decomposition of the forward and inverse DFT and representing the compute stages in a high level language, one can apply optimizations in a systematic way across the various compute stages. In this work, we use the SPL notation defined within the SPIRAL framework to represent and apply optimizations on the entire algorithm. However, the optimizations are not bound to the language, any high level language that is expressive enough can be used. We simply use SPL since it has been successfully used with expressing and optimizing the DFT computation and it can easily be used to represent DFT-based operations. For example, the Algorithm 1 implemented with the proposed API as seen in Figure 1.3 can further be described using SPL as

$$\mathrm{Conv}_n = \mathrm{iDFT}_n \cdot \mathrm{Diag}_n^H \cdot \mathrm{DFT}_n, \tag{1.1}$$

where the $\mathrm{DFT}_n$ and $\mathrm{iDFT}_n$ represent the the forward and inverse DFT of size $n$. Here, the $\mathrm{Diag}_n^H$ stores the pre-computed values of $H[k]$. The $\mathrm{DFT}_N$ and $\mathrm{iDFT}_N$ are further decomposed using algorithms like Cooley-Tukey. Using properties of the DFT like $\mathrm{iDFT} = \mathrm{DFT}_n^H$, where $(\cdot)^H$ represents the conjugate-transpose operation, the decompositions of the forward and inverse transform can be re-organized so as to alleviate the burden of merging the compute stages. Chapter 3 primarily presents the systematic approach of decomposing the DFTs and identifying the appropriate decompositions that allows for efficient stage merging. In Chapter 5, we show the benefits of merging the compute stages for both 1D and multi-dimensionl DFT-related operations. We offer a thorough analysis of when dealing with merging various stages of the algorithm, based on the different flavors of DFT-based operations.

## 1.4 Contributions

In this dissertation, the goal is to show that the DFT computation can systematically be integrated with the surrounding computation. This work provides the following contributions to the spectral methods domain:

- We identify that, although the DFT is an important mathematical kernel, the focus of optimizations should be on the DFT-based operations and not solely on the DFT. Most DFT-based operations follow the same pattern, where a forward DFT is followed by a point-wise operation and an inverse DFT. Thus, having a view of the entire algorithm offers more opportunities for optimizations.

- We show that there is a systematic way of applying optimizations across the DFT and non-DFT stages. Having domain specific knowledge about the DFT decomposition and its properties, one can easily reason on how to optimize across the various compute stages.

- We outline that fusing is important, especially when dealing with DFT and point-wise operations, which are well known memory bound problems. Each stage of the DFT-based operation stalls waiting for data to be brought from memory. However, fusing the stages reduces the stall time by improving data locality and cache utilization.

- We provide a thorough analysis of the benefits provided by fusing the compute stages. We discuss different cases where the point-wise computation may have symmetry properties and show that efficiently fusing the compute stages can bring up to 2x performance improvements over the implementations where the compute stages cannot be merged.

- We build a simple API that allows users to express most DFT-based operations. The API is built on top of the SPIRAL code generator. The code generator is extended to optimize and generate code for DFT-based operations, targeting most modern CPU architectures. Most of the optimizations carried across the DFT and non-DFT stages are automatically applied.

## 1.5 Outline

Chapter 2 presents the basic information related to the Fourier transform. In addition, the basic DFT properties are also outlined. Each property specifies that complicated operations in one domain are converted into point-wise operations in the dual domain. The chapter also introduces the language used to describe algorithms for computing the DFT. The language is further used to describe DFT-based operations like convolutions, correlations, interpolations and PDE solvers.

Chapter 3 presents presents the systematic approach of fusing the compute stages. It first starts with what loop fusion is and how loop fusion connects to some of the constructs used within the SPL notation. In the second part of the chapter, the principles behind fusing the compute stages are presented. While the discussion focuses on 1D DFT-based operations, the approach can easily be extended to higher dimensional operations.

Chapter 4 discusses the details related to implementing an end-to-end framework that allows the expression of DFT-based algorithms and that automatically generates and optimizes the code. The chapter is broken down into two main sections, namely the front-end that deals with the high level API and the back-end that are mainly represented by the code generator.

Chapter 5 presents a detailed analysis of the benefits of fusing the compute stages for DFT-based operations. The chapter starts with the analysis for the 1D DFT-based convolution. Then the discussion is moved to the 3D equivalent operation, where the point-wise operation is the one that will give different possibilities of fusing. The different scenarios for fusing offer different performance improvements.

Chapter 6 presents concluding remarks and future directions.

# Chapter 2

# Notation and Language to Describe the DFT

The first part of this chapter presents what the Fourier transform is and outline its importance. We briefly describe the different variants of the Fourier transforms, however focus more on the discrete Fourier transform (DFT) since it is the only one that can be done on a computer. We present both the 1D DFT computation, but also extend the definitions to multi-dimensional DFTs. We further present some of the basic DFT properties and emphasize that complicated operations in one domain are reduced to point-wise computations in the dual domain. The properties holds for both 1D and multi-dimensional data-sets.

The second part of the chapter introduces the domain specific language (DSL) used to describe the DFT and the various algorithms meant for the DFT computation. The goal of having a mathematical language to describe the entire algorithm is to be able to apply optimizations at that level of abstraction rather than analyzing and optimizing C code. In this work, we use the Signal Processing Language

(SPL), which is already used within the SPIRAL framework. While in prior work the language was used to describe various algorithms for computing the DFT and other transforms, we use the language to express DFT-based operations. We give the basic definitions and properties of the language, outlining some of the important aspects that allow us to reason about how to fuse the separate compute stages.

## 2.1 What is a Fourier Transform?

The Fourier transform belongs to the Spectral Methods domain as described in "The Landscape of Parallel Computing Research: A View from Berkeley" [4]. The Spectral Methods domain deals with solving various problems such partial differential equations (PDE) by performing computation in the frequency domain rather than the time or space domain. The premise is that computation in the frequency domain is simpler and less expensive. The Fourier transform is one of the most widely known and used transforms to map a sequence from the time or space domain to the frequency domain.

### 2.1.1 Definition

The Fourier transform decomposes a given function into its fundamental sinusoids or harmonics, where each sinusoid is described by a fundamental frequency that shows how fast the sinusoid oscillates. For example, figure 2.1 depicts the decomposition of the continuous time function (signal) $d(t)$ into its fundamental frequencies. It is important to notice that the function can be expressed as a superposition of three fundamental sinusoids. Each sinusoid oscillates at a specific frequency. The continuous function can be fully described by the three composing frequencies, each having a amplitude and a phase.

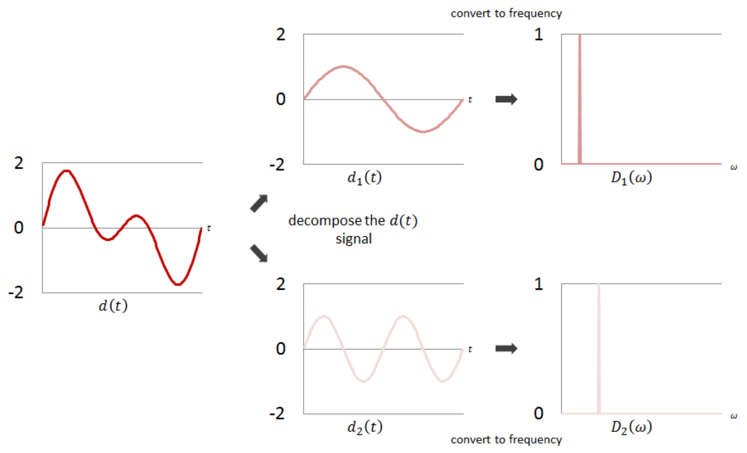


Figure 2.1: The decomposition of a continuous sequence into its composing sinusoids. Each sinusoid oscillates with a different fundamental frequency. The Fourier transform maps the continuous time representation $d(t)$ to the frequency representation.

Any function or sequence can be expressed as a finite or infinite sum of scaled harmonics or sinusoids. For example, a random continuous aperiodic function $x(t)$ can be decomposed as an infinite sum of harmonics such as

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(j\omega) e^{j\omega t} \mathrm{d}\omega, \tag{2.1}$$

where the continuous complex exponential $e^{j\omega t}$ represents the harmonic function. The $X(j\omega)$ term is the frequency representation of the $x(t)$ function which is computed as

$$X(j\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} \mathrm{d}t. \tag{2.2}$$

The forward transform determines the frequency representation of a given signal or function, while the inverse Fourier transform reconstructs the original time domain sequence given the frequency domain representation. The Fourier transform always goes in pairs. The forward and inverse Fourier transform can also be viewed as projections of continuous functions $x(t)$ and $X(j\omega)$ onto the space spanned by the complex harmonics [46].

There are four Fourier representations based on the properties of the input and output signals or functions [46], namely discrete-time Fourier transform (DTFT), continuous-time Fourier transform (FT), discrete-time Fourier series (DTFS) and continuous-time Fourier series (CTFS). If the function is continuous and does not have periodicity then the Fourier transform (FT) can be used to compute the frequency representation. However, if the sequence is discrete and periodic the discrete time Fourier series (DTFS) is the appropriate transform to determine the frequency representation. However, there are methods to link the four transforms by introducing mathematical tools like the $\delta$ function, sampling and interpolation operations. The ability to switch between the transforms is important since certain operations or properties like differentiation can only be applied on continuous functions and not discrete sequences. However, data points stored on a computer are typically discretized samples of the original continuous function, in which case operations like differentiation must be simulated.

It is important to state that out of the four Fourier representations the discrete time Fourier series (DTFS) is the only transform that can be computed on a computer. The DTFS is also known in the literature as the discrete Fourier transform (DFT). Intrinsically the DFT assumes the sequence is periodic and computes the frequency representation only on one period of the entire sequence

$$X[k] = \sum_{n=0}^{N-1} x[n]\omega_N^{kn}, \text{ where } k = 0..N-1 \text{ and } \omega_N^{kn} = e^{-j\frac{2\pi}{N}kn}. \tag{2.3}$$

The $X[k]$ sequence represents the spectral or frequency representation of the one dimensional sequence $x[n]$. The length of the sequence $X[k]$ is the same as $x[n]$. In addition, $X[k]$ has the same period $N$ as the input sequence $x[n]$. The complex terms $\omega_N^{kn} = e^{-j\frac{2\pi}{N}kn}$ represent the complex roots of unity and they form the so-called fundamental harmonics for the sequence $x[n]$.

### 2.1.2 Extending the DFT to Higher Dimensions

The 1D DFT computation can be extended to higher dimensions. For example, given a high dimensional discrete periodic sequence $x[n_0, \ldots, n_{p-1}]$, where $n_0 = 0..N_0, \ldots,$ $n_{p-1} = 0..N_{p-1}$ and $p$ represents the number of dimensions, the $p$-dimensional DFT is defined as

$$X[k_0, \ldots, k_{p-1}] = \sum_{n_0=0}^{N_0} \cdots \sum_{n_{p-1}=0}^{N_{p-1}} x[n_0, n_1, \ldots, n_{p-1}] e^{-j2\pi\left(\frac{k_0 n_0}{N_0} + \ldots + \frac{k_{p-1} n_{p-1}}{N_{p-1}}\right)}. \quad (2.4)$$

The complex exponential can be decomposed as a product of complex exponentials such as

$$e^{-j2\pi\left(\frac{k_0 n_0}{N_0} + \ldots + \frac{k_{p-1} n_{p-1}}{N_{p-1}}\right)} = e^{-j\frac{2\pi}{N_0}k_0 n_0} \cdot \ldots \cdot e^{-j\frac{2\pi}{N_{p-1}}k_{p-1} n_{p-1}}. \quad (2.5)$$

Given this property the summation terms can be grouped.

For example, given a three dimensional function $x[n_0, n_1, n_2]$, the 3D DFT can be re-written using the splitting of the complex exponentials as

$$X[k_0, k_1, k_2] = \sum_{n_0=0}^{N_0} \sum_{n_1=0}^{N_1} \sum_{n_2=0}^{N_2} x[n_0, n_1, n_2] e^{-j\frac{2\pi}{N_0}k_0 n_0} e^{-j\frac{2\pi}{N_1}k_1 n_1} e^{-j\frac{2\pi}{N_2}k_2 n_2}. \quad (2.6)$$

The summations can be grouped and can be computed in any order. For example, grouping the terms as follows

$$X[k_0,k_1,k_2] = \sum_{n_0=0}^{N_0}\left(\sum_{n_1=0}^{N_1}\left(\sum_{n_2=0}^{N_2} x[n_0,n_1,n_2]e^{-j\frac{2\pi}{N_2}k_2n_2}\right)e^{-j\frac{2\pi}{N_1}k_1n_1}\right)e^{-j\frac{2\pi}{N_0}k_0n_0}. \tag{2.7}$$

suggests that the 3D function $x[n_0,n_1,n_2]$ is gradually converted to its Fourier representation. First a 1D DFT is applied on the function in the third dimension $n_2$. Then 1D DFTs are applied in the dimension $n_1$ and lastly 1D DFTs are applied in the dimension $n_0$. This process of gradually applying 1D DFTs in each dimension requires a temporary array $X_{t_0}[n_0,n_1,k_2]$ to store the intermediary results. The next two 1D DFTs are applied on the temporary $X_0$ as

$$X[k_0,k_1,k_2] = \sum_{n_0=0}^{N_0}\left(\sum_{n_1=0}^{N_1} X_0[n_0,n_1,k_2]e^{-j\frac{2\pi}{N_1}k_1n_1}\right)e^{-j\frac{2\pi}{N_0}k_0n_0}. \tag{2.8}$$

The algorithm of applying the 1D DFT in each dimension is called the pencil-pencil-pencil algorithm. It requires three passes through the data, and each must apply all its 1D DFTs before proceeding to the subsequent stages.

Grouping the terms in a different way gives a different algorithm. For example, grouping two of the dimensions together and letting the third dimension be computed at the end gives the following algorithm

$$X[k_0,k_1,k_2] = \sum_{n_0=0}^{N_0}\left(\sum_{n_1=0}^{N_1}\sum_{n_2=0}^{N_2} x[n_0,n_1,n_2]e^{-j\frac{2\pi}{N_2}k_2n_2}e^{-j\frac{2\pi}{N_1}k_1n_1}\right)e^{-j\frac{2\pi}{N_0}k_0n_0}. \tag{2.9}$$

The algorithm is called the slab-pencil algorithm since first a 2D discrete Fourier transform is applied followed by a 1D discrete Fourier transform. In this example, we grouped the first two dimensions. However, the last dimensions can be grouped as well. In addition, due to the commutativity property of the Kronecker product the order of operations can be changed and hence other dimensions can be grouped to give other algorithms. All these re-orderings and groupings have effects on performance. In the following sections, we show how to capture this information using a domain specific language and in the following chapters we discuss how to optimize data movement when dealing with high dimensional DFTs applied on high dimensional data points.

## 2.2 Why Use the Fourier Transform?

Most sequences or functions can be expressed as a sum of scaled harmonics or complex exponentials. In the case of a discrete time sequence $x[n]$ of size $N$, the sequence can be written as

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{j\frac{2\pi}{N}kn}, \tag{2.10}$$

where $X[k]$ terms are the complex coefficients computed using the forward DFT applied on the sequence $x[n]$. The importance of expressing any sequence as a sum of complex exponentials can be outlined in the context of the convolution operation, or circular convolution since the DFT intrinsically assumes that the sequences are periodic with fundamental period $N$.

The circular convolution of two discrete sequences $x[n]$ and $h[n]$ is expressed as

$$y[n] = x[n] \circledast h[n] = \sum_{k=0}^{N-1} x[n-k]h[k]. \tag{2.11}$$

The result of the convolution is another discrete sequence $y[n]$ that shows how the sequence $x[n]$ is modified by the sequence or system $h[n]$. It can be seen that convolution operation requires one of the sequence to be inverted in time. For example, in the above formula the sequence $x[n]$ is inverted as $x[n-k]$, where $k$ is the summation variable. Changing the input sequence $x[n]$ to a complex exponential of the form $e^{j\frac{2\pi}{N}ln}$ modifies the circular convolution as follows

$$\begin{aligned} y_e[n] &= e^{j\frac{2\pi}{N}n} \circledast h[n] \\ &= \sum_{k=0}^{N-1} e^{j\frac{2\pi}{N}l(n-k)} h[k] \\ &= e^{j\frac{2\pi}{N}ln} \sum_{k=0}^{N-1} e^{-j\frac{2\pi}{N}lk} h[k] \\ &= e^{j\frac{2\pi}{N}ln} H[i]|_{i=l}, \end{aligned} \tag{2.12}$$

where $H[i]$ represents the complex coefficients of the discrete sequence $h[n]$. $y_e[n]$ the result of the circular convolution is a scaled version of the original input, where $H[i]$ is evaluated at $i = l$.

This hints at eigenvalues and eigenvectors, where given a linear operator $A_m$, then the following holds

$$Av = \lambda v \tag{2.13}$$

where $\lambda$ is a scalar. Therefore, instead of computing the matrix-vector multiplication, one can easily scale the vector $v$ with the scalar $\lambda$. The same can be said about the circular convolution. Instead of computing the circular convolution with the complex exponential $e^{j\frac{2\pi}{N}kl}$, one can pre-compute the values $H[i]$ and simply scale the complex exponentials. This can be extended to arbitrary sequences $x[n]$

that can be expressed as a sum of scaled complex exponentials. The convolution operator is a linear operator, which means that the circular convolution can be applied to each term independently as shown in equation **??**. Computing the complex coefficients $X[k]$ and $H[k]$ with the help of the DFT the circular convolution becomes the well know point-wise multiplication

$$\begin{aligned} y[n] &= x[n] \circledast h[n] \qquad (2.14) \\ &= \frac{1}{N}\left(\sum_{k=0}^{N-1} X[k]e^{j\frac{2\pi}{N}kn}\right) \circledast h[n] \\ &= \frac{1}{N}\sum_{k=0}^{N-1} X[k]\left(e^{j\frac{2\pi}{N}kn} \circledast h[n]\right) \\ &= \frac{1}{N}\sum_{k=0}^{N-1} \underbrace{(X[k]\,H[k])}_{\text{point-wise multiplication}} \quad e^{j\frac{2\pi}{N}kn} \\ &= \frac{1}{N}\sum_{k=0}^{N-1} Y[k]e^{j\frac{2\pi}{N}kn}, \end{aligned}$$

where $Z[k] = X[k]Y[k]$ represents the complex coefficients for the discrete sequence $y[n]$, the result of the circular convolution.

The convolution becomes a point-wise multiplication in the frequency domain. However, the convolution operation is not the only operation that becomes a point-wise computation in the frequency domain. Other properties of the DFT as shown in Table 2.1 simplify computation to point-wise operations. In the following we present discuss in more detail the DFT properties that reduce computation in one domain to point-wise operations in the dual domain.

| Name | Time domain $\leftrightarrow$ Frequency domain |
|---|---|
| Linearity | $z[n] = x[n] + y[n] \leftrightarrow Z[k] = X[k] + Y[k]$ |
| Time Shift | $y[n] = x[n-\alpha] \leftrightarrow Y[k] = e^{-j\frac{2\pi}{N}\alpha k}X[k]$ |
| Frequency Shift | $y[n] = e^{\frac{2\pi}{N}n\alpha}x[n] \leftrightarrow Y[k] = X[k-\alpha]$ |
| Convolution | $z[n] = x[n] \circledast y[n] \leftrightarrow Z[k] = X[k] \cdot Y[k]$ |
| Multiplication | $z[n] = x[n] \cdot y[n] \leftrightarrow Z[k] = \frac{1}{N}X[k] \circledast Y[k]$ |
| Correlation | $z[n] = x[n] \star y[n] \leftrightarrow Z[k] = X[k] \cdot Y^*[k]$ |
| Time Difference | $\sum_{i=0}^{M-1}\beta_i y[n-i] = \sum_{i=0}^{P-1}\alpha_i x[n-i] \leftrightarrow Y[k] = \frac{\sum_{i=0}^{P-1}\alpha_i e^{-j\frac{2\pi}{N}ki}}{\sum_{i=0}^{M-1}\beta_i e^{-j\frac{2\pi}{N}ki}}X[k]$ |
| Differentiation | $y(t) = \frac{d}{dt}x(t)\vert_{t=kn} \leftrightarrow Y[k] = X[k] \cdot H_{\text{Diff}}[k]$ |

Table 2.1: The main DFT properties.

### 2.2.1 Linearity

The Fourier transform is a linear operator. Given any linear combination

$$y[n] = \sum_{i=0}^{p-1}\alpha_i x_i[n], \tag{2.15}$$

where $x_0[n]$, $x_1[n]$, …, $x_{p-1}[n]$ are sequences that have the same length as $y[n]$ and $\alpha_0$, $\alpha_1$, …, $\alpha_{p-1}$ are constants, the DFT of the sequence $y[n]$ is be computed as

$$Y[k] = \sum_{i=0}^{p-1}\alpha_i X_i[k], \tag{2.16}$$

where the $X_i[k]$ represents the DFT of each of the components.

Although this property does not require any point-wise multiplications, it is still an important property when dealing with combining and separating sequences. Combining sequences may prove useful when number of DFT computations need to be reduced. Instead applying the DFT $p$ times on each of the components $x_i[n]$, the

DFT is applied once on the resultant sequence. Computing the stress and strain of a material [47] makes use of this property. Once in frequency domain, they do the point-wise computation, however they also reduce the dimensions of the sequence. Overall they reduce the number of inverse DFT computations required to get the sequence back to the time domain.

Combining sequences is also useful when computing the DFT of real sequences. The DFT of real sequences has conjugate symmetry. This implies that computation can be reduced by almost half since only half of the complex coefficients are required. Therefore, two sequences can be combined to form a complex sequence and one complex DFT is required for computation. Given two real sequences $x[n]$ and $y[n]$, we can define $z[n]$ as

$$z[n] = x[n] + jy[n]. \tag{2.17}$$

Using linearity, the DFT of $z[n]$ is computed as

$$Z[k] = X[k] + jY[k], \tag{2.18}$$

where $X[k]$, $Y[k]$ and $Z[k]$ represent the Fourier representations for the three sequences. Using the conjugate symmetry properties of the DFT coefficients, $X[k]$ and $Y[k]$ can be extracted by performing some extra computation.

### 2.2.2 Time and Frequency Shift

The time shift property refers to delaying or advancing a sequence by a specific amount $\alpha$. Shifting the sequence in time translates into the frequency domain to a point-wise multiplication with complex exponentials. Given a sequence $x[n]$ of length $N$ with $X[k]$ the DFT coefficients, the DFT coefficients of the the delayed sequence $y[n] = x[n - \alpha]$ is computed as

$$Y[k] = e^{-j\frac{2\pi}{N}k\alpha}X[k]. \tag{2.19}$$

Discrete time sequences are only defined at integer value locations. However, multiplying the frequency representation of a given sequence with the complex exponential $e^{-j\frac{2\pi}{N}\frac{1}{2}k}$ means shifting the time domain representation by half a sample. The result of the operation constructs the interpolated sequence. This means that the result gives the samples between the integer values. Therefore multiplying the complex representation of sequence with the complex exponential $e^{-j\frac{2\pi}{N}\alpha k}$, where $\alpha$ is between 0 and 1 shifts the sequence by a non-integer number.

The Fourier transform is a dual operator. There is an equivalent operation for the frequency domain. Hence, any shift in frequency translates to a complex multiplication in the time domain. For example, the relationship $Y[k] = X[k - \alpha]$ between two frequency coefficients translates in time domain to a point-wise complex multiplication as follows

$$y[n] = e^{j\frac{2\pi}{N}n\alpha}x[n]. \tag{2.20}$$

The operation is similar with the difference that the sign changes. Shifting in frequency by $-\alpha$ means that in time domain representation is multiplied by the complex exponential $e^{j\frac{2\pi}{N}\alpha n}$, where the $-$ sign is dropped.

### 2.2.3 Convolution and Modulation

The convolution property is one of the most important properties when it comes to the Fourier transform. Equation 2.14 shows how the time domain circular convolution becomes a point-wise multiplication in the frequency domain. The circular convolution assumes that the two sequence are periodic. However, a linear convolution can be cast in term of circular convolutions if the two sequences are padded with zero. For example, if both sequences $x[n]$ and $y[n]$ of length $N$ are padded with zeros to a length of at least $2N$ then the circular convolution becomes a linear convolution.

Similar to the time/frequency shift property, there is an equivalent property for doing convolution in the frequency domain. In other words, computing a scaled circular convolution in frequency domain translates to a point-wise multiplication in time domain such as

$$z[n] = x[n] \cdot y[n] \iff Z[k] = \frac{1}{N} X[k] \circledast Y[k], \tag{2.21}$$

where $N$ represents the length of the two sequences. Either in time or in frequency convolution is simplified to a point-wise multiplication.

### 2.2.4 Correlation

The correlation is also a widely used operation to measure the similarity or difference between two input sequences. The correlation operation is similar to the convolution operation. In contrast to the convolution operation, none of the sequence requires a time reversal. The correlation operation is defined as

$$z[n] = x[n] \star y[n] = \sum_{k=0}^{N-1} x[k]y[k+n] \tag{2.22}$$

Similar to the convolution property, the DFT simplifies the computation by converting the correlation operation to a point-wise multiplication as

$$Z[k] = X[k] \cdot Y^*[k], \tag{2.23}$$

where $Y^*[k]$ represents the complex conjugate of the frequency representation $Y[k]$. Complex conjugate requires flipping the sign in front of the imaginary component of the complex number. Given $c = a + jb$, the complex conjugate $c^*$ is defined as $a - jb$.

### 2.2.5 Time Difference

Time differences appear frequently in linear systems where summations of shifted versions of the input sequence are equal to summations of shifted versions of the output sequence. In other words two discrete time sequences $x[n]$ and $y[n]$ are connected by the following formula

$$\sum_{i=0}^{M-1} \beta_i y[n-i] = \sum_{i=0}^{P-1} \alpha_i x[n-i], \tag{2.24}$$

the $\alpha_i$ and $\beta_i$ values are constant values. Solving such systems may become cumbersome. However, using the DFT, linearity and time shift properties computation is significantly simplified. In the frequency domain the expression becomes

$$Y[k] = \frac{\sum\limits_{i=0}^{P-1} \alpha_i e^{-j\frac{2\pi}{N}ki}}{\sum\limits_{i=0}^{M-1} \beta_i e^{-j\frac{2\pi}{N}ki}} X[k]. \tag{2.25}$$

It can be seen that once again the values of $Y[k]$ are computed by point-wise multiplying the frequency representation $X[k]$ with a constant value. The sum $\frac{\sum_{i=0}^{P-1} \alpha_i e^{-j\frac{2\pi}{N}ki}}{\sum_{i=0}^{M-1} \beta_i e^{-j\frac{2\pi}{N}ki}}$ may become complicated. However, the values can be pre-computed. The summation may even have a closed form solution in which case it can be computed on the fly, and no extra storage is required. Finally, to compute the time domain sequence the inverse DFT is applied on the result.

### 2.2.6 Trigonometric Interpolation and Differentiation

All computations on a computer are done on discretized data. Therefore, differentiation must also be done on discretized data. "Notes on FFT-Based differentiation" [48] presents algorithms for simulating the differentiation operator on discrete sequences using the DFT. The algorithms are based on the idea that the DFT is an approximation of the Fourier series described by the equation

$$y(t) = \sum_{k=-\infty}^{\infty} Y[k] e^{\frac{2\pi}{N}kt}. \tag{2.26}$$

Applying differentiation on the continuous function gives

$$\frac{\mathrm{d}}{\mathrm{d}t} y(t) = \sum_{k=-\infty}^{\infty} \left( j\frac{2\pi}{N}k \cdot Y[k] \right) e^{\frac{2\pi}{N}kt}. \tag{2.27}$$

This suggests that the Fourier coefficients of the derived function $\frac{\mathrm{d}}{\mathrm{d}t}y(t)$ can be computed by point-wise multiplying the coefficients of the original function with constant values.

This result seems appealing since it follows the computation pattern discussed so far in this dissertation. However, computing the derivative requires the continuous function and converting the discrete sequence to the continuous function may not be possible. In such cases, differentiation must be applied on the interpolated values of the sequence. One simple interpolation is the one where the time-shift property is used. Recall that shifting in time a sequence by a value $\alpha$ requires a complex multiplication in the frequency domain. Moreover, if $\alpha$ is chosen between 0 and 1 then the samples between the original samples can be obtained. This interpolation though has significant errors due to aliasing. The sequences obtained from interpolation are not unique. Now for the sequence itself, aliasing is not a problem, however the differentiated sequence may suffer from significant errors. More details can be found in [48, 49].

An alternative solution is to use a better trigonometric interpolation that minimizes errors and the so called oscillations between the samples. One interesting property of the DFT is the trigonometric interpolation polynomial defined as

$$y(t) = Y[0] + \sum_{0<k<\frac{N}{2}} \left( Y[k] e^{j\frac{2\pi}{N}kt} + Y[N-k] e^{-j\frac{2\pi}{N}kt} \right) + Y_{\frac{N}{2}} \cos(\pi t), \tag{2.28}$$

where the $Y[k]$ coefficients represent the DFT of coefficients for the sequence $y[n]$. The $Y_{\frac{N}{2}}$ term appears when the value $N$ is even. If $N$ is odd the term disappears. This interpolation has two interesting properties. First, the polynomial consists of sinusoids that have the smallest possible values for the frequencies and thus the smallest oscillations. Second, if the original sequence $y[n]$ is real then the polynomial

**Algorithm 2** Computing the first-order derivative of a discrete time sequence $y[n]$ using the DFT.

```
1: function FIRSTDERIV($x, y, N$)    ▷ Where x - input array, y - output array, N - length
2:     $X = \text{dft}(x)$
3:     for $k = 0$ to $N - 1$ do
4:         if (k < N/2) $Y[k] = j2\pi k/NX[k]$
5:         if (k == N/2) $Y[k] = 0$
6:         if (k > N/2) $Y[k] = j2\pi(k - N)/NX[k]$
7:     end for
8:     $y = \text{idft}(Y)$
9: end function
```

only returns real samples. Details about the interpolation polynomial can be found in [46].

Using the trigonometric interpolation polynomial, one can determine and define the specific algorithms for computing the first and second order derivatives for the discrete sequence $y[n]$. Taking the first order derivative of the polynomial and sampling it at sample rate of $N$ samples, the discrete derived sequence can be computed as

$$y'[n] = \sum_{0<k<\frac{N}{2}} j\frac{2\pi}{N}k\left(Y[k]e^{j\frac{2\pi}{N}kt} - Y[N-k]e^{-j\frac{2\pi}{N}kt}\right). \tag{2.29}$$

Given the inverse DFT where $y'[n] = \sum_{k=0}^{N-1} Y'[k]e^{j\frac{2\pi}{N}kn}$ and using pattern matching, the $Y'[k]$ coefficients can be computed. The $Y'[k]$ coefficients represent the DFT coefficients of the derived sequence. Using algorithm 2, one can simulate the first order derivative of the discrete time sequence $y[n]$. Similarly, the second order can be computed. Applying the derivation step a second time and again sampling the function at $N$ samples the second order derivative can be computed following algorithm 3. There are other methods that provide more accurate results for computing the derivative [9].

**Algorithm 3** Computing the second-order derivative of a discrete time sequence $y[n]$ using the DFT.

```
function secondDeriv($x, y, N$) ▷ Where x - input array, y - output array, N - length
    $X = \mathrm{dft}(x)$
    for $k = 0$ to $N - 1$ do
        if (k <= N/2) $Y[k] = -[2\pi k/N]^2 X[k]$
        if (k > N/2) $Y[k] = -[2\pi(k - N)/N]^2 X[k]$
    end for
    $y = \mathrm{idft}(Y)$
end function
```

It can be seen that all these operations and properties require a forward Fourier transform, followed by a point-wise multiplication and an inverse Fourier transform. The properties are shown in the context of 1D operations, however the properties can easily be extended to higher dimensions. All follow the pattern emphasized in this dissertation. This shows that although the DFT is an important computational kernel, the focus should be on the algorithms that use the DFT as a building block. Optimizing the algorithms as a whole rather than optimizing the DFT in isolation may prove to be more beneficial. Using a domain specific language to express the entire computation allows one to reason about what optimizations can be applied across the compute stages. In the following section we introduce the SPL language already used to describe DFT computation. However, we use to describe DFT-base operations and apply optimizations across the compute stages. More details about the optimizations can be found in Chapter 3 and 4.

## 2.3 SPL, the Language to Represent DFTs

In this section, we focus on the language used to describe Fourier transforms and algorithms that decompose the Fourier transform. The language we use to capture various algorithms is called the Signal Processing Language (SPL) [26, 27]. The language is built around the Kronecker product [50] and structured sparse matrices. In the following paragraphs we first present the Kronecker product and the structured sparse matrices like diagonal or permute matrices. We then introduce the DFT and some of the algorithms used to decompose the DFT computation using the SPL notation. We set the ground for the next chapter, where we use the SPL notation to express DFT-based operations and show optimizations can easily applied. In Chapter 3, we mostly focus on the step by step process of merging the compute stages. While in Chapter 4, we present some other low level optimizations required to obtain competitive implementations.

### 2.3.1 Kronecker Product

The Kronecker product also know the tensor product is the back-bone of the Signal Processing Language (SPL). Given two dense square matrices $A_m \in \mathrm{R}^{m\times m}$ and $B_n \in \mathrm{R}^{n\times n}$, the Kronecker product is defined as

$$A_m \otimes B_n = \begin{bmatrix} a_{0,0}B_n & a_{0,1}B_n & \dots & a_{0,m-1}B_n \\ a_{1,0}B_n & a_{1,1}B_n & \dots & a_{1,m-1}B_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0}B_n & a_{m-1,1}B_n & \dots & a_{m-1,m-1}B_n \end{bmatrix}, \tag{2.30}$$

where $a_{i,j}$ are the elements within the $A_m$ matrix and the product $a_{i,j}B_n$ represents the multiplication of each element of $B_n$ with the scalar element $a_{i,j}$. The result is a larger matrix of size $mn \times mn$. $A_m$ and $B_n$ are square matrix. However, the Kronecker product accepts rectangular matrices as well, e.g. $A_{n,m} \in \mathrm{R}n \times m$.

First, if the $A_m$ matrix is replaced with the identity matrix then the above

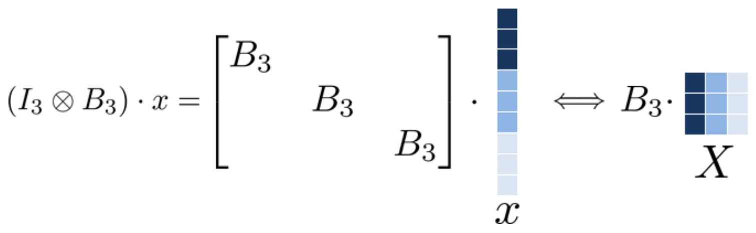

$(I_3 \otimes B_3) \cdot x =$
$B_3$
$B_3$
$B_3$
$x$
$\Longleftrightarrow B_3 \cdot$
$X$

Figure 2.2: The figure on the left shows the first interpretation of the construct $I_m \otimes B_n$ where the matrix $B_n$ is multiplied with $n$ contiguous blocks. The figure on the right shows the same construct, however viewed as a matrix-matrix multiply.

expression becomes

$$I_m \otimes B_n = \begin{bmatrix} B_n & O & \dots & O \\ O & B_n & \dots & O \\ \vdots & \vdots & \ddots & \vdots \\ O & O & \dots & B_n \end{bmatrix}, \tag{2.31}$$

where $O$ is the zero matrix that has the same size as $B_n$. If the $I_m \otimes B_n$ construct is applied on a vector $x$ of size $mn$, then the matrix $B_n$ is applied on continuous blocks of size $n$. The matrix $B_n$ is applied $m$ times on $m$ disjoint blocks. A common practice is to view the 1D array $x$ of size $mn$ as a 2D array $X$ of size $n \times m$ stored in column major. Then the $I_m \otimes B_n$ construct can be viewed as multiplying the matrix $B_n$ with the matrix $X$ as seen in figure 2.2. The $I_m \otimes B_n$ construct is well suited for thread parallelism [33], where each thread applies the same operation $B_n$ on its own disjoint portion of the array $x$.

Second, if matrix $B_n$ is replaced by the identity matrix $I_n$ then the Kronecker

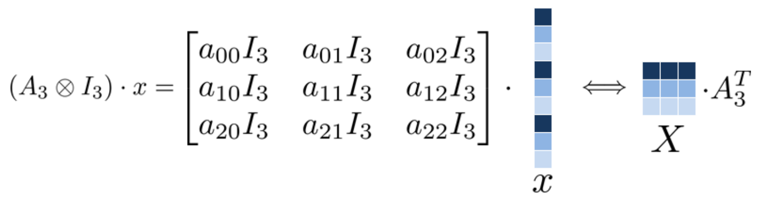

$(A_3 \otimes I_3) \cdot x = \begin{bmatrix} a_{00}I_3 & a_{01}I_3 & a_{02}I_3 \\ a_{10}I_3 & a_{11}I_3 & a_{12}I_3 \\ a_{20}I_3 & a_{21}I_3 & a_{22}I_3 \end{bmatrix} \cdot$
$x$
$\Longleftrightarrow$
$\cdot A_3^T$
$X$

Figure 2.3: The figure on the left shows the first interpretation of the construct $A_m \otimes I_n$ where the matrix $A_m$ is multiplied with elements located at a stride of $n$. The figure on the right shows the matrix interpretation.

product changes again

$$A_m \otimes I_n = \begin{bmatrix} a_{0,0}I_n & \dots & a_{0,m-1}I_n \\ \vdots & \ddots & \vdots \\ a_{m-1,0}I_n & \dots & a_{m-1,m-1}I_n \end{bmatrix}. \tag{2.32}$$

Each value of $A_m$ is multiplied with the identity matrix. Since only the diagonal of the identity is non-zero, all the elements of $A_m$ are replicated on the diagonals. The construct is used for representing operations that favor SIMD instructions [51]. If the construct $A_m \otimes I_n$ is multiplied with an array $x$ of size $mn$, then matrix $A_m$ is multiplied with data points located at a stride of $n$ elements apart. Similarly splitting the 1D array into a 2D matrix $X$ of size $n \times m$ stored in column major, the $A_m \otimes I_n$ construct can be viewed as the matrix multiplication between $X$ and $A_m$ as shown in figure 2.3. Note that for this construct matrix $A_m$ appears on the right hand size of the multiplication.

The Kronecker product is separable. Given two matrix $A_n$ and $B_m$, the tensor product between $A_n$ and $B_m$ can be separated as

$$A_n \otimes B_m = (A_n \otimes I_m) \cdot (I_n \otimes B_m)\,, \tag{2.33}$$

where the identity matrices have the same size as $A_n$ and $B_m$. This property suggests that the overall operation can be split into two stages, where matrix $B_m$ is applied on contiguous data and matrix $A_n$ is applied on strided data.

The Kronecker product allows compute stages to be merged. Given two matrices $A_n$ and $C_n$, where the number of columns of $A_n$ is equal to the number of rows of $C_n$, fusing computation is done as

$$(A_n \otimes I_m) \cdot (C_n \otimes I_m) = ((A_n \cdot C_n) \otimes I_m) \tag{2.34}$$

$$(I_m \otimes A_n) \cdot (I_m \otimes C_n) = (I_m \otimes (A_n \cdot C_n)) \tag{2.35}$$

This two properties are useful when dealing with DFT-based convolutions. The idea is that decomposing the forward and inverse DFT properly and exposing such constructs, stages can be automatically be merged. There are other properties; for more details the reader is pointed towards the following papers [52].

### 2.3.2 Specialized Matrices

Constructs can be built by simply using the identity matrix along side the Kronecker product. However, besides the identity matrix, SPL offers other specialized matrices such as diagonal matrices or various permutation matrices.

The diagonal matrix $D_n$ defined as

$$D_n = \begin{bmatrix} d_{0,0} & 0 & \dots & 0 \\ 0 & d_{1,1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_{n-1,n-1} \end{bmatrix}, \tag{2.36}$$

is the matrix where only the diagonal elements are non-zero. This matrix is used to compute a point-wise multiplication or scaling operation. The elements on the diagonal can take any values. However, in most cases, the values are pre-determined and most often pre-computed. The diagonal matrix can also be combined with the identity matrix through the Kronecker product. This translates to duplicating elements in different forms. For example, the construct $I_m \otimes D_n$ creates $m$ replicas of $D_n$, while $D_n \otimes I_m$ duplicates each element of $D_n$ $m$ such that the duplicate elements are consecutive.

SPL allows various permutation matrices that shuffle the input data based on a given permutation. For example, the $L$ matrix defined as

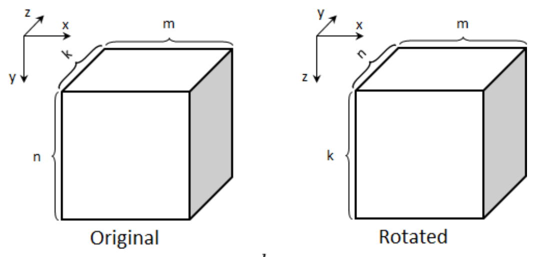
$$L_m^{mn} : in + j \mapsto jm + i, \quad 0 \leq i < m,\ 0 \leq j < n \tag{2.37}$$

represents the stride permutation. The $L$ operator is applied on a 1D array. However, viewing the input data as 2D array of size $n \times m$, the $L$ operator transposes the original matrix into a matrix of size $m \times n$. The $L$ operator commutes the $A_m$ and $B_n$ matrix within the Kronecker product as follows

$$(A_m \otimes B_n) = L_m^{mn} (B_n \otimes A_m) L_n^{mn} \tag{2.38}$$

$$(A_m \otimes B_n) L_m^{mn} = L_m^{mn} (B_n \otimes A_m) \tag{2.39}$$

Transposing or applying the Hermition symmetry on $L_m^{mn}$, gives another operator $L_n^{mn}$ that transposes the matrix from $m\times n$ to $n\times m$ as shown in Equation 2.40. Multiplying the same $L$ operator with the transposed version of itself give the identity matrix as shown in equation 2.41. The transposition matrices can be decomposed into block transpositions and transpositions within blocks as shown in Equation 2.42



$$L_n^{nk} \otimes I_m$$

Figure 2.4: The figure presents a block transposition.

and Equation 2.43.

$$\left(L_m^{mn}\right)^H = \left(L_m^{mn}\right)^T = L_n^{mn} \tag{2.40}$$

$$\left(L_m^{mn}\right)^H \cdot L_m^{mn} = I_{mn} \tag{2.41}$$

$$L_n^{kmn} = \left(L_n^{kn} \otimes I_m\right) \cdot \left(I_k \otimes L_n^{mn}\right) \tag{2.42}$$

$$L_{km}^{kmn} = \left(I_k \otimes L_m^{mn}\right) \cdot \left(L_k^{kn} \otimes I_m\right). \tag{2.43}$$

Transposing blocks or transposing within blocks is expressed the Kronecker product between the $L$ operator and the identity matrix $I$. For example, the $I_k \otimes L_m^{mn}$ construct suggests applying the permutation matrix locally on $k$ chunks of data, while the $L_n^{nk} \otimes I_m$ construct performs a permutation of blocks of size $m$ as shown in figure 2.4.

SPL allows users to define their own permutations. For example, the following two permutations

$$W_p : i \mapsto g^i \bmod p, \quad 0 \leq i < p, \tag{2.44}$$

$$V_n : i \mapsto \left(k \left\lfloor \frac{i}{k} \right\rfloor + m\,(i \bmod k)\right) \bmod n, \quad 0 \leq i < p.$$

are used later in this section within algorithms like Rader or Good-Thomas, used to decompose the DFT under specific conditions.

### 2.3.3 SPL Constructs and Extensions

The Kronecker products $A_m \otimes I_n$ and $I_m \otimes B_n$ replicate the elements from the same matrix $A_m$ and $B_m$, respectively. However, there are case that will be discussed in Chapter 3, where matrices need to be blocked into disjoint and distinct blocks. The blocks are different from each other. A more general construct that allows indexing is the direct sum $\bigoplus$ operator. Given $m$ matrices $B_n^{(i)}$, the direct sum is defined as

$$\bigoplus_{i=0}^{m-1} B_n^{(i)} = \begin{bmatrix} B_n^{(0)} & O & \dots & O \\ O & B_n^{(1)} & \dots & O \\ \vdots & \vdots & \ddots & \vdots \\ O & O & \dots & B_n^{(m-1)} \end{bmatrix}. \tag{2.45}$$

Each matrix $B_n^{(i)}$ is applied on disjoint data points. Now, if all the matrices $B_n^{(i)}$ are equal to the same matrix $B_n$, then the direct sum is identical to the Kronecker product $I_m \otimes B_n$ as

$$\bigoplus_{i=0}^{m-1} B_n = \begin{bmatrix} B_n & O & \dots & O \\ O & B_n & \dots & O \\ \vdots & \vdots & \ddots & \vdots \\ O & O & \dots & B_n \end{bmatrix}. \tag{2.46}$$

The direct sum can be used to describe the Kronekcer-equivalent construct for $A_m \otimes I_n$, when the matrix $A_m$ depends on an index value $i$. Recall that the permutation matrix $L$ commutes the Kronecker product such that $A_m \otimes I_n = L_m^{mn} \left(I_n \otimes A_m\right) L_n^{mn}$, where the construct $I_n \otimes A_m$ is replaced with $\bigoplus_{i=0}^{n-1} A_m$. This suggests that given $n$ distinct matrices $A_m^i$, the equivalent construct can be written using the direct sum as

$$L_m^{mn} \cdot \left( \bigoplus_{i=0}^{n-1} A_m^{(i)} \right) \cdot L_n^{mn}. \tag{2.47}$$

The direct sum is useful when blocking the diagonal matrix into disjoint components such as

$$D_{mn} = \bigoplus_{i=0}^{m-1} D_n^{(i)}, \tag{2.48}$$

where $D_n^{(i)}$ represents a disjoint part of the main diagonal. Using this property the diagonal can be fused with the surrounding computation when computing a DFT-based operation. The forward and inverse DFT dictate how to block the diagonal.

Some operations require zero padding and un-padding. For example, the construct

$$I_{m \times n} = \begin{bmatrix} I_n \\ O_{m-n \times n} \end{bmatrix}, \quad m \geq n, \tag{2.49}$$

where $O_{m-n \times n}$ is a rectangular matrix with all elements equal to 0, defines the zero-padding operation. The matrix $I_{m \times n}$ reads an array of size $n$ and appends $m - n$ 0s to it. The transposed version $I_{n \times m}$ where $m \geq n$ does the opposite, un-padds the array. Such operators are useful when dealing with odd size DFTs and DFT-based operations as shown in the implementation of the phase shift interpolation [3] presented in Chapter 4.

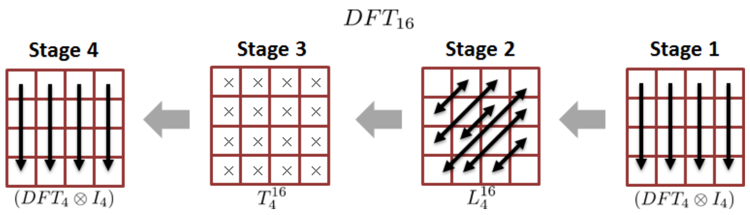


Figure 2.5: The four stages of a DFT of size 16. The input data is viewed as a matrix of size $4 \times 4$. The first stage applies the DFT in the columns, the second stage transposes the matrix, the third stage does a Hadamard product and finally the last stage applies the DFT again in the columns.

### 2.3.4 The Decomposition of the 1D DFT

The DFT of $n$ input samples $x_0, \ldots, x_{n-1}$ is defined as the matrix vector product $y = \mathrm{DFT}_n x$ with

$$\mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n} \quad \text{with} \quad \omega_n = \exp(-2\pi j/n). \tag{2.50}$$

Computing the DFT as a matrix-vector multiplication incurs a $O(n^2)$ arithmetic complexity. Fast Fourier transform (FFT) algorithms, such as the Cooley-Tukey algorithm, reduce complexity to $O(n \log(n))$. The Cooley-Tukey algorithm works when the problem size $n$ is a composite number. However, for prime numbers algorithms like Rader or Bluestein need to be used. Basically the Rader and Bluestein algorithm are DFT based convolutions on data that is padded with zeros.

In this section, we present the different algorithms used to decompose the DFT. All the algorithms are expressed in terms of SPL. We start with the Cooley-Tukey algorithm [53] that can be applied on composite problem sizes where $N = mn$. The algorithm can be expressed using SPL as follows

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) \cdot T_n^{mn} \cdot L_m^{mn} \cdot (\mathrm{DFT}_n \otimes I_m)\,. \tag{2.51}$$

The algorithm recursively decomposes the DFT of size $mn$ into smaller DFTs of size $m$ and $n$. If $n$ and $m$ are composite numbers the Cooley-Tukey decomposes the $\text{DFT}_m$ and $\text{DFT}_n$. The algorithm stops when it hits the base cases. For power of two problem size the base case is the so called butterfly matrix $\text{DFT}_2$ defined as

$$\text{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.52}$$

For composite numbers that are not powers of two, the base cases are prime sizes for which the Cooley-Tukey algorithm cannot be applied. Other algorithms like Rader, Bluestein can be applied. In some cases for small size prime numbers it is beneficial to simply compute the matrix-vector multiplication directly.

The 1D DFT is applied on 1D arrays. However, based on the decomposition of the 1D DFT using Equation 2.51, the input array can be viewed as a 2D matrix, where the size of the matrix is $n \times m$ given the decomposition of the problem $N = mn$. Hence, the four stages of the DFT decomposition can easily be explained as operations on the 2D matrix representation. The first stage applies the $\text{DFT}_n$ on the input column direction $m$ times. The data then is transposed from a matrix of size $n \times m$ to a matrix of size $m \times n$. The result is scaled by the values stored in the diagonal matrix $T_n^{mn}$ defined as

$$T_n^{mn} = [e^{-j\frac{2\pi}{mn}kl}], \text{ where } 0 \leq k < m-1 \text{ and } 0 \leq l < n-1, \tag{2.53}$$

the so-called twiddle diagonal matrix. Finally, the last stage applies the $\text{DFT}_m$ on the column direction $n$ times.

Each DFT decomposition adds another dimension to the data-set. Decomposing the $\text{DFT}_{mn}$ allowed one to view the computation as being applied on a 2D

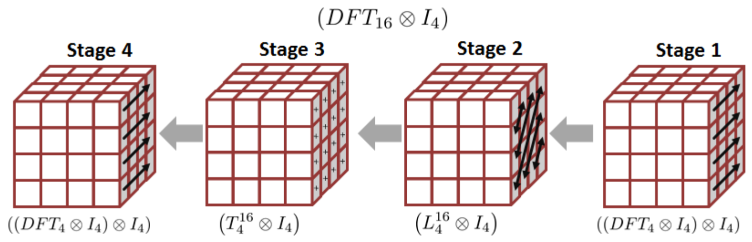


Figure 2.6: Recursively decomposing the $(\mathrm{DFT}_m \otimes I_n)$ gradually adds another dimension to the data set. The DFT computation always starts with the newly added dimension.

matrix. Decomposing the children recursively increases the number of dimensions as shown in Figure 2.6. In addition, the computation of the smaller DFTs is always applied in the newly created dimension. Assuming data is stored in row-major order, the DFT computation always applies the DFTs in the slowest dimension in memory. This implies that data is always accessed at large strides. This are covered in more details in Chapter 4, where we discuss optimizations to reduce the effects of non-unit strided accesses.

There are other algorithms for powers of two such as Stockham or Pease. All these algorithms can be expressed using SPL and are simply different decompositions of the main problem. The Stockham [54, 26] algorithm, for example, prefers sorting the data as computation is performed, while Pease has very regular computation that maps nicely to field-programmable gate arrays (FPGAs) implementations [54, 26], however it requires an expensive permutation before computation can proceed. More details about the algorithms and thei representation can be found in Van Loan [26]. It is though important to state that there are ways of going from one algorithm to the other by modifying the SPL formulation and using the Kronecker properties.

There are other algorithms that decompose DFT computation. However, the different decompositions work if certain conditions about the problem size are satisfied. For example, Good-Thomas or the Prime Factor algorithm can only be applied if the problem size $N = mn$ satisfies $gcd(m, n) = 1$, where $gcd$ represents the greatest common divisor. The algorithm has at its core the Chinese Reminder Theorem and can be formulated again using SPL as follows

$$\mathrm{DFT}_{mn} = V_n^T \cdot (\mathrm{DFT}_m \otimes I_n) \cdot (I_m \otimes \mathrm{DFT}_n) \cdot V_n, \tag{2.54}$$

where the $V_n$ matrix is the shuffle matrix defined above. The decomposition is similar to the Cooley-Tukey algorithm. However, one key difference is that the twiddle factors disappear at the cost of more expensive data shuffling operations before and after the computation.

For prime numbers either Rader [55] or Bluestein [56, 57] can be applied to compute the frequency representations of a given sequence. Both algorithms perform a convolution. Since the convolution can be computed using the Fourier transform, both algorithms resort to a DFT-based convolution approach. The Rader algorithm for a given prime size $p$ is defined as

$$\mathrm{DFT}_p = W_p^{-1} \cdot (I_1 \oplus \mathrm{iDFT}_{p-1}) \cdot E_p \cdot (I_1 \oplus \mathrm{DFT}_{p-1}) \cdot W_p, \tag{2.55}$$

where $W_p$ is the shuffle matrix defined previously, $\oplus$ represents the mathematical representation of the direct sum, and $E_p$ represents a pseudo-diagonal matrix with two additional off diagonal elements. The representation gives a strong resemblance to a convolution operation of size $p - 1$ with some extra operations for the $0^{\text{th}}$ frequency element. The downside of the Rader algorithm is that the input and output shuffle operations are expensive since the mapping function is a power function followed by a modulo function of prime numbers, e.g mod $p$ where $p$ is prime. On most modern architectures power functions incur long latencies.

Bluestein is an alternative to the Rader algorithm. Given a prime size $p$, the Bluestein algorithm computes a convolution of size $n \geq 2p - 1$. The algorithm increases the problem size to sizes that may benefit from more efficient implementations like those offered by the Cooley-Tukey algorithm. The algorithm requires some extra operations before performing the DFT-based convolution. The entire algorithm can easily be expressed using the SPL notation as

$$\mathrm{DFT}_p = B'_p \cdot I_{p \times n} \cdot (\mathrm{iDFT}_n \cdot D_n \cdot \mathrm{DFT}_n) \cdot I_{n \times p} \cdot B_p, \tag{2.56}$$

where $B'_p = B_p^H$ and $B_p$ is the diagonal matrix defined as

$$B_p = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & e^{-j\frac{\pi}{p}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots e^{-j\frac{\pi(p-1)}{p}} & \end{bmatrix} \tag{2.57}$$

The two matrices are required to pre and post-scale the input and the output. The two matrices $I_{p \times n}$ and $I_{n \times p}$ are the constructs that perform the padding and unpadding of data from the size $p$ to the size $n$. The operation in the middle can easily be identified as a DFT-based convolution, where the matrix $D_n$ is a diagonal matrix. $D_n$ has a closed form solution since it is computed from a circulant matrix defined by the Bluestein algorithm. Both Rader and Bluestein are built from DFT-based convolution, therefore merging compute stages should provide improvements.

### 2.3.5 Extending to Higher Dimensions

All of the algorithms described above focused on the 1D DFT decomposition. However, the language can also be used to capture higher dimensional DFTs since higher dimensional DFTs have a matrix like representation. Recall that $p$-dimensional DFT is defined as the $p$ sums

$$X[k_0,\ldots,k_{p-1}] = \sum_{n_0=0}^{N_0} \cdots \sum_{n_{p-1}=0}^{N_{p-1}} x[n_0,n_1,\ldots,n_{p-1}]e^{-j2\pi\left(\frac{k_0 n_0}{N_0}+\ldots+\frac{k_{p-1}n_{p-1}}{N_{p-1}}\right)}. \tag{2.58}$$

Given the product of exponents, the DFT matrix for a $p$-dimensional DFT is represented as

$$\mathrm{DFT}_{n_0\times n_1\times\ldots\times n_{p-1}} = \mathrm{DFT}_{n_0}\otimes\mathrm{DFT}_{n_0}\otimes\ldots\otimes\mathrm{DFT}_{n_{p-1}}, \tag{2.59}$$

where $n_0$, $n_1$, …, $n_{p-1}$ represent the sizes of the $p$ dimensions. One can use the properties of the Kronecker product to separate and group the stages together. Therefore algorithms such as the pencil-pencil or slab-pencil decomposition can be expressed using SPL.

We continue by briefly describing the algorithms used for the 2D and 3D DFT. We start with the 2D DFT and expand to the 3D DFT. The 2D DFT is expressed as a dense matrix $\mathrm{DFT}_{n\times m}$ that can be decomposed using the Kronecker as

$$\mathrm{DFT}_{n\times m} = \mathrm{DFT}_n \otimes \mathrm{DFT}_m.$$

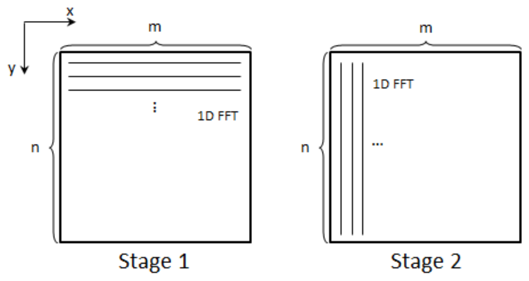


Figure 2.7: The implementation of the 2D DFT algorithm. Data is viewed as a 2D matrix of size $n \times m$, with the $x$-dimension laid out in the fast dimension in memory. First 1D DFTs are applied in the rows and then 1D DFTs are applied in the columns. Image is reproduced from [1].

Using the separation properties, the 2D DFT can be further decomposed as

$$\mathrm{DFT}_{n\times m} = \underbrace{\left(\mathrm{DFT}_n \otimes I_m\right)}_{\text{Stage 2}} \cdot \underbrace{\left(I_n \otimes \mathrm{DFT}_m\right)}_{\text{Stage 1}}.$$

The overall operation described mathematically above is depicted in Figure 2.7. The data is viewed as a $n \times m$ matrix stored in row-major order. The first stage of the 2D DFT applies 1D DFTs of size $m$ in the rows, whereas the second stage applies 1D DFTs of size $n$ in the columns direction. The above decomposition of the 2D DFT is called the pencil-pencil decomposition, where each pencil refers to a 1D DFT applied in each dimension. Recall that the 1D DFTs require strided access, therefore for large 2D DFTs each of the pencils require non-unit stride memory accesses.

Similarly, using SPL one can decompose the 3D DFT. The 3D DFT is also a matrix $\mathrm{DFT}_{k\times n\times m}$ that can be decomposed using the separability property and

SPL representation as

$$\mathrm{DFT}_{k\times n\times m} = \underbrace{\left(\mathrm{DFT}_k \otimes I_{mn}\right)}_{\text{Stage 3}} \cdot \underbrace{\left(I_k \otimes \mathrm{DFT}_n \otimes I_m\right)}_{\text{Stage 2}} \cdot \underbrace{\left(I_{kn} \otimes \mathrm{DFT}_m\right)}_{\text{Stage 1}}.$$

Data is viewed as a 3D cube of size $k \times n \times m$, stored in row major order with the $x$-dimension corresponding to the size $m$ laid out in the fastest memory dimension. The pencil-pencil-pencil decomposition of the 3D DFT applies 1D DFTs in each of the three dimensions. Figure 2.8 shows the pencil-pencil-pencil implementation of the 3D DFT algorithm. However, stages can be grouped together and algorithms such as the slab-pencil decomposition can be expressed. Grouping the last DFTs applied in the $m$ and $n$ dimension one gets the slab-pencil decomposition described once again using SPL as

$$\mathrm{DFT}_{k\times n\times m} = \underbrace{\left(\mathrm{DFT}_k \otimes I_{mn}\right)}_{\text{Stage 2}} \cdot \underbrace{\left(I_k \otimes \left(\left(\mathrm{DFT}_n \otimes I_m\right) \cdot \left(I_n \otimes \mathrm{DFT}_m\right)\right)\right)}_{\text{Stage 1}}.$$

Choosing between pencil-pencil-pencil and slab-pencil decompositions depends on the problem size and the underlying architecture. Optimizations and more details about the decomposition are discussed in Chapter 4.

## 2.4 Summary

In this chapter, we talked about the Fourier transform and some of its properties. We mostly focused on the discrete Fourier transform (DFT) for signals or sequences of any dimension. We presented some of the basic DFT properties where it was

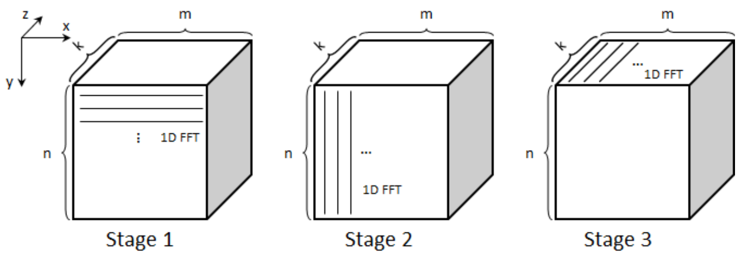


Figure 2.8: The implementation of the 3D DFT algorithm. Data is viewed as a 3D cube of size $k \times n \times m$, with the $x$-dimension laid out in the fast dimension in memory. The 1D DFTs are applied in all three dimensions one by one. Image is reproduced from [1].

shown that complicated operations become point-wise computations after applying the DFT. This suggested that although the Fourier transform is an important mathematical kernel, the focus should be on DFT-based operations. Hence, frameworks should focus on capabilities of expressing and optimizing entire DFT-based operations. Therefore, we discuss in Chapter 4 some implementation details on how to build a simple API to allow users to express most common DFT-based operations.

In the second part of the chapter, we have presented the high level mathematical language that we use to represent DFT-based operations. The SPL representation has successfully been used to represent multi-dimensional DFTs and various algorithms like Cooley-Tukey, Rader or Bluenstein. However, we extend its functionality by looking at DFT-based operations. Having a high level language to express entire algorithms enables more opportunities for optimization. In Chapter 3, we discuss applying loop fusion across DFT and non-DFT stages and we use the SPL language to outline the step by step process of achieving efficient stage merging.

# Chapter 3

# Fusing the DFT with the Surrounding Computation

The goal of this dissertation is to apply optimizations across the DFT and non-DFT stages in a systematic way. One such optimization is loop fusion [58]. Current implementations of any DFT-based operation keeps the DFT and non-DFT stages separate. Each stage requires a full pass through the data and as the problem size increases data needs to be read from the lower levels of the memory hierarchy, incurring longer latencies. This becomes a problem especially for memory bound computations like the DFT and the point-wise operation. Hence, fusing the DFT and non-DFT stages improves data locality and thus overall performance.

In the first part of this chapter, we focus on the definition of loop fusion and the necessary conditions required by two or more loops to satisfy in order for loop fusion to be applied. Typically a compiler needs to analyze the loops and decide whether loop fusion can be applied. However, due to problems like pointer aliasing, complicated loops and complicated indexing, they lack the necessary information and most likely loops will not get merged. As such, in this work we resort to using a mathematical representation, namely SPL, to represent the computation and apply optimizations like loop fusion at this level of abstraction. Therefore, in the second part of the chapter we outline the systematic approach of fusing the compute stages for both 1D and multi-dimensional DFT-based operations. We provide a step by step description of how to efficiently merge the compute stages.

## 3.1 From SPL to Loop Fusion

In this section, we talk about loop fusion and the difficulties a compiler must handle in order to determine whether loops can be fused or not. Due to problems like pointer aliasing, complicated loops and complicated indexing a compiler cannot efficiently fuse loops. However, expressing the computation in a high level mathematical language one can reason on how to merge the DFT and non-DFT stages.

### 3.1.1 Loop Fusion

Loop fusion is a compiler optimization and one of the seven loop transforms [58, 20, 59]. As the name says, the transform fuses loops together to improve data locality. However, in order to fuse loops the following conditions must be satisfied:

- the loop iteration count must match
- the data dependencies must be preserved

For example, the two loops in Figure 3.1a can be merged into one single loop as shown in Figure 3.1b. The two loops have the same iteration count and the data dependencies are not broken once the loops are fused. Fusing the loops also improves data locality since the value $b[i]$ is updated and immediately used

```c
int a[32], b[32], c[32];

for(int i = 0; i != 16; ++i) {
  b[i] = a[i] + 3;
}

for(int i = 0; i != 16; ++i) {
  c[i] = b[i] + 5;
}
```

(a) Two loops that can be fused.

```c
int a[32], b[32], c[32];

for(int i = 0; i != 16; ++i) {

  b[i] = a[i] + 3;

  c[i] = b[i] + 5;

}
```

(b) The result of fusing the code in 3.1a.

```c
int a[32], b[32], c[32];

for(int i = 0; i != 16; ++i) {
  b[i] = a[i] + 3;
}

for(int i = 0; i != 16; ++i) {
  c[i] = b[i + 1] + 5;
}
```

(c) Two loops that cannot be fused.

```c
int a[32], b[32], c[32];

for(int i = 0; i != 16; ++i) {

  b[i] = a[i] + 3;

  c[i] = b[i + 1] + 5;

}
```

(d) The result of fusing the code in 3.1b.

Figure 3.1: Example of loops that can be and cannot be fused.

in the next instruction. However, fusion cannot be applied on the loops shown in Figure 3.1c. Even though the two loops have the same iteration count, fusing them breaks the dependencies. The second loop requires the value of `b[i + 1]` which needs to be computed before it can be used. If the loops are left as is, the first loop fully updates the array *b* and the second loop reads the updated version of the array. However, if the loops are fused as shown in Figure 3.1d, each iteration updates `b[i]` but `b[i + 1]` does not get the updated value until the following iteration. Fusing the loops in Figure 3.1c requires other loop transformations like loop peeling, thus increasing the likelihood that the compiler will not optimize the code.

In addition to the extra steps required to perform loop fusion, the code may also be more complex. Applications typically have complicated loops with

```
complex in[16], t[16], out[16];

for(int i = 0; i != 4; ++i) {
  complex *inx = (in + 4 * i);
  complex *tx = (t + 4 * i);

  dft_comp(4, inx, 1, tx, 1);
}

for(int i = 0; i != 4; ++i) {
  complex *tx = (t + 4 * i);
  complex *outx = (out + 4 * i);

  dft_comp(4, tx, 1, outx, 1);
}
```

(a) Two loops around a DFT of size 4.

```
complex in[16], t[16], out[16];

for(int i = 0; i != 4; ++i) {
  complex *inx = (in + 4 * i);
  complex *tx = (t + 4 * i);

  dft_comp(4, inx, 1, tx, 1);
}

for(int i = 0; i != 4; ++i) {
  complex *tx = (t + i);
  complex *outx = (out + i);

  dft_comp(4, tx, 4, outx, 4);
}
```

(b) Two loops around a DFT of size 4.

Figure 3.2: Example of more complicated loops. The loop in Figure 3.2a can be fused, while the loop in Figure 3.2b cannot be fused.

complicated memory access patterns. For example, Figure 3.2a and Figure 3.2b show two code snippets where batches of DFTs are applied one after the other. Each batch is basically a loop around a DFT function call. The DFT computation is represented by the function `dft_comp`, which takes as arguments the size of the DFT, the input and output arrays and their corresponding leading dimensions. Because of the extra components like the offsets into the arrays and the leading dimensions, the compiler cannot identify which loops need to be merged. However, using a high level mathematical representation to represent the loops and the computation can alleviate the problem of how to fuse the stages. Therefore, in this work we opt for using the SPL notation to represent and optimize the computation.

### 3.1.2 Fusion in the Context of SPL

In Chapter 2, we presented the SPL notation and some of the constructs. Each SPL construct has a loop based equivalent as shown in [25, 31]. For example, the

```
complex x[16], y[16];

for(int i = 0; i != 4; ++i) {
  complex *xf = (x + 4 * i);
  complex *yf = (x + 4 * i);

  dft(4, xf, 1, yf, 1);
}
```

(a) The C/C++ code representation of the $I_4 \otimes \mathrm{DFT}_4$ construct.

```
complex x[16], y[16];

for(int i = 0; i != 4; ++i) {
  complex *xf = (x + i);
  complex *yf = (x + i);

  dft(4, xf, 4, yf, 4);
}
```

(b) The C/C++ code representation of the $\mathrm{DFT}_4 \otimes I_4$ construct.

Figure 3.3: The C/C++ loop equivalent of some of the SPL constructs.

construct $I_4 \otimes \mathrm{DFT}_4$ which applies a $\mathrm{DFT}_4$ on four contiguous blocks of size four is translated into a loop of four iterations around a $\mathrm{DFT}_4$ as shown in Figure 3.3a. Since the DFT is applied on contiguous blocks of data, the leading dimension is one and the offset into the input and output arrays is four. Similarly, the construct $\mathrm{DFT}_4 \otimes I_4$ which applies a $\mathrm{DFT}_4$ on four elements located at a stride distance of four elements is translated into a loop of four iterations around a the $\mathrm{DFT}_4$ as shown in Figure 3.3b. For this construct, the leading dimension is equal to four and the offset into the arrays is equal to one.

Given this mapping, the loops in Figure 3.2a and Figure 3.2b can be rewritten using the SPL as follows. The code snippet in Figure 3.2a is translated into SPL as

$$(I_4 \otimes \mathrm{DFT}_4) \cdot (I_4 \otimes \mathrm{DFT}_4)\,, \tag{3.1}$$

while the code snippet in Figure 3.2b is translated into SPL as

$$(I_4 \otimes \mathrm{DFT}_4) \cdot (\mathrm{DFT}_4 \otimes I_4)\,. \tag{3.2}$$

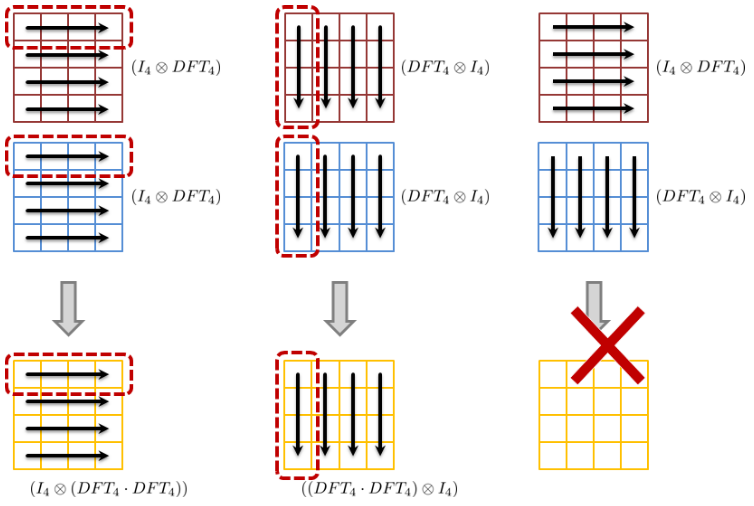


Figure 3.4: The Kronecker product and fusing the compute stages. The first two constructs can be merged since the DFTs are applied in the same direction. The last constructs cannot be merged since the DFTs are applied in different directions.

Looking at the Kronecker product properties described in Chapter 2, only the expression in Equation 3.1 can be merged since both DFTs are applied in the same direction. The expression in Equation 3.2 cannot be merged since the DFTs in each stage access data in different directions as shown in Figure 3.4. The first stage requires elements at a stride while the second stage requires contiguous elements. The second stage needs to wait for first stage finish computation before starting its computation. Fusing the two stages would break the dependencies, similar to how the dependencies were broken when fusing the code in Figure 3.1c. This implies that SPL constructs can be merged if the expressions have the following form

$$(I_m \otimes A) \cdot (I_m \otimes C) = (I_m \otimes (A \cdot C)), \tag{3.3}$$

$$(B \otimes I_n) \cdot (D \otimes I_n) = ((B \cdot D) \otimes I_n), \tag{3.4}$$

where the number of columns in $A$ is equal to the number of rows in $C$ and the number of columns in $B$ is equal to the number of rows in $D$.

The fusion operation can be extended to the direct sum. Recall that the direct sum is similar to the Kronecker product just that the construct deals with operators that are dependent on an iteration count. Given two operators $A^{(i)}$ and $B^{(i)}$ the fusion between the two direct sum is expressed as

$$\left(\bigoplus_{i=0}^{n-1} A^{(i)}\right) \cdot \left(\bigoplus_{i=0}^{n-1} B^{(i)}\right) = \bigoplus_{i=0}^{n-1} \left(A^{(i)} \cdot B^{(i)}\right). \tag{3.5}$$

The iteration count for the direct sums must match and the number of columns in matrix $A^{(i)}$ must be equal to the number of rows in $B^{(i)}$ for all values of $i$. Fusion can also be done between the direct sum and the Kronecker product as follows

$$(I_n \otimes A) \cdot \left(\bigoplus_{i=0}^{n-1} B^{(i)}\right) = \bigoplus_{i=0}^{n-1} \left(A \cdot B^{(i)}\right) \tag{3.6}$$

$$\left(\bigoplus_{i=0}^{n-1} B^{(i)}\right) \cdot (I_n \otimes A) = \bigoplus_{i=0}^{n-1} \left(B^{(i)} \cdot A\right) \tag{3.7}$$

Recall that the Kronecker product $(I_n \otimes A)$ is a direct sum where the matrix $A$ does not depend on the index $i$. Again the conditions for being able to merge the two constructs is that the iteration counts must be the same and the number of columns in $A$ must be equal to the number of rows in $B^{(i)}$ for all values of $i$.

## 3.2 Systematic Way of Merging the DFT and non-DFT Stages

Most DFT-based operations follow a specific pattern, namely a forward DFT is applied on the input data, followed by a point-wise multiplication and finally an inverse DFT. The entire algorithm can be represented using SPL as

$$\mathrm{Conv}_N = \mathrm{iDFT}_N \cdot D_N \cdot \mathrm{DFT}_N, \tag{3.8}$$

where $D_N$ represents the diagonal matrix required for performing the point-wise multiplication. $D_N$ stores the pre-computed values required for the point-wise computation. Having a wider view of the entire algorithm and knowing up-front the decompositions of the forward and inverse DFTs, we make the argument that there is a systematic way of merging the DFT and non-DFT stages. In the following sections, we cover the following

- the systematic approach of merging the stages within a 1D DFT-based convolution for problem sizes that are power of two
- the systematic approach of merging the stages within a 1D DFT-based convolution for prime number problem sizes
- the systematic approach of merging the stage within multi-dimensional DFT-based operations

### 3.2.1 Merging the Compute Stages for 1D DFT-based Convolutions for Power of Two Problem Sizes

In this subsection, we discuss the step by step approach of fusing the compute stages for the 1D DFT-based convolution. Since all other computations boil down to a point-wise computation in frequency domain, the steps for merging can be extended to the other computations. We assume that the problem size $N = 2^p$ is a power of two. Hence, the forward and inverse transforms can be decomposed using the Cooley-Tukey algorithm. Since the forward and inverse DFT are independent, there are cases that depended on the decompositions. $N$ can have different factorizations which implies different algorithms. For example, if $N = 8$ then one factorization is $n_1 = 2$ and $n_2 = 4$ and another factorization is $n_3 = 4$ and $n_4 = 2$. For a problem size $N = n_1 n_2$ the forward DFT can be decomposed as

$$\mathrm{DFT}_N = \left(\mathrm{DFT}_{n_1} \otimes I_{n_2}\right) \cdot T^{n_1 n_2}_{n_2} \cdot L^{n_1 n_2}_{n_1} \left(\mathrm{DFT}_{n_2} \otimes I_{n_1}\right), \tag{3.9}$$

while the inverse DFT can be decomposed following a different factorization of $N = n_3 n_4$ and

$$\mathrm{iDFT}_N = \left(\mathrm{iDFT}_{n_3} \otimes I_{n_4}\right) \cdot {T^{n_3 n_4}_{n_4}}^{*} \cdot L^{n_3 n_4}_{n_3} \left(\mathrm{iDFT}_{n_4} \otimes I_{n_3}\right). \tag{3.10}$$

The inverse DFT follows a similar decomposition of the as the forward DFT. The only difference is that the twiddle factors must be complex conjugated. Therefore, the decomposition of the iDFT can be thought as first decomposing a forward DFT following the factorization $N = n_3 n_4$ and then complex conjugating each term in the twiddle diagonals.

The twiddle diagonals $T^{n_1 n_2}_{n_2}$ and ${T^{n_3 n_4}_{n_4}}^{*}$ and the permutation matrices $L^{n_1 n_2}_{n_1}$ and $L^{n_3 n_4}_{n_3}$ can be fused with the DFT computation following the steps outline in [31]. The paper does not show how to merge the DFT computation. It only covers the fusion of various permutations and point-wise computations. However, looking closely at the decompositions of the forward and inverse DFTs, it can be seen that the right child of the inverse DFT, the diagonal matrix representing the point-wise computation and the left child of the forward DFT can further be merged

$$\begin{gathered}(\mathrm{iDFT}_{n_3} \otimes I_{n_4}) \cdot {T_{n_4}^{n_3 n_4}}^* \cdot L_{n_3}^{n_3 n_4} \\ \underbrace{(\mathrm{iDFT}_{n_4} \otimes I_{n_3}) \cdot D_N \cdot (\mathrm{DFT}_{n_1} \otimes I_{n_2})}_{\text{can be fused}} \cdot \\ T_{n_2}^{n_1 n_2} \cdot L_{n_1}^{n_1 n_2} (\mathrm{DFT}_{n_2} \otimes I_{n_1}) .\end{gathered} \tag{3.11}$$

Given the two DFT decompositions, the relationship $n_1 n_2 = n_3 n_4 = N$ must be satisfied. Based on the values $n_1$, $n_2$, $n_3$ and $n_4$ there are two cases that are worth investigating:

**1.** $n_2 > n_3$ **or** $n_2 < n_3$**:** Let's assume that $n_2 > n_3$. Since $n_2 > n_3$ and $N$ is a power of two that can be factorized as $N = n_1 n_2 = n_3 n_4$, there exists an integer $p > 1$ such that $n_2 = n_3 p$. The only constructs from within the DFT-based convolution that can be merged, are

$$(\mathrm{iDFT}_{n_4} \otimes I_{n_3}) \cdot D_N \cdot (\mathrm{DFT}_{n_1} \otimes I_{n_2}) . \tag{3.12}$$

Replacing the value $n_2$ with $n_3 p$ in above equation gives

$$(\mathrm{iDFT}_{n_4} \otimes I_{n_3}) \cdot D_N \cdot (\mathrm{DFT}_{n_1} \otimes I_{n_3 p}) . \tag{3.13}$$

Based on the property that $I_{n_3 p} = I_{n_3} \otimes I_p$ and the associative property of the Kronecker product, the above formula can be re-written as

$$(\mathrm{iDFT}_{n_4} \otimes I_{n_3}) \cdot D_N \cdot ((\mathrm{DFT}_{n_1} \otimes I_p) \otimes I_{n_3}) . \tag{3.14}$$

Now, using the fact that the permutation matrix $L$ commutes the tensor product, we can re-write the above construct as

$$L_{n_4}^{n_3n_4} \cdot (I_{n_3} \otimes \mathrm{iDFT}_{n_4}) \cdot L_{n_3}^{n_3n_4} \cdot D_N \cdot L_{n_1p}^{n_1pn_3} \cdot (I_{n_3} \otimes (\mathrm{DFT}_{n_1} \otimes I_p)) \cdot L_{n_3}^{n_1pn_3}, \quad (3.15)$$

where $n_1p = n_4$. The middle construct $L_{n_3}^{n_3n_4} \cdot D_N \cdot L_{n_1p}^{n_1pn_3}$ simply shuffles the data within the diagonal matrix as shown in Figure 3.5. Let $D'_N = L_{n_3}^{n_3n_4} \cdot D_N \cdot L_{n_1p}^{n_1pn_3}$ be the diagonal that stores the shuffled data. Replacing the new diagonal in the above expression gives

$$L_{n_4}^{n_3n_4} \cdot (I_{n_3} \otimes \mathrm{iDFT}_{n_4}) \cdot D'_N \cdot (I_{n_3} \otimes (\mathrm{DFT}_{n_1} \otimes I_p)) \cdot L_{n_3}^{n_1pn_3}. \quad (3.16)$$

The two constructs $I_{n_3} \otimes DFT_{n_4}$ and $I_{n_3} \otimes (DFT_{n_1} \otimes I_p)$ have the same identity matrix $I_{n_3}$ and the both $DFT_{n_4}$ and $(DFT_{n_1} \otimes I_p)$ are square matrices of size $n_4 \times n_4$. This suggests that the two constructs can be merged following the idea presented earlier in this chapter. In order to fuse the two constructs, the diagonal matrix needs to be blocked into $n_3$ disjoint chunks of size $n_4$. Using the direct sum, the diagonal $D'_N$ is re-expressed as

$$D'_N = \left( \bigoplus_{i=0}^{n_3-1} D'^{(i)}_{N/n_3} \right), \quad (3.17)$$

where $D'^{(i)}_N$ represent a disjoint component of size $n_4$ from the main diagonal. Replacing the diagonal with the direct sum expression gives

$$L_{n_4}^{n_3n_4} \cdot \left( \bigoplus_{i=0}^{n_3-1} \mathrm{iDFT}_{n_4} \right) \cdot \left( \bigoplus_{i=0}^{n_3-1} D'^{(i)}_{N/n_3} \right) \cdot \left( \bigoplus_{i=0}^{n_3-1} (\mathrm{DFT}_{n_1} \otimes I_p) \right) \cdot L_{n_3}^{n_3n_4} \quad (3.18)$$

It can be seen that the iteration count of all three direct sums is equal to $n_3$. In addition, the matrices within each direct sum have the same size. All constructs

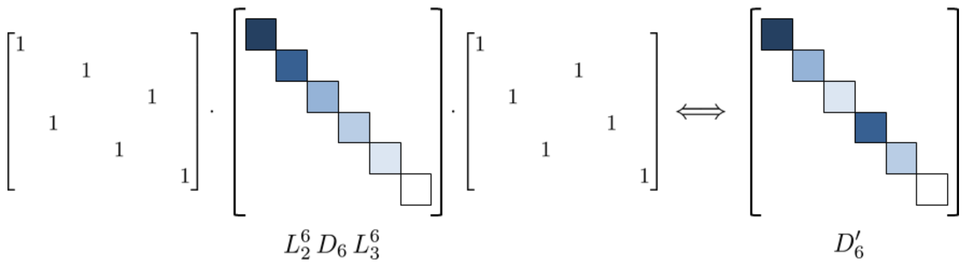


Figure 3.5: Shuffling the data within the diagonal for the SPL construct $L_{n_3}^{n_3 n_4} \cdot D_{N/n_3} \cdot L_{n_1 p}^{n_1 p n_3}$.

are matrices of size $n_4 \times n_4$. Following the ideas described in the beginning of this chapter the three stages can be merged as

$$L_{n_4}^{n_3 n_4} \cdot \left( \bigoplus_{i=0}^{n_3 - 1} \left( \mathrm{iDFT}_{n_4} \cdot D_{N/n_3}^{\prime(i)} \cdot (\mathrm{DFT}_{n_1} \otimes I_p) \right) \right) \cdot L_{n_3}^{n_3 n_4} \tag{3.19}$$

The permutation matrices before and after the direct sum suggest that the operation within the direct sum is applied on strided elements. If the input data is viewed as a 2D matrix of size $n_4 \times n_3$, the the operation $\left( \mathrm{iDFT}_{n_4} \cdot D_{N/n_3}^{\prime(i)} \cdot (\mathrm{DFT}_{n_1} \otimes I_p) \right)$ is applied in the columns of the matrix.

The stages within the construct $\mathrm{iDFT}_{n_4} \cdot D_{N/n_3}^{\prime(i)} \cdot (\mathrm{DFT}_{n_1} \otimes I_p)$ can also be merged if the $\mathrm{iDFT}_{n_4}$ can further be decomposed. However, the merging process requires some extra steps:

1. the $\mathrm{iDFT}_{n_4}$ must be decomposed using $n_4 = n_{41} n_{42}$

$$(\mathrm{iDFT}_{n_{41}} \otimes I_{n_{42}}) \cdot {T_{n_{42}}^{n_{41} n_{42}}}^{*} \cdot L_{n_{41}}^{n_{41} n_{42}} \cdot (\mathrm{iDFT}_{n_{42}} \otimes I_{n_{41}}) \cdot D_{N/n_3}^{\prime(i)} \cdot (\mathrm{DFT}_{n_1} \otimes I_p) \tag{3.20}$$

2. the right child of the newly decomposed inverse DFT can be merged given

that $n_{41}n_{42} = n_1 p$

$$(\mathrm{iDFT}_{n_{41}} \otimes I_{n_{42}}) \cdot T_{n_{42}}^{n_{41}n_{42}^*} \cdot L_{n_{41}}^{n_{41}n_{42}} \cdot \underbrace{(\mathrm{iDFT}_{n_{42}} \otimes I_{n_{41}}) \cdot D'^{(i)}_{N/n_3} \cdot (\mathrm{DFT}_{n_1} \otimes I_p)}_{\text{can be fused}} \tag{3.21}$$

3. the three stages are similar to the ones in equation 3.12, thus the the merging process repeats by analyzing whether $n_{41}$ is smaller, equal or greater than $p$.

The steps are repeated until the problem sizes are DFTs of size two, namely the base cases are hit.

If $n_2 < n_3$ then the process of merging is very similar. Given that $n_2 < n_3$ and $N$ is a power of two that can be factorized as $N = n_1 n_2 = n_3 n_4$, there exists an integer $q$ such that $n_3 = qn_2$. Replacing this in equation 3.12 gives

$$(\mathrm{iDFT}_{n_4} \otimes I_{qn_2}) \cdot D_N \cdot (\mathrm{DFT}_{n_1} \otimes I_{n_2}) \,. \tag{3.22}$$

The left child can be grouped such that $\mathrm{iDFT}_{n_4} \otimes I_{qn_2} = (\mathrm{iDFT}_{n_4} \otimes I_q) \otimes I_{n_2}$. This exposes the $I_{n_2}$ term, which is the same for both the left and right child. Repeating the steps from the case where $n_2 > n_3$, the compute stages are fused as

$$L_{n_1}^{n_1 n_2} \cdot \left( \bigoplus_{i=0}^{n_2 - 1} \left( (\mathrm{DFT}_{n_4} \otimes I_q) \cdot D''^{(i)}_{N/n_2} \cdot \mathrm{iDFT}_{n_1} \right) \right) \cdot L_{n_2}^{n_1 n_2}, \tag{3.23}$$

where $D''^{(i)}_{N/n_2}$ represents the blocked diagonal that stores the shuffled data points.

The main problem with this way of decomposing the forward and inverse DFT is that the children become unbalanced after the first fusion step as shown in Figure 3.6. One of the stages is a simple DFT while the other stage is a construct of the form $(\mathrm{DFT} \otimes I)$. Applying fusion in the subsequent stages involves some extra

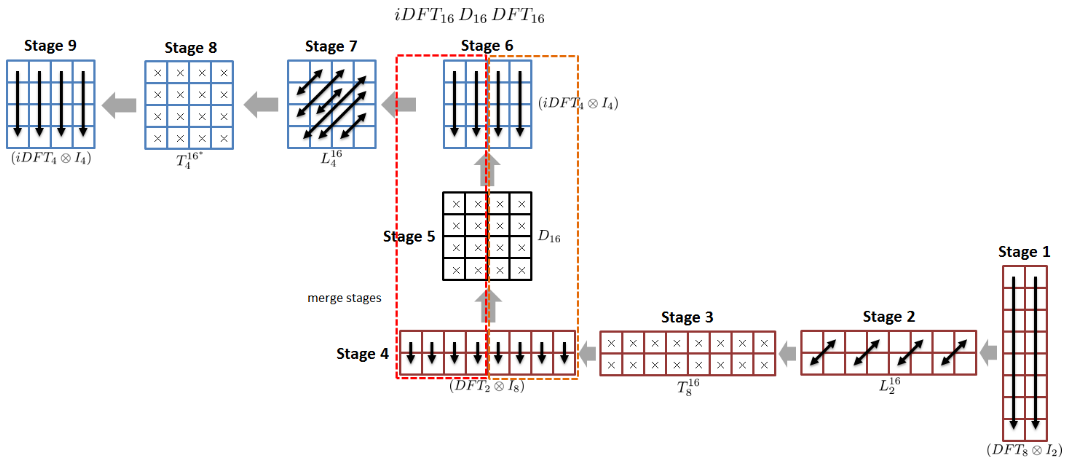


Figure 3.6: Zooming inside the convolution operation where the right child of the inverse DFT, the main diagonal and the left child of the forward DFT need to be merged. The data-sets need to be reshaped in order for the fusing operation to succeed, since the assumption is that $n_2 > n_3$.

operations as shown above. For example, given that $n_2 > n_3$, the left child must be decomposed first. The right child of the decomposition can be fused with the diagonal and the other construct. However, if $n_2 < n_3$ then the roles are flipped. The right child must be first decomposed, its left child must then be merged with the diagonal and right the construct.

**2.** $n_2 = n_3$**:** Under this scenario the forward and inverse DFT are mirror images of each other, more specifically the factorization of the inverse DFT is the mirror image of the factorization of the forward DFT. For example, for a problem size of $N = 8$ the inverse DFT decomposed using $n_1 = 4$ and $n_2 = 2$ is the mirror image of the forward DFT which is decomposed following $n_3 = 2$ and $n_4 = 4$. This suggests that the right child of the inverse DFT and the left child of the forward DFT are identical

$$(\mathrm{iDFT}_{n_1} \otimes I_{n_2}) \cdot D_N \cdot (\mathrm{DFT}_{n_1} \otimes I_{n_2})\,. \tag{3.24}$$

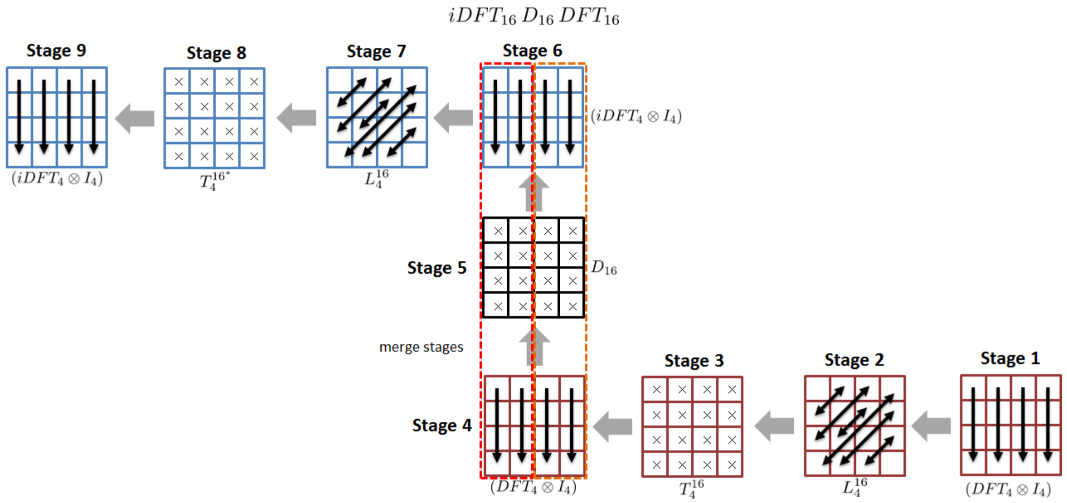


Figure 3.7: Zooming inside the convolution operation where the right child of the inverse DFT, the main diagonal and the left child of the forward DFT need to be merged. No reshaping is need since $n_2 = n_3$.

The left and right child have the same $\otimes I_{n_2}$, therefore stages can be merged. The steps discussed for the previous cases are repeated for this case as well. First the tensor products are commuted using the $L$ operator. Second the data within the diagonal matrix $D_N$ is shuffled as shown in Figure 3.5. $D_N$ is surrounded by the $L$ matrices that commuted the tensor products. The diagonal $D'_N$ that stores the shuffled data is blocked into $n_2$ disjoint chunks of size $n_1$ using the direct sum expression. The fused version of the expression from equation 3.24 follows

$$L_{n_2}^{n_1 n_2} \cdot \left( \bigoplus_{i=0}^{n_2-1} \left( \mathrm{iDFT}_{n_1} \cdot D'^{(i)}_{N/n_2} \cdot \mathrm{DFT}_{n_1} \right) \right) \cdot L_{n_1}^{n_1 n_2} \tag{3.25}$$

The fused stage $\left( \mathrm{iDFT}_{n_1} \cdot D'^{(i)}_{N/n_2} \cdot \mathrm{DFT}_{n_1} \right)$ looks like a 1D DFT-based convolution, however smaller than the original. Therefore, the process of merging the operations can be repeated.

Further optimizations can be brought to this decomposition. For example, the stages can be merged repeatedly. Keeping everything in mirror image alleviates the problem of deciding which of the stages needs to be first decomposed and then merged. Under this scenario, both stages are recursively decomposed and merged as shown in Figure 3.7.

Given $N = n_1 n_2 = n_3 n_4$ and the condition that $n_2 = n_3$, the decompositions of the forward and inverse DFT are tied together. However, a stronger condition can be imposed on the inverse DFT. It is know that the inverse transform is the conjugate transposed of the forward transform, e.g. $\text{iDFT} = \text{DFT}^H$, where $(\cdot)^H$ represents the conjugate transpose operator. Using this property and replacing the inverse DFT with the conjugate transposed version of the forward DFT gives

$$\begin{aligned} \text{Conv}_N = & \underbrace{\left(\text{DFT}_{n_2}^H \otimes I_{n_1}\right) \cdot L_{n_2}^{n_1 n_2}}_{\text{Stage3}} \cdot \\ & \underbrace{T_{n_2}^{n_1 n_2 *} \cdot \left(\text{DFT}_{n_1}^H \otimes I_{n_2}\right) \cdot D_N \cdot \left(\text{DFT}_{n_1} \otimes I_{n_2}\right)}_{\text{Stage2}} \cdot \\ & \underbrace{T_{n_2}^{n_1 n_2} \cdot L_{n_1}^{n_1 n_2} \cdot \left(\text{DFT}_{n_2} \otimes I_{n_1}\right)}_{\text{Stage1}} . \end{aligned} \tag{3.26}$$

Following the steps from [31] for fusing the twiddle and permutation matrices and applying the necessary loop transformations to fuse the computation from the second stage, the 1D DFT-based convolution is modified as follows

$$\begin{aligned} \text{Conv}_N = & L_{n_2}^{n_1 n_2} \cdot \left( \bigoplus_{i=0}^{n_1 - 1} \text{DFT}_{n_2}^H \right) \cdot \\ & L_{n_1}^{n_1 n_2} \cdot \left( \bigoplus_{i=0}^{n_2 - 1} T_{n_2}^{\prime n_1 n_2 * (i)} \cdot \text{DFT}_{n_1}^H \cdot D_{N/n_2}^{\prime (i)} \cdot \text{DFT}_{n_1} \right) \cdot L_{n_2}^{n_1 n_2} \cdot \\ & \left( \bigoplus_{i=0}^{n_1 - 1} T_{n_2}^{\prime n_1 n_2 (i)} \cdot \text{DFT}_{n_2} \right) \cdot L_{n_1}^{n_1 n_2} . \end{aligned} \tag{3.27}$$

It can be seen that the middle stage can be computed in-place. The reading and writing pattern are transposed versions of each other, therefore no extra buffer is required for storage. The computation surrounding the middle stage are transposed versions of each other. The right child reads data at a stride and outputs it sequential, while the left child reads data sequentially and outputs it at the same stride as the right child. This construct favors in place computation. In addition, the inverse DFT can simply be obtained by transposing the data-flow of the forward transform and subsequently complex conjugating all the twiddle matrices.

All of the approaches described in this subsection require the diagonal and twiddle matrices to be repartitioned, shuffled and reindexed. The problem is that as the diagonal matrix is fused with the computation around it, data must be shuffled. Recall that after reorganizing the DFT computation, the diagonal matrix is surrounded by $L$ operators. These operators permute data within the diagonal as shown in Figure 3.5. In the current implementation of the code generator, as the diagonal matrices get reshaped and repartition, we a book keeping of the new mapping in order to be able to construct the appropriate index mapping.

### 3.2.2 Merging the Compute Stages for 1D DFT-based Convolutions for Prime Number Problem Sizes

The Cooley-Tukey algorithm works for composite numbers of the form $N = mn$. However, if the problem size $N$ is prime, then algorithms like Rader or Bluestein fit better. Both algorithms are basically DFT-based convolutions that require extra permutations and data re-orderings. For example, Rader requires expensive permutations based on power functions and modulos with prime numbers. The implementation of these functions on most modern systems are expensive and incur long latencies. Bluestein on the other hand requires zero-padding and extra complex multiplications. Similarly, zero padding instructions are expensive. However, having a view of the entire 1D DFT-based convolution, knowing the decomposition of the forward and inverse DFT and imposing the condition that the iDFT $=$ DFT$^H$ can help to reduce redundant operations while merging the stages.

In this subsection, we present the step by step process of merging the compute stages when dealing with prime number problem sizes. We first focus on applications that use the Rader algorithm and then continue with the convolutions based on the Bluestein algorithm. For both cases we outline the possibilities of reducing the expensive operations and also the opportunities of merging the compute stages. The Rader algorithm decomposes the DFT for a prime problem size $p$ as follows

$$\mathrm{DFT}_p = W_p^{-1} \cdot (I_1 \oplus \mathrm{DFT}_{p-1}^H) \cdot E_p \cdot (I_1 \oplus \mathrm{DFT}_{p-1}) \cdot W_p, \tag{3.28}$$

where $W_p$ represents the Rader permutation matrix. The permutation is based on the function

$$W_p : i \mapsto g^i \bmod p, \quad 0 \leq i < p. \tag{3.29}$$

The power and modulo functions are expensive instructions. Using the property iDFT $=$ DFT$^H$, the 1D DFT-based convolution for prime sizes can be expressed with SPL as

$$\begin{aligned}\mathrm{Conv}_p = W_p^{-1} \cdot (I_1 \oplus \mathrm{DFT}_{p-1}^H) \cdot E_p^* \cdot \\ (I_1 \oplus \mathrm{DFT}_{p-1}) \cdot W_p \cdot D_p \cdot W_p^{-1} \cdot (I_1 \oplus \mathrm{DFT}_{p-1}^H) \cdot \\ E_p \cdot (I_1 \oplus \mathrm{DFT}_{p-1}) \cdot W_p,\end{aligned} \tag{3.30}$$

where $E_p^*$ represents the complex conjugate of $E_p$. Given the Rader permutation, the construct $W_p \cdot D_p \cdot W_p^{-1}$ shuffles the diagonal matrix $D_p$ into $D'_p$. The diagonal $D'_p$ can again be blocked using the direct sum. The difference now is that the direct sum blocks the matrix into two matrices of different sizes. The first matrix $D''_N$ represents the first element of the main diagonal, $D''_1$ is a $1 \times 1$ matrix. The second matrix $D'''_{p-1}$ represents the diagonal matrix that contains the remaining $p-1$ elements, the matrix is of size $(p-1) \times (p-1)$. The compute stages can be merged as follows

$$\begin{aligned} \text{Conv}_p = W_p^{-1} \cdot (I_1 \oplus \text{DFT}_{p-1}^H) \cdot E_p^* \cdot \\ \left(D''_1 \oplus \left(\text{DFT}_{p-1} \cdot D'''_{p-1} \cdot \text{DFT}_{p-1}^H\right)\right) \cdot \\ E_p \cdot (I_1 \oplus \text{DFT}_{p-1}) \cdot W_p \end{aligned} \tag{3.31}$$

The Rader-based 1D convolution gains from reducing the number of expensive permutations. Data still needs to be permuted at the beginning and end of the 1D DFT-based convolution, however no more permutations are required within the computation. The diagonal must be shuffled, however this can be done apriori. For convolutions that use the Bluestein algorithms, the merging simplifies data packing and reduces extra point-wise computation. Recall that the Bluestein algorithm increases the problem size from $p$ to $N > 2p - 1$ such as

$$\text{DFT}_p = B'_p \cdot I_{p \times N} \cdot \left(\text{DFT}_N^H \cdot D_N \cdot \text{DFT}_N\right) \cdot I_{N \times p} \cdot B_p, \tag{3.32}$$

where $B_p$, $B'_p$ are extra point-wise operations required to scale the input and output sequence and the $I_{N \times p}$ and $I_{p \times N}$ operators are used to perform zero-padding and un-padding. Once again, we use the fact that the inverse DFT is the conjugate transpose of the forward DFT, $\text{iDFT}_N = \text{DFT}_N^H$. This insight allows us to re-write the 1D convolution for the Bluestein decomposition as

$$\begin{aligned}\mathrm{Conv}_p = B_p^H \cdot I_{p\times N} \cdot \left(\mathrm{DFT}_N^H \cdot D_N^* \cdot \mathrm{DFT}_N\right) \cdot \\ I_{N\times p} \cdot B_p'^H \cdot G_p \cdot B_p' \cdot I_{p\times N} \cdot \\ \left(\mathrm{DFT}_N^H \cdot D_N \cdot \mathrm{DFT}_N\right) \cdot I_{N\times p} \cdot B_p,\end{aligned} \tag{3.33}$$

where $G_p$ represents the convolution point-wise computation. Since $D_N$ is a diagonal matrix and $B_p'^H$ and $B_p'$ are complex conjugate diagonal matrices of each other, $B_p'^H$ and $B_p'$ cancel themselves. This brings the $I_{N\times p}$ and $I_{p\times N}$ operators closer to the diagonal $G_p$. The operators simply increase the size of the diagonal $G_p$ to $N$ by appending zero elements to the $G_p$ diagonal. Let $G_N'$ be the diagonal that contains the 0 elements. This convolution is simplified as

$$\begin{aligned}\mathrm{Conv}_p = B_p^H \cdot I_{p\times N} \cdot \\ \left(\mathrm{DFT}_N^H \cdot D_N^* \cdot \mathrm{DFT}_N\right) \cdot G_N' \cdot \left(\mathrm{DFT}_N^H \cdot D_N \cdot \mathrm{DFT}_N\right) \cdot \\ I_{N\times p} \cdot B_p,\end{aligned} \tag{3.34}$$

where $G_N'$ represents the expanded diagonal matrix. Merging the compute stages for the Bluestein-based convolution reduces computation and also reduces data padding. In addition, it exposes more stages that can further be merged. Typically, the problem size $N$ is increased to a power of two so that the Cooley-Tukey algorithm can be used for the forward and inverse DFTs. Therefore, the above compute stages can be merged following the steps presented in the previous subsection.

### 3.2.3 Merging the Compute Stages for Multi-Dimensional DFT-based Convolutions

The step by step process of merging the compute stages can be extended to multi-dimensional DFT-based operations. We use the DFT-based convolution as an example, however we focus on the 2D DFT-based convolution. The steps can be extended to any multi-dimensional DFT. However, the 2D DFT-based convolution is sufficient to illustrate the cases where the point-wise computation exhibits symmetry properties. Depending on such properties the point-wise computation is non-separable or separable on the dimensions of the data-set. We explore and show the merging process for both cases.

**2D DFT-based operation with non-separable point-wise.** We first discuss the case where the point-wise operation does not have symmetry properties and thus cannot be decomposed on the dimensions of the data. The point-wise has to wait for the 2D DFT to be fully computed before it can applied. Using SPL, the 2D DFT-based convolution is expressed as

$$\mathrm{Conv}_{m\times n} = \mathrm{iDFT}_{m\times n} \cdot D_{mn} \cdot \mathrm{DFT}_{m\times n}, \tag{3.35}$$

where the $\mathrm{iDFT}_{m\times n}$ and $\mathrm{DFT}_{m\times n}$ are the 2D DFTs and $D_{mn}$ is the diagonal matrix for computing the point-wise multiplication. High dimensional DFTs are separable operations. Therefore, using the pencil-pencil algorithm, where a 1D DFT is applied in each dimension, the 2D DFT-based convolution can be re-written as

$$\begin{aligned}\mathrm{Conv}_{m\times n} = (\mathrm{iDFT}_m \otimes I_n) \cdot (I_m \otimes \mathrm{iDFT}_n) \cdot D_{mn} \cdot \\ (\mathrm{DFT}_m \otimes I_n) \cdot (I_m \otimes \mathrm{DFT}_n)\end{aligned} \tag{3.36}$$

Now, forcing the inverse transform to be the conjugate transposed representation of the forward transform, re-shuffles the compute stages of the inverse 2D DFT such that

$$\begin{aligned}\text{Conv}_{m\times n} = &\left(I_m \otimes \text{DFT}_n^H\right) \cdot \\ &\underbrace{\left(\text{DFT}_m^H \otimes I_n\right) \cdot D_{mn} \cdot (\text{DFT}_m \otimes I_n)}_{\text{can be fused}} \cdot \\ &(I_m \otimes \text{DFT}_n)\end{aligned} \tag{3.37}$$

The re-ordering exposes the compute stages that can be merged. In this case the $\left(\text{DFT}_m^H \otimes I_n\right)$ construct can be merged with the diagonal $D_{mn}$ and the $(\text{DFT}_m \otimes I_n)$. Similar to the 1D case, we use the property that the $L$ operator commutes the Kronecker products and obtain the following

$$\begin{aligned}\text{Conv}_{m\times n} = &\left(I_m \otimes \text{DFT}_n^H\right) \cdot \\ &\cdot L_m^{mn} \cdot \left(I_n \otimes \text{DFT}_m^H\right) \cdot L_n^{mn} \cdot D_{mn} \cdot L_m^{mn} \cdot (I_n \otimes \cdot\text{DFT}_m) \cdot L_n^{mn} \cdot \\ &(I_m \otimes \text{DFT}_n)\end{aligned} \tag{3.38}$$

Once again, the permutation matrices around the diagonal matrix $D_{m,n}$ simply shuffle data within the diagonal. Using the direct sum to replace the Kronecker products and to block the diagonal matrix into disjoint components allows us to merge the middle compute stages as

$$\begin{aligned}\mathrm{Conv}_{m\times n} = \left(I_m \otimes \mathrm{DFT}_n^H\right) \cdot \\ \cdot L_m^{mn} \cdot \left(\bigoplus_{i=0}^{n-1} \left(\mathrm{DFT}_m^H \cdot D_m^{\prime(i)} \cdot \mathrm{DFT}_m\right)\right) \cdot L_n^{mn} \cdot \\ \left(I_m \otimes \mathrm{DFT}_n\right)\end{aligned} \tag{3.39}$$

The 2D convolution operation reduces to two DFTs applied on consecutive chunks of data and 1D convolution applied on strided data. The construct $\left(\mathrm{DFT}_m^H \cdot D_m^{\prime(i)} \cdot \mathrm{DFT}_m\right)$ can be optimized by further merging the compute stages following the steps presented in this chapter. Basically the construct is a smaller 1D convolution.

In this example, we computed the forward 2D DFT by first applying the 1D DFT in the row direction followed by a 1D DFT in the column direction as shown in Figure 3.8. Due to the commutativity property of the Kronecker product, the 2D DFT can be compute by first applying 1D DFTs in the columns and then in the rows. Since the forward and inverse DFT are tied through the property that $\mathrm{iDFT} = \mathrm{DFT}^H$, changing the order of computing the forward transform changes the order of the inverse transform as well. This reordering gives a different schedule of the compute stages, which in turns exposes the DFT compute stages done in the row direction to be fused as follows

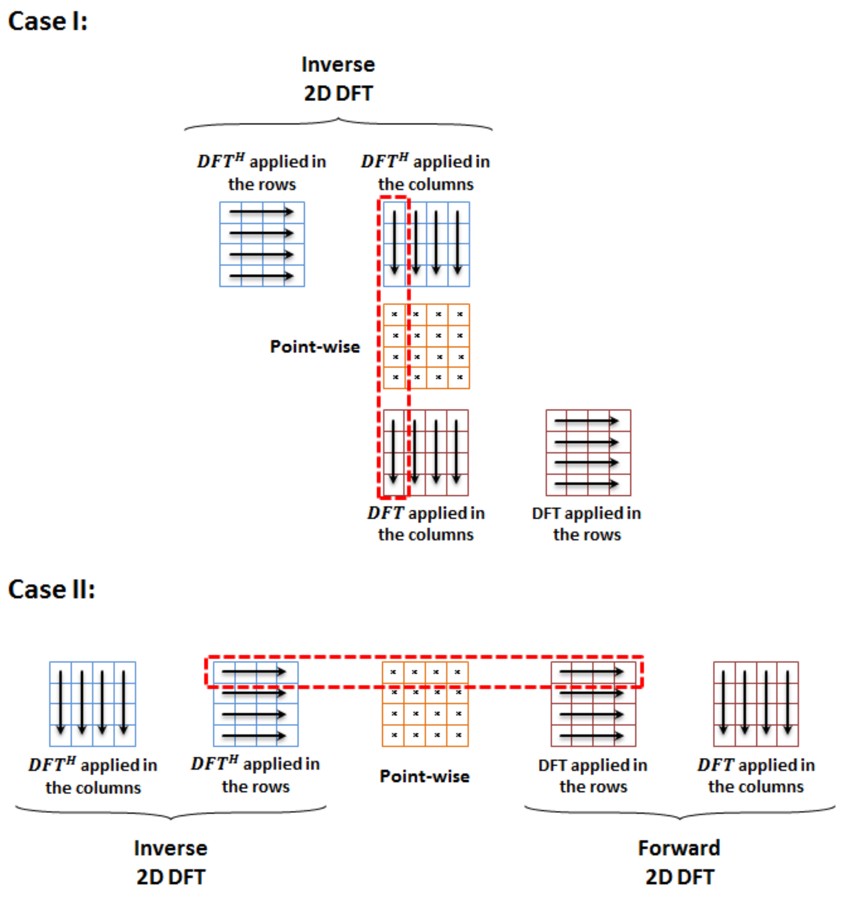

Case I:
Inverse 2D DFT
$DFT^H$ applied in the rows
$DFT^H$ applied in the columns
Point-wise
$DFT$ applied in the columns
DFT applied in the rows
Case II:
$DFT^H$ applied in the columns
$DFT^H$ applied in the rows
Point-wise
DFT applied in the rows
$DFT$ applied in the columns
Inverse 2D DFT
Forward 2D DFT

Figure 3.8: The two possibilities of fusing the stages that compose the 2D DFT-based convolution. The first method fuses the stages so that the 1D convolution is applied in the column direction, while the second method fuses the compute stages so that the 1D convolution is applied in the row direction.

$$\text{Conv}_{m\times n} = \left(\text{DFT}_m^H \otimes I_n\right) \cdot \left(I_m \otimes \text{DFT}_n^H\right) \cdot D_{mn} \tag{3.40}$$
$$\left(I_m \otimes \text{DFT}_n\right) \cdot \left(\text{DFT}_m \otimes I_n\right).$$

For higher dimensional operations the number of possible schedules increases. For example, for the 3D DFT there are three possibilities of re-organizing the compute order depending on which dimension is computed first.

**2D DFT-based operation with separable point-wise.** In this part, we discuss the case where the point-wise computation is separable on the dimensions of the data-set since the point-wise computation exhibits symmetry. In the above example, the assumption is that in order to apply the point-wise multiplication the entire 2D DFT must be applied. However, there are cases where the point-wise multiplication does not require the sequence to be fully converted to the frequency domain. Take for example the differentiation operator $\frac{\mathrm{d}^2}{\mathrm{d}x\mathrm{d}y}$. The operation is separable on the $x$ and $y$ dimensions. Let $i[n_0, n_1]$ be a 2D sequence of size $m$ by $n$. The 2D sequence is the discretized version of the continuous function on which the differentiation operator is applied to. The basic procedure for computing the operator $\frac{\mathrm{d}^2}{\mathrm{d}x\mathrm{d}y}$ is to first apply a 2D DFT on the 2D sequence, apply a point-wise multiplication with complex values defined as

$$D_{mn}[k_0, k_1] = \begin{cases} j\frac{2\pi}{m}k_0 \cdot j\frac{2\pi}{n}k_1,\ k_0 < \frac{m}{2} \text{ and } k_1 < \frac{n}{2} \\ j\frac{2\pi}{m}k_0 \cdot j\frac{2\pi}{n}(k_1 - n),\ k_0 < \frac{m}{2} \text{ and } k_1 > \frac{n}{2} \\ j\frac{2\pi}{m}(k_0 - m) \cdot j\frac{2\pi}{n}k_1,\ k_0 > \frac{m}{2} \text{ and } k_1 < \frac{n}{2} \\ j\frac{2\pi}{m}(k_0 - m) \cdot j\frac{2\pi}{n}(k_1 - n),\ k_0 > \frac{m}{2} \text{ and } k_1 > \frac{n}{2} \\ 0,\ k_0 = \frac{m}{2} \text{ or } \frac{n}{2} \end{cases} \tag{3.41}$$

and finally apply an inverse 2D DFT to obtain the differentiated sequence. Now, it can be seen that $D[k_0, k_1] = D_m[k_0]D_n[k_1]$ (is symmetric), where the values of $D_m$ and $D_n$ are simply the differentiation values obtained when differentiating 1D sequences by $\frac{\mathrm{d}}{\mathrm{d}x}$ and $\frac{\mathrm{d}}{\mathrm{d}y}$ respectively.

$$D_m[k_0] = \begin{cases} j\frac{2\pi}{m}k_0,\ k_0 < \frac{m}{2} \\ j\frac{2\pi}{m}(k_0 - m),\ k_0 > \frac{m}{2} \\ 0,\ k_0 = \frac{m}{2} \end{cases} \tag{3.42}$$

$$D_n[k_1] = \begin{cases} j\frac{2\pi}{n}k_1,\ k_1 < \frac{n}{2} \\ j\frac{2\pi}{n}(k_1 - n),\ k_1 > \frac{n}{2} \\ 0,\ k_1 = \frac{m}{2} \end{cases} \tag{3.43}$$

The point-wise multiplication can be decomposed on the dimensions of the data-set. Due to the Kronecker product properties, the computation can be grouped so that point-wise computation is locally done. The steps of computing the 2D differentiation algorithm knowing the point-wise computation is separable is done as follows

- apply $m$ DFTs of size $n$ in the row direction
- perform $m$ point-wise multiplications with $D_n[k_1]$
- apply $m$ inverse DFTs of size $n$ in the row direction
- apply $n$ DFTs of size $m$ in the column direction
- perform $n$ point-wise multiplications with $D_m[k_0]$
- apply $n$ inverse DFTs of size $m$ in the column direction

The compute stages can be merged. The 2D differentiation operation can be broken down in two parts, first apply the differentiation in the $m$ dimension, followed by the differentiation in the $n$ dimension.

Using the SPL notation, the diagonal $D_{m,n}$ is re-written as

$$D_{mn} = D_m \otimes D_n = (D_m \otimes I_n) \cdot (I_m \otimes D_n), \tag{3.44}$$

where the matrices $D_{mn}$, $D_m$ and $D_n$ are the smaller diagonal matrices. The Kronecker product connects the large diagonal $D_{mn}$ with the smaller matrices $D_m$ and $D_n$, respectively. The smaller matrices are the point-wise operations that are applied in each dimension. The 2D DFT-based differentiation operator $\text{Diff}_{m\times n}$ can be re-written as

$$\begin{aligned}\text{Diff}_{m\times n} = {} & \left(\text{DFT}_m^H \otimes I_n\right) \cdot \left(I_m \otimes \text{DFT}_n^H\right) \cdot \\ & (\text{DFT}_m \otimes I_n) \cdot (I_m \otimes D_n) \cdot (\text{DFT}_m \otimes I_n) \cdot (I_m \otimes \text{DFT}_n)\end{aligned} \tag{3.45}$$

Using the properties of the Kronecker product, the compute stages can be rearranged. In addition, the compute stages can be grouped together and merged. The fused version of the 2D DFT-based differentiation is expressed using SPL as

$$\text{Diff}_{m\times n} = \left(\left(\text{DFT}_m^H \cdot D_m \cdot \text{DFT}_m\right) \otimes I_n\right) \left(I_m \otimes \left(\text{DFT}_n^H \cdot D_n \cdot \text{DFT}_n\right)\right) \tag{3.46}$$

The $\left(\text{DFT}_m^H \cdot D_m \cdot \text{DFT}_m\right)$ and $\left(\text{DFT}_n^H \cdot D_n \cdot \text{DFT}_n\right)$ operations are 1D convolution-like operations. Given that, the compute stages can be further merged following the details discussed in this chapter.

The 2D DFT-based convolution has two cases for the point-wise computation. It can be non-separable and thus the 2D DFT must fully be computed before the point-wise can be applied. Or, it can be separable in which case the computation is grouped together and locally done. However, as the dimensions of the operation increases the number of possible algorithms increase as well. Take for example a 3D DFT-based convolution. Depending on the symmetry properties of the point-wise computation, there exist five possible algorithms. The point-wise computation can be non-separable, it can be separable only on two of the dimensions or it can be fully separable on all three dimensions.

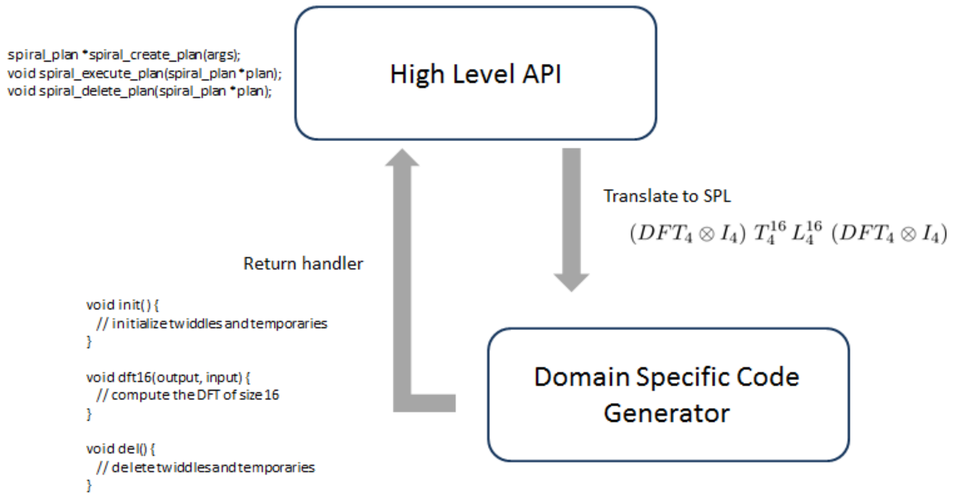
## 3.3 Summary

In this chapter, we focused on fusing the various stages that compose any DFT-based operation. As stated, most DFT-based operations follow a specific pattern, e.g. a forward DFT is applied, followed by a point-wise operation and an inverse DFT. We showed that there is a systematic way of fusing the compute stages if the decomposition of the forward and inverse DFT is known up-front and if constraints like $iDFT = DFT^H$ are imposed. We used the SPL notation to show the step by step process of fusing the computation. However, the optimizations are not bound to the language. The idea is that optimizations like fusion can be done easier if the computation is expressed in a high level mathematical language such as SPL compared to analyzing and optimizing C/C++. In addition, since SPL is used within the SPIRAL framework, we extended the framework with the steps required to automatically perform the fusion of the DFT and non-DFT stages. Moreover, we developed a simple API to express most DFT-based operations built on top of the SPIRAL code generator. In the next chapter, we present the implementations details related to the proposed framework, namely we introduce the simple API meant to express most DFT-based operations and show some other optimizations that are dependent on the underlying architecture but are required to obtain competitive implementations.

# Chapter 4

# From High Level Representation to Optimized Implementation

In this chapter, we look at some of the details regarding the implementation of an end-to-end framework to describe and optimize DFT-based operations. The entire framework is built on the two main ideas presented in this work. First, most DFT-based operations follow a specific pattern, namely a forward DFT followed by a point-wise computation and an inverse DFT. Second, there is a systematic way of applying loop transformations across the DFT and non-DFT stages. Therefore, in this chapter we describe the structure of the entire framework. First, we focus on the front-end part where we present a simple API to capture most common DFT-based operations. Second, we move to the back-end part where we discuss some low level optimizations that are dependent on underlying architecture but are required to achieve competitive code.


High Level API
Translate to SPL
$(DFT_4 \otimes I_4)\, T_4^{16}\, L_4^{16}\, (DFT_4 \otimes I_4)$
Return handler
Domain Specific Code Generator

```
spiral_plan *spiral_create_plan(args);
void spiral_execute_plan(spiral_plan *plan);
void spiral_delete_plan(spiral_plan *plan);
```

```
void init( ) {
  // initialize twiddles and temporaries
}

void dft16(output, input) {
  // compute the DFT of size 16
}

void del() {
  // delete twiddles and temporaries
}
```

Figure 4.1: The structure of the framework. The front-end deals with the API and the translation layer, while the back-end deals with generating and optimizing the code for a specific problem and architecture. The generate code is linked back to the front-end.

## 4.1 The Structure of the Framework

The goal is to offer a framework to describe and optimize DFT-based operations for various architectures. The proposed framework has two components as shown in figure 4.1. The front-end part offers the basic functions to create and execute DFT-based operations, while the second part generates and optimizes the code. Some optimizations are automatically performed, however there still are some optimizations that need automation and thus are currently done by hand. We leave those optimizations as future work.

The framework's front-end is represented by the high level API and the translation layer. As shown in Figure 4.2 the API offers the appropriate data structures and the functions that allow users to create, execute and delete a wide variety of DFT-based operations. We go into details about each function and data structure in the following subsections. The most important functions are the create functions, since they read the specification provided by the user, translate the specification into the mathematical representation and invoke the code generator to generate and optimize the code for that specific DFT-based problem, targeting a given architecture. The code generator returns a handle to the created functions. That handle is afterwards used by the execute and clean-up function to run the code and then clean-up when computation is not required anymore. The idea of creating a plan is not new, FFTW offers such functionalities for generating DFT code. We build upon that idea and extend it to DFT-based operations. The incentive is that scientific applications already use FFTW. Developers are familiar with the FFTW interface, thus modifying their code and replacing the FFTW code-base with the proposed approach can be easily be done.

The framework's back-end is the code-generator. The code generator interprets the mathematical representation, generated and optimize the code for a specific architecture. We use the SPL notation to express DFT-based operations and extend the functionalities of the SPIRAL framework by adding the capabilities to automatically fuse the DFT and non-DFT stages as described in Chapter 3. There are other high level loop optimizations meant to improve the execution time of both DFT and DFT-based operations, like loop fission, loop interchange, loop unrolling. Besides, the high level loop transforms, we also present some low level optimizations that are dependent on the underlying architecture. Most systems provide Single Instruction Multiple Data (SIMD) instructions. Therefore, we apply SIMD vectorization across the DFT and non-DFT stages for DFT-based operations for both power of two and non-power of two problem sizes. We automatically perform data format changes to make computation efficient. We automatically apply efficient padding and un-padding for problem sizes that are not powers of two.

```
spiral_plan *plan;

typedef struct {
  double *data_pointer;
  void (*func)(double*, double*, int, int);
} spiral_pointwise;

typedef struct{
  int size;
  int istr;
  int ostr;
} spiral_dft;

typedef struct{
  int dims;
  spiral_dft *forward_op;
  spiral_dft *backward_op;
  spiral_pointwise *pointer;
} spiral_dft_op;

spiral_plan *spiral_create_basic_op(size_t *dims, int n,
    spiral_pointwise *ptr, unsigned int config);
spiral_plan *spiral_create_advanced_op(spiral_dft_op *ops, int n,
    unsigned int config);

void spiral_execute_plan(spiral_plan *plan, double *output, double *
    input);
void spiral_delete_plan(spiral_plan *plan);
```

Figure 4.2: The data-structures and functions provided by the proposed API.

## 4.2 High Level Interface for DFT-based Operations

In this section we discuss the functionalities that allow developers to describe and create DFT-based operations for 1D, 2D and 3D data-sets. We first focus on some of the basic functions for describing basic DFT-based operations. The assumption for this type of operations is that the point-wise multiplication is non-separable. We then present the more advanced API to describe more complicated DFT-based operations. The advanced interface allows users to specify data rotations/shuffles and separable point-wise computations. The high level interface provides a multiple create functions, an execution and a clean-up function as shown in Figure 4.2.

### 4.2.1 Basic Interface for DFT-based Operations

The basic API handles the plain three stage convolution-like operations, where a forward DFT is followed by a point-wise multiplication and an inverse DFT. The forward and inverse DFT can be swapped, since convolution in time/space or frequency domain maps to point-wise multiplication in the dual domain. Therefore, the inverse DFT can be applied first, followed by the point-wise multiplication and finally the forward DFT. The DFT can also take any dimensions depending on the problem type. The only fixed assumption is that the point-wise operation is assumed to be non-separable. Figure 4.3 shows the basic API for creating such problems. We show only the create function since the execute and delete functions do not change. The API format is very similar to the API offered by FFTW [34] for creating DFT code.

```
#ifndef _MY_SPIRAL_OP
#define _MY_SPIRAL_OP

// data structures
spiral_plan *plan;

// the first bit decides the direction
#define SPIRAL_FORWARD 0
#define SPIRAL_BACKWARD 1

// the second bit decides the type of pointer
#define SPIRAL_DATAPOINTER 0
#define SPIRAL_FUNCPOINTER 2

// functions
spiral_plan *spiral_create_basic_op(size_t *dims, int n,
    spiral_pointwise *ptr, unsigned int config);
#endif
```

Figure 4.3: The basic API for creating DFT-based operations for which the point-wise multiplication is non-separable. The point-wise computation is passed as a pointer of type `spiral_pointwise`. The `spiral_pointwise` data structure provides either a pointer to an array or a pointer to a predefined function.

The create function `spiral_create_basic_op` requires four parameters or arguments. The first argument specifies a pointer to an array that contains the sizes of each dimension, while the second argument specifies the number of dimensions. If the operation is a 1D operation, then the first argument is an array of length 1 and the second argument is equal to 1. If the operation is a 3D operation, the the first argument is an array of length 3 containing the sizes for each dimension and the second argument is equal to 3. The third argument is a pointer of type `spiral_pointwise`. This argument specifies what kind of point-wise computation the code generator should expect, e.g. is the point-wise computation done with pre-computed values stored in an array or is the point-wise compute on the fly by a pre-defined function. The `spiral_pointwise` data structure has two fields. The first field accepts a pointer to an array, while the second field accepts a pointer to a predefined function. The last argument specifies some configuration parameters. For example, if the parameter `SPIRAL_FORWARD` is specified then the first stage of the convolution-like operation is a forward DFT and the last stage is an inverse DFT. If the parameter `SPIRAL_DATAPOINTER` is specified then the point-wise operation are passed in as a pointer to a data-structure data store the precomputed values. The configuration arguments can be combined with the logic `OR` operator.

### 4.2.2 Passing the Point-wise Computation as a Pointer to a Data Structure

The third argument in the create function `spiral_create_basic_op` is a pointer to either an array that stores per-computed values or to a function that computes the values and the point-wise multiplication on the fly. If the configuration argument `SPIRAL_DATAPOINTER` is chosen, then the function will know that the pointer is going to be towards a array. The array holds the pre-computed values. Under this scenario, the user has to first create the array, initialize it with the precompute values and finally pass it to the create function. The user must also take care of de-allocating the temporary array once the required operations is not needed.

Figure 4.4 shows an example where a 1D DFT-based convolution is created specifying that the point-wise operation is done using precomputed values. Recall,

```
#include <spiral_header.h>
// data structures
spiral_plan *plan;
spiral_pointwise spointwise;

int main(int argc, char **args) {
  // ...
  double *temp_array = (double*) malloc(2 * n * sizeof(double));
  spointwise.data_pointer = temp_array;

  // initialize the array
  initialize(temp_array);

  plan = spiral_create_basic_op(&n, 1, &spointwise, SPIRAL_FORWARD |
    SPIRAL_DATAPOINTER);

  // ...
  return 0
}
```

Figure 4.4: Example of creating a basic plan for a 1D DFT-based operation where the the point-wise multiplication will be computed using a temporary array that stores the pre-computed values. The configuration parameters specify that the operation uses the forward DFT as the first stage and enforces the fact that the third argument is a data pointer that requires to reorderd.

in Chapter 3 we discussed about fusing the point-wise operation represented as a diagonal matrix with the DFT stages. The diagonal matrix is always required to be shuffled and partitioned based on how the surrounding computation is decomposed. Therefore, the code generator automatically re-organizes the data as the diagonal matrix is fused with the computation. Re-organizes the data is important since it helps improve the access patterns to the array. Data is always accesses sequentially, no strided accesses into the array are required.

The advantage of using an array to store pre-computed values is that the code generator is in charge of generating the code around the array. The code generator generates the load, compute and store instructions. Therefore, the code generator can choose how the appropriate data format for having an efficient DFT and point-wise computation. The downside of this approach is the fact that extra storage is required. All DFT-based applications require an array equal in size to the input and output arrays. Therefore, as the problem size increases, the amount of data required may become a problem.

The DFT-based convolution specified using the API is translated to SPL, which is interpreted and optimized by the code generator. For example, the 1D convolution described in Figure 4.4 is translated into the following SPL representation

$$\mathrm{Conv}_N = \mathrm{iDFT}_N \cdot D_N \cdot \mathrm{DFT}_N, \tag{4.1}$$

where the diagonal matrix $D_N$ contains the stored values.

### 4.2.3 Passing the Point-wise Computation as a Pointer to a Function

The point-wise computation can also be computed on the fly if a closed form solution exists. Hence, the pointer of type `spiral_pointwise` may be towards a pre-defined function. The user must specify the `SPIRAL_FUNCPOINTER` flag. This flag will enforce the condition that the third argument is a pointer to function and therefore the code-generator will not expect an array. The code generator will simply insert function calls to the pre-defined computation within the generated code. This method allows users to define their own computation. However, the user must follow a specific function header so that the generated code knows about which function to invoke.

Figure 4.5 shows an example of creating a 1D DFT-based operation where the pointer is to a function. The operation itself is a time shift operation computed using the DFT method, which was covered in Chapter 3. Recall that a time shift operation translates to a point-wise multiplication with complex exponentials $e^{-j\frac{2\pi}{N}k\alpha}$, where

```c
#include <spiral_header.h>
int n; // problem size
spiral_plan *plan;
spiral_pointwise spointwise;

// computing a time shift in the time domain
void compute_func(double *in, double *out, int size, int frequency) {
  for(int i = 0; i != size; ++i) {
    int freq_point = frequncy + i
    double re = cos(2 * PI * freq_point * 3 / n);
    double im = -sin(2 * PI * freq_point * 3 / n);

    out[2 * i + 0] = re * in[2 * i + 0] - im * in[2 * i + 1];
    out[2 * i + 1] = re * in[2 * i + 1] + im * in[2 * i + 0];
  }
}

int main(int argc, char **args) {
  // ...
  spointwise.func = compute_func;

  plan = spiral_create_basic_op(&n, 1, &spointwise, SPIRAL_FORWARD |
    SPIRAL_FUNCPOINTER);

  // ...
  return 0
}
```

Figure 4.5: Example of creating a DFT-based time shift. The configuration parameters specify that the operation uses the forward DFT as the first stage, the pointer is to a function and computation is done on the fly.

$\alpha$ represents the amount of time the sequence is shifted. The user must write a small snippet of code to perform the computation as seen in figure 4.5. We have shown that all DFT properties require the point-wise multiplication to be dependent on the frequency points $k$. Typically, the input and output array and the frequency point should suffice.

Since the point-wise computation is recursively fused with the computation until the DFT bases cases are met and since the DFT computation is typically implemented using SIMD instructions, the point-wise function must allow users to specify a frequency range. The function header imposes the following constraints: the first two arguments must be pointers to the input and output arrays, the third argument is an integer used to specify the frequency ranges and the last argument specifies the starting frequency point. The frequency range is given by the granularity at which the DFT computation is applied. Typically, the computation is at the granularity of a cacheline. A typical cacheline can store four complex numbers, therefore the function will start at the frequency point $k$ and apply the computation for $k$, $k+1$, $k+2$ and $k+3$. More details about this aspect are offered in the second part of this chapter.

The advantage of providing a pointer to function is that the point-wise computation is done on the fly. In addition, no extra storage is required compared to the previous method. The function though may require expensive operations as shown in Figure 4.5, however the latencies of computing those instructions may prove to be smaller compared to the latencies of accessing data in main memory. The main downside of this approach is that it falls on the user to write the code. The user must follow the specified contract to write the function, otherwise the generated code cannot invoke the function. In addition, the user must optimize the code snippet. He/she must understand the data format between the DFT compute stages and write the code appropriately, preferably using SIMD instructions. However, the code snippet does not require a large amount of code.

Similarly to the previous case, the 1D convolution described in Figure 4.5 is translated to SPL using the translation layer such as

$$\mathrm{Conv}_n = \mathrm{iDFT}_n \cdot D_n\left(\mathrm{func}\right) \cdot \mathrm{DFT}_n. \tag{4.2}$$

In this case the diagonal is just a place-holder for the function. No data is stored.

```
#ifndef _MY_SPIRAL_OP
#define _MY_SPIRAL_OP

// dft data structure
typedef struct{
  int size;
  int istr;
  int ostr;
} spiral_dft;

// dft operation data structure
typedef struct{
  int dims;
  spiral_dft *forward_op;
  spiral_dft *backward_op;
  spiral_pointwise *pointer;
} spiral_dft_op;

// functions
spiral_plan *spiral_create_advanced_op(spiral_dft_op *ops, int n,
    unsigned int config);
#endif
```

Figure 4.6: The advanced interface for expressing convolution-like operations where the point-wise multiplication is separable on dimensions. The `dft` and `dft_op` represent the two data structures to create the DFT and convolution-like operations.

Instead of re-shuffling the data as the diagonal is fused with the computation, the code generator tracks how the indexing into the function changes and provides an index mapping function.

### 4.2.4 Advanced Interface for DFT-based Operations

The interface discussed so far only deals with the case where the point-wise operation is non-separable, meaning that the point-wise multiplication cannot be decomposed on the dimension of the data set. However, there are applications where the point-wise operation can be separated. Recall the 2D differentiation operation discussed in Chapter 3. The differentiation is gradually applied on the data, immediately after the 1D DFT the point-wise and inverse DFT are also applied. This reduces the number of passes through the data and applies computation locally. In this section, we focus on the API to capture separable operations.

We are required to add some extra data structures to express convolutions with separable point-wise multiplications. Figure 4.6 shows the two data structures and the create function required to create the separable convolution-like operations. The first data structure `spiral_dft` is identical to the FFTW's [34] data structures, e.g. `fftw_iodim`. It is used to describe the forward and inverse DFT stages and the dimensions in which the DFTs are applied. The first argument specifies the size of the DFT, while the other two arguments specify the data access patterns. For example, if the DFT size is 16, `istr` is equal to 1 and `ostr` is equal to 4, then the DFT computation requires 16 consecutive elements and produces 16 elements, where each element is written at a stride of 4 elements from each other.

The second data structure `spiral_dft_op` specifies how to combine the DFTs to create the DFT-based operations. The first parameter, `dims`, specifies the dimensions of the DFT-based operation. Both the forward and inverse DFT must have the same dimension. The DFT constructs `spiral_dft` are passed as pointers to the fields `forward_dft` and `backward_dft`. The two pointers point to the same DFT configuration if the forward and inverse problem do not change the problem size. The only operation that requires different problem sizes is the interpolation operation. Typically an interpolation increases the number of samples, thus the forward DFT is smaller than the inverse DFT. Finally, the last argument is the `spiral_pointwise` pointer to specifies whether the point-wise computation either a arrays that stores the pre-computed complex values or to a specific function that computes everything on the fly.

```
spiral_plan *plan;
spointwise sr, sc;

int main(int argc, char **argv) {
  double *diff_row, *diff_col; //initialize differentiation arrays
  spiral_dft dft_rows[1], dft_cols[1]; spiral_dft_op op[2];
  dft_rows[0].size = N1;
  dft_rows[0].istr = 1;
  dft_rows[0].ostr = 1;
  dft_cols[0].size = N0;
  dft_cols[0].istr = N1;
  dft_cols[0].ostr = N1;
  sr.data_pointer = diff_row;
  sc.data_pointer = diff_col;

  op[0].dims = 1;
  op[0].forward_op = dft_rows;
  op[0].backward_op = dft_rows;
  op[0].pointer = sr;
  op[1].dims = 1;
  op[1].forward_op = dft_cols;
  op[1].backward_op = dft_cols;
  op[1].pointer = sc;

  plan = spiral_create_advance_op(op, 2, SPIRAL_FORWARD |
    SPIRAL_DATAPOINTER);
  return 0;
}
```

Figure 4.7: C/C++ description of the 2D differentiation operation using the proposed framework. The 2D differentiation operation has a separable point-wise computation.

The data structures are then passed to the create function. This function takes three arguments. The first argument is a pointer to an array of type `spiral_dft_op`. The array stores the convolution-like configurations. The size of the array depends on the how many dimensions can the point-wise computation be decomposed upon. The size is specified in the second argument. For example, for the 2D differentiation where the complex values are separated in the row and column direction the value of $n$ must be equal to two. The first argument is an array that contains two distinct convolution-like operations. The last argument specifies the different configuration options, similarly to the basic interface.

Figure 4.7 shows the C/C++ description of the 2D differentiation problem using the proposed API. The first step is to declare the DFTs applied in the rows and in the columns (lines 7 - 14). The first DFT is applied in the rows since the `istr` and `ostr` are both equal to 1. The second DFT is applied in the column direction, since The `istr` and `ostr` are set to $N_1$. Lines 16 - 23 describe how the DFT-based operations are constructed. Each construct defines the convolution-like operation applied in the row and in the column direction. The `forward_op` and `backward_op` fields point to the same array. This suggests that the dimensions of the input and output do not change. Finally, each `spiral_dft_op` has the pointer `pointer` linked to the specific arrays. The constructs are then passed to the create function. The create function translates the specification into the appropriate SPL representation

$$\begin{aligned} Diff_{N_0 \times N_1} = & \left( \left( iDFT_{N_0} \cdot D_{\alpha} \cdot DFT_{N_0} \right) \otimes I_{N_1} \right) \\ & \left( I_{N_0} \otimes \left( iDFT_{N_1} \cdot D_{\beta} \cdot DFT_{N_1} \right) \right), \end{aligned} \tag{4.3}$$

where $D_{\alpha}$ and $D_{\beta}$ represent the diagonal matrices that store the complex values used for differentiation. The representation is then passed to the SPIRAL code generator.

Both the basic and advanced create function translate the specifications into the SPL representation and invoke the SPIRAL framework to automatically generate and optimize the code. The code generator applies high level loop transformations specified in Chapter 3 and merges the compute stages. However, since the DFT computation is regular, the code is also automatically vectorized for a given architecture. It is well known that the DFT exhibits data parallelism, thus both the DFT and the DFT-based operations must make us of the SIMD instructions for improved performance. In the following section, we discuss some of the challenges required to apply SIMD vectorization on operations that require complex multiplication and also operations that require padding since not all problem sizes are power of two and thus divisible by the SIMD vector length.

## 4.3 Low Level Optimizations

DFT computation requires basic arithmetics like addition, subtraction and multiplication with complex numbers. Moreover, the DFT computation requires all the input samples to participate in the computation of all the outputs. This translates to a transposition. The Cooley-Tukey for power of two problem sizes explicitly has a transposition in the decomposition and a point-wise multiplication with complex numbers. Therefore, generating an efficient implementation becomes difficult especially when using CPU features like the Single Instruction and Multiple Data (SIMD) instructions. To make things more complicated, some problem sizes are not powers of two and require padding, thus implementing the computation with SIMD instructions becomes cumbersome. In the next paragraphs we discuss some of these challenges and offer solutions. First though we give a brief introduction to

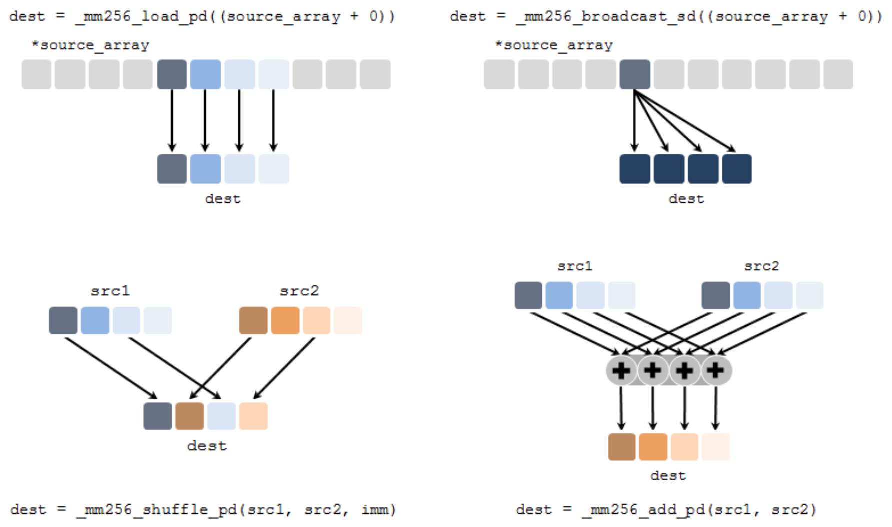
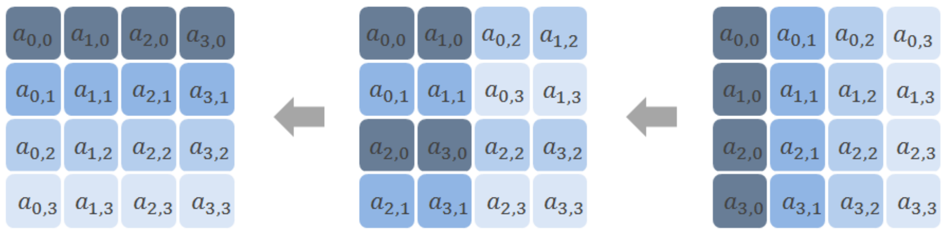

dest = _mm256_load_pd((source_array + 0))
*source_array
dest
dest = _mm256_broadcast_sd((source_array + 0))
*source_array
dest
src1
src2
dest
dest = _mm256_shuffle_pd(src1, src2, imm)
src1
src2
dest
dest = _mm256_add_pd(src1, src2)

Figure 4.8: AVX instructions used within the DFT computation. The first instruction loads four contiguous elements in the vector register. The second instruction replicates one element in all four locations. The third instruction performs a so called in-register shuffle. The fourth instruction applies a separate addition to all four data points.

the SIMD programming model.

### 4.3.1 Single Instruction Multiple Data (SIMD)

Most modern architectures from vendors like Intel, AMD, ARM offer instructions that apply the same operation on multiple data-points, e.g SSE, AVX [60] for Intel, AMD and Neon [61] for ARM. They offer wider registers that can store more elements. The number of elements they can store is called the width of the SIMD register $\nu$. For example, Intel's AVX can store up to four double precision or eight single precision elements into one SIMD register, where the typical number of SIMD registers is equal to 16 or 32. The underlying architecture stitch functional units to be able to load and store multiple points at the same time. For example, the AVX instruction `_mm256_load_pd` loads four double precision elements at a time and


$a_{0,0}$ $a_{1,0}$ $a_{2,0}$ $a_{3,0}$
$a_{0,1}$ $a_{1,1}$ $a_{2,1}$ $a_{3,1}$
$a_{0,2}$ $a_{1,2}$ $a_{2,2}$ $a_{3,2}$
$a_{0,3}$ $a_{1,3}$ $a_{2,3}$ $a_{3,3}$
$a_{0,0}$ $a_{1,0}$ $a_{0,2}$ $a_{1,2}$
$a_{0,1}$ $a_{1,1}$ $a_{0,3}$ $a_{1,3}$
$a_{2,0}$ $a_{3,0}$ $a_{2,2}$ $a_{3,2}$
$a_{2,1}$ $a_{3,1}$ $a_{2,3}$ $a_{3,3}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$

```
dest1 = _mm256_permute2f128_pd(temp1, temp3, 0 | (2 << 4));
dest3 = _mm256_permute2f128_pd(temp1, temp3, 1 | (3 << 4));
dest2 = _mm256_permute2f128_pd(temp2, temp4, 0 | (2 << 4));
dest4 = _mm256_permute2f128_pd(temp2, temp4, 1 | (3 << 4));
```

```
temp1 = _mm256_shuffle_pd(src1, src2, 0 | (0 << 1) | (0 << 2) | (0 << 3));
temp2 = _mm256_shuffle_pd(src1, src2, 1 | (1 << 1) | (1 << 2) | (1 << 3));
temp3 = _mm256_shuffle_pd(src3, src4, 0 | (0 << 1) | (0 << 2) | (0 << 3));
temp4 = _mm256_shuffle_pd(src3, src4, 1 | (1 << 1) | (1 << 2) | (1 << 3));
```

Figure 4.9: Performing a data transposition for data stored in SIMD vector registers. Each SIMD register contains four elements. If the data stored in the SIMD registers is viewed as the rows of a matrix, the SIMD algorithm performs a matrix transposition, where the registers store the columns of the matrix.

stores them into a vector register, while instructions `_mm256_broadcast_sd` reads one double precision element and replicates it four times within the SIMD vector registers as seen in figure 4.8.

Most floating point operations like addition, subtraction and multiplication have a SIMD counterpart. Similar to the load instructions, multiple functional units that perform addition, subtraction and multiplication are as well stitched together to be able to perform the same operation on different elements. Instead of computing one operation per cycle, $\nu$ operations are computed in parallel. For example, the AVX SIMD addition instruction depicted in figure 4.8 does four double precision adds per cycle. Most modern architectures even offer more complicated instructions like fused-multiply add (FMAs) instructions. The operation fuses the multiplication and addition in one single instruction. While this is important for matrix-matrix multiplication, operations like DFT do not see so much improvement when using it, especially that the DFT is dominated by additions and subtractions.

There are also instructions that shuffle data within the SIMD registers. These operations are important for transposing and rotating data. Take for example, the permutation in figure 4.9, where 16 double precision elements are stored into four

SIMD registers. Assuming that the four SIMD registers store the rows of a 4x4 double precision matrix, the transposition shuffles the data so that at the end of it the registers contains the columns of the matrix. The transposition operation within the SIMD registers requires four SIMD shuffle instructions. The first two instructions permute two elements locally, while the other two instructions permute the blocks of two elements. The downside of shuffle instructions is that on most architectures there is only one functional unit that executes shuffle operations. Therefore, in the case of computing the DFT, the shuffle instructions become a significant bottleneck.

### 4.3.2 Arithmetic Operations on Complex Data Points

All DFT-based operations compute on complex data points. In addition, we assume the complex data points to be stored in the typical complex interleaved format, where the real and imaginary components are stored in consecutive order. While this format does not influence computations like addition or subtraction, it affects the multiplication operation especially when dealing with SIMD registers and operations. Addition and subtraction of two complex numbers is straight forward. Given two complex number $z_0 = a + jb$ and $z_1 = c + jd$, where $j^2 = -1$, the addition is expressed as

$$z = z_0 + z_1 = (a + c) + j\,(b + d)\,. \tag{4.4}$$

It can be seen that adding two complex numbers means adding the real parts and the imaginary parts independently. Switching to the SIMD implementation requires replacing the scalar registers with SIMD registers and changing the addition/subtraction with the corresponding SIMD instruction. However, computing the complex

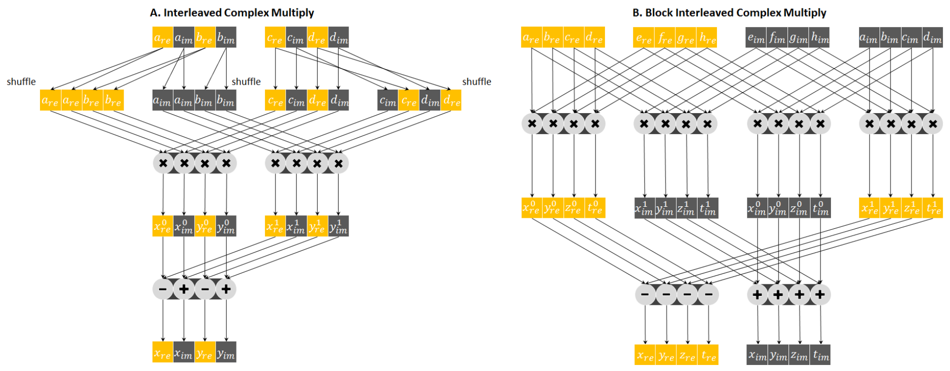

A. Interleaved Complex Multiply
B. Block Interleaved Complex Multiply
shuffle
shuffle
shuffle

Figure 4.10: Computing complex multiplication when data is stored in complex interleaved versus block complex interleaved. The complex interleaved format requires shuffle instructions interleaved with the compute instructions. The block complex interleaved format requires data to be packed and unpacked in advance. Picture is reproduced from [2].

multiplication of the two complex numbers $z_0$ and $z_1$ is a bit more involved

$$w = z_0 \cdot z_1 = (ac - bd) + j\,(ad + bc) \tag{4.5}$$

The real and imaginary components of the result depend on all the components of the two inputs. The multiplication requires more operations than simply replacing the scalar registers and instructions with the SIMD counterparts. SIMD shuffle instructions and specialized instructions such as `addsub` must be added.

In the "Mixed Data Layout Kernels for Vectorized Complex Arithmetic" [2], we show the differences between two variants of computing the complex multiplication as seen in figure 4.10. The first variant assumes data is stored in complex interleaved. Four complex numbers are loaded into two SIMD registers. In order to compute the complex multiplication, the data stored in the registers needs to be shuffled so that the real and imaginary components are correctly aligned. Three shuffle instructions are needed to perform this alignment. Data then can be mul-

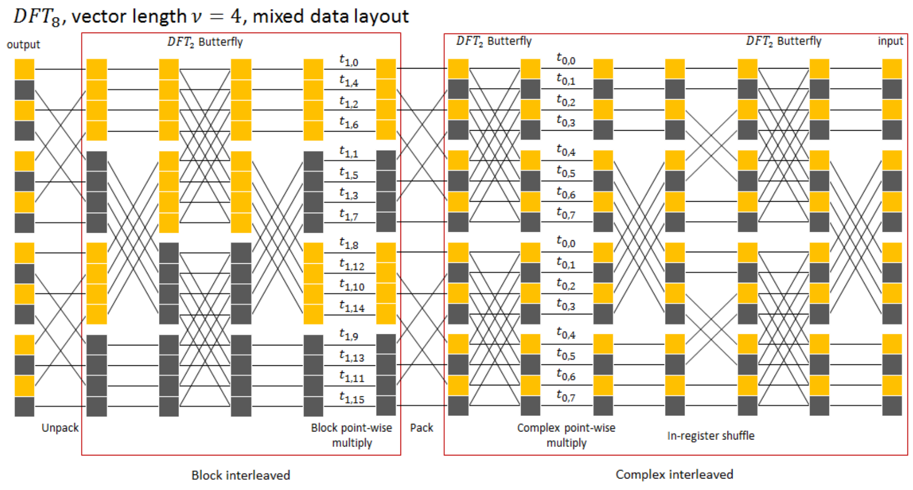

DFT_8, vector length ν = 4, mixed data layout
output
DFT_2 Butterfly
$t_{1,0}$ $t_{1,4}$ $t_{1,2}$ $t_{1,6}$ $t_{1,1}$ $t_{1,5}$ $t_{1,3}$ $t_{1,7}$ $t_{1,8}$ $t_{1,12}$ $t_{1,10}$ $t_{1,14}$ $t_{1,9}$ $t_{1,13}$ $t_{1,11}$ $t_{1,15}$
DFT_2 Butterfly
$t_{0,0}$ $t_{0,1}$ $t_{0,2}$ $t_{0,3}$ $t_{0,4}$ $t_{0,5}$ $t_{0,6}$ $t_{0,7}$ $t_{0,0}$ $t_{0,1}$ $t_{0,2}$ $t_{0,3}$ $t_{0,4}$ $t_{0,5}$ $t_{0,6}$ $t_{0,7}$
DFT_2 Butterfly
input
Unpack
Block point-wise multiply
Pack
Complex point-wise multiply
In-register shuffle
Block interleaved
Complex interleaved

Figure 4.11: Hiding the data format change within the shuffle instructions for double precision AVX instructions. Part of the computation is done using the complex interleaved format, while part of it is done using the block complex interleaved. Picture is reproduced from [2].

tiplied and finally an `addsub` instruction is need to compute the final results. The main advantage is that data is kept in the original format. However, the three shuffle operations become bottlenecks since most modern systems can only execute one shuffle instruction per cycle. Due to the one shuffle instruction per cycle and due to the dependencies, multiply instructions cannot be issued in parallel even though there are typically two pipelines that execute multiply instructions. Another problem that appears is that there are architectures that do not have support for `addsub` instructions. These instructions need to be emulated, thus higher latencies are incurred.

The second variant assumes that the data format is changed from complex interleaved to block complex interleaved. The block complex interleaved format implies that $\nu$ real components are interleaved with $\nu$ imaginary components, where $\nu$ represents the SIMD vector length. The complex multiplication is similar to the scalar implementation, however instead of using scalar registers and operations, everything is replaced with the SIMD counterparts. This removes the shuffle and `addsub` instructions. The computation still requires shuffle instructions, since data needs to be packed from complex interleaved to block complex interleaved. However, the packing of the data can be done in advance and in the case of the DFT the format change can be hidden within the permute stages as seen in figure 4.11. In addition, for DFT-based operations the format change can be done once at the start of the entire algorithm. It can be maintained across compute stages so that operations like the point-wise multiplication can be done in block complex interleaved format. The data format can be changed back once computation has finished.

### 4.3.3 Data Permutations using SIMD

The Cooley-Tukey algorithm requires a transposition stage as shown in equation 2.51, since all inputs participate in the computation of each output. Since computation is implemented using SIMD instructions, the $L$ matrix must be blocked to the appropriate vector length and merged with the computation. Recall that the 1D DFT convolution requires the forward DFT and the conjugate transpose DFT. The forward transform fuses the $L$ matrix with the right $DFT \otimes I$ child, while the the conjugate transposed version fuses the $L$ matrix with the $DFT \otimes I$ left child as shown in 3.27. In this section we present the step by step approach of fusing the transposition with the DFT computation for both the forward and inverse DFT.

We add one more constraint, namely all implementations change the format from complex interleaved to block complex interleaved. Hence, the computation and the permutation must be blocked to a block greater than the SIMD vector length $\nu$. The change of format requires at least two vector registers, one to store the real components and one to store the imaginary components. Blocking the computation without the permutation is trivial. The SPL representation is as follows

$$(DFT_m \otimes I_n) = \left(L_m^{mn/b} \otimes I_b\right) \cdot \left(I_{m/b} \otimes (\mathrm{DFT}_m \otimes I_b)\right) \cdot \left(L_{m/b}^{mn/b} \otimes I_b\right), \qquad (4.6)$$

where $b$ represents the block size and $b \geq 2\nu$. The blocking operation says the $DFT_m$ is applied on $b$ columns at a time.

Blocking and fusing the permutation with the computation is more challenging. However, the process can be systematically derived. Using the properties of the $L$ matrix defined in Chapter 2, the transposition can be decomposed into a block permutation and multiple permutations within the blocks such as

$$\left(I_{m/b} \otimes L_b^{nb}\right) \cdot \left(L_{m/b}^{mn/b} \otimes I_b\right) \cdot (DFT_n \otimes I_m). \qquad (4.7)$$

The block permutation commutes some of the elements within the $DFT \otimes I$ construct as follows

$$\left(I_{m/b} \otimes L_b^{nb}\right) \cdot \left(I_{m/b} \otimes (DFT_n \otimes I_b)\right) \cdot \left(L_{m/b}^{mn/b} \otimes I_b\right). \qquad (4.8)$$

Both the in-block permutation and the DFT computation have the same $I_{m/b}$. Therefore, the stages can be merged as follows

$$\left(I_{m/b} \otimes \underbrace{\left(L_b^{nb} \cdot (DFT_n \otimes I_b)\right)}_{\text{further decomposes}}\right) \cdot \left(L_{m/b}^{mn/b} \otimes I_b\right). \qquad (4.9)$$

This implies that the data is permuted immediately after the computation.

The inner child $L_b^{nb} \cdot (DFT_n \otimes I_b)$ can further be decomposed if $n > b$. Assuming that $n > b$ and $n = n_0 n_1$, the Cooley-Tukey can be recursively applied. However, the blocking $I_b$ construct must be moved close to the computation as follows

$$L_b^{nb} \cdot (DFT_{n_0} \otimes I_{n_1 b}) \cdot \left(T_{n_1}^{n_0 n_1} \otimes I_b\right) \cdot \left(L_{n_0}^{n_0 n_1} \otimes I_b\right) \cdot (DFT_{n_1} \otimes I_{n_0 b}). \tag{4.10}$$

Distributing the $I_b$ construct is equivalent to applying the unroll and jam optimization [62, 63] on nested loops, where the outer loop is unrolled and then the inner loops are merged together. Using the properties of the $L$ matrix, the permutation construct $L_b^{nb}$ can be decomposed as

$$\begin{gathered}\left(L_b^{n_0 b} \otimes I_{n_1}\right) \cdot \left(I_{n_0} \otimes L_b^{n_1 b}\right) \cdot (DFT_{n_0} \otimes I_{n_1 b}) \cdot \\ \left(T_{n_1}^{n_0 n_1} \otimes I_b\right) \cdot \left(L_{n_0}^{n_0 n_1} \otimes I_b\right) \cdot (DFT_{n_1} \otimes I_{n_0 b}).\end{gathered} \tag{4.11}$$

The terms can be re-arranged and grouped together to merge the compute stages as follows

$$\begin{gathered}\underbrace{\left(L_b^{n_0 b} \otimes I_{n_1}\right) \cdot (DFT_{n_0} \otimes I_{n_1 b})}_{\text{stage 2}} \cdot \\ \underbrace{\left(I_{n_0} \otimes L_b^{n_1 b}\right) \cdot \left(T_{n_0}^{n_0 n_1} \otimes I_b\right) \cdot (I_{n_1} \otimes (DFT_{n_1} \otimes I_b)) \cdot \left(L_{n_0}^{n_0 n_1} \otimes I_b\right)}_{\text{stage 1}}.\end{gathered} \tag{4.12}$$

The first stage applies the computation on blocks of size $b$, point-wise multiplies the data with the twiddle factors and immediately after permutes the data within the blocks. The second stage applies the same computation on blocks of size $n_1$ and immediately after it permutes those blocks. The entire operation is shown in Figure 4.12. Merging the stages, the decomposition process can repeat until the

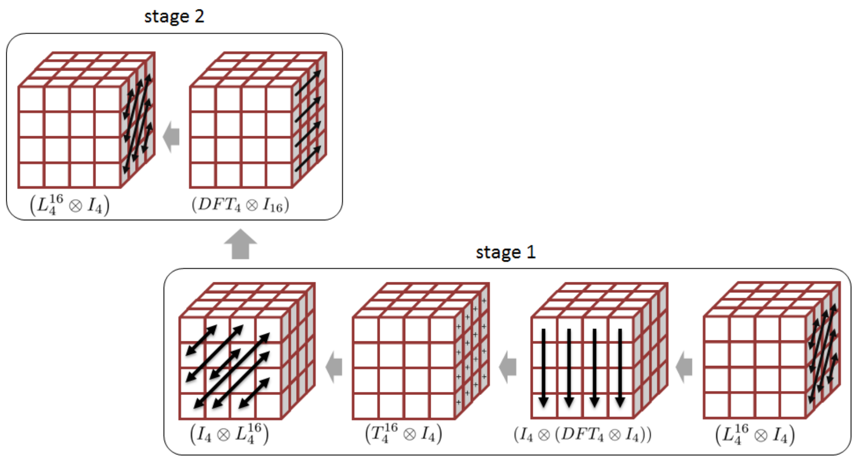


Figure 4.12: The pictorial representation of the SPL notation from equation 4.12.

base case is met.

Let the base case be when $n = b$. For the moment we do not handle the case where $n < b$. The base is represented as

$$L_b^{b^2} \left( DFT_b \otimes I_b \right). \tag{4.13}$$

It has been shown in [64] that this construct represents the base case for vectorization when using SIMD SSE instructions ($b = 2$ and $b = 4$). The paper has shown that the SPL construct can easily be mapped to SIMD instructions. Of interest is the permutation $L_b^{b^2}$. In the paper it has been shown that the permutation requires two or four in-register shuffle instructions depending on the data type (float or double). However, SSE only stores 128 bits. AVX and more recently AVX512 can store more elements, which implies that the implementation of the transposition operation $L_b^{b^2}$ requires more in-register shuffle instructions with the increase in SIMD vector length. For example, the AVX512 implementation using single precision numbers requires 64 shuffle instructions. Similarly to the SIMD implementation of the complex multiply using the interleaved format, the permute instructions are on the critical path and create a serious bottleneck. We use some of the information from [65, 66] and hand-implement the construct. We break the permutation into smaller permutations and hand interleave the shuffle instructions with the compute instructions. For the moment, we have implemented several computational kernels by hand for various blocking parameters and SIMD ISAs, however we leave as future work the mathematical representation to capture these optimizations and the automation component.

The same steps can be repeated for the inverse DFT, or the conjugate transposed version of the forward DFT. The conjugate transpose operation simply flips the representation of the $L^{nb} \cdot (DFT_n \otimes I_b)$ construct to obtain the following SPL representation

$$\left(DFT_n^H \otimes I_b\right) L_b^{nb}. \tag{4.14}$$

The steps for decomposing the construct are similar. If $n > b$ then the DFT is decomposed using, for example, the Cooley-Tukey algorithm. The $I_b$ construct is grouped closer to the computation. All the terms are re-arranged, merged and the decomposition process is repeated. The main difference is the base case, when $n = b$. The permutation precedes the computation as follows

$$\left(DFT_b^H \otimes I_b\right) L_b^{b^2}. \tag{4.15}$$

Similarly to the previous case, as the blocking $b$ increases, implementing the permutation becomes more expensive. However, breaking the permutation into smaller permutations and interleaving them with computation re-leaves the pressure on the permute pipeline. We implement the $L_b^{b^2}(DFT_b \otimes I_b)$ construct by hand. We leave the automation part of expressing and decomposing the permutations as future work.

### 4.3.4 Zero-padding for Non-Powers of Two

Not all scientific applications require problem sizes that are divisible by the SIMD vector length. For example, ONETEP [67] is such an application that requires non-power of two problem that are also odd. Hence, the DFT-based interpolation used within ONETEP and presented in [3] has to deal with non-powers of two. Mapping the application to SIMD instructions becomes non-trivial. Data must be padded with 0s to the closest multiple of the SIMD vector length $\nu$ so that computation can be done efficiently. In this section, we show that there is a systematic way of inserting 0s so that the memory footprint is not drastically increased. In addition, we show that for DFT-based operations where we have the entire view of the application we can reduce the number of padding and un-padding operations. Note, that similarly to the permute instructions, padding and un-padding instructions are expensive.

To start with a simple example, we have shown in equation 4.6 that the construct $DFT_m \otimes I_n$ can be vectorized easily if $n$ is divisible by the SIMD vector length $\nu$. However, if $n$ is not divisible we need to use constructs like $I_{m\times n}$ and $I_{n\times m}$ for $m > n$, which were defined in Chapter 2, to perform data padding and un-padding. Using the ideas presented in [68] data can zero-padded as it gets loaded. Computation is applied in a SIMD fashion and the result is then stored, removing the redundant zeros. Formally, this can be represented using the SPL notation as follows

$$DFT_m \otimes I_n = \left(I_m \otimes I_{n\times\nu\lceil n/\nu\rceil}\right)\left(DFT_m \otimes I_{\nu\lceil n/\nu\rceil}\right)\left(I_m \otimes I_{\nu\lceil n/\nu\rceil\times n}\right), \qquad (4.16)$$

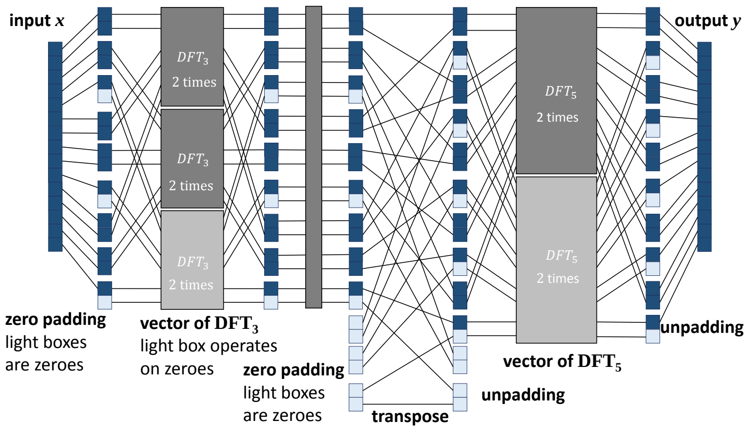


Figure 4.13: DFT decomposition with zero-padding for size $N = 15$ and vector length $\nu = 2$. Light boxes represent inserted zero value cells, and staggered pairs of cells represent SIMD vectors or SIMD vectorized DFT kernels. Data flows from right to left. Picture is reproduced from [3].

where $I_{\nu\lceil n/\nu\rceil \times n}$ and $I_{n\times\nu\lceil n/\nu\rceil}$ are the operators that perform the data zero-padding and un-padding. $\rceil$ represents the mathematical ceiling function. We make use of this operator to construct the padded version of the Cooley-Tukey algorithm.

We derive the short vector Cooley Tukey algorithm starting from the equation

$$\mathrm{DFT}_{mn} = \big(\mathrm{DFT}_m \otimes I_n\big) T_n^{mn} L_m^{mn} \big(\mathrm{DFT}_n \otimes I_m\big). \tag{4.17}$$

Next we carefully zero-pad to the minimum extent necessary. We define $m' = \nu\lceil m/\nu\rceil$, $n' = \nu\lceil n/\nu\rceil$ and we extend the Cooley-Tukey algorithm as

$$\begin{aligned}\mathrm{DFT}_{mn} = (I_m \otimes I_{n\times n'})\big(\mathrm{DFT}_m \otimes I_{n'}\big) \\ (I_m \otimes I_{n'\times n}) T_n^{mn} L_m^{mn} (I_n \otimes I_{m\times m'}) \\ \big(\mathrm{DFT}_n \otimes I_{m'}\big)(I_n \otimes I_{m'\times m}).\end{aligned} \tag{4.18}$$

The twiddle matrix $T_n^{mn}$ is merged with the right hand child, namely the $DFT_n \otimes I_{m'}$ construct. The twiddle factors need to be first permuted because of the $L$ matrix, and then zeros need to be inserted because of the $(I_n \otimes I_{m \times m'})$. Let $S_{m'}^{m'n}$ be the modified twiddle matrix. Replacing this construct in the above expression gives the following

$$\begin{aligned} \mathrm{DFT}_{mn} = (I_m \otimes I_{n \times n'})\big(\mathrm{DFT}_m \otimes I_{n'}\big) \\ (I_m \otimes I_{n' \times n}) L_m^{mn} (I_n \otimes I_{m \times m'}) \\ S_{m'}^{m'n}\big(\mathrm{DFT}_n \otimes I_{m'}\big)(I_n \otimes I_{m' \times m}). \end{aligned} \tag{4.19}$$

The goal is to vectorize the entire DFT. Computation can be vectorized because of the padding. The twiddle computation can also be vectorized, since the diagonal has been extended to a multiple of the SIMD vector length. The permute matrix though cannot. Neither $n$ nor $m$ are divisible by the SIMD vector length. However, due the surrounding padding, the permute matrix can be replaced with a permute matrix $L_{m'}^{m'n'}$ so that it can be SIMD vectorized. More details, can be found in [3]. The final expression of the Cooley-Tukey implementation that can be implemented using SIMD instructions is represented using SPL as follows

$$\begin{aligned} \mathrm{DFT}_{mn} = (I_m \otimes I_{n \times n'})\big(\mathrm{DFT}_m \otimes I_{n'}\big) \\ I_{mn' \times m'n'} L_{m'}^{m'n'} I_{m'n' \times m'n} \\ S_{m'}^{m'n}\big(\mathrm{DFT}_n \otimes I_{m'}\big)(I_n \otimes I_{m' \times m}). \end{aligned} \tag{4.20}$$

Pictorially the implementation is shwon in Figure 4.13. It can be seen, that the transpose stage requires some extra zeros to perform the transposition. Some of the zeros are dropped. The entire process is represented by the $I_{mn' \times m'n'} L_{m'}^{m'n'} I_{m'n' \times m'n}$

construct. Similar steps can be applied to Rader, Bluestein and Good-Thomas. In addition, the padding can be applied across the DFT and non-DFT stages of any DFT-based operation. Knowing the decomposition and knowing where the zeros are insert redundant padding can be removed. The padding is done at the beginning and end of the DFT-based convolution. No padding is needed in between the stages.

## 4.4 Summary

In this chapter, we presented the implementation details concerning the entire framework. First, we discussed the front end of the framework, namely the high level API. The API offers the capabilities to express most common DFT-based operations like convolution, correlations and Poisson solvers. We introduced the basic operations that permit one to create 1D, 2D, 3D DFT-based operations with non-separable point-wise computation. We then augmented the basic interface with the functions that allow users to express DFT-based operations where the point-wise computation can be separated on the different dimensions. The role of the API is to capture the specification and pass it to the underlying code generator.

We then presented the back-end component of the framework, namely the code generator. While we stated that merging the computed stages is important and presented things in more details in Chapter 3, the code generator also targets other low level optimizations. DFT and DFT-based computations are regular and must be mapped on SIMD instructions. However, DFT computation requires complex multiplication, data permutations and in some cases, depending on the problem size, zero padding operations. We showed that there is a systematic way of applying SIMD optimizations within the DFT and across the DFT and non-DFT stages. In the next chapter, we show that all these optimizations produce competitive code.

# Chapter 5

# Results and Discussion

In this chapter, we provide a thorough analysis of the benefits of merging the DFT and non-DFT stages for 1D and 3D DFT-based convolutions. For the 3D DFT-based convolution we analyze the improvements for both the non-separable and separable point-wise computation. In addition, we tackle both power of two and non-power of two problem sizes. We compare the results of the code that is automatically fused and vectorized against implementations that use MKL and FFTW for the DFT computation and require hand-written code for the point-wise operation. For each case, we also provide a breakdown of the performance and show the key factors that help the overall execution time.

## 5.1 Methodology

For all DFT-based operations, we compare our approach against implementations that use DFT library calls to MKL and FFTW. Our method merges the DFT and non-DFT stages and generates the entire DFT-based convolution. However, the applications that use MKL and FFTW as DFT library calls do not merge

| Feature | Intel Haswell 4770k | Intel Kaby 7700k |
|---|---|---|
| Cores | 4 | 4 |
| Threads | 8 | 8 |
| Frequency | 3.5 GHz | 4.5 GHz |
| SIMD FMAs/cycle | 2 | 2 |
| SIMD Adds/cycle | 1 | 2 |
| SIMD Muls/cycle | 2 | 2 |
| L1 Data Cache | 32 KB | 32 KB |
| L2 Data Cache | 256 KB | 256 KB |
| L3 Data Cache | 8 MB | 8 MB |
| Memory | 32 GB | 64 GB |

Table 5.1: The architectures used to obtain experimental results. The table outlines some of the main CPU features.

the computation and require the point-wise computation to be implemented by hand. Both the generated code and the point-wise computation are implemented using SIMD instructions. We use two architectures to validate our assumptions, namely the Intel Haswell 4770k and the Intel Kaby Lake 7700k. Table 5.1 shows the characteristics of each architecture. We choose the two architectures because of the number of pipelines that can execute SIMD floating point additions. We use different compilers and packages, therefore we split the experiments into two. The first category deals with problem sizes that are powers of two. In this case, all the code for computing 1D and 3D DFT-based convolutions are compiled with the Intel C++ Compiler 18.0.2. In addition, we compare the performance of our implementation will MKL 2018 and FFTW 3.3.6. The second category deals with problem sizes that are non-power of two problem sizes. The reason is that most of the results are taken from our paper [3]. The code was compiled with the Intel C++ Compiler 14.0.1 and compared against the implementations that used Intel MKL 11.0.0 and FFTW 3.3.4.

For power of two problem sizes, we report performance numbers as float-

| DFT | Number of Additions | Number of Multiplications | $5n\log(n)$ |
|---|---|---|---|
| $\text{DFT}_{16}$ | 160 | 64 | 320 |
| $\text{DFT}_{32}$ | 448 | 256 | 800 |
| $\text{DFT}_{64}$ | $1,024$ | 512 | $1,920$ |
| $\text{DFT}_{128}$ | $2,560$ | $1,536$ | $4,480$ |
| $\text{DFT}_{256}$ | $5,632$ | $3,072$ | $10,240$ |
| $\text{DFT}_{512}$ | $13,312$ | $8,192$ | $23,040$ |
| $\text{DFT}_{1024}$ | $28,672$ | $16,384$ | $51,200$ |
| $\text{DFT}_{2048}$ | $65,536$ | $40,960$ | $112,640$ |
| $\text{DFT}_{4096}$ | $139,264$ | $81,920$ | $245,760$ |
| $\text{DFT}_{8192}$ | $311,296$ | $196,608$ | $532,480$ |

Table 5.2: The DFT sizes and the corresponding number of additions and multiplications.

ing point operations per cycle (FLOPS/cycle). For each DFT we count the exact number of additions and multiplications as shown in table 5.2. We measure each execution using the Intel's performance counters and record the number of cycles. For powers of two, the DFT computation is dominated by additions and subtractions. The only multiplications appear when computing the point-wise multiplication with the twiddle factors. Therefore, the performance of the DFT and any DFT-based operations is determined by how efficient the addition pipeline is being used. We state that on the Intel Haswell 4770K the peak performance for a DFT and DFT-based operations is equal to 4 floating point operations per cycle using AVX double precision floating point numbers. On the Intel Kaby Lake 7700K the peak performance is equal to 8 floating point operations. Therefore, the top line in each plot represents the peak performance that can be achieved on that given architecture.

For non-power of two problem sizes, we report pseudo-floating point operations per cycle. With non-power of two problem sizes, the main difficulty is determining the exact number of floating point operations, since most DFT decomposition use algorithms like Good-Thomas, Rader or Bluestein. Good-Thomas trades off the point-wise multiplication with more expensive permutations, while Rader and Bluestein are basically convolution operations. We use the formula $5n\log(n)$ for the DFT problems of size $n$. However, this number does not reflect the actual number of floating point operations, it is essentially used as normalization factor for showing inverse run-time [34], which gives an indication of the performance one sees. We only use this metric for the 3D interpolation operation used in the paper [3].

Since all DFT-based operations use a point-wise multiplication, we use the formula $6n$ for the total number of floating point operations, where $n$ represents the size of the arrays that get point-wise multiplied. The point-wise operation is a complex multiplication, therefore it requires four multiplications and two additions. This allows us to split the total number of floating point operations into $4n$ floating point multiplications and $2n$ floating point additions.

## 5.2 To Fuse or Not To Fuse the 1D DFT-based Convolution

In this section, we analyze the benefits of fusing the compute stages for the 1D DFT-based convolution. We assume all problem sizes are power of two. Hence we use the Cooley-Tukey algorithm to decomposed the forward and inverse DFTs. In Chapter 2, we discussed that the Cooley-Tuley algorithm decomposes the DFT into four stages, namely two compute stages of smaller DFTs, a data permutation stage and point-wise multiply stage with the twiddle factors. The permute stage and the twiddle scaling stage are fused with the computation as shown in [31], giving two compute stages that cannot be further merged. The last stage must wait until the first stage fully computes its data points. If implemented withe MKL and FFTW, a

1D DFT-based convolution has five stages, two from the forward DFT, two from the inverse DFT and one stage represented bt the point-wise computation. However, using the steps described in Chapter 3, the 1D DFT-based convolution is reduced to having three stages. The last stage of the forward DFT is merged with the point-wise computation and the first stage of the inverse DFT. Mathematically, the stages are represented using SPL as follows

$$\begin{aligned}\mathrm{Conv}_N = L_{n_2}^{n_1 n_2} \cdot \left( \bigoplus_{i=0}^{n_1-1} \mathrm{DFT}_{n_2}^H \right) \cdot \\ L_{n_1}^{n_1 n_2} \cdot \left( \bigoplus_{i=0}^{n_2-1} T_{n_2}^{\prime n_1 n_2 {}^*(i)} \cdot \mathrm{DFT}_{n_1}^H \cdot D_{N/n_2}^{\prime(i)} \cdot \mathrm{DFT}_{n_1} \right) \cdot L_{n_2}^{n_1 n_2} \\ \left( \bigoplus_{i=0}^{n_1-1} T_{n_2}^{\prime n_1 n_2 (i)} \cdot \mathrm{DFT}_{n_2} \right) \cdot L_{n_1}^{n_1 n_2},\end{aligned} \tag{5.1}$$

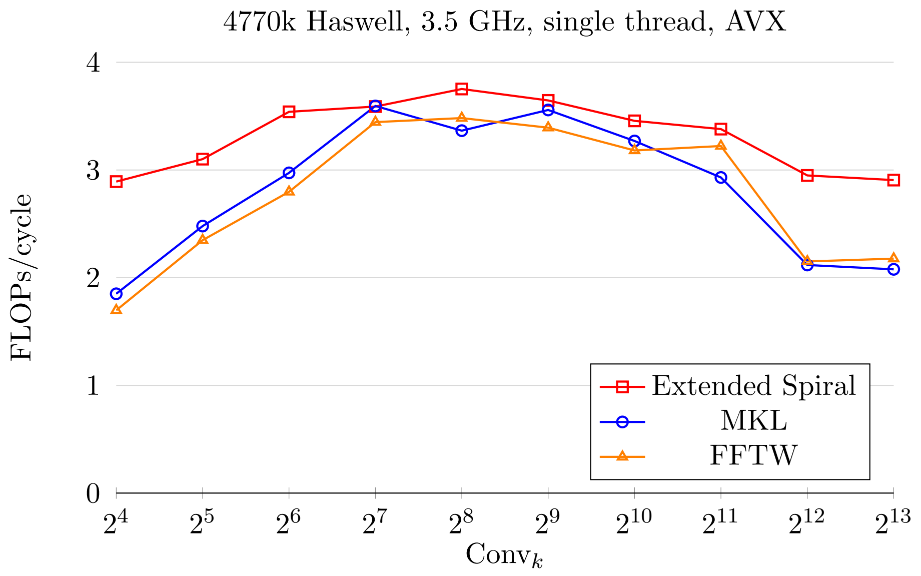
Note that for both the 1D and 3D DFT-based convolution we use the property $i\mathrm{DFT} = \mathrm{DFT}^H$. We generate the code for the SPL expression 5.1 using AVX SIMD instructions. We compare the generated implementation with variants of the 1D DFT-based convolution where the DFT stages are library calls to MKL and FFTW. Since, there is no way of expressing the point-wise multiplication, we implement it by hand using SIMD instructions as seen in Figure 5.1. Note that we make the assumption that the format for the complex layout is complex interleaved, therefore we require shuffle instructions to perform the point-wise multiplication on complex data points when using SIMD instructions.

Figure 5.2 shows the performance results of the three implementations. The top lines represent the peak performance that can be achieved on the given architecture for the DFT algorithm. It can be seen that the fused convolution gets closer to peak and in addition it outperforms the implementations that keep the compute

```
void pointwise(__m256d *input, __m256d *scale, int size) {
  __m256d *inputx = input;
  __m256d *scalex = scale;

  for(int i = 0; i != 0; ++i) {
    __m256d v0 = *(inputx);
    __m256d v1 = *(scalex);

    // shuffle data before complex multiplication
    __m256d t0 = _mm256_permute_pd(v0,  0);
    __m256d t1 = _mm256_permute_pd(v0, 15);
    __m256d t2 = _mm256_permute_pd(v1,  5);

    __m256d t3 = _mm256_mul_pd(t1, t2);
    // use the fused multiply addsub instruction
    t3 = _mm256_fmaddsub_pd(t0, v1, t3);

    *(inputx) = t3;

    inputx++;
    scalex++;
  }
}
```

Figure 5.1: The hand-written code for computing the point-wise multiplication for the 1D DFT-based convolution when using MKL and FFTW for the DFT calls.

stages separate and use MKL and FFTW for the DFT computation. where the compute stages remain separate. The fused convolution generated using our approach outperforms by almost 1.5x the other two implementations even for small sizes like 16, 32 and 64. The data-set for these problem sizes still fit within the L1 cache. However the gain in performance comes from the fact that our approach does not have overhead. It is well know that MKL and FFTW incur overhead when invoking their functions. Recall that in our approach, the API translates the C/C++ description into SPL, which is then passed to the code generator that generates the C code. The C code is then compiled into a shared library object (.so) and read back into the main program.



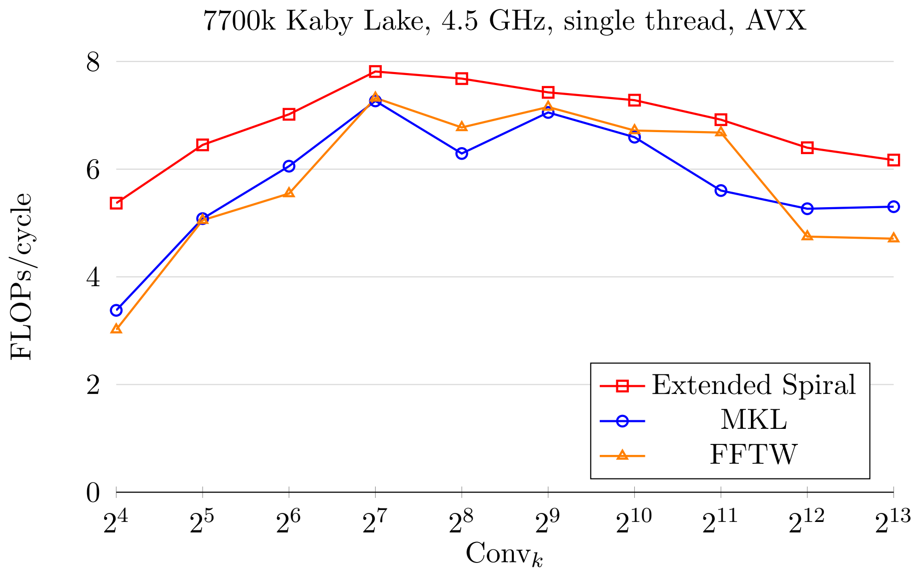
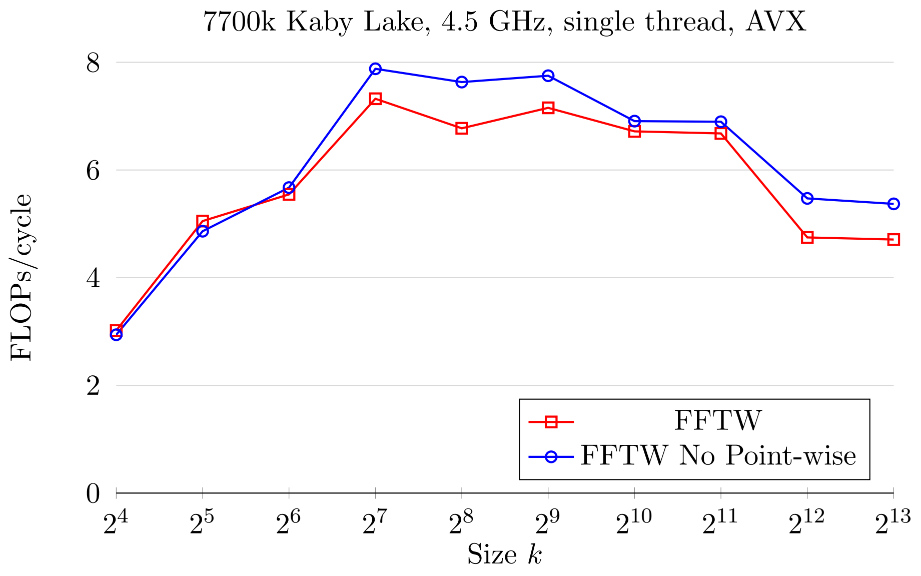


Figure 5.2: The performance for the 1D DFT-based convolution of our approach and the implementations that use FFTW and MKL as library calls. The figure on the left shows the performance on the Intel Haswell, while the figure on the right shows the performance on the Intel Kaby Lake.

For the sizes 128, 256, 512, 1024 and 2048, all three implementations achieve similar performance. Data fits into the L1 or L2 cache. Fusing the stages does not necessarily mean performance gains, since accessing data from L1 or L2 does require long latencies. Both Haswell and Kaby Lake require three and 12 cycles to read data from L1 and L2, respectively. However, as the problem size increased beyond the L2's capacity, the performance difference between the implementation that fuses the stages and the two other implementation can be seen once again. For a 1D DFT-based convolution of size 4096 the performance of the implementations that use MKL and FFTW drops abruptly. This is caused by the increased latency of accessing the data from the last level cache. The code generated using our approach does not see that rapid change since compute stages are merged and data locality is improved.

The first question we tackle is how much the point-wise computation influences the overall performance. Our approach fuses the point-wise operation with the computation. Therefore, the cost of loading in the data to perform the point-wise computation is amortized by the DFT compute stages. Figure 5.3 shows this aspect. The plot on the left presents the effects of removing the point-wise computation from the 1D DFT-based operation implemented with the help of FFTW. For small sizes the effects are negligible. The performance of the variant that has the point-wise computation and the one that does not are almost identical. However, as the problem size increases and data needs to be read from the lower levels of the memory hierarchy the effects can easily be spotted. The FFTW implementation that does not have the point-wise computation sees a 10% performance improvement compared to the case where the point-wise computation present. The same though cannot be said about our implementation. Fusing the DFT and non-DFT stages hides the effects of computing the point-wise multiplication.



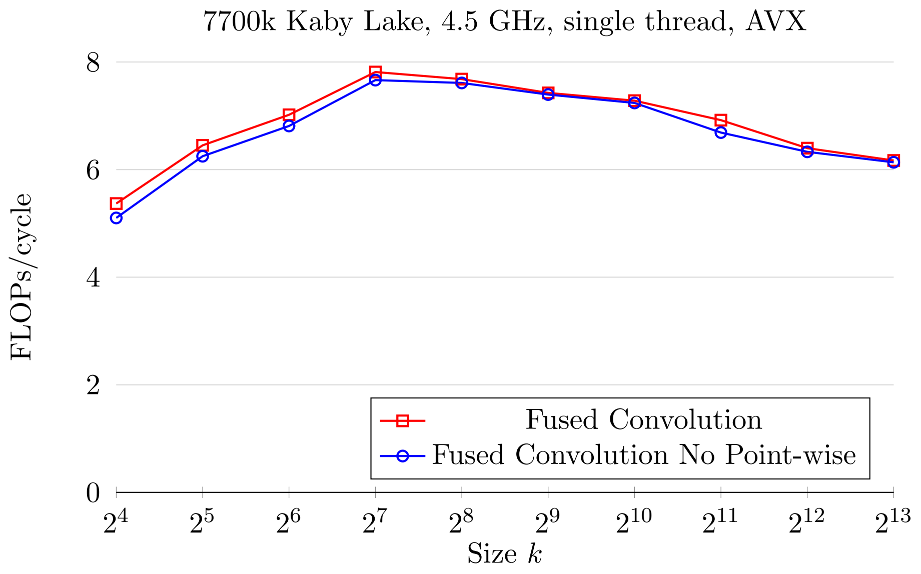


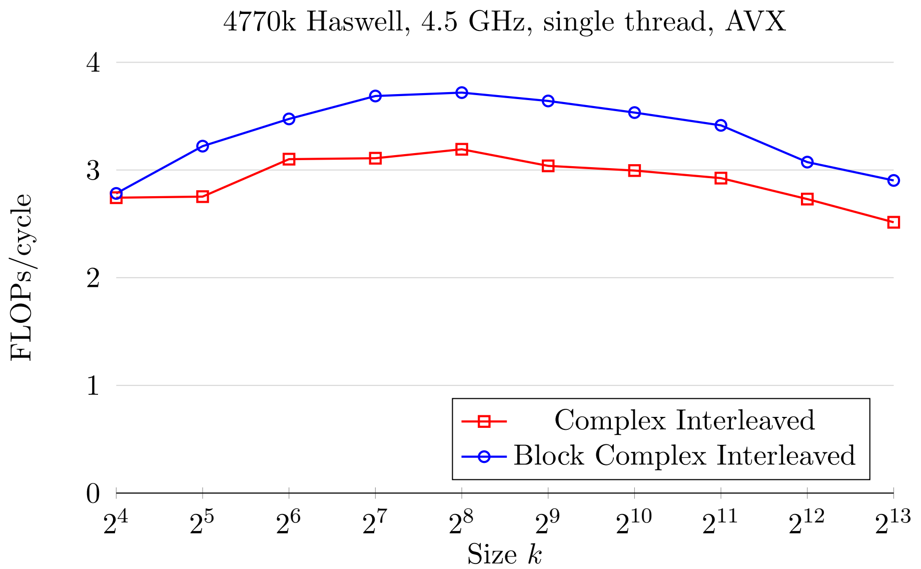
Figure 5.3: The performance improvement when removing the point-wise multiplication from the 1D DFT-based convolution. The left figure shows a performance improvement of almost 10% when removing the point-wise multiplication. The right figure emphasize the fact that the point-wise multiplication does not modify performance if the forward and inverse DFT are merged.

It can be seen that the two implementations, one with the point-wise and one without, achieve the same performance. Two conclusions can be drawn from this experiment:

- Fusing the DFT and non-DFT stages amortizes the cost of loading data to perform the point-wise computation. The effects of computing the point-wise are negligible

- Part of the performance improvement comes from fusing the point-wise computation. However, the main performance boost comes from fusing the stages of the forward and inverse DFT.

Choosing the correct instructions to implement the complex multiplication with SIMD instructions also makes a difference. We have shown in [2] that the block complex interleaved format provides improvements over keeping data in the complex interleaved layout. The block complex interleaved format removes the shuffle instructions from the critical path, therefore operations like additions and multiplications can be issued in parallel whenever possible. Assuming data is stored in interleaved format, the computation still requires shuffles, however the shuffle instructions are hidden with the permute instructions as shown in Chapter 4. Figure 5.4 shows the performance difference when using complex interleaved and block complex interleaved for computing just the forward and inverse DFT, without the point-wise operation. The difference in performance is more prominent on the Kaby Lake processor. The Intel Kaby Lake can issue two SIMD addition instructions per cycle, the same rate at which the SIMD multiply and fused-multiply add instructions are issued. However, the number of pipelines that execute shuffle instructions has not been changed. Therefore, removing the shuffle instructions from the critical path gives the processor the possibility of issuing the SIMD additions in parallel.



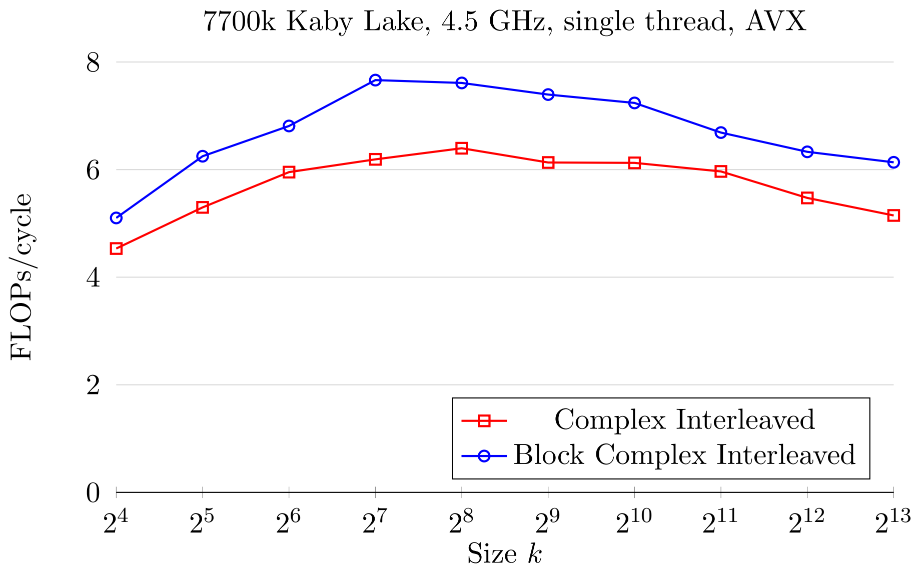


Figure 5.4: The performance results when using the complex interleaved versus block complex interleaved data format. The results are for the fused version of the forward and inverse DFT without the point-wise computation. The left image shows the results on the Intel Haswell, while the right image shows the results on the Intel Kaby Lake.

Haswell also sees a difference in performance between the two implementations.

Removing the shuffle instructions from the critical path also decreases the length of dependent instructions. In addition, more instructions that can be executed in parallel are exposed, namely add and multiply instructions.

Overall, there are performance gains that can be achieved when fusing the stages within a 1D DFT-based convolution. For problem sizes that fit within the upper levels of the memory hierarchy, the performance improvements might be smaller. However, as the problem size increases and data needs to be read from the lower levels of the memory hierarchy, the performance difference can be more significant. We discussed problems that go up to the L3 cache. In the next section, we focus on 3D DFT-based operations that reside wither in the last level cache or in main memory.

## 5.3 Fusing 3D DFT-based Operations

In this section, we focus on 3D DFT-based operations with non-separable and separable point-wise computation. We present results for three implementations:

- The first implementation deals with the basic 3D-DFT based convolution, where the point-wise computation is non-separable. In addition, all the problem sizes are powers of two.

- The second implementation is that of a 3D DFT-based differentiation operation for which the point-wise computation is separable on the dimensions. Similarly to the previous implementation, the problem sizes are powers of two.

- The third implementation is that of a 3D DFT-based interpolation. The interpolation is separable on the three dimensions, however the sizes are odd numbers smaller than 128.

We compare the implementations generated using our approach and the implementations that use MKL and FFTW. We emphasize the fact that the FFTW and MKL implementations require the user to write code around the library calls. In most cases in order to achieve good performance the user is forced to know the underlying system when implementing the glue code around the DFT calls.

### 5.3.1 Non-Separable Point-wise Multiplication

Recall, that the 3D DFT-based convolution with a non-separable point-wise multiplication is expressed using SPL as

$$\begin{aligned} \mathrm{Conv}_{k\times m\times n} = &\underbrace{\left(I_{km}\otimes \mathrm{DFT}_n^H\right)\cdot\left(I_k\otimes \mathrm{DFT}_m^H\otimes I_n\right)}_{\text{can be fused}}\cdot \\ &\underbrace{\left(\mathrm{DFT}_k^H\otimes I_{mn}\right)\cdot D_{kmn}\cdot\left(\mathrm{DFT}_k\otimes I_{mn}\right)}_{\text{can be fused}}\cdot \\ &\underbrace{\left(I_k\otimes \mathrm{DFT}_m\otimes I_n\right)\cdot\left(I_{km}\otimes \mathrm{DFT}_n\right)}_{\text{can be fused}}. \end{aligned} \tag{5.2}$$

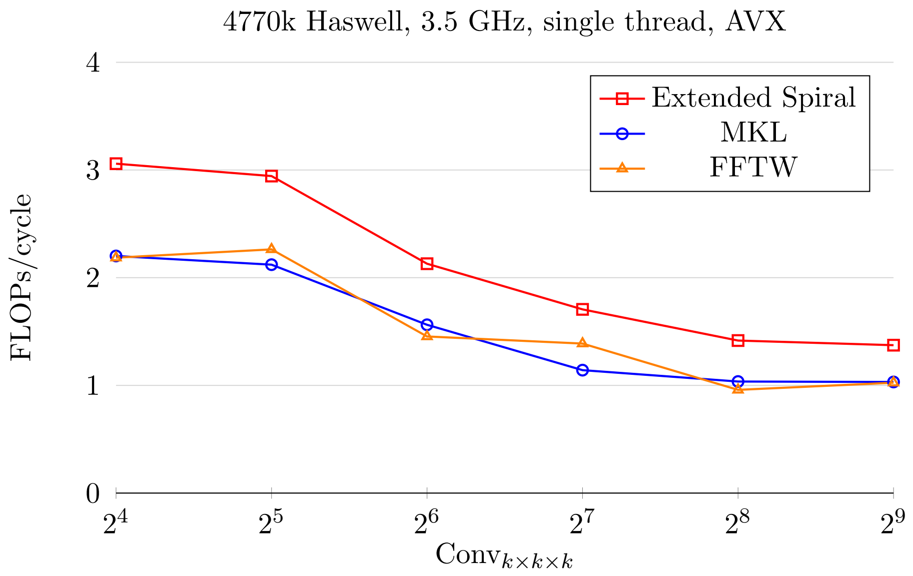
Note that, we use the property that the inverse DFT is the complex conjugate transpose of the forward DFT. In Chapter 3, we have shown that this property re-arranges the DFT stages within the inverse 3D DFT to better facilitate the fusion process with the forward transform. It can be seen that there are two groups of computation where fusion can be applied. First, the middle operations represented by $\left(\mathrm{DFT}_k^H\otimes I_{mn}\right)$, $D_{kmn}$ and $\left(\mathrm{DFT}_k\otimes I_{mn}\right)$ constructs can be fused in one single compute stage, using the steps described in Chapter 3. Let $P_{k,mn} = L_m^{kmn}\cdot\left(\bigoplus_{i=0}^{mn-1}\left(\mathrm{DFT}_k^H\cdot D_k'^{(i)}\cdot \mathrm{DFT}_k\right)\right)\cdot L_n^{kmn}$ be the fused stage. Replacing the construct in the above expression gives

$$\begin{aligned}\text{Conv}_{k\times m\times n} = \left(I_{km}\otimes \text{DFT}_n^H\right)\cdot\left(I_k\otimes \text{DFT}_m^H\otimes I_n\right)\cdot P_{k,mn}\cdot \\ \left(I_k\otimes \text{DFT}_m\otimes I_n\right)\cdot\left(I_{km}\otimes \text{DFT}_n\right).\end{aligned} \tag{5.3}$$

Second, the first two forward DFTs and the last two inverse DFTs can also be merged. Instead of applying 1D DFTs one after the other first in the $n$ dimension and then in the $m$ dimension, a 2D DFT of size $DFT_{m\times n}$ can be applied $k$ times. Merging the DFT compute stages gives the following SPL representation

$$\text{Conv}_{k\times m\times n} = \left(I_k\otimes\left(\text{DFT}_{m\times n}^H\right)\right)\cdot P_{k,mn}\cdot\left(I_k\otimes\left(\text{DFT}_{m\times n}\right)\right) \tag{5.4}$$

The assumption is that for the 3D DFT computation, MKL and FFTW merge the DFT computations applied in the $m$ and $n$ dimensions. They resort to a slab-pencil decomposition of the 3D DFT as described in Chapter 2. However, the DFT-based convolution using MKL and FFTW does not fuse the forward DFT, point-wise and inverse DFT. Figure 5.5 shows the performance results of the 3D convolution with non-separable point-wise for the three implementations. The red line represents the performance obtained by the fused code. The other two lines (blue and orange lines) represent the 3D convolutions implemented using FFTW and MKL. It can be seen that our approach achieves from 1.2 to 1.6x performance improvements over the library based implementations. Compared to peak performance, our implementation gets closer. For problem sizes that fit within the memory hierarchy, our approach is within 75% of peak. However, as the problem size increases and data needs to be read from the lower levels of the memory hierarchy the performance of the generated code drops to almost 40% of peak.



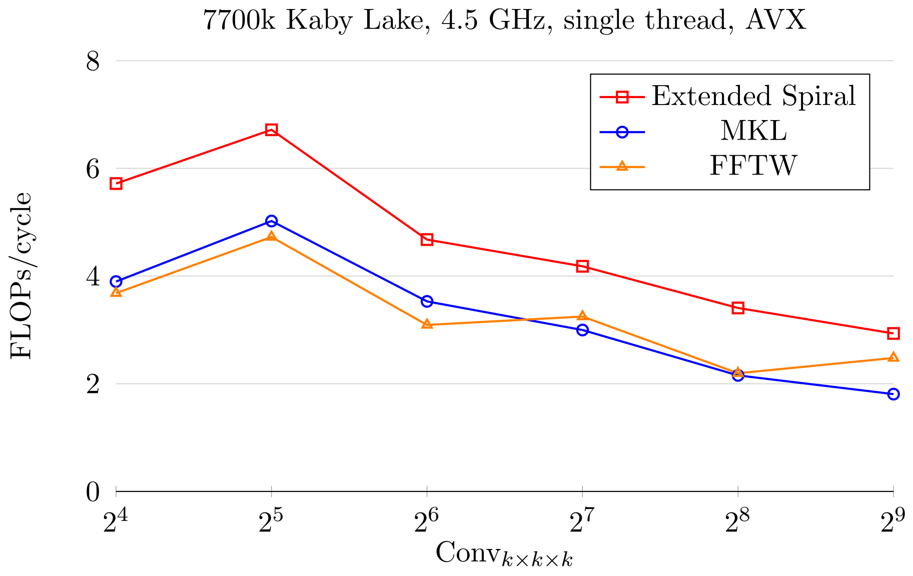
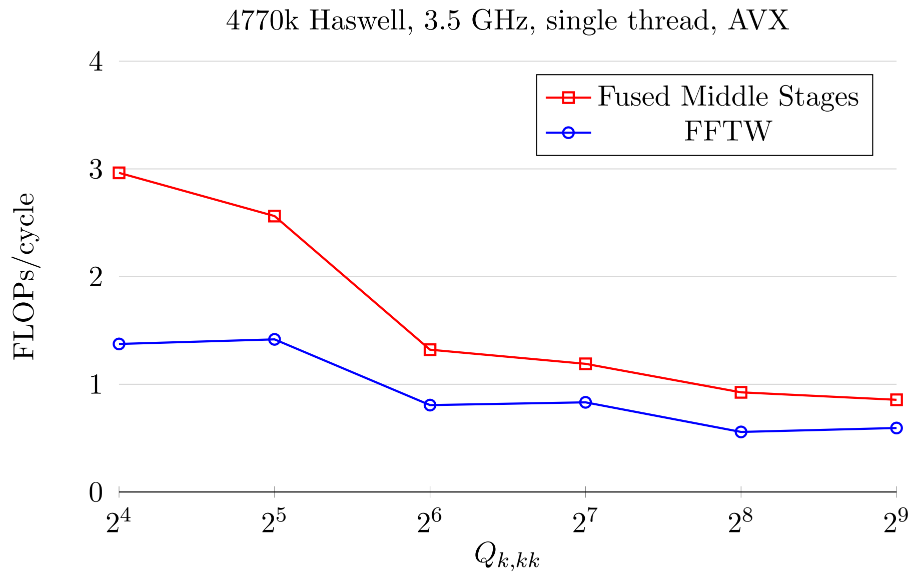


Figure 5.5: The performance results for the 3D DFT-based convolution. The red line represents the performance results for the generated convolution using our approach, where some of the compute stages are fused. The other two lines represent the 3D DFT-based convolutions where the 3D DFTs are computed using the implementations offered by MKL and FFTW.

Merging the stages improves performance, however the DFT memory accesses and the long latencies to main memory influence the overall performance. Therefore, the main questions that need to be answered are how much of the performance comes from merging the middle stages and how much of the performance comes from the DFT computation.

We first focus on answering how much the performance improvement comes from simply merging the middle stages. Figure 5.6 shows the performance improvements when merging the DFT and inverse DFT applied in the $k$ dimensions and the point-wise computation. It can be seen that fusing the stages improves performance by up to 2x compared to the FFTW implementation. We do not compare against the case where the DFT computation is implemented using the MKL library, since MKL does not provide an interface similar to FFTW's Guru Interface. Writing the forward and inverse DFT using MKL, requires hand-written packing routines, task that becomes cumbersome. Compared to the FFTW implementation, fusing the stages improves data locality. Once data is brought into the upper levels of the cache hierarchy, the forward DFT, part of the point-wise computation and the inverse DFT are applied immediately before writing the data back to main memory. Our implementation is cache aware, while FFTW's DFT implementation is cache oblivious [69]. The merged stages are tailored to the parameters of the cache hierarchy. However, both implementations see a performance drop when the problem sizes do not fit in the last level cache. The DFT and point-wise computation is applied at large strides. Accessing data at large strides causes cache conflicts, since most caches are set associative. In addition, due to the non-unit stride nature of the DFT algorithm bandwidth is not efficiently utilized.

We focus next on the performance obtained by the batch 2D DFT applied at the beginning and at the end of the 3D DFT-based convolution. Figure 5.7 shows the performance results for only the forward batch transform.



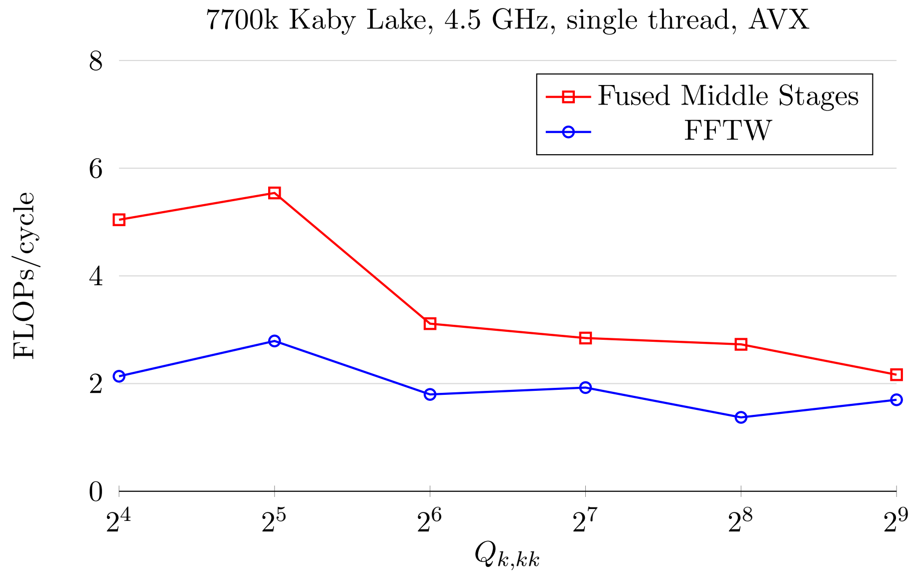


Figure 5.6: The advantages of fusing the compute stages for the construct $Q_{k,mn} = \left(\mathrm{DFT}_k^H \times I_{mn}\right) D_{kmn} \left(\mathrm{DFT}_k \otimes I_{mn}\right)$ as seen in equation 5.3.

The performance of the inverse transform is similar. We compare our approach (red line) against the FFTW and MKL implementations. Our implementation is generated using AVX SIMD instructions and the change of format as described in Chapter 4. For the FFTW implementation we use the Guru Interface. Since MKL does not offer an interface for batch 2D DFTs, we simply loop around the 2D DFT library call. It can be seen, that the generated code using our method is competitive on both the Intel Haswell and Intel Kaby Lake. All implementations fall short to getting close to peak performance. However, we have to state that the implementation we generate is not fully optimized. Optimizations like unrolling and constant simplification are still left on the table. Putting everything together, fusing just the middle stages one can achieve up to 1.6x performance improvements compared to the implementations where the DFT and point-wise stages are left un-fused. It is important to notice that, although the performance of merging the middle stages is significant in isolation, the execution time of the batch 2D DFTs overshadows these improvements.

The implementation of the 3D DFT-based convolution with non-separable point-wise computation can also pe parallelized. Figure 5.8 shows the performance improvements on the Intel Kaby Lake processor, when increasing the number of threads from one to four. Notice that the peak performance is 32 floating point operations per cycle. The processor has four cores, each core having a peak of eight floating point operations per cycle. When increasing the number of threads the performance of the 3D DFT-based convolution increases almost linearly. It is also important to notice that for problem sizes that fit within the cache hierarchy the overall performance is withn 40% from peak. However, the performance drops significantly when the problem size resides in main memory. The performance is expected to drop since the performance of the DFT drops when problem sizes do not fit on on-chip cache.

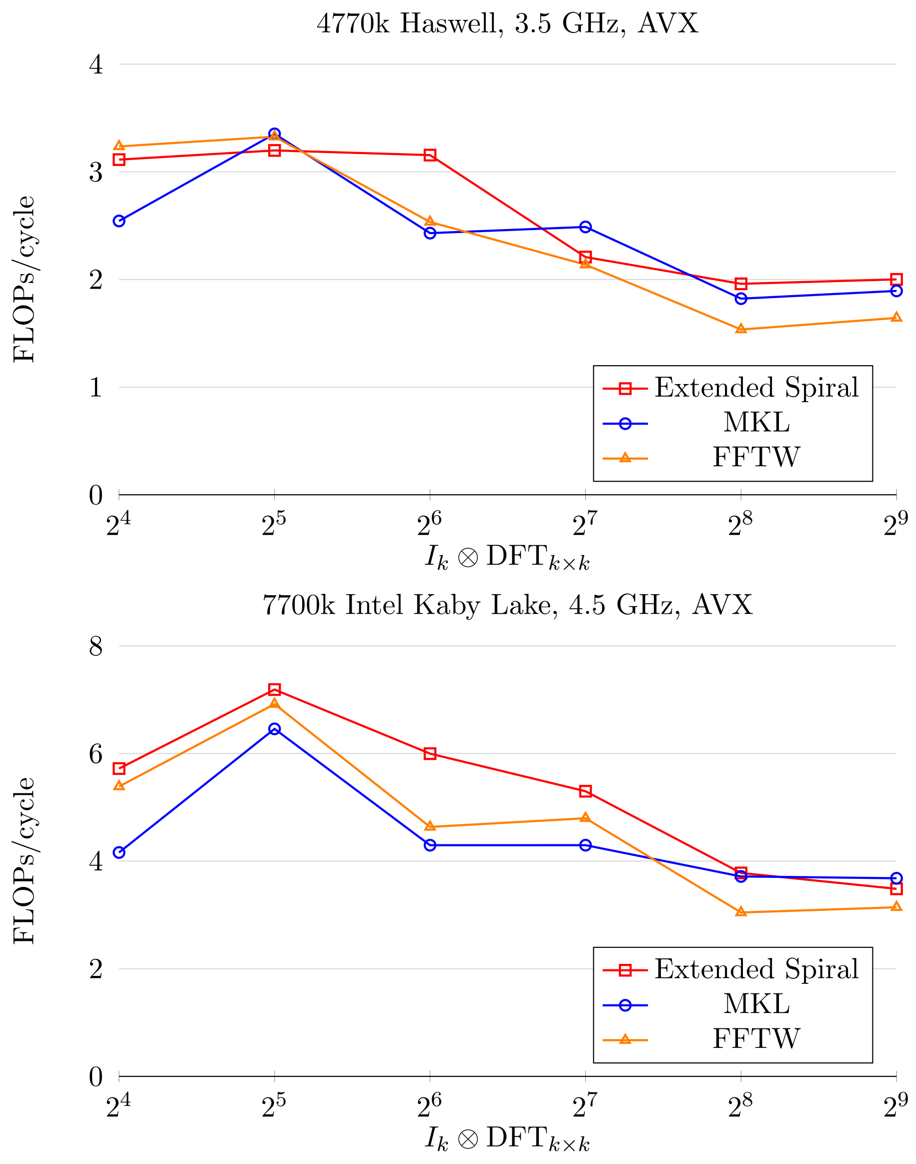


Figure 5.7: Performance results for the batch 2D DFT. The red line represents the performance of the generated code using our in-house code generator, while the other two lines represent the performance of the 2D DFTs using MKL and FFTW.

The DFT is memory latency bound, computation cannot hide the latency of reading

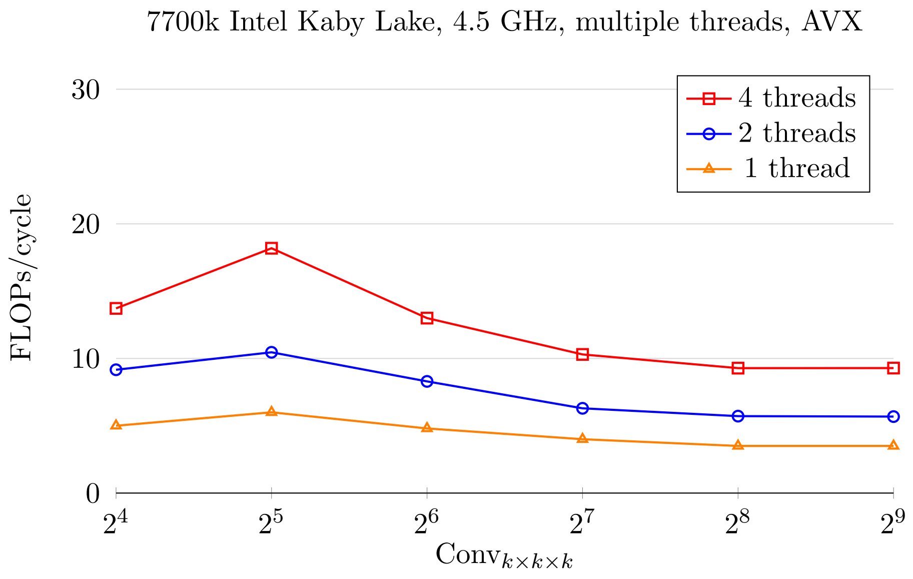
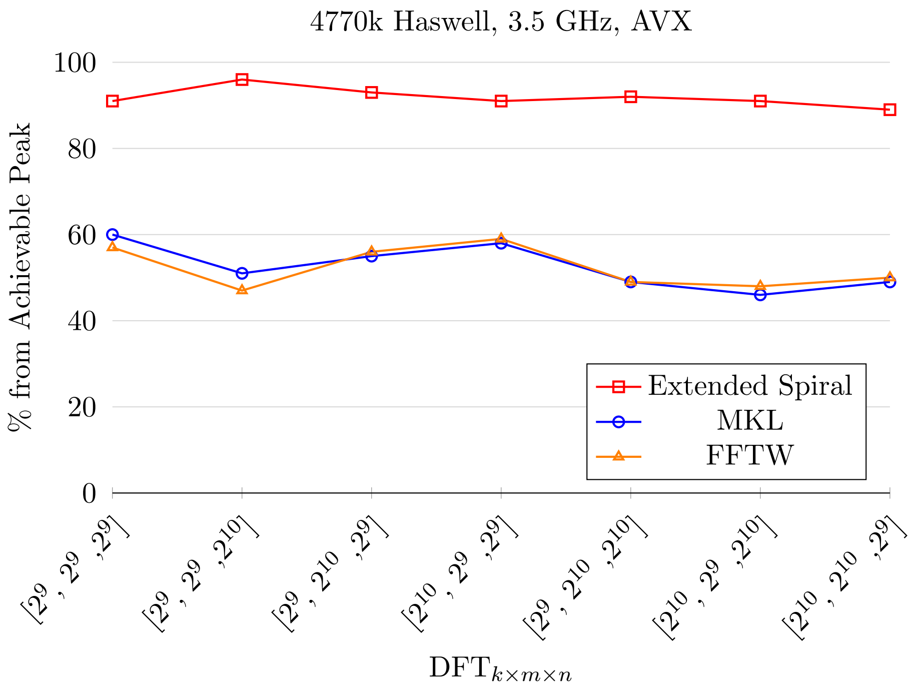


Figure 5.8: Performance results of the 3D DFT convolution when increasing the number of threads from 1 to 4 on the Intel Kaby Lake.

data from main memory.

However, we have shown in [1] that the performance of the DFT can be improved so that it runs at the speed of the bandwidth. In the paper, we show that for large problem sizes that do not fit in the last level cache, one can apply a double buffering technique, which technically is the Out-of-Core [70, 71, 72] execution mapped to a cache syste, to improve the overall execution for multi-dimensional DFTs. The approach is basically software pipelining [73] applied at the granularity of a task, e.g compute and data movement tasks. The threads are split into compute and data threads. The compute threads only apply 1D DFTs on cached data, while data threads move data to and from main memory into a shared local buffer that resides within the last level cache. In Figure 5.9, we show the performance results when using double buffering on large 3D DFTs.



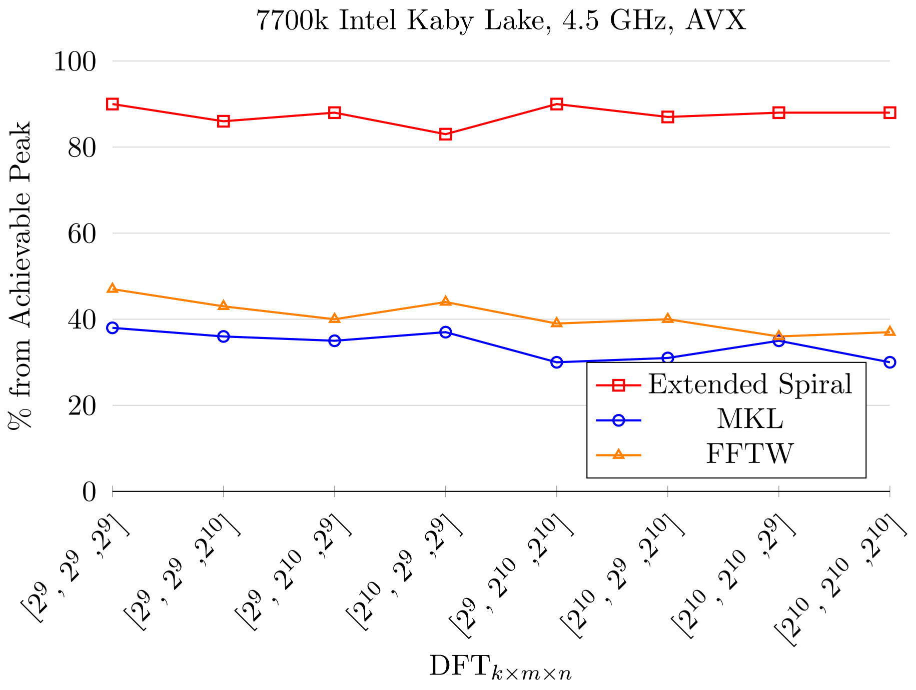


Figure 5.9: The performance results for the optimized version of the 3D DFT using the double buffering approach as suggested in [1]. We compare the performance numbers against the achievable peak performance if bandwidth is efficiently utilized.

We compare the 3D DFT implementation using the double buffering technique against FFTW and MKL and show a performance improvement of almost 3x. In addition, we compare our implementation against the achievable peak when assuming that data is stream at full bandwidth speed determined using the STREAM benchmark [74]. The double buffering approach achieves within 10% of peak performance. Hence, a similar performance boost is expected when implementing the double buffering technique on DFT-based operations.

Overall, fusing can improve performance for 3D DFT-based convolution with non-separable point-wise computation. Depending on the problem sizes, the performance improvement can vary from 1.3 to 1.6x compared to the implementations that do not fuse the middle compute stages. Although fusing the middle stages sees significant performance improvements, the overall performance is overshadowed by the 2D forward and inverse DFT computation that can not be fused. We state that as the dimensions of the multi-dimensional DFT-based operation increases, the effects of merging the middle stages become less obvious. However, performance can further be improved if optimizations such as thread level parallelism [33] or double buffering [1] are used on the DFT computation.

### 5.3.2 Separable Point-wise Multiplication

In this section, we analyze the performance improvements for 3D DFT-based operations where the point-wise computation is separable on the dimensions. We present two cases. First, we tackle the 3D DFT-based differentiation operation for power of two problem sizes. Recall that the differentiation property requires a point-wise computation that is separable on the dimensions. Since MKL does not offer interfaces to easily express this separable operation, we use the basic 3D DFT interface

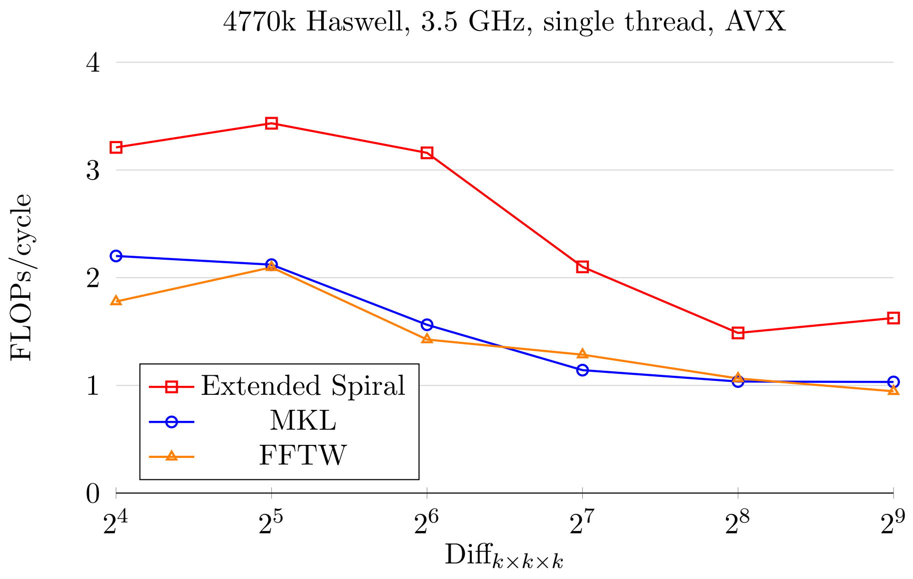
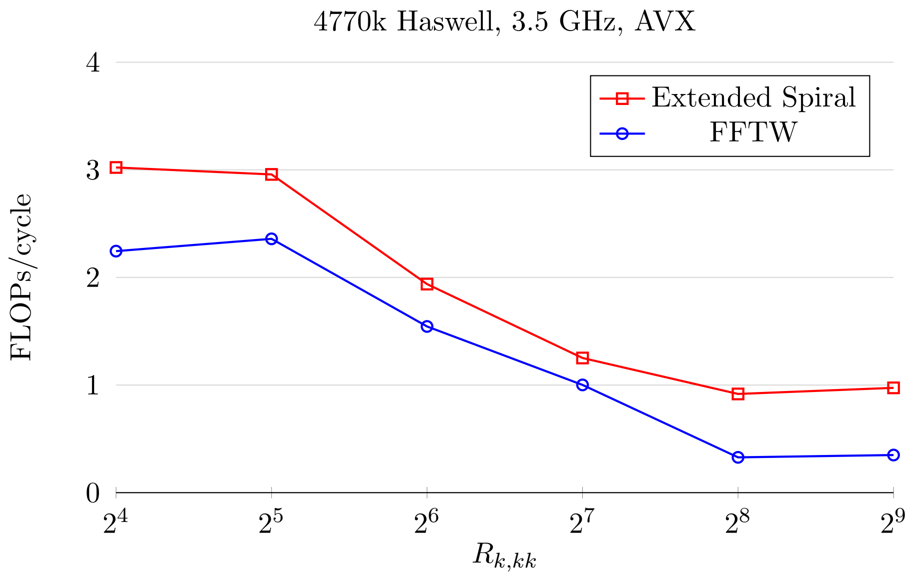


Figure 5.10: Performance plot for the 3D DFT-based differentiation operation on Intel Haswell 4770K. The first implementation (red line) represents the performance achieved by our approach. The other two implementations use MKL and FFTW forthe DFT stages and have hand-written code to compute the point-wise multiplication.

and leave the point-wise computation as if it is non-separable. Second, we present a 3D DFT-based interpolation algorithm based on the shifting in time property (Chapter 2) for problem sizes that are non-power of two. The algorithm is based on the work from [67, 75] and presented in more details in [3]. In the latter, we showed that applying SIMD instructions requires zero-padding operations as described in Chapter 4 and thus having a wider view of the entire algorithm allows one to remove the redundant padding and un-padding operations.

**3D DFT-based Differentiation.** Recall that there are DFT-based operations like differentiation, where the point-wise computation is separable since it exhibits symmetry properties. For example, computing the $\frac{d^3}{dx\,dy\,dz}$ operation in the frequency domain does not require the point-wise multiplication to stall for the full



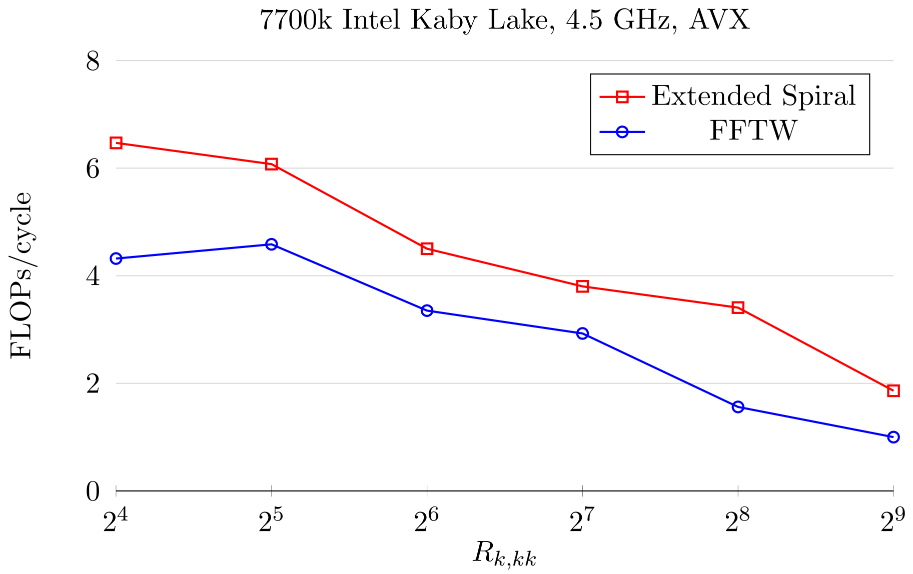


Figure 5.11: The advantages of fusing the compute stages for the construct $R_{k,mn} = \left(\mathrm{DFT}_k^H \cdot D_k \cdot \mathrm{DFT}_k\right) \otimes I_{mn}$ as seen in equation 5.5.

3D DFT. The point-wise multiplication can be separated on all three dimensions. This suggests that the 3D DFT-based operation can be expressed using SPL as follows

$$\begin{aligned}\mathrm{Diff}_{k\times m\times n} = {} & \left(\left(\mathrm{DFT}_k^H \cdot D_k \cdot \mathrm{DFT}_k\right) \otimes I_{mn}\right) \\ & \left(I_k \otimes \left(\mathrm{DFT}_m^H \cdot D_m \cdot \mathrm{DFT}_m\right) \otimes I_n\right) \\ & \left(I_{km} \otimes \left(\mathrm{DFT}_n^H \cdot D_n \cdot \mathrm{DFT}_n\right)\right),\end{aligned} \tag{5.5}$$

where $D_k$, $D_m$ and $D_n$ represent the diagonal matrices that store the pre-computed values of the differentiation operation applied in each dimension. It can be seen that in each dimension, there is a local forward DFT, followed by the local point-wise multiplication and finally a local inverse DFT. Moreover, the computation in each dimension is actually a 1D DFT-based convolution, that can further be fused using the steps described in Chapter 3.

Figure 1.1 showed the performance improvements when fusing the compute stages for the 3D DFT-based differentiation on the Intel Kaby Lake processor. The figure showed that having the view of the entire algorithm and optimizing across the DFT and non-DFT stages improves performance by almost 2x compared to the FFTW and MKL implementations. The same trend can be seen on the Intel Haswell in Figure 5.10, where the generated code (red line) achieves the same 2x. For the FFTW baseline, we use the FFTW Guru interface for the DFT computation and we hand-write the loops around the DFT calls. In addition, we hand-write the point-wise computation using SIMD instructions. The hand-written implementation can be seen in Appendix A. For the MKL implementation, we resort to the 3D DFT function call. MKL does not have an interface similar to the one provided by FFTW. Hence, we keep the point-wise computation as being non-separable and apply the full forward and inverse 3D DFT computation. An alternative solution for MKL would be to hand-write the data movement to rotate the data so that the basic 1D

DFT can be applied. However, the amount of work is too great.

Fusing the compute stages reduces the number of passes through the data and hence improves data locality and performance. However, fusion is not the only optimization that improves the code. The way the code is generated has also an effect on the performance. Figure 5.11 shows the performance improvements for the 1D DFT convolution $R_{k,mn} = \left(\mathrm{DFT}_k^H \cdot D_k \cdot \mathrm{DFT}_k\right) \otimes I_{mn}$, where we compare the generated code using our approach against the implementation that uses the FFTW Guru interface. Our implementation (red line) is cache-aware, while FFTW is cache oblivious. Each read and write are done at the granularity of a cache line, not the granularity of the SIMD vector length. All the temporary arrays required by the forward and inverse DFT are tailored to the size of the L1 and L2 cache. In addition, we force the temporary arrays to be re-used through-out the entire computation. Using all these optimizations, the generated code outperforms the FFTW implementation on both Intel Haswell and Kaby Lake.

Similarly to the non-separable operation, the 3D DFT-based differentiation can also be parallelized. Figure 5.12 shows the performance for the 3D DFT-based differentiation when increasing the number of threads. It can be seen that increasing the number of threads from one to four, performance almost doubles each time. However, this operations still suffers when the problem size increases beyond the last level cache. The performance degrades since the DFT memory accesses are still non-unit stride and computation is still not sufficient to hide the long latency, even though computation is merged. Thus, it makes perfect sense to apply the double buffering technique described in [1] similarly to the non-separable case. Data is brought in the last level cache, and computation can hide the latency of reading data from the last level cache a lot easier.

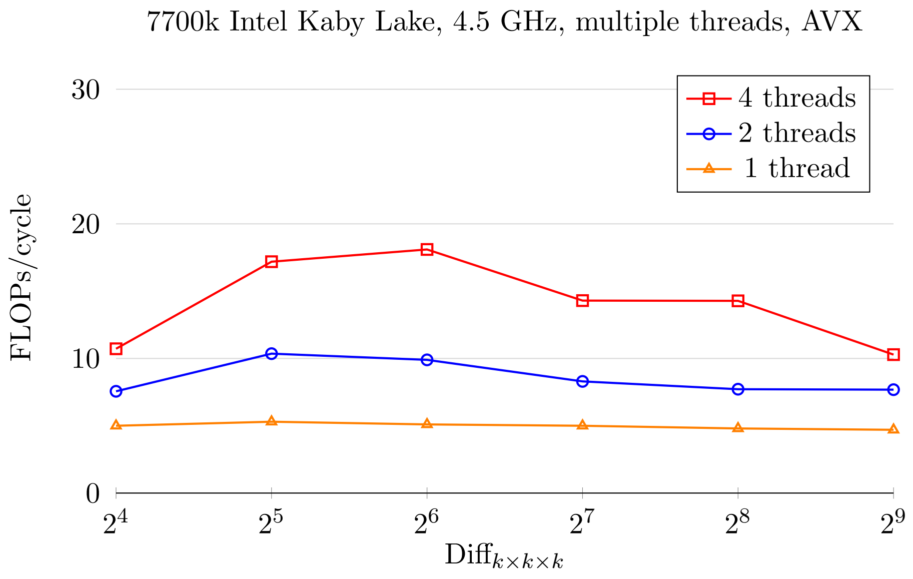


Figure 5.12: Performance results of the 3D DFT differentiation when increasing the number of threads from 1 to 4 on the Intel Kaby Lake.

**3D DFT-Based Interpolation.** Some problems are not always divisible by the SIMD vector length. Some applications require central frequencies, hence they must deal with non power of two problem sizes that are not divisible by the SIMD vector length. The ONETEP project [67] is such an application that requires an interpolation operation to avoid aliasing while performing computation. Basically, the DFT-based interpolation is trigonometric interpolation as the one presented in [76]. For a 1D sequence, the interpolation requires a forward DFT, a point-wise computation that performs a phase shift and an inverse DFT. Following the trend, the operation can be expressed using SPL as

$$Z_n^k = \text{iDFT}_n \cdot D_n^k \cdot \text{DFT}_n, \tag{5.6}$$

with

$$D_n^k = \text{diag}\left(1, \omega_n^k, \omega_n^{k^2} \ldots, \omega_n^{k^{n-1}}\right) \text{ and } \omega_n = \exp(-2\pi j/n).$$

The diagonal is the point-wise computation required when doing a time shift. Recall that for integer values of $k$ the matrix $Z_n^k$ simply performs a cyclic shift of the samples. However, setting the value of $k = 1/2$ allows one to compute the samples located between the original samples. Therefore, the phase shift interpolation defined as $\mathcal{U}_n^2$, can be expressed using SPL as

$$\mathcal{U}_n^2 = L_n^{2n} \begin{bmatrix} I_n \\ Z_n^{1/2} \end{bmatrix}. \tag{5.7}$$

It can be seen that the identity matrix copies the original sequence. The bottom operation performs the time shift by half a sample and the $L$ operator interleaves the values to obtain the interpolated sequence. The algorithm can be extended to three dimensions using the Kronecker product. In addition, the interpolation is a separable operation, so each dimension can be gradually increased by a factor of two. The SPL representation of the entire 3D DFT interpolation is as follows

$$\mathcal{U}_{k\times m\times n}^{2\times 2\times 2} =(\mathcal{U}_k^2 \otimes I_{4mn}) \cdot (I_k \otimes \mathcal{U}_m^2 \otimes I_{2n}) \cdot (I_{km} \otimes \mathcal{U}_n^2). \tag{5.8}$$

It can be seen that the 3D interpolation is very similar to any 3D DFT computation, with the difference that after each compute stage the data-set is increased by a factor of two in each dimension. Computation can be grouped using the properties of the Kronecker product. For example, the interpolation in the $n$ and $m$ dimension can be fused. Hence, the interpolation can first focus on the $mn$-slab and then the interpolation in the $k$ dimension computes the final samples. SIMD vectorization can also be applied. The problem is that the sizes are odd sizes that

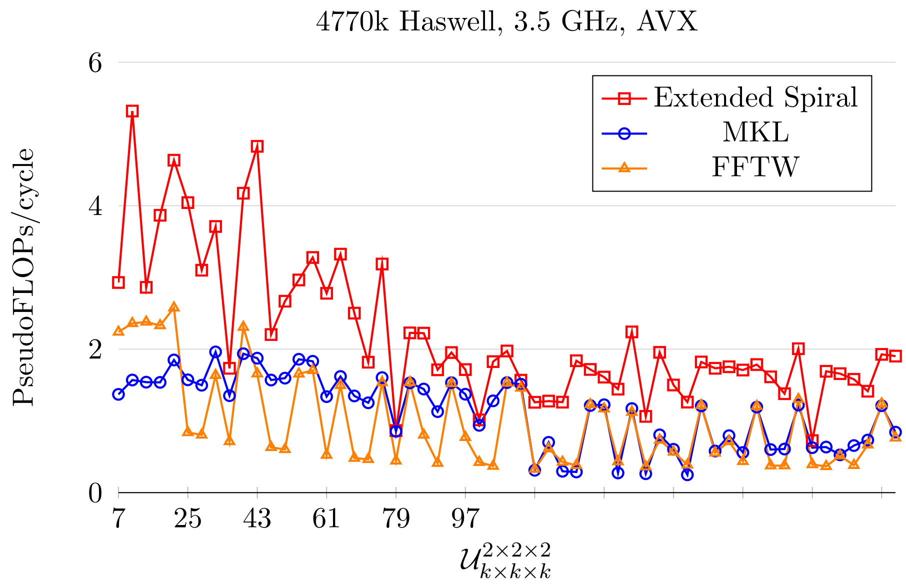


Figure 5.13: The performance results of the 3D DFT-based interpolation for non-power of two problem sizes. The blue line represents the performance achieved by our approach. While the red and brown lines represent the performance of the implementations that use FFTw and MKL for the DFT computation. For this experiment we use PseudoFLOPs/cycle.

are not divisible by the SIMD vector length. However, using the optimizations steps from Chapter 4, the computation can be vectorized with padding and un-padding. In addition, having the entire view of the algorithm reduces the number of padding operations.

Figure 5.13 shows the performance results on the Intel Haswell processor for the 3D DFT-based interpolation implemented using our approach, MKL and FFTW. Since both MKL and FFTW do not allow for the point-wise computation to be merged with the DFT stages, the two implementations required hand-written point-wise computations. In addition, MKL requires extra rotations since the library does not offer an interface like FFTW. Overall, our approach outperformed the two other implementations by almost three times. Details about the performance breakdown can be found in the paper [3]. It is important to state that the problem sizes were non-powers of two. The DFT computation required algorithms like Good-Thomas, Rader or Bluestein. And second, implementing the algorithms using SIMD instructions required zero-padding and un-padding, which are expensive on most Intel systems. Similarly to the permute instructions, execution stalls waiting for data to be padded and un-padded. However, having the wide view of the entire algorithm and optimizing across the DFT and non-DFT stages allows for stages to be fused and also allows for redundant data padding to be discarded. The data is zero-padded efficiently at the beginning of the entire operation. The zeros are smartly inserted and using the information from Chapter 4, they are kept for the entire computation until data needs to be written out.

Overall, separable DFT-based operations see better performance improvements since the number of passes through the data is drastically decreased. Data locality is improved because the forward DFT, point-wise computation and inverse DFT are all applied before writing the data back to memory. Whether the problem size is a power of two or not, our generated codes see almost 3x performance improvements compared to implementations that require glue code around DFT library calls to FFTW and MKL.

## 5.4 Summary

In this chapter we focused on providing a detailed analysis of the benefits of merging and applying other low level optimizations across the DFT and non DFT-stages. First, we discussed the benefits of fusing the stages for 1D DFT-based convolutions. We showed that for problem sizes that fit in the L1 and L2 caches the performance difference is not that significant, however as the problem size increases the generated code achieves almost 1.3x performance improvements compared to the implementations that use FFTW and MKL for the DFT computation. Second, we focused on 3D DFT-based convolution operations for problem sizes that require data to reside in last level cache or main memory. We focused on both separable and non-separable point-wise operations. In both cases, the generated code achieves performance improvements against the competitors. However, non-separable operations see less significant performance improvements compared to the separable ones, since the effects of fusing only the middle stages are overshadowed by the surrounding DFT computation that cannot be merged. Irrespective of the separability property, the performance of any DFT-based operations decreases when data cannot fit in the last cache level. There are other methods like double buffering, but those are orthogonal to optimizations applied to DFT-based operations.

# Chapter 6

# Concluding Remarks

*To re-iterate, the thesis of this dissertation is that for most Fourier-based algorithms there is a systematic way of achieving efficient code through cross-stage optimizations. Most Fourier-based algorithms like convolutions, correlations, interpolations or partial differential equation (PDE) solvers typically follow the same pattern, where discrete Fourier transform (DFT) stages are interleaved with other compute stages. This insight suggests that the focus of optimizations should be on the DFT-based algorithms rather than the DFT itself. Performance gains can be achieved by having a wider view of the entire algorithm, since high level loop transformations and low level optimizations can be applied across compute stages.*

Up to now, there has not been any attempt to expose the implementation of the DFT and merge it with the surrounding computation, although though it is well known that the DFT computation is memory bound. Because of the complexity of writing efficient DFT code, most developers avoid the decomposition of the DFT and thus do not integrate it with the surrounding computation. In most cases, developers resort to writing glue code around highly optimized DFT implementations offered by libraries like MKL and FFTW. Even frameworks like AccFFT or Indigo resort to the library call implementation even though they recognize the fact that the DFT is just part of the entire application.

Therefore, this work makes two contributions. The first contribution is that we identified that most DFT-based operations like convolutions, correlations, interpolations and PDE solvers follow a pattern, where DFT stages are interleaved with other compute stages. Hence, we make the statement that the focus should be on the DFT-based algorithms rather the DFT itself. Libraries should provide support for describing and optimizing entire DFT-based operations. We tackle this aspect first by showing that one can easily develop a simple API to capture most DFT-based operations. The second contribution is that we state that there is a systematic way of applying high level optimizations like loop merging and low level optimizations like SIMD vectorization and zero-padding across the DFT and non-DFT stages, optimizations that are otherwise not possible if the implementation is done using black box library calls to MKL and FFTW. We use the SPL notation to express entire DFT-based algorithms and extend SPIRAL's capabilities to apply all the optimizations automatically.

In this work, we tackled 1D, 2D and 3D DFT-based operations with separable and non-separable point-wise operations. However, we showed results only for 1D and 3D DFT operations. For each case we provided a detailed analysis of the benefits of fusing the computation and we outlined what the main contributors to the improved performance are. In particular, we focused on the following aspects:

- We have showed that for 1D DFT-based convolutions merging the forward and inverse DFT with the point-wise multiplication gives almost 1.3x performance improvements compared to the implementations that use MKL and FFTW for the DFT computation and leave the point-wise operation separated. Although the problem sizes we tackled still fit in the last level cache, we highlighted the fact that merging the point-wise operation with the computation is important. Loading the data to perform the point-wise multiplication is fully amortized by the DFT computation. In addition, data locality is improved and the cache hierarchy is more efficiently utilized.

- Performance improvements can be seen for large 3D DFT-based convolutions with non-separable point-wise, where the problem size require data to be loaded directly from either the last level cache or main memory. The generated code optimized using the steps outlined in Chapter 3 and 4 outperforms by almost 1.6x the implementations that use FFTW and MKL for computing the DFT stages. Performance improvement is achieved even if only some of the stages are merged. The point-wise multiplication does not have any symmetry properties and as such requires the entire data to be fully converted to the frequency domain before applying it. Even so, data locality and thus overall performance is improved.

- The 3D DFT-based convolutions with separable point-wise multiplication see the most benefits when efficiently merging the stages. The point-wise multiplication is separated on the dimensions and thus it can be grouped with the local 1D DFTs. Grouping the forward DFT, point-wise computation and inverse DFT drastically reduces the passes through the data, improving data locality and cache utilization. Hence, we showed that the generated code achieves up to 2.2x performance gains compared to hand-written implementations that use either the FFTW advanced interface or write data packing code around MKL's basic 1D DFT interface. Once again, merging the stages and making the computation cache aware helps achieve performance.

Overall, we showed that focusing on the entire DFT-based algorithms performance can be improved. We have showed that DFT-based operations see performance improvements when applying optimizations like loop fusion across the DFT and non-DFT stages. It is well know that both the DFT computation and the point-wise operation are memory latency bound problems that do not efficiently use the underlying memory hierarchy. Hence, merging the stages improves data locality and cache utilization, thus improves overall performance. *As such, we believe that most memory bound applications should be integrated with the surrounding computation. Achieving performance requires one to express the before and after stages to any memory bound operation and to systematically apply high level optimizations like fusion and other low level optimizations across the entire compute stages.*

## 6.1 Limitations

While we tackle a wide variety of DFT-based operations and show performance improvements, the work has limitations. Some of the limitations are left to be tackled as future work.

- The current state of the API only allows the description of 1D, 2D and 3D DFT-based operations with or without separable point-wise operations. The framework does not allow users to declare more complicated computations that can be done in the frequency domain like matrix-vector multiplications [11] or tensor contractions [47, 77, 78].
- We only focused on complex data sets. However, there are applications that compute the DFT on real data sets. Addressing this problem requires modi-

fications to the code generator so that it can generate real DFTs using SIMD instructions.

- We have only covered DFT-based operations that were computed using the double precision data type. However, there are applications that require different precisions like floats or double double. While the code generator does not need to know the underlying SIMD architecture, the codelets need to be re-implemented taking into account the new data-types.
- We create small codelets for the DFTs of size two and four, where we use SIMD AVX instructions, targetting double precision floating point. In addition, we schedule the code using the information from [79]. Each codelet has different features such as point-wise computations and/or in-register permutations. The permutation may be on the right or left hand side of the DFT computation. While for AVX, double precision there are almost 30 codelets, the number of kernels/codelets changes as the SIMD vector length increases or the data type changes.
- The framework generates code for the memory hierarchy based on empirical results. However, there are works in the linear algebra domain [80, 81, 82, 83] or in the regular mesh methods [84, 85, 86, 87] where it has been shown that key parameters may be chosen based on mathematical models.
- The framework focuses on single and multi-threaded DFT-based operations. However, the parallelism is done in the traditional sense, where each thread receives the same task. While the 3D DFT computation using double buffering is implemented for both single and multisocket systems capable of NUMA domains [88], we have not extended it to any DFT-based operations.

## 6.2 Future work

In this section we discuss the possible directions for future work.

- Make the high level API more flexible to allow users to define different operations more easily and also to allow for more complicated access patterns. This extension may require though a restructuring of the entire API or even a re-definition of the entire front-end.
- Develop a systematic way of generating and optimizing DFT computations given the underlying architecture, namely the capabilities of the computational units and the structure of the memory system. It has been shown that modeling works for other domains, it should also work for generating efficient implementations for both the DFT and DFT-based computations.
- Fully automate the process of applying optimizations like loop fission, loop interchange, loop unrolling in a systematic way. While loop fusion is done automatically, the other optimizations are still done by hand.
- Automate the process of generating the codelets for any SIMD vector length and even GPUs. It has been shown in [89] that based on the execution pipeline, efficient micro-kernels can be automatically generated for computing the matrix-matrix multiplication. Thus, generating DFT kernels can be done in a similar fashion.
- Automatically parallelize the code and automatically apply the double buffering technique when problem sizes require data to be read from main memory.
- Use the framework in actual applications. The framework is a proof of concept to show that having a representation to capture both the DFT and non-DFT

stages is helpful. Optimizations like fusion and data packing can be done across the compute stages and thus improve overall execution time. We tackled a multitude of DFT properties, however the framework should be used within real applications.

# Appendix A

# Using the FFTW Interface To Create and Execute 3D DFT-based Convolutions

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <limits.h>
#include <fftw3.h>
#include <math.h>
#include <immintrin.h>
#include "rdtsc.h"

#define SIZE 512

void compute_pointwise(double *pointwise, double *temp) {
  __m256d *pp = (__m256d*) pointwise;
  __m256d *tt = (__m256d*) temp;
  for(size_t x = 0; x != (SIZE / 2); ++x) {
    __m256d v0 = *(pp);
    __m256d v1 = *(tt);

    __m256d t0 = _mm256_permute_pd(v0, 0 | (0 << 1) | (0 << 2) | (0 <<
    3));
    __m256d t1 = _mm256_permute_pd(v0, 1 | (1 << 1) | (1 << 2) | (1 <<
    3));
    __m256d t2 = _mm256_permute_pd(v1, 1 | (0 << 1) | (1 << 2) | (0 <<
    3));

    __m256d t3 = _mm256_mul_pd(t1, t2);
    t3 = _mm256_fmaddsub_pd(t0, v1, t3);

    *(tt) = t3;

    pp++;
    tt++;
  }
}

int main(int argc, char **argv) {
  int runs = atoi(argv[1]);

  size_t total_size = 2 * SIZE * SIZE * SIZE * sizeof(double);
  size_t buff_size = 2 * SIZE * sizeof(double);

  double *input = (double*) memalign(64, total_size);
  double *output1 = (double*) memalign(64, total_size);

  double *temp = (double*) memalign(64, 4 * buff_size);
  double *pointwiseX = (double*) memalign(64, buff_size);
  double *pointwiseY = (double*) memalign(64, buff_size);
  double *pointwiseZ = (double*) memalign(64, buff_size);

  fftw_iodim many2_dfts_in, single2_dft_in, many2_dfts_out,
    single2_dft_out, many3_dfts_in, single3_dft_in, many3_dfts_out,
    single3_dft_out;
  single2_dft_in.n = SIZE;
  single2_dft_in.is = SIZE;
  single2_dft_in.os = 1;

  single2_dft_out.n = SIZE;
  single2_dft_out.is = 1;
  single2_dft_out.os = SIZE;

  many2_dfts_in.n = 4;
  many2_dfts_in.is = 1;
  many2_dfts_in.os = SIZE;

  many2_dfts_out.n = 4;
  many2_dfts_out.is = SIZE;
  many2_dfts_out.os = 1;

  single3_dft_in.n = SIZE;
  single3_dft_in.is = SIZE * SIZE;
  single3_dft_in.os = 1;

  single3_dft_out.n = SIZE;
  single3_dft_out.is = 1;
  single3_dft_out.os = SIZE * SIZE;
```

```
  many3_dfts_in.n = 4;
  many3_dfts_in.is = 1;
  many3_dfts_in.os = SIZE;

  many3_dfts_out.n = 4;
  many3_dfts_out.is = SIZE;
  many3_dfts_out.os = 1;

#ifdef FFTFORWARD
  fftw_plan plan_forward1 = fftw_plan_dft_1d(SIZE, (fftw_complex*)
    input, (fftw_complex*) temp, FFTW_FORWARD, FFTW_MEASURE);
  fftw_plan plan_backward1 = fftw_plan_dft_1d(SIZE, (fftw_complex*)
    temp, (fftw_complex*) output1, FFTW_BACKWARD, FFTW_MEASURE);
  fftw_plan plan_forward2 = fftw_plan_guru_dft(1, &single2_dft_in, 1, &
    many2_dfts_in, (fftw_complex*) input, (fftw_complex*) temp,
    FFTW_FORWARD, FFTW_MEASURE);
  fftw_plan plan_backward2 = fftw_plan_guru_dft(1, &single2_dft_out, 1,
    &many2_dfts_out, (fftw_complex*) temp, (fftw_complex*) output1,
    FFTW_BACKWARD, FFTW_MEASURE);
  fftw_plan plan_forward3 = fftw_plan_guru_dft(1, &single3_dft_in, 1, &
    many3_dfts_in, (fftw_complex*) input, (fftw_complex*) temp,
    FFTW_FORWARD, FFTW_MEASURE);
  fftw_plan plan_backward3 = fftw_plan_guru_dft(1, &single3_dft_out, 1,
    &many3_dfts_out, (fftw_complex*) temp, (fftw_complex*) output1,
    FFTW_BACKWARD, FFTW_MEASURE);
#else
  fftw_plan plan_forward1 = fftw_plan_dft_1d(SIZE, (fftw_complex*)
    input, (fftw_complex*) temp, FFTW_BACKWARD, FFTW_MEASURE);
  fftw_plan plan_backward1 = fftw_plan_dft_1d(SIZE, (fftw_complex*)
    temp, (fftw_complex*) output1, FFTW_FORWARD, FFTW_MEASURE);
  fftw_plan plan_forward2 = fftw_plan_guru_dft(1, &single2_dft_in, 1, &
    many2_dfts_in, (fftw_complex*) input, (fftw_complex*) temp,
    FFTW_BACKWARD, FFTW_MEASURE);
  fftw_plan plan_backward2 = fftw_plan_guru_dft(1, &single2_dft_out, 1,
    &many2_dfts_out, (fftw_complex*) temp, (fftw_complex*) output1,
    FFTW_FORWARD, FFTW_MEASURE);
  fftw_plan plan_forward3 = fftw_plan_guru_dft(1, &single3_dft_in, 1, &
    many3_dfts_in, (fftw_complex*) input, (fftw_complex*) temp,
    FFTW_BACKWARD, FFTW_MEASURE);
  fftw_plan plan_backward3 = fftw_plan_guru_dft(1, &single3_dft_out, 1,
    &many3_dfts_out, (fftw_complex*) temp, (fftw_complex*) output1,
    FFTW_FORWARD, FFTW_MEASURE);
#endif

  for(size_t i = 0; i < 2 * SIZE * SIZE * SIZE; ++i) {
    input[i] = (double) (rand() / ((double) INT_MAX));
    output1[i] = 0.0;
  }

  double value;
  FILE *fpoinwise = fopen("pointwise.bin", "rb");

  for(size_t i = 0; i < 2 * SIZE; ++i) {
    fread(&value, sizeof(double), 1, fpointwise);
    pointwiseX[i] = value;
  }

  for(size_t i = 0; i < 2 * SIZE; ++i) {
    fread(&value, sizeof(double), 1, fpointwise);
    pointwiseY[i] = value;
  }
```

```

for(size_t i = 0; i < 2 * SIZE; ++i) {
  fread(&value, sizeof(double), 1, fpointwise);
  pointwiseZ[i] = value;
}

fclose(fpointwise);

long long sum0 = 0, sum1 = 0;
tsc_counter t0, t1;

for(int i = 0; i != runs; ++i) {
  RDTSC(t0);
  for(size_t j = 0; j != SIZE; ++j) {
    double *inputx = (input + j * 2 * SIZE * SIZE);
    double *outputx = (output1 + j * 2 * SIZE * SIZE);

    for(size_t k = 0; k != SIZE; ++k) {
fftw_execute_dft(plan_forward1, (fftw_complex*) (inputx + k * 2 *
  SIZE), (fftw_complex*) temp);
compute_pointwise(pointwiseX, temp);
fftw_execute_dft(plan_backward1, (fftw_complex*) temp, (fftw_complex
  *) (outputx + k * 2 * SIZE));
    }
    for(size_t k = 0; k != (SIZE / 4); ++k) {
fftw_execute_dft(plan_forward2, (fftw_complex*) (outputx + 8 * k), (
  fftw_complex*) temp);
for(size_t r = 0; r != 4; ++r) {
  compute_pointwise(pointwiseX, (temp + 2 * SIZE * r));
}
fftw_execute_dft(plan_backward2, (fftw_complex*) temp, (fftw_complex
  *) (outputx + 8 * k));
    }
  }
  for(size_t k = 0; k != (SIZE * SIZE / 4); ++k) {
    fftw_execute_dft(plan_forward3, (fftw_complex*) (output1 + 8 * k)
  , (fftw_complex*) temp);
    for(size_t r = 0; r != 4; ++r) {
compute_pointwise(pointwiseX, (temp + 2 * SIZE * r));
    }
    fftw_execute_dft(plan_backward3, (fftw_complex*) temp, (
  fftw_complex*) (output1 + 8 * k));
  }
  RDTSC(t1);

  sum0 += COUNTER_DIFF(t1, t0, CYCLES);
}

printf("Correctness:\t%lf\tFFTW:\t%lf\n", output1[0], (double) (sum0
  / (runs * 1.0)));

fftw_destroy_plan(plan_forward1);
fftw_destroy_plan(plan_backward1);
fftw_destroy_plan(plan_forward2);
fftw_destroy_plan(plan_backward2);
fftw_destroy_plan(plan_forward3);
fftw_destroy_plan(plan_backward3);
free(input);
free(temp);
free(pointwiseX);
free(pointwiseY);
free(pointwiseZ);
free(output1);

return 0;
}
```

# Appendix B

# Using the API To Create and Execute 3D DFT-based Convolutions

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <limits.h>
#include <math.h>
#include "rdtsc.h"
#include "include/spiral_header.h"

#define SIZE 512

int main(int argc, char **argv) {
  int runs = atoi(argv[1]);

```

```
size_t total_size = 2 * SIZE * SIZE * SIZE * sizeof(double);
size_t buff_size = 2 * SIZE * sizeof(double);

double *input = (double*) memalign(64, total_size);
double *output1 = (double*) memalign(64, total_size);

double *pointwiseX = (double*) memalign(64, buff_size);
double *pointwiseY = (double*) memalign(64, buff_size);
double *pointwiseZ = (double*) memalign(64, buff_size);

spiral_dft dft_rows[1], dft_cols[1], dft_depth[1];
spiral_dft_op op[3];

dft_rows[0].size = SIZE;
dft_rows[0].istr = 1;
dft_rows[0].ostr = 1;

dft_cols[0].size = SIZE;
dft_cols[0].istr = SIZE;
dft_cols[0].ostr = SIZE;

dft_depth[0].size = SIZE;
dft_depth[0].istr = SIZE * SIZE;
dft_depth[0].ostr = SIZE * SIZE;

spiral_pointwise sr, sc, sd;
sr.data_pointer pointwiseX;
sc.data_pointer pointwiseY;
sd.data_pointer pointwiseZ;

op[0].dims = 1;
```

```
  op[0].forward_op = dft_rows;
  op[0].backward_op = dft_rows;
  op[0].pointer = &sr;

  op[1].dims = 1;
  op[1].forward_op = dft_cols;
  op[1].backward_op = dft_cols;
  op[1].pointer = &sc;

  op[2].dims = 1;
  op[2].forward_op = dft_depth;
  op[2].backward_op = dft_depth;
  op[2].pointer = &sd;

#ifdef FFTFORWARD
  spiral_plan *plan = spiral_create_advance_op(op, 3, SPIRAL_FORWARD |
    SPIRAL_DATAPOINTER);
#else
  spiral_plan *plan = spiral_create_advance_op(op, 3, SPIRAL_BACKWARDD
    | SPIRAL_DATAPOINTER);
#endif

  for(size_t i = 0; i < 2 * SIZE * SIZE * SIZE; ++i) {
    input[i] = (double) (rand() / ((double) INT_MAX));
    output1[i] = 0.0;
  }

  double value;
  FILE *fpoinwise = fopen("pointwise.bin", "rb");

  for(size_t i = 0; i < 2 * SIZE; ++i) {
    fread(&value, sizeof(double), 1, fpointwise);
    pointwiseX[i] = value;
  }

  for(size_t i = 0; i < 2 * SIZE; ++i) {
    fread(&value, sizeof(double), 1, fpointwise);
    pointwiseY[i] = value;
  }

  for(size_t i = 0; i < 2 * SIZE; ++i) {
    fread(&value, sizeof(double), 1, fpointwise);
    pointwiseZ[i] = value;
  }

  fclose(fpointwise);

  long long sum0 = 0, sum1 = 0;
  tsc_counter t0, t1;

  for(int i = 0; i != runs; ++i) {
    RDTSC(t0);
    spiral_execute(plan, output1, input);
    RDTSC(t1);

    sum0 += COUNTER_DIFF(t1, t0, CYCLES);
  }

  printf("Correctness:\t%lf\tSPIRAL:\t%lf\n", output1[0], (double) (
    sum0 / (runs * 1.0)));

  spiral_destroy(&plan);
  free(input);
  free(output1);
  free(pointwiseX);
  free(pointwiseY);
  free(pointwiseZ);

  return 0;
}
```

# Bibliography

[1] T. Popovici, T.-M. Low, and F. Franchetti, “Large bandwidth-efficient FFTs on multicore and multi-socket systems,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2018. (document), 2.7, 2.8, 5.3.1, 5.9, 5.3.1, 5.3.2

[2] D. Popovici, F. Franchetti, and T. M. Low, “Mixed data layout kernels for vectorized complex arithmetic,” in *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*, pp. 1–7, 2017. (document), 4.10, 4.3.2, 4.11, 5.2

[3] T. Popovici, F. Russell, K. Wilkinson, C.-K. Skylaris, P. H. J. Kelly, and F. Franchetti, “Generating optimized Fourier interpolation routines for density functional theory using SPIRAL,” in *Workshop on Compilers for Parallel Computing (CPC)*, 2015. (document), 2.3.3, 4.3.4, 4.13, 4.3.4, 5.1, 5.1, 5.3.2, 5.3.2

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, “The landscape of parallel computing research: A view from berkeley,” tech. rep., Techni-

cal Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. 1.1, 2.1

[5] S. Plimpton, R. Pollock, and 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. 1.1

[6] T. W. Sirk, S. Moore, and E. F. Brown, “Characteristics of thermal conductivity in classical water models,” *The Journal of chemical physics*, vol. 138, no. 6, p. 064505, 2013. 1.1

[7] S. J. Plimpton and A. P. Thompson, “Computational aspects of many-body potentials,” *MRS bulletin*, vol. 37, no. 5, pp. 513–521, 2012. 1.1

[8] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann, “Hacc: Extreme scaling and performance across diverse architectures,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’13, (New York, NY, USA), pp. 6:1–6:10, ACM, 2013. 1.1

[9] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, *et al.*, “Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures,” *New Astronomy*, vol. 42, pp. 49–65, 2016. 1.1, 2.2.6

[10] C.-S. Chang, S. Ku, and H. Weitzner, “Numerical study of neoclassical plasma pedestal in a tokamak geometry,” *Physics of Plasmas*, vol. 11, no. 5, pp. 2649–2667, 2004. 1.1

[11] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko,

R. Lehe, A. Myers, C. Ng, *et al.*, “Warp-X: A new exascale computing platform for beam–plasma simulations,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 2018. 1.1, 6.1

[12] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, *et al.*, “High performance computational chemistry: An overview of NWChem a distributed parallel application,” *Computer Physics Communications*, vol. 128, no. 1-2, pp. 260–283, 2000. 1.1

[13] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, *et al.*, “NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations,” *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010. 1.1

[14] T. Straatsma, E. Bylaska, H. van Dam, N. Govind, W. de Jong, K. Kowalski, and M. Valiev, “Advances in scalable computational chemistry: NWChem,” in *Annual Reports in Computational Chemistry*, vol. 7, pp. 151–177, Elsevier, 2011. 1.1

[15] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, “Nyx: A massively parallel amr code for computational cosmology,” *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013. 1.1

[16] Intel, “Math Kernel Library.” http://developer.intel.com/software/products/mkl/, 2018. 1.1, 1.2.1

[17] M. Frigo and S. G. Johnson, "Fftw: Fastest fourier transform in the west," *Astrophysics Source Code Library*, 2012. 1.1

[18] Nvidia, "Nvidia cuda based fft library." https://developer.nvidia.com/cufft, 2018. 1.1

[19] arm, "Arm performance library." https://developer.arm.com/products/software-development-tools/hpc/arm-performance-libraries/getting-started, 2018. 1.1

[20] J. Ullman, A. V. Aho, and R. Sethi, "Compilers: Principles, techniques and tools," 2018. 1.1, 1.2.1, 3.1.1

[21] A. Gholami, J. Hill, D. Malhotra, and G. Biros, "Accfft: A library for distributed-memory fft on cpu and gpu architectures," *arXiv preprint arXiv:1506.07933*, 2015. 1.2, 1.2.2

[22] A. Gholami, A. Mang, K. Scheufele, C. Davatzikos, M. Mehl, and G. Biros, "A framework for scalable biophysics-based image analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 19, ACM, 2017. 1.2, 1.2.2

[23] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," 2012. 1.2, 1.2.2

[24] M. Driscoll, B. Brock, F. Ong, J. Tamir, H.-Y. Liu, M. Lustig, A. Fox, and K. Yelick, "Indigo: A domain-specific language for fast, portable image reconstruction," 1.2, 1.2.2

[25] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005. 1.2.1, 3.1.2

[26] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992. 1.2.1, 2.3, 2.3.4

[27] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. Programming Language Design and Implementation (PLDI)*, pp. 298–308, 2001. 1.2.1, 2.3

[28] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura, "Discrete Fourier transform on multicores: Algorithms and automatic implementation," *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, vol. 26, no. 6, pp. 90–102, 2009. 1.2.1

[29] M. Püschel and J. M. F. Moura, "The algebraic approach to the discrete cosine and sine transforms and their fast algorithms," *SIAM Journal of Computing*, vol. 32, no. 5, pp. 1280–1316, 2003. 1.2.1

[30] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. of Computation*, vol. 19, pp. 297–301, 1965. 1.2.1

[31] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal

transforms," in *Programming Languages Design and Implementation (PLDI)*, pp. 315–326, 2005. 1.2.1, 3.1.2, 3.2.1, 3.2.1, 5.2

[32] F. Franchetti, M. Püschel, J. M. F. Moura, and C. W. Ueberhuber, "Short vector SIMD code generation for DSP algorithms," in *High Performance Extreme Computing (HPEC)*, 2002. 1.2.1

[33] F. Franchetti, Y. Voronenko, and M. Püschel, "FFT program generation for shared memory: SMP and multicore," in *Supercomputing (SC)*, 2006. 1.2.1, 2.3.1, 5.3.1

[34] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 216–231, 2005. 1.2.1, 4.2.1, 4.2.4, 5.1

[35] S. G. Johnson and M. Frigo, "A modified split-radix FFT with fewer arithmetic operations," *IEEE Trans. Signal Processing*, vol. 55, no. 1, pp. 111–119, 2007. 1.2.1

[36] F. F. T. in The East, "Ffte." `http://www.ffte.jp/`, 2018. 1.2.1

[37] D. Takahashi and Y. Kanada, "High-performance radix-2, 3 and 5 parallel 1-d complex fft algorithms for distributed-memory parallel computers," *The Journal of Supercomputing*, vol. 15, pp. 207–228, Feb 2000. 1.2.1

[38] D. Takahashi, "A parallel 1-d fft algorithm for the hitachi sr8000," *Parallel Computing*, vol. 29, no. 6, pp. 679–690, 2003. 1.2.1

[39] D. Takahashi, "An implementation of parallel 3-d fft with 2-d decomposition on a massively parallel cluster of multi-core processors," in *International Confer-*

*ence on Parallel Processing and Applied Mathematics*, pp. 606–614, Springer, 2009. 1.2.1

[40] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013. 1.2.2

[41] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, “Automatically scheduling halide image processing pipelines,” *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, p. 83, 2016. 1.2.2

[42] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic linear algebra subprograms for Fortran usage,” *ACM Trans. Math. Soft.*, vol. 5, pp. 308–323, Sept. 1979. 1.2.2

[43] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, “An extended set of FORTRAN basic linear algebra subprograms,” *ACM Trans. Math. Soft.*, vol. 14, pp. 1–17, March 1988. 1.2.2

[44] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, “A set of level 3 basic linear algebra subprograms,” *ACM Trans. Math. Soft.*, vol. 16, pp. 1–17, March 1990. 1.2.2

[45] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, “Efficient algorithms for all-to-all communications in multiport message-passing systems,” *IEEE Transactions on parallel and distributed systems*, vol. 8, no. 11, pp. 1143–1156, 1997. 1.2.2

[46] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, “Signals and systems, vol. 2,” *Prentice-Hall Englewood Cliffs, NJ*, vol. 6, no. 7, p. 10, 1983. 2.1.1, 2.2.6

[47] R. A. Lebensohn, A. K. Kanjarla, and P. Eisenlohr, “An elasto-viscoplastic formulation based on fast fourier transforms for the prediction of micromechanical fields in polycrystalline materials,” *International Journal of Plasticity*, vol. 32, pp. 59–69, 2012. 2.2.1, 6.1

[48] S. G. Johnson, “Notes on FFT-based differentiation,” *MIT Applied Mathematics*, no. April, 2011. 2.2.6, 2.2.6

[49] J. Starn, “A simple fluid solver based on the FFT,” *Journal of graphics tools*, vol. 6, no. 2, pp. 43–52, 2001. 2.2.6

[50] C. F. Van Loan, “The ubiquitous Kronecker product,” *Journal of computational and applied mathematics*, vol. 123, no. 1-2, pp. 85–100, 2000. 2.3

[51] F. Franchetti and M. Püschel, “Short vector code generation for the discrete Fourier transform,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2003. 2.3.1

[52] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “SPIRAL: Code generation for DSP transforms,” *Proceedings of the IEEE*, vol. 93, no. 2, 2005. special issue on ”Program Generation, Optimization, and Adaptation”. 2.3.1

[53] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965. 2.3.4

[54] P. N. Swarztrauber, "FFT algorithms for vector computers," *Parallel Computing*, vol. 1, no. 1, pp. 45–63, 1984. 2.3.4

[55] C. M. Rader, "Discrete fourier transforms when the number of data samples is prime," *Proceedings of the IEEE*, vol. 56, no. 6, pp. 1107–1108, 1968. 2.3.4

[56] L. Rabiner, R. Schafer, and C. Rader, "The chirp z-transform algorithm," *IEEE Transactions on Audio and Electroacoustics*, vol. 17, pp. 86–92, June 1969. 2.3.4

[57] P. N. Swarztrauber, R. A. Sweet, W. L. Briggs, J. Otto, *et al.*, "Bluestein's fft for arbitrary n on the hypercube," *Parallel computing*, vol. 17, no. 6-7, pp. 607–617, 1991. 2.3.4

[58] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Languages and Compilers for Parallel Computing* (U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds.), (Berlin, Heidelberg), pp. 301–320, Springer Berlin Heidelberg, 1994. 3, 3.1.1

[59] T. M. Low, "A calculus of loop invariants for dense linear algebra optimization," 2013. 3.1.1

[60] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Paper*, pp. 1–21, 2011. 4.3.1

[61] V. G. Reddy, "Neon technology introduction," *ARM Corporation*, 2008. 4.3.1

[62] S. Carr and K. Kennedy, "Compiler blockability of numerical algorithms," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pp. 114–124, IEEE Computer Society Press, 1992. 4.3.3

[63] D. Nuzman and A. Zaks, "Outer-loop vectorization: revisited for short SIMD architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 2–11, ACM, 2008. 4.3.3

[64] F. Franchetti, Y. Voronenko, and M. Püschel, "A rewriting system for the vectorization of signal transforms," in *High Performance Computing for Computational Science (VECPAR)*, vol. 4395 of *Lecture Notes in Computer Science*, pp. 363–377, Springer, 2006. 4.3.3

[65] F. Franchetti and M. Püschel, "Generating SIMD vectorized permutations," in *International Conference on Compiler Construction (CC)*, vol. 4959 of *Lecture Notes in Computer Science*, pp. 116–131, Springer, 2008. 4.3.3

[66] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, "Automatic simd vectorization of fast fourier transforms for the larrabee and avx instruction sets," in *Proceedings of the International Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 265–274, ACM, 2011. 4.3.3

[67] C.-K. Skylaris, P. D. Haynes, A. A. Mostofi, and M. C. Payne, "Introducing ONETEP: Linear-scaling density functional simulations on parallel computers," *The Journal of Chemical Physics*, vol. 122, no. 8, p. 084119, 2005. 4.3.4, 5.3.2, 5.3.2

[68] F. Franchetti and M. Püschel, "SIMD vectorization of non-two-power sized FFTs," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 2, pp. II–17, 2007. 4.3.4

[69] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. 40th Ann. Symp. on Foundations of Comp. Sci. (FOCS)*, pp. 285–297, IEEE Comput. Soc., 1999. 5.3.1

[70] D. H. Bailey, "FFTs in external of hierarchical memory," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, pp. 234–242, ACM, 1989. 5.3.1

[71] W. M. Gentleman and G. Sande, "Fast Fourier Transforms: For fun and Profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pp. 563–578, ACM, 1966. 5.3.1

[72] T. H. Cormen, J. Wegmann, and D. M. Nicol, "Multiprocessor out-of-core ffts with distributed memory and parallel disks (extended abstract)," in *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '97, (New York, NY, USA), pp. 68–78, ACM, 1997. 5.3.1

[73] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, (New York, NY, USA), pp. 318–328, ACM, 1988. 5.3.1

[74] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995. 5.3.1

[75] F. Russell, K. A. Wilkinson, P. Kelly, and C.-K. Skylaris, "Optimised three-dimensional fourier interpolation: An analysis of techniques and application to a linear-scaling density functional theory code," vol. 187, 10 2014. 5.3.2

[76] L. Yaroslavsky, "Efficient algorithm for discrete sinc interpolation," *Applied Optics*, vol. 36, no. 2, pp. 460–463, 1997. 5.3.2

[77] R. A. Lebensohn, “N-site modeling of a 3d viscoplastic polycrystal using fast fourier transform,” *Acta materialia*, vol. 49, no. 14, pp. 2723–2737, 2001. 6.1

[78] A. Kanjarla, R. Lebensohn, L. Balogh, and C. Tomé, “Study of internal lattice strain distributions in stainless steel using a full-field elasto-viscoplastic formulation based on fast fourier transforms,” *Acta Materialia*, vol. 60, no. 6-7, pp. 3094–3106, 2012. 6.1

[79] R. Veras, D. T. Popovici, T. M. Low, and F. Franchetti, “Compilers, hands-off my hands-on optimizations,” in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, WPMVP ’16, (New York, NY, USA), pp. 4:1–4:8, ACM, 2016. 6.1

[80] K. Goto and R. van de Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Trans. Math. Soft.*, vol. 34, pp. 12:1–12:25, May 2008. 6.1

[81] K. Yotov, X. Li, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill, “Is search really necessary to generate high-performance BLAS?,” *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”*, vol. 93, no. 2, 2005. 6.1

[82] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, “Analytical modeling is enough for high-performance BLIS,” *ACM Trans. Math. Softw.*, vol. 43, pp. 12:1–12:18, August 2016. 6.1

[83] T. M. Smith, R. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, “Anatomy of high-performance many-threaded matrix multiplication,” in *IPDPS ’14: Proceedings of the International Parallel and Distributed Processing Symposium*, 2014. To appear. 6.1

[84] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache, "Iterative optimization in the polyhedral model: Part i, one-dimensional time," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, (Washington, DC, USA), pp. 144–156, IEEE Computer Society, 2007. 6.1

[85] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 13–24, ACM, 2013. 6.1

[86] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *ACM SIGPLAN Notices*, vol. 49, pp. 65–76, ACM, 2014. 6.1

[87] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, *et al.*, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pp. 138–149, IEEE, 2015. 6.1

[88] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996. 6.1

[89] R. M. Veras, T. M. Low, T. M. Smith, R. A. van de Geijn, and F. Franchetti, "Automating the last-mile for high performance dense linear algebra," *CoRR*, vol. abs/1611.08035, 2016. 6.2