IMPLEMENTING AND OPTIMIZING FAST FOURIER TRANSFORMS IN LIFT

BASTIAN KÖPCKE

Masterthesis
Computer Science Department
Parallel and Distributed Systems
University of Münster

FIRST SUPERVISOR: Prof. Dr. habil. Sergei Gorlatch
(University of Münster)

SECOND SUPERVISOR: Prof. Dr. Herbert Kuchen
(University of Münster)

September 2018

ACKNOWLEDGMENTS

First of all, I thank my supervisor Sergei Gorlatch for giving me the freedom to pursue my research interest. I thank Bastian Hagedorn who pointed me to my research topic and could always be relied upon when I had questions regarding Lift. Furthemore, many thanks go to Jens Gutsfeld who was a great help in the last phase of this thesis and had to deal with my awful punctuation. Last but not least, I thank my family for their support of my studies over all these years.

CONTENTS

LIST OF FIGURES

LISTINGS

1 INTRODUCTION AND BACKGROUND

In modern high-performance computing, Graphics Processing Units (GPUs) provide an inherently parallel architecture that can be exploited to speed up the computations of applications such as Fast Fourier Transforms (FFTs) and many others. Fast Fourier Transforms form a class of divide and conquer algorithms that efficiently compute the Discrete Fourier Transform (DFT). They have a wide range of applications in physics [3], biochemistry [15], image manipulation [36], digital signal processing [27] and deep neural networks [28] and a lot of research has been directed towards them in the past. Some applications of FFTs (e. g. in deep neural networks) rely on large amounts of sometimes very large FFTs. Therefore, it is advantageous to optimize the performance of their computation as much as possible. In this thesis, we investigate the expression of FFTs in the high-level functional language Lift as compositions of its high-level functional primitives. Preserving the semantics, we rewrite the expressions into an Intermediate Language (IL) that is based upon low-level functional data parallel primitives for GPUs. Then, the rewritten expressions are used to generate executable OpenCL code.

1.1 MOTIVATION

Fast Fourier Transforms are challenging to implement efficiently on GPUs because of their often irregular data access patterns. An initial recursively formulated version was provided by Cooley and Tukey [5]. The algorithm works by dividing the input sequence to the FFT into smaller parts on which the FFT can be applied recursively. The way the input sequence is sorted and split depends on given factorizations of the input size. Also, FFTs can be expressed using sparse matrix factorizations of the DFT, hence providing sequential algorithms [44] that can then be rewritten for execution on GPUs exploiting the parallelism provided by the GPU's large number of cores.

While GPUs can be used as parallel accelerators in many applications, they are known to be notoriously hard to program. The added complexity when programming for parallel systems makes it challenging to achieve correctness and even more so efficiency. Therefore, the effort that needs to be put into programming high performing GPU applications grows increasingly with the application's complexity. In his Ph.D. thesis [4], Cole introduced the idea of abstracting recurring patterns in parallel algorithms into easy to use functional algorithmic skeletons. This approach has been built upon by Lift [17, 38,

41] which uses expressions of high-level functional primitives to generate high-performance code for modern GPUs. The Lift framework transforms high-level expressions into expressions in an intermediate language based on low-level functional data parallel primitives that are closely related to the OpenCL framework. For that transformation, it uses a formal rewrite system that is based on rewrite rules.

FFT algorithms suited for GPUs that work on arbitrary factorizations of the input sizes have been published in [14, 26]. These are based on a version of FFT called Stockham formulation and are expressed in a pseudo form of low-level languages similar to CUDA or OpenCL. While Govindaraju et al. [14] provide different algorithms, these algorithms mainly differ in hardware specific optimizations for different sizes of the input sequence rather than changes of the underlying algorithm. This emphasizes the fact that the performance of programs written in low-level languages such as OpenCL or CUDA is highly dependent on the underlying hardware, e. g. number of cores and memory hierarchy, and thus expert knowledge on the side of the programmer is necessary [1, 30].

One attempt to solve this problem are highly optimized library implementations. However, the performance of these libraries is platform specific and hardware changes make new optimizations by hand necessary. Therefore, Lift offers a promising approach to programming FFTs in a performance-portable manner that has worked well with other applications before [18, 34, 40].

The main goal of this thesis is to find a way to express different FFT algorithms in Lift in order to generate high-performance OpenCL kernels for different GPU architectures.

1.2 BACKGROUND

This section is dedicated to providing important background information on concepts that are needed to understand this thesis. We start by explaining the general architecture of Graphics Processing Units (GPUs) and their differences to multi-core Central Processing Units (CPUs), followed by the OpenCL framework that is used to create programs that can be executed on GPUs. Then, we give an overview over Fast Fourier Transforms and their underlying ideas. Finally, the Lift system itself is introduced.

1.2.1 *Graphics Processing Unit*

Figure 1.1 shows abstract overviews of typical multi-core CPU architectures and GPU architectures. The Central Processing Unit (CPU) consists of several cores in combination with two levels of caches. An L3 cache that is very large is accessible by all the cores. These caches are used to hide memory latencies when requesting memory

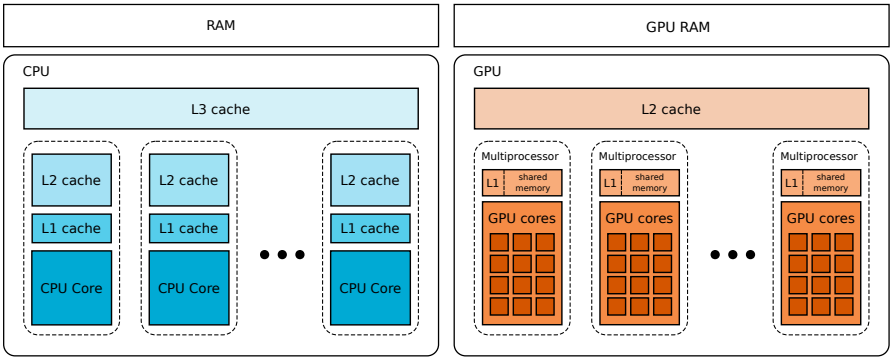

Figure 1.1: Abstract overview of a CPU and GPU architecture (from [16]).

accesses from the *Random Access Memory (RAM)*. The architecture of a CPU core itself is highly sophisticated. It consists of multiple execution units and provides several technologies to execute instructions as quickly as possible (e. g. branch prediction). Furthermore, every core performs computations completely independently of the others.

In comparison to CPUs, the Graphics Processing Units (GPUs) are based on far more, sometimes thousands of cores. Cores are very simple and lightweight, e. g. they do not support branch prediction, and are therefore much cheaper to produce. They are grouped into *Multiprocessors* that provide a small L1 cache and a very fast programmer managed memory segment called *shared memory* that is available to all the cores of a Multiprocessor and private to others. Multiprocessors are largely independent from each other and provide their own control units, registers and execution pipelines. On every Multiprocessor, threads are scheduled for execution in groups that are called *warps*. Every thread of a warp executes the same instruction at the same time on different data (*Single Instruction Multiple Data (SIMD)*). The amount of memory that is provided on the chip, i. e. L2, L1 caches and shared memory, is much smaller than the amount of on-chip memory in CPUs. Every GPU also contains a slow off-chip Random Access Memory, that is accessible from GPU and CPU and of several gigabytes in size. Hence, to exploit the architecture of a GPU for high performance, it is best for problems to be massively parallel over large amounts of data and it is often beneficial to reduce repeated RAM accesses as much as possible by using shared memory cleverly.

1.2.2 *The OpenCL Framework*

Because GPUs were developed for real-time 3D rendering, for a long time, non-graphics applications on GPUs still needed to be expressed using the terms of graphics pipelines in a GPU [31]. Since the advent of general purpose programming models for GPUs such as OpenCL and CUDA programming non-graphics applications for GPUs has be-

come more popular, although many problems, namely performance portability and the difficulty of writing high performance programs, remain.

The OpenCL framework [20] developed by the Khronos Group is an open standard for parallel programming of a variety of processors like multi-core CPUs or desktop and mobile GPU. OpenCL is based on multiple models: platform model, execution model, memory model and programming model. The following paragraphs are dedicated to describing each of them.

PLATFORM MODEL The platform model describes the relationship between a *host* and one or more connected *devices*. An OpenCL application running on the host executes the so-called *host program* that submits commands containing programs called *kernels* to the devices. A device is divided into one or more *compute units (CUs)* which in turn are divided into one or more *processing elements (PEs)* in which the computations eventually occur. For the purpose of this thesis, we assume that the host code is always executed on a CPU and kernels are executed on a single GPU device. Our focus will be on generating high-performance kernel code. Note also how the descriptions of CUs and PEs map perfectly to Multiprocessors and GPU cores described in the previous section.

EXECUTION MODEL The execution model describes how kernels execute. On a device, multiple instances of a kernel are executed at the same time. A thread executing an instance of a kernel is called a *work-item (WI)*. Work-items are organized in *work-groups (WGs)*. Every work-item and work-group is assigned an index number that uniquely identifies it. A kernel program has access to these indices and can therefore compute different memory accesses based on them. When the host program submits a kernel for execution, it provides a number that represents the amount of work-items to execute. Similarly, it provides a number that represents how many of these work-items are to be grouped in a single work-group. The indices can be organized in up to three dimensions (e. g. the host program could provide an index range of 16 in the first and 8 in the second dimension, starting 128 WIs that are uniquely identifiable by their indices in both dimensions). Note that the correct execution of some kernels can depend on a fixed amount of WIs or WGs. For kernels that do not depend on a fixed amount of WIs finding a value that maps well to the underlying hardware in terms of performance can be very challenging. Auto-tuning frameworks have been developed to tackle this problem [33].

MEMORY MODEL The memory model differentiates between the *host memory* and the *device memory*. A host program has direct access

to the host memory and can issue commands to transfer data from the host memory to the device and the other way around. A kernel program has direct access to the device memory, but there is no way for a work-item executing a kernel to access the host memory. The device memory is further divided into four distinct *memory regions*. The *global memory* region can be accessed by all the work-items and allows read and write accesses at an arbitrary position inside an allocated memory space. The *constant memory* region is a read-only global memory. Usually, the amount of constant memory supported is significantly smaller than that of global memory. A *local memory* region is local to a work-group. This means that work-items of one work-group cannot access the local memory of another, but WIs inside a work-group can exchange data via local memory. A *private memory* region is private to a single work-item and invisible to all other WIs. Data transfers between host memory and device memory can only occur using the global and constant memory regions. Additionally, it is possible to synchronize work-items inside a single work-group using *global or local memory barriers*. This means that at a given synchronization point, local or global memory are consistent across the entire work-group but not necessarily between different work-groups. The only synchronization point to achieve global memory consistency across all work-groups occurs when all the WIs executing a kernel terminate. The memory model has a lot of similarities to the architecture of a GPU. Global memory represents the GPU's RAM, local memory corresponds to the fast shared memory and private memory represents the registers.

PROGRAMMING MODEL The programming model supports a *data parallel* programming model as well as a *task parallel* programming model. Under the task parallel model a kernel is executed by a single work-item only and parallelism is achieved by issuing multiple tasks and using vector data types. The execution model as described above is primarily designed to support the data parallel model under which multiple instances of the same program are executed on different data (SPMD - Single Program Multiple Data). For the purposes of this thesis we will focus on the data parallel model.

1.2.3 *Fast Fourier Transforms*

Fast Fourier Transforms (FFTs) are algorithms that compute the matrix-vector product that results in the *Discrete Fourier Transform (DFT)* of a vector with less arithmetic operations than with a natural implementation. Equation (1.1) shows the Discrete Fourier Transform:

$$V_k = \sum_{j=0}^{n-1} v_j \cdot \omega_n^{jk} \tag{1.1}$$

with $\omega_n = e^{-2\pi i/n}$, where ω_n is complex with i denoting the imaginary number. The vector v holds samples of an input signal and v_j represents the element at position $0 \leq j < n$. The vector V represents the result of the DFT and V_k represents the element at position $0 \leq k < n$.

The original idea for an FFT was provided by Cooley and Tukey [5] with single processor machines in mind. It exploits regularities in the ω_n values that make it possible to compute smaller DFTs over parts of the input vector and reuse some of the results. Hence, instead of computing a matrix-vector product which would have a computational complexity of $\mathcal{O}(n^2)$ the Cooley-Tukey method is able to compute the DFT in $\mathcal{O}(n\ log(n))$ operations.

Since this first version, many variations have been developed and examined in terms of parallel computations. Distributed-memory FFTs have been developed in [22, 23, 42]. In [2], approaches for vector computers are described, based on a modification of the Stockham algorithm [14]. Based on the same idea the authors of [24] show how vector computers can be used to compute multiple transforms in parallel. Other methods for shared-memory systems have been developed as well [14].

By introducing the matrix

$$DFT_n = \begin{pmatrix} \omega_n^{0\cdot 0} & \cdots & \omega_n^{0\cdot(n-1)} \\ & \ddots & \\ \vdots & \omega_n^{jk} & \vdots \\ & & \ddots \\ \omega_n^{(n-1)\cdot 0} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix}, \tag{1.2}$$

the DFT can be expressed in matrix-vector notation:

$$V = DFT_n v \tag{1.3}$$

We use this matrix vector notation to give a notion of the idea behind FFTs.

1.2.3.1 *FFT Idea*

The main idea on which FFTs are based is that the input vector v of size n can be sorted and split into parts. Over these parts smaller DFTs are computed. Then, in another step, an element from every smaller DFT result is combined with an element from all the other parts, e. g. the first element from the first part with all the first elements of the other parts. To provide an intuition for this idea, we give the following example based on DFT_4.

EXAMPLE 1.1. *Set $n = 4$, then*

$$DFT_4 = \begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ \omega_4^0 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}$$

By decomposing the matrix into matrix multiplications, we are able to see the following pattern when multiplying DFT_4 to v:

$$\begin{aligned} DFT_4 v &= \left(\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array}\right) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_4 \end{pmatrix} \\ &= \left(\begin{array}{cc|cc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -i \\ \hline 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & i \end{array}\right) \left(\begin{array}{cc|cc} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{array}\right) \Pi_{2,4}^T \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_4 \end{pmatrix} \\ &= \left(\begin{array}{cc|c} 1 & 0 & \Omega_2 \\ 0 & 1 & \\ \hline 1 & 0 & -\Omega_2 \\ 0 & 1 & \end{array}\right) \left(\begin{array}{c|cc} DFT_2 & 0 & 0 \\ & 0 & 0 \\ \hline 0 \;\; 0 & \multicolumn{2}{c}{\multirow{2}{*}{DFT_2}} \\ 0 \;\; 0 & & \end{array}\right) \left(\begin{array}{c} v_0 \\ v_2 \\ \hline v_1 \\ v_4 \end{array}\right) \end{aligned}$$

The permutation matrix $\Pi_{2,4}^T$ reorders the elements in the vector v such that all the elements with even indices are in the first half and the elements with the odd indices are in the second half of the array. The 2 in the index indicates that the vector is sorted and split into 2 parts (odd and even). The 4 in the index indicates the size of the vector or DFT. Then the smaller DFT that has the size of the parts, i. e. $4/2 = 2$, is computed over both of them. In the end, values from the smaller DFTs are combined to form the result of applying DFT_4 by multiplying a so-called *Butterfly* matrix (the left matrix in Example 1.1). We always combine a single value from every smaller DFT and multiply a factor stored in the Ω_2 matrices to elements from the second part. We do this $n = 4$ times.

For this example, n has been set to 4 for the sake of brevity. Unfortunately, this size is too small for us to immediately see the benefits in terms of the amount of arithmetic operations, when performing this split. We therefore provide another example in which we roughly assess the amount of arithmetic operations that we have to perform to compute the DFT over the entire input vector.

EXAMPLE 1.2. *Set $n = 8$ and split the vector v into 2 parts based on the even and odd indices of its elements. Assume that a multiplication and addition can be computed in a single operation. We now multiply DFT_4 to both*

```
1  complex[] fft(complex[] xs) {
2    int n = xs.length
3
4    if (n = 1) return xs[0]
5    else
6      int m = n/2
7
8      complex[] firstHalf = fft(reorder(xs).firstHalf)
9      complex[] secondHalf = fft(reorder(xs).secondHalf)
10
11     complex omega = exp(e, -2*PI*i/n)
12     complex[] twiddles =
13       [exp(omega, 0), exp(omega, 1), ..., exp(omega, m-1)]
14
15     complex[] ys
16     for(int j=0; j < m; j++)
17       ys[j] = firstHalf[j] + twiddle[j] * secondHalf[j]
18     for(int j=0; j < m; j++)
19       ys[m+j] = firstHalf[j] + -1*twiddle[j] * secondHalf[j]
20
21     return ys
22 }
```

Listing 1.1: Pseudo-code recursive implementation of the FFT idea based on a radix-2 splitting for an input vector of size n.

parts. This results in 32 *operations from multiplying the* 4×4 *matrix twice. Then, combine the values from every part. This results in* 16 *operations from combining values from* 2 *parts* $n = 8$ *times.*

Therefore, computing the DFT of size 8*, based on this split amounts to* 48 *operations. In comparison, computing a DFT of size* 8 *by simple matrix-vector multiplication would result in* 64 *operations.*

One can now imagine that the same method can be used for bigger inputs and that it can be applied recursively to the resulting smaller DFTs leading to further savings in arithmetic operations. Listing 1.1 shows an implementation of the FFT idea that recursively splits smaller DFTs in pseudo-code. The call to the function `exp` with `exp(e,-2*PI*i/n)` computes the complex value $e^{-2\pi i/n}$. Therefore, the array `omegas` holds the values that correspond to the elements in Ω_2 from Example 1.1. The term *twiddle factor* is used throughout literature to describe the values that need to be multiplied before results from different DFTs can be combined to form a larger one.

Both examples sort and split the vector into two parts. But not every vector has an even size and thus it is not always possible to split a vector in such a way. This is no problem however, because the idea works for any split into $p > 1$ parts where $n = pm$ for some positive integer m. Then, the smaller DFTs that are computed are of size m. We can split the computation of the smaller DFTs of size m recursively

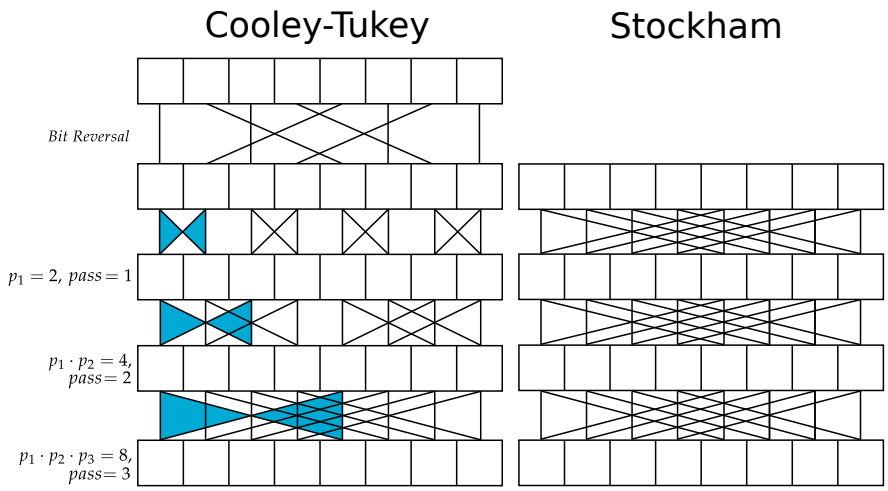

Figure 1.2: Example of the dataflow in a Cooley-Tukey and Stockham FFT over an input array of size 8. Highlighted are some of the so-called Butterflies.

into $p' > 1$ parts where $m = p'm'$ for some positive integer m'. The values p or p' are called *radices*. By providing a complete factorization $\rho = \langle p_1, \ldots, p_t \rangle$ of n with $n = p_t \cdot \ldots \cdot p_1$, we receive a way to specify the splits in every recursion step.

In this thesis, we will focus on two variants of this idea, the Cooley-Tukey and the Stockham method. Then, we go on to use these methods to express the Four-Step. We will show the differences between Cooley-Tukey and Stockham and how they use the FFT idea in the following.

1.2.3.2 *Cooley-Tukey and Stockham*

A key step in the Cooley-Tukey method is an initial permutation step called *Bit-Reversal*. In this step, the permutation that we saw in Example 1.1 is combined with all the permutations that occur when recursively splitting smaller DFTs. The permutation reorders the input values, such that smaller DFTs that are applied to parts of the input vector access consecutive memory segments. The term Bit-Reversal originates from the way that the elements are reordered for a pure radix-2 splitting. The new position of the index results from reversing the bits in the index's binary form (e. g. $2 = 10_b$ is reordered to position $1 = 01_b$). Then, the results of the smaller DFTs are combined in a *pass*, which performs so-called *Butterfly* operations. A Butterfly combines values by computing small DFTs, after multiplying with different twiddle factors. The operation gets its name from the dataflow pattern that occurs in passes of a radix-2 splits. Two elements are combined twice and the results are stored at the positions of the original two values. Figure 1.2 depicts the dataflow of the iterative Cooley-Tukey and Stockham methods for an input array of size 8, with all radices set to 2. After the Bit-Reversal step, the first pass which cor-

responds to the operations in the last level of the recursion tree in a recursive FFT, is applied. The resulting Butterfly in the dataflow pattern is highlighted. The following passes represent the operations of levels going up the recursion tree. Note that each pass q has a radix p_q and combines elements such that DFTs of size $L_q = p_1 \cdot \ldots \cdot p_q$ are computed.

The Stockham method does not rely on an initial Bit-Reversal step and accesses elements at their original position in the array. Simply storing the results in natural order for every pass leads to the computation of a full DFT in the last pass. An advantage of the Stockham algorithm is that the often expensive incoherent memory accesses of the Bit-Reversal step can be avoided. Furthermore, the access pattern of Stockham passes promises better results for GPUs. Therefore, for the purposes of this thesis, we will put our initial focus on Stockham passes.

1.2.4 *Lift*

Lift aims at providing high-level programming with performance portability [41]. Instead of expressing programs in an imperative language based on loops that require index calculations and explicit synchronizations, the user provides a high-level functional expression that encapsulates the algorithmic semantics within a combination of high-level algorithmic building blocks. In this regard, Lift is similar to algorithmic skeleton libraries such as SkelCL [39], MUESLI [8] or Skandium [25]. The high-level functional expression is automatically rewritten and optimized by the Lift system and eventually compiled into a high-performance OpenCL kernel for a target device.

The Lift compilation process is depicted in Figure 1.3. It consists of three major steps with the first step expecting expressions of high-level building blocks that are either so-called high-level (algorithmic) primitives which operate over multi-dimensional arrays (e. g. the well known *map* or *reduce*) or compositions of said primitives. These building blocks are used to create high-level expressions that can be rewritten into other semantically equivalent high-level expressions or into low-level expressions. Low-level expressions consist of so-called functional low-level data parallel primitives that are closely related to the OpenCL framework. They preserve the semantics of high-level expressions, but add further information regarding for example the distribution of parallel computations over work-items or the memory regions to store intermediate results in. By rewriting these low-level expressions, Lift is able to generate a search space of semantically equivalent expressions that represent common optimizations. From these low-level expressions, OpenCL code is generated after possibly applying standard compiler techniques such as loop unrolling. The generated OpenCL code is executed on the target platform and its per-

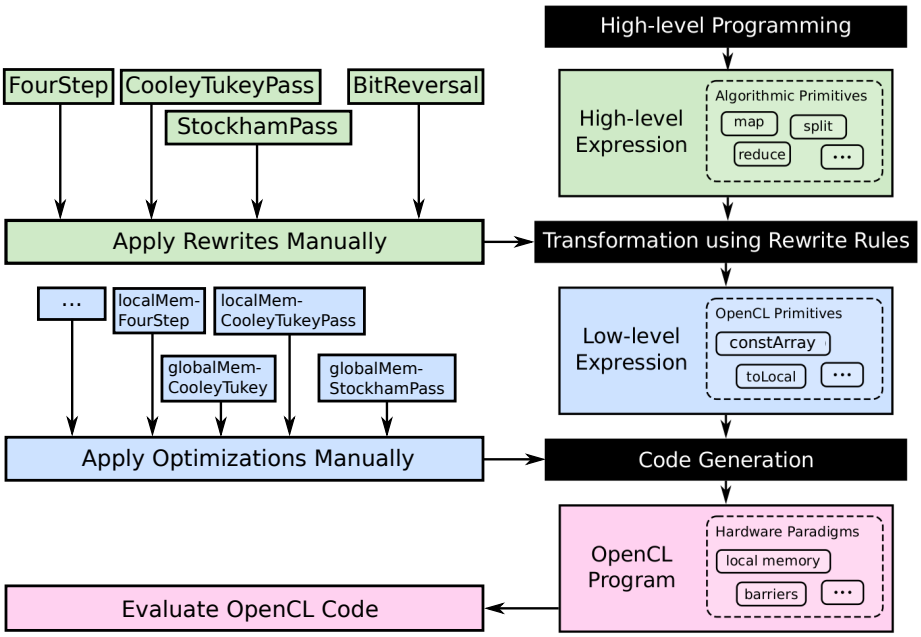

Figure 1.3: The compilation flow of the Lift system in relation to the work of this thesis (based on [16]).

formance is compared against OpenCL code generated from other expressions to choose the best performing one as a result. This approach has proven to be successful with sparse linear algebra [19], General Matrix Multiplication (GEMM) [40] and stencil computations [18].

In this thesis, we will implement an expressions for Bit-Reversal, Cooley-Tukey and Stockham passes and the Four-Step method. We will then provide formal rewrite rules to *lower* these expressions and eventually generate OpenCL kernel code that we can evaluate empirically.

1.3 RELATED WORK

Fast Fourier Transforms have long been a subject of study to the computer science community and much work has been done toward the variety of algorithms that compute FFTs and toward their optimization. We will summarize existing research and compare it to the work done in this thesis.

DYNAMIC GENERATION OF ALGORITHMS FFTW [12], [13] is well known for generating high-performance FFT implementations on different architectures. It creates a search space of different FFT algorithms where performance critical parts are generated by a special-purpose compiler [11] and attempts to find the fastest algorithm at runtime. Although modern versions of FFTW support the Message Passing Interface (MPI) and OpenMP for distributed and parallel DFT computations, it is not possible to generate programs that are exe-

cutable on GPUs. Spiral [10] is a system that automatically generates high-performance programs for linear transforms, which includes Fast Fourier Transforms. Implementations generating programs for GPUs have been developed [9]. While Spiral probably comes closest to our approach, there are some significant differences. First of all, Spiral is solely focused on linear transforms and uses a DSL (Domain Specific Language), for this purpose. Lift on the other hand, is a general purpose high-level functional language that can be used to express far more than linear transforms. While Spiral decomposes the DFT into basic building blocks, like we do, these blocks are rather abstract (e. g. matrix-vector multiplication) and still linear transform specific. We go a step further and break these operations down into more basic building blocks and therefore retain more flexibility and possibilities in the rewrite and code generation process. Furthermore, a generated algorithm expressed with Spiral's building blocks is compiled into low-level imperative code that can be compiled for the target platform (e. g. C code with OpenMP directives), without using an IL (Intermediate Language) that retains the semantics of our functional primitives for use in the code generation process. Our approach therefore allows us to use the IL to search for platform specific optimizations with the additional information provided by our low-level functional primitives.

GENERAL PURPOSE GPU CODE GENERATION More general high-level languages such as Halide [32] and others [29, 43] provide parallel patterns to abstract and simplify the programming of GPUs. These approaches can be used to implement FFTs as well, but similar to Spiral, they do not rely on a functional data parallel IL like Lift.

HIGH PERFORMANCE FFT LIBRARIES There exist several libraries that provide easy to use interfaces for FFTs on GPUs. Nvidia's cuFFT library [6] offers professionally optimized FFT implementations based on CUDA and enables the user to compute FFTs on Nvidia GPUs. The clFFT library [21], has been implemented and optimized by AMD for their GPUs and enables the user to compute FFTs using OpenCL. Because clFFT is based on OpenCL, it can be executed on many different platforms including Nvidia GPUs and CPUs. Both libraries are well and actively maintained and currently provide some of the best performing options for computing FFTs on GPUs. Nonetheless, the performance of both of these libraries is platform specific and hardware changes make new optimizations by hand necessary.

1.4 CONTRIBUTIONS

EXPRESSING FFTS WITH HIGH-LEVEL DATA PARALLEL PRIMITIVES We use the Lift system to express passes of Fast Fourier Transforms

based on high-level data parallel primitives by decomposing the Cooley-Tukey, Stockham and Four-Step methods into their fundamental data parallel building blocks.

GENERATION OF HIGH-PERFORMANCE FFT CODE FOR GPUS We extend the Lift system with a new primitive called *carray*, inspired by an optimization done in the clFFT library. With this primitive we are able to generate high-performance FFT kernels that are competitive with clFFT.

PERFORMANCE COMPARISON OF DIFFERENT FFT METHODS By generating OpenCL kernel code for the different FFT methods, we are able to compare their performance on different GPU devices against each other and learn how to further improve our implementations in the future.

2

FORMAL DEFINITION OF FFTS IN MATRIX-VECTOR NOTATION

In this chapter, first, we will introduce a matrix-vector notation to describe different passes of a Fast Fourier Transform. Then, we continue by formally defining functional expressions for different FFT passes based on this notation. The matrices used have a lot of zero entries, i. e. they are sparse. Sparse matrix multiplications can be implemented by skipping the multiplication with their zero elements, resulting in less arithmetic expressions. Van Loan [44] showed that indeed, implementing the following matrix-vector products this way leads to Fast Fourier Transforms. A benefit of the matrix-vector notation is, that we can use it to derive decompositions of the functional expressions into basic high-level building blocks later on.

2.1 EXPRESSING FFTS BASED ON MATRIX-VECTOR NOTATION

In order to express FFTs in matrix-vector notation, we need to introduce the *Kronecker product* operator. We will then show how the Permutation step, Butterflies and the entire Combine step can be defined. Afterwards, based on these definitions, we will provide formal definitions of expressions that represent the *Cooley-Tukey* and *Stockham* FFTs.

KRONECKER PRODUCT The Kronecker product as defined by [44] is used to express matrices with a highly regular structure, where the matrix is composed of multiple versions of a smaller matrix that only differ in a factor applied to all the elements of this smaller matrix. This structure arises in many matrix decompositions of the DFT matrix.

DEFINITION 2.1. *Let $A \in \mathbb{C}^{r \times s}$ and $B \in \mathbb{C}^{m \times n}$ be complex matrices, where r, s, m and n are the respective sizes of the matrices. Then, the Kronecker product over matrices $\otimes$ is formally defined by*

$$A \otimes B = \begin{pmatrix} a_{0,0}B & \cdots & a_{0,s-1}B \\ \vdots & & \vdots \\ a_{r-1,0}B & \cdots & a_{r-1,s-1}B \end{pmatrix} \in \mathbb{C}^{rm \times sn},$$

where $a_{j,k}$ for $0 \leq j < r$, $0 \leq k < s$ is an element of A in row j and column k.

The B entries inside the resulting matrix are called *block matrices*. We talk about a block matrix when refering to values of a matrix

that are contained in another matrix, i. e. the matrix resulting from a Kronecker product contains $r \times s$ block matrices. Example 2.1 shows how the Kronecker product and block matrix notation can be used to express the multiplication of the same matrix with consecutive parts of a vector.

EXAMPLE 2.1. *Let I_2 be the identity matrix of size 2 and $A \in \mathbb{C}^{2\times 2}$. Let v be a vector of size 4 with elements v_j, where $0 \leq j < 4$. The multiplication of A with the first and second half of v can be expressed using the Kronecker product:*

$$(I_2 \otimes A)v = \begin{pmatrix} 1 \cdot A & 0 \\ 0 & 1 \cdot A \end{pmatrix} v = \left(A \begin{pmatrix} v_0 \\ v_1 \end{pmatrix}, A \begin{pmatrix} v_2 \\ v_3 \end{pmatrix} \right)^T$$

where the result is a vector that is expressed in block matrix notation.

RADICES As mentioned in Chapter 1 Section 1.2.3, the way a DFT is decomposed into smaller DFTs to form an FFT is decided by the so-called radices. A radix is a number that determines the amount of smaller DFTs that are going to be combined in one recursion step. Also, a radix has another property. It describes the size of the Butterflies, which themselves consist of a DFT computation, and a multiplication with twiddle factors. E. g. a radix-2 split assumes a smaller DFT has been computed for 2 parts of the array, and 2 elements from both parts are combined by multiplying twiddle factors and computing a DFT of size 2, which is equal to the radix. We refer to the Butterfly matrix of Example 1.1 to depict how Butterflies consist of twiddle factors and DFTs.

EXAMPLE 2.2. *Decompose the Butterfly matrix of Example 1.1 into matrix multiplications:*

$$\left(\begin{array}{cc|c} 1 & 0 & \multirow{2}{*}{Ω_2} \\ 0 & 1 & \\ \hline 1 & 0 & \multirow{2}{*}{$-\Omega_2$} \\ 0 & 1 & \end{array}\right) = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -i \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & i \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -i \end{pmatrix}$$

$$= (DFT_2 \otimes I_2)\ diag(I_2, \Omega_{2,2})$$

We see that, when multiplying the Butterfly matrix to a vector from the left, we can first multiply the so-called *twiddle factors* in $diag(I_2, \Omega_{2,2})$ and then compute 2 DFTs of size 2 to combine the previous results into a larger DFT.

We already mentioned that the radices can be provided as a collection that is a factorization of the size of the input vector:

$$\rho = \langle p_1, \ldots, p_t \rangle \tag{2.1}$$

with $n = p_t \cdot \ldots \cdot p_1$ with $t > 1$. This means that for a particular ρ, an FFT can be computed in t passes. The first pass represents the lowest level of the recursion tree in a recursive FFT and uses radix p_1. The last pass with radix p_t represents the root of the recursion tree in a recursive FFT and the initial split.

Furthermore, this means that in every pass q with $1 \leq q \leq t$, DFTs of size

$$L_q = p_1 \cdot \ldots \cdot p_q \tag{2.2}$$

are created from the results of the previous pass.

PERMUTATION MATRICES The bit-reversal step as well as the permutations at the beginning of every Stockham pass can be viewed as a combination or single application of permutation matrices. These matrices have exactly one 1 per row and column. The rest of the elements in the same row and column are 0. Multiplying such a matrix with a vector results in a vector that is either unchanged (i. e. for the identity matrix) or holds the same elements in a different order. We introduce the so-called *mod-p sort* permutation matrix $\Pi^T_{p,m}$ that is used in the Stockham method and which, in its transposed form, can be used to define the Bit-Reversal step of the Cooley-Tukey method.

DEFINITION 2.2. *Let $v \in \mathbb{C}^n$ be a complex vector of size n, with $n = pm$. The mod-p sort permutation matrix $\Pi^T_{p,n}$ is then defined as follows:*

$$\Pi^T_{p,n} v = \begin{pmatrix} v(0 : p : n-1) \\ v(1 : p : n-1) \\ \vdots \\ v(p-1 : p : n-1) \end{pmatrix},$$

with $v(j : p : n-1)$ for $0 \leq j < p$ representing the elements that are $p-1$ elements apart in v starting at the index j (e. g. $(1 : 2 : 4)$ returns the indices 1 and 3.) The elements for all j together form a vector.

Let us consider how the mod-p sort permutation matrix can be used to express the even-odd sort from Example 1.1 and how it groups the elements based on the mod p value of the indices.

EXAMPLE 2.3. *Set $n = 4$ and $p = 2$ and let v be a vector of size 4 with elements v_j where $0 \leq j < n$. Then*

$$\Pi^T_{2,4} v = \begin{pmatrix} v(0 : 2 : 3) \\ v(1 : 2 : 3) \end{pmatrix} = \begin{pmatrix} v_0 \\ v_2 \\ v_1 \\ v_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} v = \Pi^T_{2,4} v$$

BUTTERFLY MATRICES A Butterfly matrix represents the Butterfly computations used to combine elements into a bigger DFT and occurs in many different versions of FFTs. This means, that depending on the current pass and the size of DFTs from previous passes, multiple Butterflies are represented in a single Butterfly matrix. In the following definition, we can see the relationship between twiddle factors and DFTs.

DEFINITION 2.3. *Let q with $1 \leq q \leq t$ be the FFT pass, p_q be a radix, $L_q = p_1 \cdot \ldots \cdot p_q$ be the size of the DFTs that are computed in this pass and $L_{q-1} = L_q / p_q$ be the size of the DFTs computed in the previous pass. The Butterfly matrix B_{p_q,L_q} is then defined as follows:*

$$B_{p_q,L_q} = (DFT_{p_q} \otimes I_{L_{q-1}}) diag(I_{L_{q-1}}, \Omega_{p_q,L_{q-1}}, \ldots, \Omega_{p_q,L_{q-1}}^{p_q-1}),$$

where

$$\Omega_{p_q,L_{q-1}} = diag(1, \omega_{L_q}, \ldots, \omega_{L_q}^{L_{q-1}-1})$$

are diagonal matrices that in turn are arranged as block matrices on a main diagonal.

When multiplying the matrix from the left to a vector, first the twiddle factors in the $\Omega_{p_q,L_{q-1}}$ are multiplied to its values. Then, L_{q-1} DFTs are computed. This definition corresponds directly to the matrices found in Example 2.2.

COOLEY-TUKEY For the Cooley-Tukey method, we define the Bit-Reversal step in the form of a single permutation matrix. The Bit-Reversal step is a combination of all the permutations that occur when recursively splitting DFTs. These permutations can be expressed as matrices and then be combined into a single Bit-Reversal matrix by multiplication.

DEFINITION 2.4. *Let n be the size of the FFT and ρ be a collection of t radices that form a factorization of n. The Bit-Reversal matrix is then defined as follows:*

$$P_n(\rho) = R_t \cdots R_1,$$

where

$$R_q = I_{\frac{n}{L_q}} \otimes \Pi_{p_q,L_q}$$

is the permutation necessary for the qs pass.

Multiplying the matrix to a vector from the left, results in the bit-reversed vector.

A Butterfly matrix combines values to form the results of a larger DFT. Unless the last pass of an FFT is computed, i. e. the larger DFT

that is formed is of size n, multiple Butterfly matrices have to be multiplied to different parts of the input vector. This is represented in the *Combine* matrix. There are different Combine matrices for different FFT methods.

DEFINITION 2.5. *Let n be the size of the FFT, q with $1 \leq q \leq t$ be the the FFT pass, p_q be a radix, $L_q = p_1 \cdot \ldots \cdot p_q$ be the size of the DFTs that are computed in this pass and I_{n/L_q} be the identity matrix of size n/L_q. The Combine matrix for the Cooley-Tukey method in pass q, C_q is then defined as follows:*

$$C_q = I_{\frac{n}{L_q}} \otimes B_{p_q,L_q}$$

The Combine matrix C_q represents an entire pass. Now, we have defined everything to provide a complete definition of the Cooley-Tukey FFT in matrix-vector notation.

THEOREM 2.1 ([44]). *Let v be a vector of size n, ρ be a collection of t radices that form a factorization of n. It then holds that:*

$$DFT_n v = C_t \cdots C_1 P_n(\rho)^T v$$

By cleverly implementing the multiplication of the given sparse matrices skipping the multiplications of their zero entries, an algorithm for the Cooley-Tukey FFT follows. We will provide an example on how this can be done for the Stockham method.

STOCKHAM Similarly to the Cooley-Tukey method, we can give a sparse matrix decomposition of DFT_n that represents the Stockham procedure. As mentioned before, the Stockham algorithm applies a permutation at the beginning of every pass. Therefore, a matrix representing a pass is a combination of a permutation and a Combine matrix.

The permutation of a Stockham pass reorders groups of adjacent elements based on the mod-p sort permutation (Definition 2.2). Thereby, elements over which *different* Butterflies are computed lie beside each other.

DEFINITION 2.6. *Let n be the size of the FFT, q with $1 \leq q \leq t$ be the FFT pass, p_q be a radix, $L_q = p_1 \cdot \ldots \cdot p_q$ be the size of the DFTs that are computed in this pass and I_{n/L_q} be the identity matrix of size n/L_q. The permutation matrix for the Stockham method in pass q, Q_{p_q,L_q} is then defined as follows:*

$$Q_{p_q,L_q} = \Pi^T_{p_q,L_q} \otimes I_{\frac{n}{L_q}}$$

As for the Cooley-Tukey method we define a Combine matrix to express how several Butterflies are applied. Instead of being separated over multiple parts of the input vector, the Butterfly matrix multiplications are now interleaved. The Combine matrix and the permutation matrices are multiplied to form the matrix of a Stockham pass.

DEFINITION 2.7. *Let n be the size of the FFT, q with $1 \leq q \leq t$ be the FFT pass, p_q be a radix, $L_q = p_1 \cdot \ldots \cdot p_q$ be the size of the DFTs that are computed in this pass and I_{n/L_q} be the identity matrix of size n/L_q. The Combine matrix for the Stockham method in pass q, S_q is then defined as follows:*

$$S_{p_q,L_q} = B_{p_q,L_q} \otimes I_{\frac{n}{L_q}},$$

Then, the matrix for Stockham pass q is defined as follows:

$$G_q = S_{p_q,L_q} \cdot Q_{p_q,L_q}$$

Now, we have defined everything to provide a complete definition of the Stockham FFT in matrix-vector notation.

THEOREM 2.2 ([44]). *Let v be a vector of size n. It then holds that:*

$$DFT_n v = G_t \cdots G_1 v$$

An example of how the sparse matrix multiplications can be implemented for a pure radix-2 split can be found in Listing 2.1. The outermost loop steps through the passes. For every pass, the matrix-vector multiplications of the Butterfly matrices are implemented by computing a twiddle factor and using it to compute the relevant Butterflies. For a radix-2 split, the Butterfly computation includes a DFT_2 that is multiplied with some twiddled values. This is represented by the addition and subtraction in lines 14 and 16 (a DFT_2 matrix only consists of 1s and a single -1 in the lower right corner). A general radix FFT implementation could simply compute the matrix multiplication over an array holding the affected elements. Finally, we point out that the permutation that occurs at the beginning of every Stockham pass is implemented in the index calculations.

Note that this implementation traverses the array that represents the input vector. Then it computes twiddle factors (and Butterfly values for general radix implementations), based on the current indices. By implementing Stockham FFTs based on functional primitives, we do not have access to loops or indices and have to find a creative way to determine these values.

2.2 FORMAL DEFINITIONS OF FFT EXPRESSIONS

Based on the matrix-vector notations provided in the previous section, we quickly define expressions that represent FFT passes. To be able to do this, we need to introduce a formal notation of types and functional primitives first. In order to decompose these expressions into high-level functional primitives, we provide formal definitions of said primitives. Then, the primitives are applied and composed to form FFT expressions by formally deriving them from the matrix-vector notations.

```
//input is v with n elements
for (int q = 1; q <= t; q++) {
    //Stockham FFT pass
    int L = pow(2,q); int LPrev = L/2;
    int y[n];
    copy(v, y);
    for (int j = 0; j < LPrev; j++) {
        double2 twiddle = {cos(-2*PI*j/L), sin(-2*PI*j/L)};
        for (int k = 0; k < (n/L); k++) {
            //Butterfly
            double2 twiddledVal =
                twiddle * y(j*n/LPrev + k + n/L);
            v(j*n/L + k) =
                y(j*n/LPrev + k) + twiddleVal;
            v((j + LPrev)*n/L + k) =
                y(j*n/LPrev + k) - twiddleVal;
        }
    }
}
```

Listing 2.1: Pseudo-code implementation of an iterative Stockham FFT.

2.2.1 *Notation*

FUNCTION APPLICATION For readability, we denote function application with the familiar notation of a function name followed by function arguments in parenthesis, e. g. $f(x, y)$. Despite this notation, we consider functions to be curried. This means, that a function applied to multiple arguments can be considered to be applied to the arguments consecutively from left to right, one argument at a time, i. e. $f(x, y) = f(x)(y)$. Every application creates a new function with the parameter set fix to the provided value. The function application is left associative. We may use infix notation for some binary operators to improve readability, i. e. $+(x, y) = x + y$. Sequential function composition uses the $\circ$ operator, e. g. $(f \circ g)(x) = f(g(x))$.

ARRAYS AND TUPLES Many of our functional primitives and expressions will operate on arrays. An array is a representation of a sequential collection of elements. For an array xs of size n with elements x_i, we write $[x_1, x_2, \ldots, x_n]$. Sometimes we want to stress that elements of an array have to be arrays themselves, i. e. the array is multi-dimensional, we then talk about *arrays of arrays* and simply write:

$$[[x_{1,1}, \ldots, x_{1,n}], \ldots, [x_{m,1}, \ldots, x_{m,n}]]$$

or

$$\begin{bmatrix} [x_{1,1}, \ldots, x_{1,n}], \\ \vdots \\ [x_{m,1}, \ldots, x_{m,n}] \end{bmatrix}$$

to highlight the similarities to a matrix. To be precise, we will differentiate between vectors or matrices and arrays. If not specified otherwise, when we talk about the *array representation* of a vector v, we mean the array holds elements equal to the elements in v and in the same order. For the array representation of a matrix we assume the same, only for a two-dimensional array. To express an array access we provide the indices of the accessed element in parenthesis, e. g. for accessing an array at index i we write $[x_1, \ldots, x_n](i) = x_i$.

For a tuple t with the first element t_1 and the second element t_2, we write $\{t_1, t_2\}$. Provided a tuple we write $t.1$ or $t.2$ to denote the first or second element respectively.

TYPING One of our goals in this thesis is to nest and compose high-level primitives to express FFT computations. Therefore, we introduce *types* that allow us to define how primitives can be combined. For an expression e of type T we write $e : T$. The type of a function f that takes n arguments $x_1, x_2, \ldots, x_n$ of types $T_1, T_2, \ldots, T_n$ and maps these arguments to a type U, is written as $f : (x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n) \to U$. Tuple types are denoted as $\{T, U\}$. Array types include information about the amount of elements in an array, i. e. their size. For an array with elements of type T and size n, we write $[T]_n$. Note that a type T can be an array type thus allowing arrays of multiple dimensions. Additionally, we will make use of an integer type Int and floating point types F, i. e. *Double* or *Float*.

2.2.2 *FFT Expressions*

Fast Fourier Transforms are based on complex numbers. Hence, we need a means to express a complex type. A complex number consists of a real and imaginary part. We can therefore define a complex type as follows:

DEFINITION 2.8. *Let F be a floating point type. A complex type $Complex$ is then defined as follows:*

$$Complex = \{F, F\}$$

where the first element of the tuple is the type of the real and the second is the type of the imaginary part.

Furthermore, to be able to perform basic operations on $Complex$ types, we define complex addition, multiplication and the complex identity.

DEFINITION 2.9. *Let x and y be of the type Complex. We define $id_{\mathbb{C}}$, $+_{\mathbb{C}}$ and $*_{\mathbb{C}}$ as follows:*

- $id_{\mathbb{C}}(x) = x$
- $+_{\mathbb{C}}(x, y) = \{x.1 + y.1, x.2 + y.2\}$
- $*_{\mathbb{C}}(x, y) = \{x.1 \cdot y.1 - x.2 \cdot y.2,\ \ x.2 \cdot y.1 + x.1 \cdot y.2\}$

The functions have the following types:

- $id_{\mathbb{C}} : Complex \to Complex$
- $+_{\mathbb{C}} : \{Complex, Complex\} \to Complex$
- $*_{\mathbb{C}} : \{Complex, Complex\} \to Complex$

For readability, we will omit the index of the functions unless it is not clear from the context, e. g. $+$ instead of $+_{\mathbb{C}}$.

We will make use of the complex values $e^{-2\pi il/n}$ for some integers l and n. To compute the complex number e^{ix} for some real x, we note that Euler's formula states

$$e^{ix} = cos(x) + isin(x) \tag{2.3}$$

and define:

DEFINITION 2.10. *Let l and n be positive integers. We define omega as follows:*

$$omega(l, n) = \{cos(-2\pi l/n),\ sin(-2\pi l/n)\}$$

which represents $e^{-2\pi il/n}$.

Finally, we are able to provide a formal definition of expressions that represent FFT passes. We begin with the *stockhamPass* that computes a pass of a Stockham FFT over an array.

DEFINITION 2.11. *Let vs be an array representation of the vector v with size n and elements v_j where $0 \leq j < n$, p a positive integer with $n = mp$ and L be a positive integer with $L = rp$. The stockhamPass expression is then defined as follows:*

$$stockhamPass(L, p, vs) = \\ [(S_{p,L}Q_{p,L}v)_0, \ldots, (S_{p,L}Q_{p,L}v)_j, \ldots, (S_{p,L}Q_{p,L}v)_{n-1}]$$

where $(\cdot)_j$ is the vector element at index j. The type of stockhamPass is as follows:

$$stockhamPass : (L : Int,\ p : Int,\ vs : [Complex]_n) \to [Complex]_n$$

The expressions for a Cooley-Tukey pass are defined similarly:

DEFINITION 2.12. *Let vs be an array representation of the vector v with size n and elements v_j where $0 \leq j < n$, p a positive integer with $n = mp$ and L be a positive integer with $L = rp$. The cooleytukeyPass expression is then defined as follows:*

$$cooleytukeyPass(L, p, vs) = \\ [(I_{n/L} \otimes B_{p,L} v)_0, \ldots, (I_{n/L} \otimes B_{p,L} v)_j, \ldots, (I_{n/L} \otimes B_{p,L} v)_{n-1}]$$

where $(\cdot)_j$ is the vector element at index j. The type of cooleytukeyPass is as follows:

$$cooleytukeyPass : (L : Int,\ p : Int,\ vs : [Complex]_n) \to [Complex]_n$$

Also, the Bit-Reversal step of a Cooley-Tukey FFT is defined:

DEFINITION 2.13. *Let vs be an array representation of the vector v with size n and elements v_j where $0 \leq j < n$. Let $\rho = \langle p_1, \ldots, p_t \rangle$ be a collection of radices that forms a factorization of n, with $n = p_1 \cdot \ldots \cdot p_t$. The expression bitrev that represents a Bit-Reversal pass is defined as follows:*

$$bitrev(\rho, vs) = [(P_n(\rho) v)_0, \ldots, (P_n(\rho) v)_j, \ldots, (P_n(\rho) v)_{n-1}]$$

where $(\cdot)_j$ is the vector element at index j. The type of bitrev is as follows:

$$bitrev : (\rho : [Int]_t,\ vs : [Complex]_n) \to [Complex]_n$$

Hence, an entire FFT can be expressed by composing the passes in accordance to a provided radix split ρ.

DEFINITION 2.14. *Let vs be an array representation of the vector v with size n and elements v_j where $0 \leq j < n$. Let $\rho = \langle p_1, \ldots, p_t \rangle$ be a collection of radices that forms a factorization of n, with $n = p_1 \cdot \ldots \cdot p_t$. A complete fft expression is then defined as follows:*

$$fft_{st}(\rho, vs) = \\ stockhamPass(p_1 \cdot \ldots \cdot p_t, p_t) \circ \ldots \circ \\ stockhamPass(p_1 \cdot p_2, p_2) \circ stockhamPass(p_1, p_1)\ (vs)$$

for a Stockham FFT or

$$fft_{ct}(\rho, vs) = \\ cooleytukeyPass(p_1 \cdot \ldots \cdot p_t, p_t) \circ \ldots \circ \\ cooleytukeyPass(p_1 \cdot p_2, p_2) \circ cooleytukeyPass(p_1, p_1) \circ \\ bitrev(\rho)\ (vs)$$

for a Cooley-Tukey FFT.

3

FFTS IN HIGH-LEVEL FUNCTIONAL PRIMITIVES

In this chapter, we decompose the formally defined expressions of FFT passes into algorithmic building blocks in the form of higher order functions. We will derive building blocks for FFTs by composing so-called *high-level functional primitives* (or in short *primitives*) by analyzing the structures of matrices from the previous chapter and expressing transformations on these matrices and the input vector. We begin by introducing the high-level functional primitives. Then, we derive the high-level expressions that are our building blocks. Afterwards, we combine these building blocks to form expressions for FFT methods that are correct.

3.1 HIGH-LEVEL FUNCTIONAL PRIMITIVES

High-level functional primitives represent simple but commonly occuring algorithmic patterns and can be combined to build powerful functions that we call expressions, by means of function composition. Going back to our introduction of FFTs, we recall that the Cooley-Tukey and Stockham method can be decomposed into two major steps:

- Combining smaller DFTs into larger ones by applying Butterfly computations to groups of elements.
- Applying permutations to the vector that is transformed.

We have already shown how the combine step can be represented in matrix notation by multiplying a sparse matrix with a vector. Hence, we want some means to express matrix-vector multiplications. Applying the same computation such as Butterfly multiple times is a well known pattern in the functional world and often solved by *map*. Also, *reduce* is often used to implement matrix-vector multiplications. In order to represent an array that can be interpreted as a matrix for example, we will introduce the *array* primitive. Other primitives such as *transpose*, *split* and *join* that let us change the order and dimension of an array can be leveraged to express the permutations needed.

This section is dedicated to introducing and formally defining the primitives that are needed to express Cooley-Tukey, Stockham and Four-Step FFT passes. All the provided primitives do already exist in Lift and we draw the definitions and the notation given in this section from [18], [16] and [37]. This means that we are able to express FFTs without introducing new primitives to Lift which does not only show the power of the existing system, but also has the added benefit of

keeping the search space for rewrite rules as small as possible. Furthermore, we are able to use general purpose primitives only, that are are completely independent from FFTs and can be used to express a host of other applications (e. g. in combination with some additional primitives, stencil computations). Therefore, rewrite rules that are introduced in the next chapter might be applied to expressions in other application domains as well.

MAP *map* is long-known and wide spread in languages that support functional programming (e. g. Lisp, Scala or Haskell) and even imperative or object-oriented languages such as C++ provide map-like functions (i. e. *transform* in C++), because of its power and simplicity. The *map* primitive returns an array that represents a function unary f being applied to each element in an input array.

DEFINITION 3.1. *Let xs be an array of size n with elements x_i where $0 < i \leq n$. Let f be a unary customizing function defined on elements. The high-level primitive map is then defined as follows:*

$$map(f,\ [x_1, x_2, \ldots, x_n]) \stackrel{def}{=} [f(x_1), f(x_2), \ldots, f(x_n)]$$

The type of map is defined as follows:

$$map : (f : T \to U,\ xs : [T]_n) \to [U]_n$$

REDUCE *reduce* (or in languages like Scala called *fold*) is another well-known element in functional programming. Traditionally, it computes a single element from a given array based on a provided binary operator that defines how to combine successive elements. Instead of returning a single element the *reduce* primitive returns an array containing a single element:

DEFINITION 3.2. *Let xs be an array of size n with elements x_i where $0 < i \leq n$. Let $\oplus$ be an associative and commutative binary customizing operator and init be an initial value. The high-level primitive reduce is then defined as follows:*

$$reduce(init,\ \oplus,\ [x_1, x_2, \ldots, x_n] \stackrel{def}{=} [init \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n]$$

The type of reduce is defined as follows:

$$reduce : (init : U,\ \oplus : (U, T) \to U,\ xs : [U]_n) \to [U]_1$$

Note that requiring the operator to be associative and commutative enables efficient parallel implementations [37]. However, this property is not used in this thesis. We keep it in our definition because it does not impair us in any way (i. e. our usage of reduce will solely be with the operator $+$ which meets above criteria) and might be useful in future considerations of FFT expressions.

ZIP *zip* combines the elements of independent arrays of the same length into a single array of tuples. A tuple contains as many elements as arrays have been combined and the positions of elements in the resulting arrays is the same to their input arrays:

DEFINITION 3.3. *Let xs and ys be arrays of length n with elements x_i and y_i where $0 < i \leq n$. The high-level primitive zip is then defined as follows:*

$$zip([x_1, x_2, \ldots, x_n],\ [y_1, y_2, \ldots, x_n]) \stackrel{def}{=} \\ [\{x_1, y_1\}, \{x_2, y_2\}, \ldots, \{x_n, y_n\}]$$

The type of zip is defined as follows:

$$zip : (xs : [T]_n,\ ys : [U]_n) \to [\{T, U\}]_n$$

SPLIT AND JOIN *split* or *join* divide an array into regularly sized parts of smaller arrays or create a coherent array from smaller ones, thereby increasing or decreasing the dimensionality of the outermost array. The two primitives are often used in conjunction, e. g. to apply a function over parts of an array using map and then joining it back together. We start by defining the *split* primitive:

DEFINITION 3.4. *Let xs be an array of size n with elements x_i where $0 < i \leq n$. Let m be an integer value that divides n, meaning $n \mod m = 0$. The split primitive is then defined as follows:*

$$split(m,\ [x_1, x_2, \ldots, x_n]) \stackrel{def}{=} \\ [[x_1, \ldots, x_m], [x_{m+1}, \ldots, x_{2m}], \ldots, [x_{n-(m-1)}, \ldots, x_n]]$$

The type of split is defined as follows:

$$split(m : Int,\ xs : [T]_n) \to [[T]_m]_{n/m}$$

The inverse operation to *split* is given by defining *join* which flattens a given array in one dimension:

DEFINITION 3.5. *Let xs be an array of size $\frac{n}{m}$ containing as elements arrays of size m with elements $x_{i,j}$ for the i's inner array, where $0 < i \leq m$ and $0 < j \leq n$. The join primitive is then defined as follows:*

$$join([[x_{1,1}, \ldots, x_{1,m}], [x_{2,1}, \ldots, x_{2,m}], \ldots, [x_{n/m,1}, \ldots, x_{n/m,m}]] \stackrel{def}{=} \\ [x_{1,1}, \ldots, x_{1,m}, x_{2,1}, \ldots, x_{2,m}, \ldots, x_{n/m,1}, \ldots, x_{n/m,m}]$$

The type of join is defined as follows:

$$join(xs : [[T]_m]_{n/m}) \to [T]_n$$

TRANSPOSE *transpose* computes a matrix transposition. Therefore, *transpose* is defined over an at least two-dimensional array:

DEFINITION 3.6. *Let xs be an array of size m containing as elements arrays of size n with elements $x_{i,j}$ for the i's inner array, where $0 < i \leq m$ and $0 < j \leq n$. The transpose primitive is then defined as follows:*

$$transpose \left(\begin{bmatrix} [x_{1,1}, x_{1,2}, \ldots, x_{1,n}], \\ [x_{2,1}, x_{2,2}, \ldots, x_{2,n}], \\ \vdots \\ [x_{m,1}, x_{m,2} \ldots, x_{m,n}] \end{bmatrix} \right) \overset{def}{=} \begin{bmatrix} [x_{1,1}, x_{2,1}, \ldots, x_{m,1}], \\ [x_{1,2}, x_{2,2}, \ldots, x_{m,2}], \\ \vdots \\ [x_{1,n}, x_{2,n} \ldots, x_{m,n}] \end{bmatrix}$$

The type of transpose is defined as follows:

$$transpose : (xs : [[T]_n]_m) \to [[T]_m]_n$$

PAD *pad* adds additional elements that are determined by a so-called *boundary handling function* on either side of an array:

DEFINITION 3.7. *Let xs be an array of size n with elements x_i, where $0 \leq i < n$. Let l and r be positive integer values and h be a binary function over two integers. The pad primitive is then defined as follows:*

$$pad(l,\ r,\ h,\ [x_0, x_1, \ldots, x_{n-1}]) \overset{def}{=} \\ [b_{-l}, \ldots, b_{-2}, b_{-1},\ x_0, \ldots, x_{n-1},\ b_n, \ldots, b_{n+r-1}]$$

with

$$b_j = x_{h(j,n)}, \qquad \forall j : -l \leq j < n + r$$

The type of pad is defined as follows:

$$pad : \big(l : Int,\ r : Int, \\ h : (j : Int,\ size : Int) \to Int, \\ xs : [T]_n\big) \to [T]_{l+n+r}$$

The arguments l and r specify how many elements to add on the left and right side of the array, respectively. Then h determines which elements to add at which position by computing the index of an element in xs and placing the element at that index at a given position j. Hence, an added element is always equal to some element that already exists in the original array xs. For our purposes we need the boundary handling function *wrap* that adds elements from the opposite side of the input array. This way, the contents of an array are simply repeated when accessing elements at indices that exceed the input array's size. Note that we declared the indices of the elements in xs to start at 0 to be able to give a nice and short definition of wrap without differentiating between the left and right boundaries:

DEFINITION 3.8. *Let j and n be integer values. The boundary handling function $wrap$ is then defined as follows:*

$$wrap(j, n) = j \bmod n$$

The following example shows the effect of *wrap* used as the boundary handling function in *pad*.

EXAMPLE 3.1. *Set $xs = [x_1, x_2, x_3, x_4]$, $l = 2$, $r = 2$ and choose the boundary function to be wrap. Applying pad to xs then results in*

$$pad(2, 2, wrap, [x_1, x_2, x_3, x_4]) = [x_3, x_4, x_1, x_2, x_3, x_4, x_1, x_2]$$

ARRAY CONSTRUCTOR *array* is a primitive that is different from the afore mentioned ones. While the other primitives compute a result array based on some provided input array, the *array* constructor primitive is used to generate an array procedurally by invoking a generator function, without providing any initial values. Therefore, when the data in an array of a specific application is always the same, only differing depending on the array's size, the generation of said array's data can be directly included in an expression, by using the *array* primitive. We give a definition for *array* based on the elements in an array created by it:

DEFINITION 3.9. *Let m and $n_1, n_2, \ldots, n_m$ be positive integer sizes of different dimensions with the indices j_i, where $0 < j_i \leq n_i$, $\forall i : 0 < i \leq m$ and g be a $2m$-ary generator function. The element generated by the m-dimensional array constructor primitive at index $j_1, j_2, \ldots, j_m$, is then defined as follows:*

$$array_m(n_1, n_2, \ldots, n_m, g)(j_1, j_2, \ldots, j_n) \stackrel{def}{=} g(j_1, j_2, \ldots, j_m, n_1, n_2, \ldots, n_m)$$

The type of array is defined as follows:

$$\begin{aligned} array_m : (&n_1 : Int,\ n_2 : Int, \ldots, n_m : Int, \\ &g : (j_1,\ j_2, \ldots, j_m, \\ &\quad n_1, n_2, \ldots, n_m) \to T) \to [[\ldots[T]_{n_m}\ldots]_{n_2}]_{n_1} \end{aligned}$$

To determine the value of an element at a specific position in the array generated by the *array* constructor, the generator function g is invoked with the indices and the sizes of the different dimensions. Based on this information, values are calculated.

USER FUNCTION As we have seen in previous definitions, some primitives expect operators that might immediately operate on scalar values, e. g. *reduce*. The operators are provided in the form of functions that accept arguments of non-array types, only. We will refer to

this kind of function as *user functions*. Note that operations like complex addition $+_C$ and multiplication $*_C$ are user functions.

This concludes our definitions of high-level functional primitives. Now, we have all the tools that we need to decompose our expressions of FFT passes.

3.2 DECOMPOSING A STOCKHAM PASS

In this section, we will use the definition of a Stockham pass in matrix-vector notation from the previous chapter to derive expressions composed of high-level primitives that are composed to form an expression for a Stockham FFT pass. Some of these expressions can be reused to build Cooley-Tukey and Four-Step FFTs as well.

3.2.1 *Expressing the Permutation of a Stockham Pass*

In a Stockham pass, the order of the elements accessed during the Combine step is determined by an initial permutation step represented by the matrix Q_{p_q,L_q} (Definition 2.6). Because multiplying the matrix with a vector only results in a reordered vector, we do not need to express any operations like addition or the computations or loading of matrix elements that are usually necessary to implement a matrix-vector multiplication. Composing the provided primitives *split*, *join* and *transpose* is sufficient to express the permutation.

Before we go on to express Q_{p_q,L_q}, we note the following helpful property of the Kronecker product.

LEMMA 3.1 ([44]). *For an arbitrary matrix A of size $r \times r$, vectors v, y of size n, the identity matrix I_m of size $m \times m$ and $n = rm$, it always holds that*

$$y = (A \otimes I_m)v \Leftrightarrow y_{r\times m} = A \cdot v_{r\times m}$$

with $v_{r\times m}$, $y_{r\times m}$ being vectors interpreted as matrices where elements that are consecutive in the corresponding vector are adjacent to each other in rows of length m in the matrix. We call these matrices the $(r \times m)$ matrix-split of the vector.

To illustrate Lemma 3.1, we provide the following example.

EXAMPLE 3.2. *Let $m = 2$, $r = 2$ and $n = 4$, with A, I_2 and v as defined before. Then, the resulting vector y of*

$$(A \otimes I_2)v = \left(\begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\right) \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

$$= \begin{pmatrix} a_{0,0}v_0 + a_{0,1}v_2 \\ a_{0,0}v_1 + a_{0,1}v_3 \\ a_{1,0}v_0 + a_{1,1}v_2 \\ a_{1,0}v_1 + a_{1,1}v_3 \end{pmatrix} = y$$

can be interpreted as

$$y_{2\times 2} = \begin{pmatrix} a_{0,0}v_0 + a_{0,1}v_2 & a_{0,0}v_1 + a_{0,1}v_3 \\ a_{1,0}v_0 + a_{1,1}v_2 & a_{1,0}v_1 + a_{1,1}v_3 \end{pmatrix}$$

$$= \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} v_0 & v_1 \\ v_2 & v_3 \end{pmatrix} = A \cdot v_{2\times 2}$$

where y has simply been split into a matrix after $m = 2$ elements.

Applying Lemma 3.1 to $Q_{p_q,L_q}v$, shows that the permutation can be computed by multiplying the mod-p sort permutation matrix with all the columns of $v_{r\times m}$. Because the same permutation is applied to every column, instead of interpreting the permutation as being applied to each column, one column at a time, we can regard the rows of $v_{r\times m}$ as elements and then apply the mod-p sort permutation over the vector containing the rows.

To motivate our following expression, we note that the mod-p sort permutation, by definition, groups elements that are $p - 1$ elements apart from each other. Now assume that $k = lp$ and we want to mod-p sort a vector v of size k. Then, we know that for the matrix $v_{l\times p}$ the elements that form groups are in the columns. By transposing $v_{l\times p}$ and reinterpreting the matrix as a vector, we receive the mod-p sorted vector of v.

DEFINITION 3.10. *Let xs be an array of size n with elements x_j, where $0 \leq j < n$. Let p be a positive integer with $n = pm$. The modpsort expression is then defined as follows:*

$$modpsort(p, xs) \stackrel{def}{=} join \circ transpose \circ split(p)\ (xs)$$

It follows that the type of *modpsort* is:

$$modpsort : (p : Int,\ xs : [T]_n) \to [T]_n$$

We directly see that the given type is correct. By splitting the array once and joining it in the end, the amount of dimensions does not

change. Furthermore, *transpose* does not change the amount or type of elements in the array. Therefore, the result of applying *modpsort* is again an array with n elements of type T. Note that T can be an array type.

Suitable for Lemma 3.1, our next expression that we name *kronapply* represents the application of a function to every column of a matrix-split of a vector. First, we transform the array representation of an input vector such that it represents the matrix it can be split into by representing the columns of the matrix as separate arrays. Then the function is applied to each array representing the columns, by utilization of *map*.

DEFINITION 3.11. *Let xs be an array of size n with elements x_j, where $0 \leq j < n$. Let r be a positive integer with $n = rm$ and f be a function of type $[T]_r \to [T]_r$. The kronapply expression is then defined as follows:*

$$kronapply(m, f, xs) \stackrel{def}{=} join \circ transpose \circ map(f) \circ transpose \circ split(m)\ (xs)$$

It follows that the type of *kronapply* is:

$$kronapply : (m : Int,\ f : [T]_r \to [T]_r,\ xs : [T]_n) \to [T]_n$$

This type is correct. Applying $split(m)$, by definition, leads to an array $[[T]_m]_r$ that is transposed to an array $[[T]_r]_m$ and with *map* and the assumption about f, the type does not change. Then transposing and joining results in $[T]_n$.

By appropriately applying *kronapply* with *modpsort* to an array representing the input vector v of an FFT, we can now express the Stockham pass permutation.

DEFINITION 3.12. *Let vs be an array representation of v with n elements v_j, where $0 \leq j < n$. Let p be a positive integer with $n = mp$ and L be a positive integer with $L = rp$. The stockperm expression that represents the permutation of a Stockham pass is then defined as follows:*

$$stockperm(p,\ L,\ vs) \stackrel{def}{=} kronapply(n/L,\ modpsort(p),\ vs)$$

It follows that the type of *stockperm* is:

$$stockperm : (p : Int,\ L : Int,\ vs : [Complex]_n) \to [Complex]_n$$

Note how we exploit the fact that functions are curried. *modpsort* expects 3 arguments, but we apply it only to two, which leads to a new function that computes the mod-p sort permutation for set p and L over an array *ys* of size m and is of type $[T]_m \to [T]_m$. Now, we immediately see that the given type is correct. The parameters are clearly defined, with *xs* being an array representation of a complex vector. The resulting type directly depends on the type of *kronapply*, where

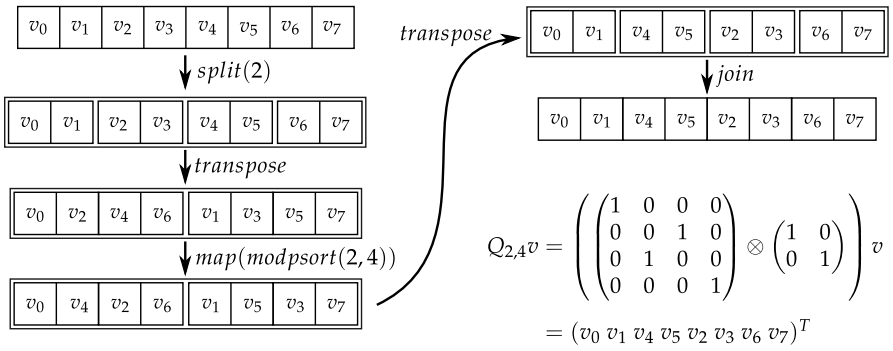

Figure 3.1: Application of *kronapply* with $p = 2$ and $L = 4$.

$T = Complex$. Figure 3.1 depicts the permutation of an array representation of a vector with 8 elements in comparison to multiplying $Q_{2,4}$ with v. The applied permutation is part of a Stockham FFT pass with the current radix being $p = 2$. The DFTs that are being computed in this pass are of size $L = 4$. We can see that the transformed array is a representation of the result vector of the multiplication. Finally, this leads us to the following Lemma and conclusion of deriving an expression for the Stockham pass permutation:

LEMMA 3.2. *Let vs be the array representation of the vector v of size n. Let p be a radix and L be a positive integer with $L = pr$. For*

$$stockperm(p,\ L,\ vs) = [y_0, y_1, \ldots, y_{n-1}] = ys,$$

it holds that:

$$y_j = \left(Q_{p,L}v\right)_j, \quad \forall j : 0 \leq j < n$$

meaning that ys is an array representation of $Q_{p,L}v$.

Proof. A proof for this Lemma can be found in the Appendix A.1.4. □

3.2.2 *Expressing the Combine Step of A Stockham Pass*

Traditionally, FFT algorithms have either been provided in an imperative style based on loops or in a functional style based on recursion. These algorithms usually utilize some means to compute needed values in-line by simply specifying a calculation rule that has knowledge about values such as the current FFT pass, the parameter used in previous passes and most importantly the indices of the accessed vector elements.

An example for this can be seen in Listing 2.1, where the index and twiddle calculations depend on some of the same values (e. g. `LPrev` or the loop variable `j`). The example implementation computes an FFT in which all radices equal 2. In a more general algorithm, the

$$B_{2,4} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -i \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & i \end{pmatrix} \longleftrightarrow \begin{bmatrix} [[1,1], & [1,-i]\ ], \\ [[1,1], & [-1,i]\ ] \end{bmatrix} \; B(2,4)$$

Figure 3.2: Example $B(p,\ L)$ with $p = 2$ and $L = 4$ in comparison to $B_{2,4}$.

radix for the current FFT pass needs to be handed to the Combine computation as well. With the use of functional primitives like *map*, index information is not immediately available to the functions applied to an array. Therefore, we have to find a different approach to providing the values that are needed during an FFT pass. Instead of focusing on the input vector alone and computing values accordingly, our idea is to have a representation of the necessary values in (multidimensional) array form, together with the vector that will be represented as a one-dimensional array, available. We can then combine the matrix with the vector as is done in matrix-vector multiplications for example.

We refer to Definition 2.7 to see that the Combine step can be expressed as the matrix

$$S_{p_q,L_q} = B_{p_q,L_q} \otimes I_{\frac{n}{L_q}}.$$

With Lemma 3.1, we see that $(B_{p_q,L_q} \otimes I_m)v$ can be calculated by consecutively computing matrix-vector multiplications over the columns of $v_{r\times m}$. Hence, instead of representing S_{p_q,L_q} in its entirety, we concentrate on providing a representation for a Butterfly matrix B_{p_q,L_q} that can then be multiplied with appropriately reordered and grouped parts of the array that represents the input vector v.

ARRAY REPRESENTATION OF A BUTTERFLY MATRIX Fortunately, a Butterfly matrix has a highly regular structure that allows us to use our previously defined primitives to express it. It is a square matrix of size $L_q \times L_q$ and consists of block matrices that are diagonal matrices. These block matrices neatly form rows and columns on their own. Figure 3.2 shows an example for $p = 2$ and $L = 4$. The first row of diagonal block matrices is marked. The array representation that can be seen is defined in the following. We can represent a Butterfly matrix in array terms by giving an expression for an array that holds all its non-zero entries. First, we have to define how the non-zero elements are calculated, based on our understanding of the Butterfly matrix's structure. We provide the definition of a user function, that computes a value depending on the row (*blr*) and column (*blc*) of block matrices it is in and its position on the diagonal (*dpos*) of its block matrix.

DEFINITION 3.13. *Let blr, blc, dpos, sizeR, sizeC and sizeD be positive integers. The function genB is then defined as follows:*

$$genB(blr,\ blc,\ dpos,\ sizeR,\ sizeC,\ sizeD) \stackrel{def}{=} \\ omega((blr \cdot sizeD + dpos) \cdot blc,\ sizeC \cdot sizeD)$$

We see immediately that the type of $genB$ is as follows:

$$genB : (blr : Int,\ blc : Int,\ dpos : Int, \\ sizeR : Int,\ sizeC : Int, sizeD : Int) \to Complex$$

Note that this user function has a type that fits into the $array_3$ primitive. We make use of this fact in an instant by defining the array representation of $B_{p,L}$.

DEFINITION 3.14. *Let p and L be positive integer values with $L = pm$. The array representation of the matrix $B_{p,L}$ is then defined as follows:*

$$B(p,\ L) \stackrel{def}{=} array_3(p,\ p,\ L/p,\ genB)$$

The type follows directly from the definition of $array_3$:

$$B : (p : Int,\ L : Int) \to [[[Complex]_{L/p}]_p]_p$$

Looking at Figure 3.2 again, we see that with our definition, rows of block matrices are regarded as a two-dimensional array of diagonal matrices that are represented by a one-dimensional array. We will soon show that this representation of $B_{p,L}$ is indeed useful as we extend it to represent a Combine matrix and provide an expression for multiplication with a vector.

EXPRESSING THE STOCKHAM COMBINE STEP Figure 3.3 shows how the rows of block matrices affect different values of a vector during matrix-vector multiplication. Note that all the values from a single row of block matrices are multiplied with different values in the vector. The pattern repeats itself over all the rows of block matrices.

To express the multiplication of the matrix with a vector, we reorder the array B such that the values found in a row of $B_{p,L}$, are grouped into one-dimensional arrays. By reordering the array representation of the vector that we multiply $B_{p,L}$ with, in a way that values that are affected by a single row of $B_{p,L}$ (e. g. the circles in Figure 3.3) are grouped together in one-dimensional arrays as well, we are able to later combine the appropriate values. We provide expressions for both, the array representation of the vector and the B array's reordering:

DEFINITION 3.15. *Let xs be an array of size L and p be a positive integer. The expression prepBmult and its inverse $prepBmult^{-1}$ are then defined as follows:*

$$prepBmult(p,\ xs) \stackrel{def}{=} transpose \circ split(L/p)\ (xs) = ys$$

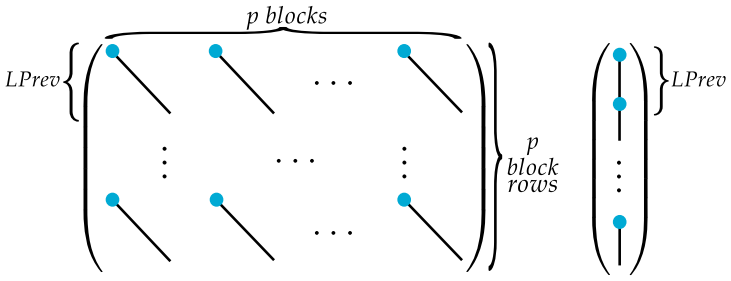

Figure 3.3: General structure of a Butterfly matrix $B_{p,L}$. The black diagonals depict entries on the main diagonals of inner block matrices, empty space represents zero entries. The circles show elements in the $B_{p,L}$ matrix that are multiplied with elements in the vector during matrix multiplication.

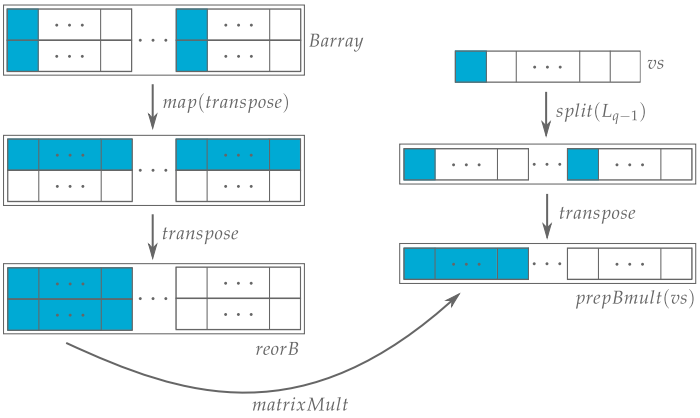

Figure 3.4: Preparing the array B and the array representation of v that we call *varr*, to express Butterflies. Elements are highlighted to depict values from B that affect *varr* during a matrix-vector multiplication.

and

$$prepBmult^{-1}(ys) \stackrel{def}{=} join \circ transpose\ (ys) = xs$$

DEFINITION 3.16. *Let n, p and L be positive integers, with $n = pm$ and $L = pk$. The array representation of the Combine matrix is then defined as follows:*

$$reorB(p,\ L) \stackrel{def}{=} transpose \circ map(transpose)\ (B(p, L))$$

Figure 3.4 shows the structure of *prepBmult* and a reordered B array. We see that when the two-dimensional arrays in *reorB* are matrix-multiplied with their counterparts in the prepared vector array, the resulting values are what is computed during a matrix-vector multiplication of B with a vector. The only difference is that the values are in the wrong order because of the previously performed transformation. By applying the inverse transformation the order would be restored. We provide the definition of an expression for matrix-vector multiplications from [40].

DEFINITION 3.17. *Let xs and ys be array representations of complex vectors of size n, M be an array representation of a $m \times n$ matrix, then*

$$matrixMult(M,\ xs) = join \circ map(dotproduct(xs))\ (M)$$

where

$$dotproduct(xs,\ ys) = reduce(+,0) \circ map(*) \circ zip(xs, ys)$$

is the array representation of the dotproduct of two vectors.

It follows that the type of *matrixMult* is:

$$matrixMult : (M : [[Complex]_n]_m,\ xs : [Complex]_n) \to [Complex]_n$$

We remember that *reduce* has the type $[T]_1$. Therefore, the *join* in *matrixMult* is necessary to receive an array of one dimension.

Finally, we can express the multiplication of B with a vector. This multiplication will then be applied several times over parts of a bigger input array, to compute the Combine step, i. e. the multiplication of S_{p_q,L_q} with a vector v.

DEFINITION 3.18. *Let p and L be positive integers with $L = pm$. Let xs be an array of size L. The expression multB is then defined as follows:*

$$\begin{aligned} multB(p,\ L,\ xs) \overset{def}{=} \\ prepBmult^{-1} \circ map(matrixMult) \circ \\ zip(reorB(p,\ L)) \circ prepBmult(p)\ \ (xs) \end{aligned}$$

We recall that we provided an expression to compute the repeated multiplication of a matrix over parts of an input vector that results from a property of the Kronecker product in conjunction with an identity matrix, i. e. the expression *kronapply* defined in Definition 3.11. Using this we can express the Combine step:

DEFINITION 3.19. *Let vs be an array representation of v with n elements v_j, where $0 \leq j < n$. Let p be a positive integer with $n = mp$ and let L be a positive integer with $L = rp$. The stockcombine expression that represents the Combine step of a Stockham pass is then defined as follows:*

$$stockcombine(p,\ L,\ vs) \overset{def}{=} kronapply(n/L,\ multB(p,L),\ vs)$$

It follows that the type of *stockcombine* is:

$$stockcombine : (p : Int,\ L : Int,\ vs : [Complex]_n) \to [Complex]_n$$

Indeed, we are allowed to use the only partly applied expression $multB(p,L)$ because its type is $[Complex]_L \to [Complex]_L$. Then the type of *stockcombine* immediately follows from *kronapply*. The *stockcombine* expression generally works for Stockham passes:

LEMMA 3.3. *For*

$$stockcombine(p, L, vs) = [y_0, y_1, \ldots, y_{n-1}] = ys,$$

it holds that:

$$y_j = (S_{p_q,L_q} v)_j, \quad \forall j : 0 \leq j < n$$

meaning that ys is an array representation of $S_{p_q,L_q} v$.

Proof. A proof for this Lemma can be found in the Appendix A.1.5. □

A COMPLETE DECOMPOSITION OF STOCKHAM PASS We were able to provide expressions for both steps of a Stockham pass that are based on high-level data parallel functional primitives. For the sake of completeness, we will conclude this section with a theorem that states just that.

THEOREM 3.1. *Let vs be an array representation of the vector v with size n and elements* v_j *where* $0 \leq j < n$, *p a positive integer with* $n = mp$ *and L be a positive integer with* $L = rp$. *It then always holds that:*

$$stockhamPass(p, L, vs) = stockcombine(p, L) \circ stockperm(p, L)\ (vs)$$

Proof. We simply apply the Lemmata for the two subexpressions.

$$\begin{aligned}
&stockcombine(p, L) \circ stockperm(p, L)\ (vs) \overset{Lemma\ 3.2}{=} \\
&\quad stockcombine(p, L)\ ([(Q_{p,L} v)_0, \ldots, (Q_{p,L} v)_{n-1}]) \overset{Lemma\ 3.3}{=} \\
&\quad\quad [(S_{p,L} Q_{p,L} v)_0, \ldots, (S_{p,L} Q_{p,L} v)_{n-1}] \overset{Def.\ 2.11}{=} \\
&stockhamPass(p, L, vs)
\end{aligned}$$

□

3.3 SEPARATING THE COMBINE STEP INTO TWIDDLE AND DFT

In the definitions of the Butterfly matrix B_{p_q,L_q}, we see that it can be decomposed into a multiplication with so-called twiddle factors and a smaller DFT computation of size p_q. Often, this decomposition is used in practice, because it allows optimizing the smaller DFT computations. Note however that, directly implementing this split leads to more arithmetic operations initially because the DFT computations need as many operations as the Butterflies and we have to additionally multiply twiddle factors. Additionally, the optimizations that are usually applied are not possible in Lift, because they are based on low-level imperative register optimizations that are specific to FFTs (i. e. they use similarities in twiddle factors to save arithmetic operations). Nonetheless, we are be able to make use of this decomposition in our own way and begin deriving an expression by defining the array representation for the matrix holding the twiddle values:

DEFINITION 3.20. *Let p and L be positive integer values. The array representation of the matrix $diag(\Omega^0_{p,\frac{L}{p}}, \Omega^1_{p,\frac{L}{p}}, \ldots, \Omega^{p-1}_{p,\frac{L}{p}})$ is then defined as follows:*

$$Twiddle(p, L) \stackrel{def}{=} array_2(L/p,\ p, genTwiddle)$$

where

$$genTwiddle(diag, exp, sizeOmegas, amountOmegas) \stackrel{def}{=} omega(diag \cdot exp, sizeOmegas \cdot amountOmegas)$$

The type follows directly from the definition of $array_2$:

$$Twiddle : (p : Int,\ L : Int) \to [[Complex]_p]_{L/p}$$

With this representation, the values that are on the diagonal of an $\Omega^j_{p,L/p}$, are stored in the columns of the *Twiddle* array.

To express the DFT computation we use an array constructor to generate a matrix representation of DFT_p. Because no values are omitted or reordered for this representation, we skip a formal definition and give the full expression using *Twiddle* right away.

DEFINITION 3.21. *Let vs be an array representation of the vector v of size n. Let p be a positive integer with $n = mp$ and let L be a positive integer with $L = rp$. The expression stockhamTwPass that represents a Stockham pass with decomposed Butterfly matrices is then defined as follows:*

$$\begin{aligned} &stocktwcombine(p, L, vs) = \\ &\quad kronapply(n/L, prepBmult^{-1} \circ map(matrixMult(DFT_p)) \circ \\ &\quad\quad map(map(*) \circ zip) \circ zip(Twiddle) \circ prepBmult(p), vs) \end{aligned}$$

With this expression, we get an equivalent alternative to our previous *stockcombine* expression.

3.4 DECOMPOSING A COOLEY-TUKEY PASS

In this short section, we use the ideas and definitions introduced previously, to decompose the expressions for a Cooley-Tukey pass and the Bit-Reversal pass. Because the approach uses exactly the same ideas, we do not go into much detail as to why the definitions are correct. We start with the Bit-Reversal expression and follow it up with the Cooley-Tukey passes using once the array B and once the array *Twiddle*.

3.4.1 *Expressing Bit-Reversal*

By definition, the Bit-Reversal matrix $P_n(\rho)$ is created with matrices R_q for pass q of the form:

$$R_q = I_{n/L_q} \otimes \Pi_{p_q, L_q}$$

with the definition of the Kronecker product, we see that R_q has the following structure:

$$R_q = \begin{pmatrix} \Pi_{p_q,L_q} & 0 & \dots & 0 \\ 0 & \Pi_{p_q,L_q} & \dots & 0 \\ & \ddots & & \\ 0 & \dots & 0 & \Pi_{p_q,L_q} \end{pmatrix}$$

where n/L_q permutation matrices form a diagonal of block matrices. Hence, consecutive parts of size L_q of an input vector are reordered according to Π_{p_q,L_q}. Therefore, we can express R_q by giving an expression for Π_{p_q,L_q} and applying it to chunks of the input array. We know that $\Pi^T_{p,L}$ is the mod-p sort permutation for a vector of size L. With an example, we show what happens, when we transpose this matrix.

EXAMPLE 3.3. *Let v be a vector of size 6 and $p = 2$. Multiplying the transposed mod-2 sort permutation to the vector results in its mod-3 $(\frac{6}{2})$ sort permutation.*

$$\Pi_{2,6}v = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T \cdot \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{pmatrix}$$
$$= \begin{pmatrix} v_0 & v_3 & | & v_1 & v_4 & | & v_2 & v_5 \end{pmatrix}^T$$

We see that the elements of v are now grouped by mod-3 classes of their indices.

We can therefore provide a definition for an R_q using our *modpsort* expression, as follows:

DEFINITION 3.22. *Let xs be an array of size n. Let p and L be positive integers. The expression cooleyperm is then defined as follows:*

$$cooleyperm(p,\ L,\ xs) = \\ \quad join \circ map(modpsort(L/p)) \circ split(L)\ (xs)$$

It follow that the type of *cooleyperm* is:

$$cooleyperm : (p : Int,\ L : Int,\ xs : [T]_n) \to [T]_n$$

After splitting xs into parts of size L, the permutations are applied over the parts. From the type of *modpsort* it follows that the reordered

parts have the same dimension as the non-reordered parts. Therefore, after applying *join* the type follows.

Now, the Bit-Reversal matrix can be expressed by composing multiple *cooleyperm* expressions with the appropriate parameters for the different passes:

LEMMA 3.4. *Let vs be an array representation of the complex vector v of size n. Let* $\rho = \langle p_1, \ldots, p_t \rangle$*, be a collection of radices that is a factorization of* n*,* $n = p_1 \cdot \ldots \cdot p_t$*. Let* L_q *be* $L_q = p_1 \cdot \ldots \cdot p_q$*, where* $1 \leq q \leq t$*. It then holds that:*

$$bitrev(\rho,\ vs) = \\ cooleyperm(\rho(t),\ L_t) \circ \ldots \circ cooleyperm(\rho(1), L_1)\ (vs)$$

Proof. We show that *cooleyperm* is well defined. Then, the lemma follows from the Definition of the Bit-Reversal matrix (Definition 2.4). With the Definition of $\otimes$ (Definition 2.1), we immediately see that

$$(I_{n/L} \otimes \Pi_{p,L})v = \begin{pmatrix} \Pi_{p,L} v(0:1:L-1) \\ \Pi_{p,L} v(L:1:2L-1) \\ \vdots \\ \Pi_{p,L} v(n-L:1:n-1) \end{pmatrix}^T$$

We apply *cooleyperm* to *vs*:

$$cooleyperm(p, L, vs) = \\ join \circ map(modpsort(L/p)) \circ split(L)\ (xs)$$

With the Definition 3.4 $split$, we see that the resulting two-dimensional array after $split(L, vs)$, holds the values $v(kL : 1 : (k+1)L - 1)$ in row k, with $0 \leq k < n/L$.

In Appendix A.1, we show that *modpsort* is well defined. Therefore, with Definition 3.1 *map*, it follows that after

$$map(modpsort(L/p))(split(L, vs)$$

the resulting array holds the values of $\Pi_{p,L} v(kL : 1 : (k+1)L - 1)$ in row k.

With the definition of *join*, the statement follows. □

The way that we decompose *cooleytukeyPass* follows from the same reasoning that we used for *cooleyperm*. In the definition of the combine matrix for a Cooley-Tukey pass (Definition 2.5) we see that Butterfly matrices are simply multiplied with consecutive chunks of a vector. This leads us to the following remark.

LEMMA 3.5. *Let vs be an array representation of the complex vector v of size n. Let p be the radix of the current FFT pass and* $n = mp$*. Let L be a positive integer with* $L = rp$*. It then holds that*

$$cooleytukeyPass(p,\ L,\ vs) = \\ join \circ map(multB(p, L)) \circ split(L)$$

Proof. Similar to the proof of Lemma 3.4, we want to split the array representation of v into consecutive parts of size L.

With the Definition 3.4 *split*, we see that the resulting two-dimensional array after $split(L, vs)$, holds the values $v(kL : 1 : (k+1)L-1)$ in row k, with $0 \leq k < n/L$.

In Appendix A.3, we show that *multB* is well defined. Therefore, with Definition 3.1 *map*, it follows that after

$$map(multB(p, L))(split(L, vs)$$

the resulting array holds the values of $B_{p,L}v(kL : 1 : (k+1)L-1)$ in row k.

With the definition of *join* it follows that *cooleyperm* is well defined. □

To conclude this section, we mention that a Cooley-Tukey pass with the Butterflies separated into *Twiddle* and DFT can be defined analogously.

3.5 FOUR-STEP FFT

The Four-Step FFT represents another method to compute FFTs. It gets its name from the fixed number of passes that were originally used to implement it. For our implementations aimed at GPUs later on, we might fuse passes and add others to adapt the algorithm to the new hardware architectures. In this section, we will give an overview over the mathematical idea behind the Four-Step method and provide a recursive definition of it based on high-level primitives. We will not provide a formal definition of the Four-Step expression without high-level primitive, because it simply represents an entire DFT computation. The Four-Step method is based on the following property of the DFT_n matrix, as shown in [44]:

$$(DFT_n v)_{n_1 \times n_2} = \\ DFT_{n_1}(DFT_n(0 : n_2 - 1, 0 : n_1 - 1) * (DFT_{n_2} v_{n_2 \times n_1}))^T$$

We use the matrix-split of a vector notation introduced with Lemma 3.1, with n_1 and n_2, where $n = n_1 \cdot n_2$. The operator $.*$ stands for the element wise product of two matrices and the matrix $DFT_n(0 : n_2 - 1, 0 : n_1 - 1)$ contains all the elements in the upper left corner of DFT_n that form a submatrix of size $n_2 \times n_1$. For the Four-Step method, these elements are commonly called twiddle factors.

Traditionally, from above formula, the following four passes were derived. First, a DFT is computed over the columns of the matrix-split of the input vector v. Then, the twiddle factors are multiplied and the resulting matrix is transposed. Afterwards, a DFT over the columns

is computed. This leads to a $n_1 \times n_2$ matrix-split of the transformed input vector. More formally, we can express the passes as follows:

1. $v_{n_2 \times n_1} := DFT_{n_2} v_{n_2 \times n_1}$
2. $v_{n_2 \times n_1} := DFT_n(0 : n_2 - 1, 0 : n_1 - 1) * v_{n_2 \times n_1}$
3. $v_{n_1 \times n_2} := v^T_{n_2 \times n_1}$
4. $v_{n_1 \times n_2} := DFT_{n_1} v_{n_1 \times n_2}$

This leads us to the following expressions.

FFT COMPUTATIONS FOUR-STEP PASS Previously, we have seen that multiplying a matrix to the matrix-split of a vector can be expressed with the *kronapply* expression, where f represents the transformation that occurs due to the matrix multiplication. Therefore, we can express a pass in which multiple FFTs are computed with:

DEFINITION 3.23. *Let vs be an array representation of the vector v of size n. Let n_1 and n_2 be positive integers with $n = n_1 \cdot n_2$ and let $fft(\rho)$ be an expression representing an initialized FFT expression. Let $\rho = \langle p_1, \ldots, p_t \rangle$ be a collection of radices that form a factorization of n_2, with $n_2 = p_1 \cdot \ldots \cdot p_t$. A pass of the Four-Step method in which FFTs are computed, can then be expressed as follows:*

$$fourfftPass(fft(\rho), n_1, vs) = kronapply(n_1, fft(\rho), vs)$$

An initialized FFT expression is an expression for which every parameter except the input array, has been set. It follows that the type of $fourfftPass$ is:

$$fourfftPass : ([Complex]_{n_2} \to [Complex]_{n_2}, \\ n_1 : Int,\ vs : [Complex]_n) \to [Complex]_n$$

This is proven by directly substituting the type of *kronapply*.

Note that this definition allows us to recursively apply the Four-Step method, later.

TWIDDLE MULTIPLICATION PASS Next, we show how to express the twiddle multiplication. We construct an array that represents the *slice* of the DFT_n matrix. Then, elements are grouped with zip and the complex multiplication is mapped over the tuples.

DEFINITION 3.24. *Let xs be an array of size n with $n = n_1 \cdot n_2$.*

$$twiddleMult(n_1, n_2, xs) = \\ join \circ transpose \circ map(map(*)) \circ map(zip) \circ \\ zip(array_2(n_1, n_2, genDFTSlice)) \circ \\ transpose \circ split(n_1)\ (xs)$$

where genDFTSlice is a generator function for the elements of a DFT_n matrix:

$$genDFTSlice(j,\ k,\ height,\ width) = omega(j \cdot k, width \cdot height)$$

Now, we have all the tools that we need to formulate an expression for a complete Four-Step FFT. This is shown with the following theorem.

THEOREM 3.2. *Let vs be an array representation of the vector v of size n. Let n_1 and n_2 be positive integers with $n = n_1 \cdot n_2$ and let $fft(\rho)$ be an expression representing an initialized FFT expression. Let $\rho_j = \langle p_{1_j}, \ldots, p_{t_j} \rangle$ be a collection of radices that form a factorization of n_j. So $n_j = p_{1_j} \cdot \ldots \cdot p_{t_j}$ with $j \in \{1, 2\}$. Then, the expression*

$$\begin{aligned} &fourStep(\rho_1, \rho_2, vs) = \\ &\quad fourfftPass(ff(\rho_1), n_2) \circ \\ &\quad join \circ transpose \circ split(n_1) \circ \\ &\quad twiddleMult(n_1, n_2) \circ \\ &\quad fourfftPass(fft(\rho_2), n_1)\ (vs) \end{aligned}$$

represents the multiplication of DFT_n with v.

Proof. The lines in the above equations represent the passes of the Four-Step method from bottom to top. The correctness of this method has been shown in [44]. The applications of $fourfftPass$ express the FFTs over the columns of the matrix-split. This follows directly from the definition of *kronapply* which is well-defined as shown in the Appendix A.4. The twiddle multiplication pass is expressed by *twiddleMult* as shown in the Appendix A.5. It remains to be shown that

$$join \circ transpose \circ split(n_1)$$

expresses transposing the matrix-split $v_{n_2 \times n_1}$. With $split(n_1)$ the $(n_2 \times n_1)$ matrix-split is created. This follows directly from the definitions of *split* in Definition 3.4 and the matrix-split in Lemma 3.1. Then, the matrix-split is transposed with *transpose* which leads to an array representation of the matrix-split $v^T_{n_2 \times n_1}$, which was the goal. □

4

REWRITING EXPRESSIONS

In this chapter, we rewrite our FFT expressions in order to obtain expressions that can be compiled to efficient OpenCL kernels by Lift. Rewrites on high-level expressions are used to change compositions of FFT passes algorithmically and to simplify them in order to *lower* them to low-level expressions. Lowering describes the process of rewriting expressions into low-level functional primitives that form Lift's intermediate language and are closely related to hardware or a specific programming model. Our focus will lie on OpenCL specific primitives. Lowering our FFT expressions will allow us to specify how certain operations are executed by OpenCL and therefore, because of their close relation, transitively by a GPU.

4.1 LOW-LEVEL OPENCL-SPECIFIC FUNCTIONAL PRIMITIVES

This section is dedicated to providing definitions of low-level functional primitives that are closely related to the OpenCL programming model. These primitives have been defined in [37] before.

PARALLEL MAP The low-level OpenCL-specific *map* primitives form a way to exploit the organizational structure of OpenCL's execution and data parallel programming model by specifying how tasks and data map to work-items and work-groups. The high-level semantics of *map*, as defined in Definition 3.1, are preserved in the low-level primitives and only extended by additional information concerning the OpenCL model. There exist two ways to assign work to threads, either by considering the OpenCL thread hierarchies or not:

> *mapGlobal*: The *mapGlobal* primitive assigns work to all the work-items, independent of their work-groups.
>
> *mapWorkgroup*/*mapLocal*: The *mapWorkgroup* primitive assigns work to a work-group. However, because it needs to be clarified how the work inside of a WG is assigned, this primitive has to be used in conjunction with at least one nested *mapLocal* primitive that assigns work to all the WI of the WG.

Furthermore, each of these primitives can be parameterized to resemble the three-dimensional space in which WIs can be organized in OpenCL. We will simply denote the dimension by adding an index $j \in \{0, 1, 2\}$ to the primitive, e. g. $mapGlobal_1$ to assign work to all the work-items in the first dimension. This index is omitted, when there

exists only one low-level *map* in the expression where the index is then implicitly 0.

SEQUENTIAL MAP AND REDUCE We have defined primitives to assign work to multiple work-items. In order to perform a mapping or reduction sequentially in a single work-item, we introduce *mapSeq* and *reduceSeq*. We could relax the definition of *reduceSeq* in comparison to *reduce* by not requiring associativity of the binary customizing operator, simply because it was only required for parallel implementations. However, we only want to use associative operators in this thesis and therefore do not provide a new definition.

MEMORY ASSIGNMENT The OpenCL memory hierarchy is represented by the *toGlobal*, *toLocal* and *toPrivate* primitives. They are used to specify where the results of a function are stored. With the *toGlobal* primitive, results are stored in global memory. For GPUs, that means it is stored in the slow off-chip GPU RAM. With the *toLocal* primitive, results are stored in local memory. For GPUs, that means they are stored in the fast on-chip shared memory. With the *toPrivate* primitive, results are stored in private memory that is only visible to the owning WI. For GPUs, that means the results are stored in the very fast registers. We provide a collective definition of the functions:

DEFINITION 4.1. *Let f be a function. Let toMem be one of the memory assignment primitives toGlobal, toLocal, toPrivate. Assigning the results of applied f to a specific memory region is then defined as follows:*

$$toMem(f) \stackrel{def}{=} f',$$

where

$$f'(x) \stackrel{def}{=} f(x),$$

for all possible x and f' is guaranteed to store the results in the memory region associated with the provided toMem.

The types of the primitives are defined as follows:

$$toMem : (f : (U \to V)) \to (U \to V)$$

Note that there does not exist a primitive to assign work to constant memory. This is simply due to the fact that data in constant memory is assigned before the execution of an OpenCL kernel. Either the host program issues a command to transfer the data or a constant memory array is provided inside an OpenCL program, which is initialized before the execution of a kernel.

```
1  dotproduct = λ((xs, ys) ⇒ reduce(+, 0) ∘ map(∗) ∘ zip(xs, ys))
2  matrixMult =
3      λ((M, xs) ⇒ join ∘ map(λ(Mrow ⇒ dotproduct(xs, Mrow))))
4
5  prepBmult^−1 = join ∘ transpose
6  prepBmult = transpose ∘ split(L/p)
7
8  reorB = transpose ∘ map(transpose)  (array_3(p, p, L/p, genB))
9
10 stockperm = join ∘ transpose  ∘
11                 map(join ∘ transpose ∘ split(p))  ∘
12                   transpose ∘ split(n/L)
13
14 stockhamPass =
15     λ(vs ⇒ join ∘ transpose  ∘
16             map(λ(vectorMatrixColumn ⇒
17                     prepBmult^−1  ∘
18                       map(matrixMult,
19                           zip(reorB, prepBmult(vectorMatrixColumn)))
20             ))  ∘ transpose ∘ split(n/L) ∘ stockperm  (vs))
```

Listing 4.1: A Stockham pass expression in lambda notation.

USER FUNCTIONS We mentioned user functions during the definitions of high-level primitives. There, they were simply regarded as functions operating on non-array types. In Lift, these functions are defined in OpenCL-C. Therefore, they can simply be declared in OpenCL kernels and there is no way to lower them even further.

4.2 LOWERING THE STOCKHAM PASS EXPRESSION

A high-level expression is rewritten into a low-level expression by applying rewrite rules that replace high-level primitives such as *map* with their low-level counterparts. Some rewrite rules can be applied only if the high-level expression is in an appropriate form, e. g. if the high-level expression contains only a single *map* then *mapWorkgroup* cannot be used, because there needs to be a nested *mapLocal* inside of it. Hence, it can be useful to rewrite a high-level expression and then lower it afterwards.

ASSIGNING WORK TO WORK-ITEMS It is important to note that not every high-level expression has to be rewritten to create a low-level expression. Many primitives do not define a computation over data and merely change its layout. We call them *data layout primitives*. Some of these primitives include *join*, *split*, *pad*, *transpose*, *zip* and even *array*. Because the changes to the data layout are the same in a low-level expression, they can simply be kept as is. Furthermore,

```
1  dotproduct = λ((xs, ys) ⇒ reduceSeq(+, 0) ∘ mapSeq(∗) ∘ zip(xs, ys))
2  matrixMult =
3      λ((M, xs) ⇒ join ∘ mapSeq(λ(Mrow ⇒ dotproduct(xs, Mrow)))
4  ...
5
6  stockhamPass =
7      λ(vs ⇒ join ∘ transpose ∘
8              mapGlobal_1(λ(vectorMatrixColumn ⇒
9                      prepBmult^−1 ∘
10                       mapGlobal_0(matrixMult,
11                           zip(reorB, prepBmult(vectorMatrixColumn)))
12             )) ∘ transpose ∘ split(n/L) ∘ stockperm (vs))
```

Listing 4.2: A lowered Stockham pass expression. Depicted are only the parts that changed.

mapping a data layout primitive or an expression that is a composition of data layout primitives does not specify computations as well. Therefore, there is no need to rewrite a *map* that only contains data layout primitives. In fact, no *map* that does not contain a user function at some point in its nested expression, either on its own or provided in a *reduce*, needs to be lowered.

We use a new notation for our expressions to increase readability and point out some specifics of the Lift system. The notation is based on so-called *lambda functions* (λ, in short simply *lambda*), anonymous functions that can wrap an expression in order to supply parameters and can be used in an expression's place. In fact, a composition of high-level primitives implicitly creates a lambda.

The decomposition of our previously defined *stockhamPass* into high-level primitives can be seen in Listing 4.1. Note that FFT specific parameters like the radix p are not part of the lambda's. This is due to the fact that the current Lift implementation does not support a way to pass non-array parameters from a lambda to data layout primitives. This problem is attributed to the types of the lambda parameters in their current Scala implementation and can be addressed in future versions of Lift. Additionally, the *stockperm* expression does not explicitly declare a lambda function because it implicitly creates one from the instantiated primitives which all expect a single array as an input. Also, *stockperm* consists of data layout primitives only.

By replacing a single *map* in *stockhamPass* with *mapGlobal* and the rest of the *map*s inside of it with *mapSeq*, we are able to distribute Butterfly multiplications over several work-items. This leads to multiple matrix-vector multiplications per WI. We can go even further and replace two *map* primitives with *mapGlobal* in two dimensions. That way, we are able to distribute the loading and combining of separate sets of array elements in a way that it is possible to perform a single matrix-vector multiplication per work-item, if enough WIs

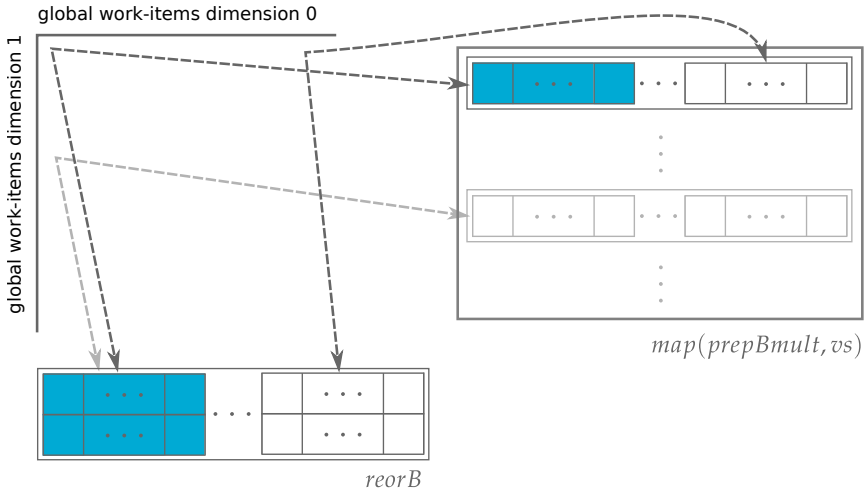

Figure 4.1: Distribution of data over work-items for a lowered *stockhamPass*.

are created. This leads to more flexibility when tuning a generated OpenCL kernel later. The resulting expression is shown in Listing 4.2 and is in fact a completely lowered expression for Lift, although many high-level *map*s remain.

Figure 4.1 depicts how the expression is distributed over the global work-items. The grid of work-items is depicted in the upper left corner. The position of the WI in dimension 0 determines which part of the reordered array that represents a column of the matrix-split of a vector is accessed together with the appropriate part of B. The position in dimension 1 determines which column of the matrix split is accessed and has no influence on the values that are accessed in B.

Transforming to the low-level expression preserves the semantics of our high-level expression. We take the rewrite rules that we need from [37], where they have been proven to be correct:

REWRITE RULE 1. *Lowering Rule*

$$map \rightarrow \quad mapGlobal \mid mapWorkgroup \mid mapLocal \mid mapSeq$$

REWRITE RULE 2. *Lowering Rule*

$$reduce(init, \oplus) \rightarrow reduceSeq(init, \oplus)$$

Listing 4.3 depicts the generated code for the computation of the dot product in a *stockhamPass* with $p = 2$. Note that the intermediary results after the multiplication are stored in a variable that points to global memory and that the result of the reduction is stored in private memory. We want to store the intermediate results in the fastest memory segment possible. In this case because the values are accessed by a single WI only, this means that we can store the results in private memory, i. e. registers. Additionally, to create a valid low-level expression, it is important that the end result is stored in global memory. We

```
kernel void KERNEL(..., global Tuple2_double_double* v__166){
...
Tuple2_double_double init = {0.0,0.0};
...
// map_seq
for (int i = 0; (i < 2); i = (1 + i)) {
  tmp[(i + (2 * v_i_157) +
      (4 * v_gl_id_156) + (16 * v_gl_id_155))] =
    cMult(v__161[(i + (2 * v_gl_id_156))],
      genB(v_i_157, i, v_gl_id_156, 2, 2, 4));
}
// end map_seq
// reduce_seq
for (int i = 0; (i < 2); i = (1 + i)) {
  init = cAdd(init, tmp[(i + (2 * v_i_157) +
                       (4 * v_gl_id_156) + (16 * v_gl_id_155))]);
}
// end reduce_seq
...
}
```

Listing 4.3: Generated OpenCL code for the dotproduct computed in the combine pass of a Stockham FFT with $p = 2$. Some variables and types have been renamed and expressions shortened for clarity.

therefore need a means to store the result of the reduction, which coincidentally computes the final values of the entire *stockhamPass* expression, in global memory. Hence, we introduce the following rules to assign memory regions:

REWRITE RULE 3. *Memory Rule*

$$
\begin{aligned}
&mapLocal(f) && \rightarrow mapLocal(toLocal(f)) \\
&mapLocal(f) && \rightarrow mapLocal(toGlobal(f)) \\
\\
&mapSeq(f) && \rightarrow mapSeq(toPrivate(f)) \\
&mapSeq(f) && \rightarrow mapSeq(toGlobal(f)) \\
&mapLocal(mapSeq(f) \circ g) && \rightarrow mapLocal(\\
& && \quad mapSeq(toLocal(f)) \circ g)
\end{aligned}
$$

These rules formalize the fact that the result of *mapLocal* can always be stored in either local or global memory because the creation of work-groups is guaranteed, since *mapLocal* has to be nested in a *mapWorkgroup*. Additionally, the rules express the fact, that the results of *mapSeq* can always be stored in either global or private memory, because all operations are performed by a single work-item. If local memory is defined, meaning *mapSeq* is inside a *mapLocal* and

with that implicitly inside a $mapWorkgroup$, then the results can be stored in local memory.

We are now able to express writing the intermediary results into private memory. Now, we are able to rewrite our Stockham pass expression such that we can assign our intermediary results to private memory. Unfortunately, the memory rules cannot be applied to the reduction. We can copy the results of a sequential reduction into global memory by writing

$$mapSeq(toGlobal(id)) \circ reduce(init, \oplus)$$

after specifying the following rule from [37]:

REWRITE RULE 4. *Identity*

$$f \rightarrow f \circ map(id) \mid map(id) \circ f$$

The expression that results from these rewrites is a valid expression to the Lift compiler. Before we generate code, we apply one more change to simplify the outcome, i. e. we reduce the amount of global work-item dimensions. In practice, this has made it easier for us to tune the generated OpenCL kernel. The rule is more sophisticated than the previous ones and uses types, i. e. the size that is coupled with an array type.

REWRITE RULE 5. *Dimensionality Change*
For $x : [[T]_n]_m$ *and* $y : [U]_n$

$$map(map(f) \circ zip(y))\ (x) \rightarrow$$
$$split(n) \circ map(f) \circ zip(pad(0,\ (m-1)n,\ wrap,\ y),\ join(x))$$

This rule expresses the fact that instead of zipping the same values and applying a function to the results in every row of x row by row, we can concatenate the same values, zip only once and then apply the function over all the values of x in a single mapping. This leads to the expression depicted in Listing 4.4. In order to be able to reason about possible rewrites that we might want to apply to our expressions, we need to understand how Lift generates OpenCL code from a low-level expression. Specifically, the so-called *view system* that generates code for the data layout primitives is of interest to us. Hence, we consider how Lift generates array accesses from composing the so-called *views* in its view system as described in [41].

LIFT VIEWS Memory allocations and read and write accesses specifically to global memory (and with that the GPU's RAM) are very slow and can cause significant losses of performance. In Lift, instead of allocating and writing new arrays in memory every time that a data layout primitive is encountered in an expression, an internal data structure called a *view* is created. It stores information about the layout changes to an underlying array. Views from different composed

```
1  dotproduct =
2      λ((xs, ys) ⇒ toGlobal(mapSeq(id)) ∘ reduceSeq(+, 0) ∘
3          toPrivate(mapSeq(∗)) ∘ zip(xs, ys)
4  ...
5
6  stockhamPass =
7      λ(vs ⇒ join ∘ transpose ∘ map(join ∘ transpose) ∘ split(L/p)∘
8          //Butterfly computations in only one dimension of WI.
9          mapGlobal_0(matrixMult) (
10             //Multiplied reordered Butterfly matrix.
11             zip(pad(0, (n − L)/p, wrap, reorB),
12                 //Simplyfied Stockham permutation expression.
13                 join ∘ map(transpose ∘ split(L/p)) ∘
14                     transpose ∘ join ∘ transpose ∘
15                         split(p) ∘ split(L/p) (vs))))
```

Listing 4.4: A completely lowered *stockhamPass* expression, with assigned memory regions.

primitives are composed as well. E. g. for $join \circ transpose\ (xs)$ a view for *transpose* which regards xs as its input view is created, while the view created for *join* regards its predecessor's view as its input view. Different primitives create different views accordingly, e. g. a *SplitView* for the *split* primitive. This process is called the *view construction*.

After a view has been constructed, the structure holds all the information required to generate an access. It can now be utilized to access an element at a given index in the current data layout, by calculating an array index expression that is used to access an underlying memory object, i. e. array. During the *view consumption*, an array index expression is calculated by consuming the provided information from top-to-bottom until for example a *MemoryView* which represents a specific object in memory, is encountered. While a MemoryView is a typical bottom most view, *array* constructor primitives are implemented via the view system as well.

ARRAY CONSTRUCTOR VIEW The *ArrayConstructorView* is created by the *array* primitive. Instead of holding information about a memory object, it knows the array constructors generator function. During OpenCl code generation, the generator function that is a user function defined in OpenCL-C code, is simply declared in the program and all accesses to the array are resolved into a function call with the appropriately calculated array indices from different views.

With this knowledge we can analyze how our lowered expression accesses memory. By fixing and keeping track of elements in an input array, we see which elements of a contiguous memory region are read

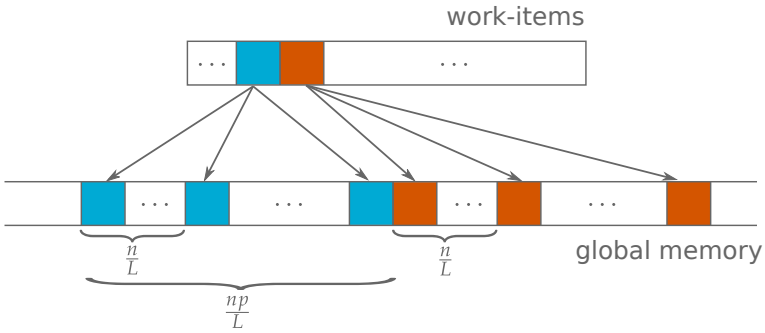

Figure 4.2: Memory access pattern in a Stockham pass.

and written by a work-item, depending on the parameters provided to the expression.

MEMORY ACCESS PATTERN As previously mentioned, accesses to the GPU's RAM are expensive operations. The GPU can optimize accesses for certain access patterns. Specifically, when threads in a warp access a contiguous memory region, the memory access requests can be coalesced into a single request by the hardware, as long as the amount of data does not exceed a certain threshold (fits into a global memory cache line). Current GPUs, for example with Nvidia's Kepler architecture, can coalesce accesses to up to 16 byte sized elements per thread.

Figure 4.2 depicts the memory access pattern during a Stockham pass. A field represents one complex value. For this thesis, we assume that a complex value is always in double precision and therefore has a size of 16 bytes (one double for the real and one for the imaginary part). The distances between accessed elements change depending on the previous passes and the current radix. But, more importantly, we see that adjacent work-items and therefore the work-items grouped in warps, do not access elements adjacent to each other. In general, this will lead to the hardware not being able to coalesce many memory accesses and as a result a lot of performance is lost. Therefore, we are going to look at a way to change the memory access pattern in a manner that allows for coalescing.

The memory access pattern in our *stockhamPass* results from directly deriving expressions from matrix-vector notations and specifically the mod-p sort permutation. However, as long as the values that are stored and read between passes remain the same, we can change the locations that values are written to. Every work-item combines p values in a pass that are all the elements that have to be read and written by it. By uniformly distributing these elements over the entire array, we create a memory pattern that changes only depending on the current radix and the input vector's size. We change *stockhamPass* as shown in Listing 4.5. This leads to the pattern depicted in 4.3 which

```
1 stockhamPass =
2     λ(vs ⇒ join ∘ map(join ∘ transpose) ∘ split(L/p)
3         ...
4             zip(...,
5                 transpose ∘ split(n/p)  (vs))))
```

Listing 4.5: A *stockhamPass* expression for an adjusted memory pattern.

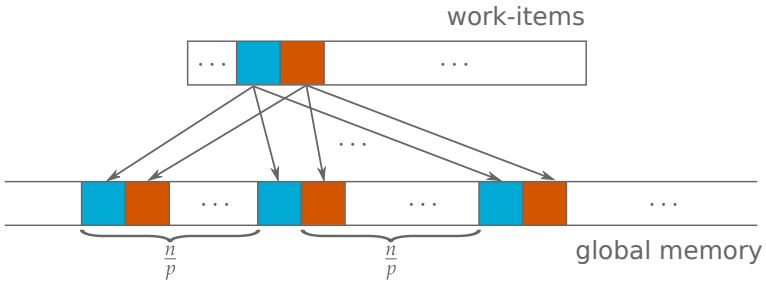

Figure 4.3: The adjusted Stockham pass memory pattern.

has been used by the authors of [14]. Now, groups of adjacent work-items access contiguous memory regions.

NOTE ON LOWERING COOLEY-TUKEY PASSES AND BIT-REVERSAL The same rewrite rules that are utilized to lower a Stockham pass can be utilized to lower the expression *cooleytukeyPass*, as well. The expression for the Bit-Reversal pass *bitrev* does not contain a user function and therefore needs to include an artificial $mapGlobal(id)$ that is prepended by composition, to be compilable. Having said that, no other rewrites have to be done for *bitrev* because only data layout primitives are used in the *map*s. To not repeat ourselves needlessly, we provide the lowered *cooleytukeyPass* expressions in Appendix A.1.

4.3 ARRAY CONSTRUCTOR BASED ON PRECOMPUTED VALUES

Depending on the underlying hardware, the *cos* and *sin* computations done in calls to *genB* are very expensive. Clearly, this has been acknowledged by the developers of clFFT as well. There, they provide constant memory arrays containing precomputed values that are declared in generated OpenCL programs. Precomputed values can be combined to calculate the element at a specific position in the array, e. g. a twiddle factor.

DEFINING CARRAY CONSTRUCTOR Based on this approach, we expand Lift by developing a new data layout primitive called *carray*. The primitive represents an array based on a view that holds a generator function, similarly to the *array* constructor primitive. The most significant difference is that additionally, the view has knowledge about

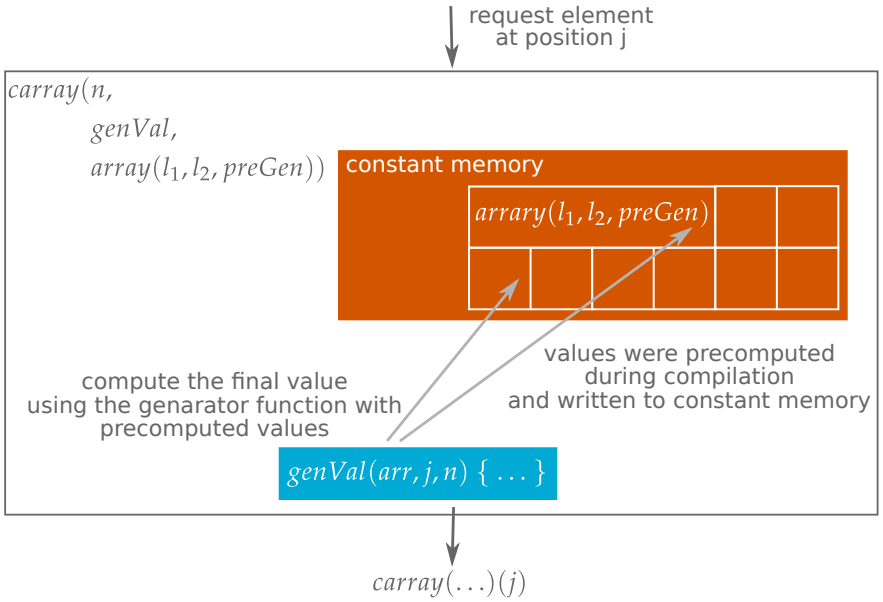

Figure 4.4: Schema for the OpenCL implementation of the *carray* primitive for a one-dimensional array. A two-dimensional precomputed array and a generator function called *genVal* are used.

a memory object with precomputed values. We start by providing a formal definition of *carray*:

DEFINITION 4.2. *Let m and $n_1, n_2, \ldots, n_m$ be positive integer sizes of different dimensions with the indices j_i, where $0 < j_k \leq n_k$, $\forall k : 0 < k \leq m$ and c be a $2m+1$-ary generator function that accepts an array constructor arr as an argument. The element generated by the m-dimensional carray constructor primitive at index $j_1, j_2, \ldots, j_m$, is then defined as follows:*

$$carray_m(n_1, n_2, \ldots, n_m,\ c,\ arr)(j_1, j_2, \ldots, j_m) \stackrel{def}{=} c(arr, n_1, \ldots, n_m, j_1, j_2, \ldots, j_m)$$

where c has access to the elements of arr. The type of carray is as follows:

$$\begin{aligned} carray_m : (&n_1 : Int,\ n_2 : Int, \ldots, n_m : Int, \\ &c : (arr : [[\ldots[T]_{l_o}]\ldots]_{l_2}]_{l_1}, \\ &\quad j_1 : Int, j_2 : Int, \ldots, j_m : Int, \\ &\quad n_1 : Int,\ n_2 : Int, \ldots, n_m : Int) \to T, \\ &arr : [[\ldots[T]_{l_o}]\ldots]_{l_2}]_{l_1}) \to [[\ldots[T]_{n_m}\ldots]_{n_2}]_{n_1} \end{aligned}$$

In comparison to the basic array constructor, the generator function takes an additional argument that refers to an array constructor. The elements in the array represented by that array constructor can be used by the generator function.

OPENCL CODE GENERATION FOR CARRAY CONSTRUCTOR Figure 4.4 depicts how *carray* works in OpenCL schematically. The generation of *OpenCL* code for *carray* accesses involves two additional

steps. First, all the values in *arr* are pre-computed, by executing *arr*'s generator function with all possible indices. Second, an OpenCL memory object for constant memory with the size of *arr* is created and assigned all the pre-computed values. As with the basic array constructor, the generator function *c* is declared in OpenCL-C code. Then, to generate an access, the declared function is called with the appropriate indices.

A CARRAY GENERATOR FUNCTION FOR TWIDDLE We use *carray* to compute twiddle factors. Because the twiddle factors depend on the exponentiation of the same base, it is possible to compute a new twiddle factor by multiplying others, e. g. ω^j and ω^k can be multiplied to form ω^{j+k}. In other words, we can express a given twiddle factor in terms of complex multiplication by finding two twiddle factors with exponents that add up to the exponent of the given twiddle factor. Assuming that the found values have been computed already, we can avoid *cos* and *sin* computations in favor of the complex multiplication.

The *Twiddle* array is an array representation of a matrix, where the element in row j and column k is a representation of ω^{jk}. The array has the type $[[Complex]_p]_{L/p}$ and hence holds L many elements. The biggest possible exponent for a twiddle factor in *Twiddle* is therefore $(L/p-1)\cdot(p-1)$. We want to provide values with which the entire range of possible exponents can be calculated by summation.

Note that every number can be expressed as a sum in terms of its digits that are multiplied with the base of an underlying numeral system (e. g. decimal system) to the power of the digits position in the expressed number. Formally, this can be written as

$$d_b = d_{n-1}\dots d_1 d_0 = \sum_{j=0}^{n-1} d_j b^j \tag{4.1}$$

for a number d_b represented in base b, with the sequence of n digits $d_{n-1}\dots d_1 d_0$. This is obvious for the decimal system, where for example $647 = 6\cdot 10^2 + 4\cdot 10^1 + 7\cdot 10^0$. It follows that we can express the range of possible exponent values based on all the digits that occur in the numeral system of base b multiplied with b^j for all $0 \leq j \leq k$, where k is such that $L \leq b^{k+1}$. In reference to equation (4.1), this means that we can express every needed number by providing the $d_j b^j$ for every possible $d_j \in \{0,\dots,b-1\}$. Note that $L \leq b^{k+1}$ always holds for $k = \log_b(L)$.

Hence, by precomputing the twiddle factors $\omega^{d_j b^j}$ for all possible digits and j as previously declared, we are able to compute every twiddle factor in *Twiddle* (and many more). This leads to the following definition for an array constructed with a generator function, for the precomputed twiddle factors of an FFT of size n.

DEFINITION 4.3. *Let n, digitpos, digit, bound and base be positive integer values. The generator function $gentw_n$ is then defined as follows:*

$$gentw_n(bound,\ base,\ digitpos,\ digit) \stackrel{def}{=} omega(digit \cdot base^{digitpos},\ n)$$

In this definition, *digitpos* stands for the position of a digit in the number that is being computed, i. e. it refers to the j in $d_j b^j$. The *digit* parameter stands for every possible digit in base *base*. Note that *bound* is not used, but provided as an argument to make $gentw_n$ a valid generator function for a two-dimensional array and stands for the largest value for *digitpos*. Now, using $gentw_n$ we define the two-dimensional array containing all the values that are to be precomputed.

DEFINITION 4.4. *Let L be the size of the DFTs computed in the current pass and p be the radix used in the current pass. Let base be the base that is used to represent the exponents. Let bound be a positive integer value such that $L \leq (base - 1)b^{bound}$. The array that provides values to compute all twiddle factors for the current FFT pass is then defined as follows:*

$$precomptw(n,\ bound,\ base) = array_2(bound,\ base,\ gentw_n)$$

By this definition, *precomptw* is an array that has the following form:

$$precomptw(n,\ k+1,\ b) = \begin{bmatrix} [\omega^{0 \cdot b^0},\ \omega^{1 \cdot b^0}, \ldots, \omega^{(b-1) \cdot b^0}], \\ [\omega^{0 \cdot b^1},\ \omega^{1 \cdot b^1}, \ldots, \omega^{(b-1) \cdot b^1}], \\ \vdots \\ [\omega^{0 \cdot b^k},\ \omega^{1 \cdot b^k}, \ldots, \omega^{(b-1) \cdot b^k}] \end{bmatrix} \tag{4.2}$$

The generator function that has access to this array needs to compute the digits of the exponent of the final twiddle factor in the provided base. We know that we can get the digit d_j at position j of the number d with:

$$\left\lfloor \frac{d}{b^j} \right\rfloor \bmod b = d_j \tag{4.3}$$

For the number $14 = 1110_2$, for example, we get the digit at position $j = 2$ in the binary system with

$$\left\lfloor \frac{14}{2^2} \right\rfloor \bmod 2 \ = \ 3 \bmod 2 \ = \ 1.$$

Therefore, by using this method to decompose the exponent that is passed to *omega* when accessing an element in *Twiddle* into its digits and loading the precomputed value for this digit from the row of $precomptw_n$ that is representing its position, we can combine the values by multiplication. This leads us to the following definition.

DEFINITION 4.5. *Let arr be a two-dimensional complex array and r, c, h, w be positive integer values. The carray generator function gentwfrarr is then defined as follows:*

$$gentwfrarr(arr,\ r,\ c,\ h,\ w) \stackrel{def}{=} \prod_{j=0}^{bound-1} arr\left(j, \left\lfloor \frac{d}{b^j} \right\rfloor \bmod base\right)$$

Finally, we have all the tools to utilize *carray* to implement a lambda that computes a Stockham pass with precomputed twiddle factors. For that, we use the array *TwiddleC* instead of *Twiddle*, which is provided with the following lemma:

LEMMA 4.1. *Let n be the size of the computed FFT. Let p be the radix of the current pass and L be the size of the DFTs that are computed in it. Let base be a positive integer base of a numeral system and bound be such that* $L \leq base^{bound}$. *For*

$$\begin{aligned} TwiddleC(p, L, bound, base) = carray_2(&L/p,\\ &p,\\ &gentwfrarr,\\ &precomptw(L, bound, base)) \end{aligned}$$

it always holds that

$$Twiddle(p,\ L)(j,\ k) = TwiddleC(p,\ L,\ bound, base)(j,\ k)$$

Proof. This follows directly from the construction of *precomptw* and *gentwfrarr* on the side of *TwiddleC* and the definition of *Twiddle* (Definition 3.20). □

While p and L change with passes, *bound* and *base* are fixed. In practice we set $base = 2^8$ and

$$bound = \left\lceil \frac{\lceil \log_2(L) \rceil}{\log_2(base)} \right\rceil.$$

This way, *base* and *bound* are always a power of 2 which enables fast implementations of *gentwfrarr* based on bit shifts and bit-wise And operations, avoiding expensive mod computations and exponentiations. Listing 4.6 shows an excerpt from a *gentwfrarr* implementation in OpenCL-C, with comments added for clarification. The digit that is being extracted from the exponent is at position 1. The bit shift is used to cut away the digits that follow the one at the desired position. Then, the bit-wise And is used to cut away all the digits in front of the one at the desired position.

In summary, we use *carray* to move the expensive computations of *cos* and *sin* into the compilation stage, so that only fast arithmetic operations like bit shift, multiplication and addition are performed to create the final result during runtime.

```
1  //current state of v is v = j*k
2
3  //v = ⌊v/256⌋
4  v = v >> 8;
5  //u = v  mod 256
6  u = v & 255;
7  //complex multiplication of previous value with value at u
8  tmp._0 = res._0 * arr[1*256+u]._0 - res._1 * arr[1*256+u]._1;
9  tmp._1 = res._1 * arr[1*256+u]._0 + res._0 * arr[1*256+u]._1;
10 res = tmp;
```

Listing 4.6: Excerpt from a fast implementation of *gentwfrarr*, as it is done in clFFT as well.

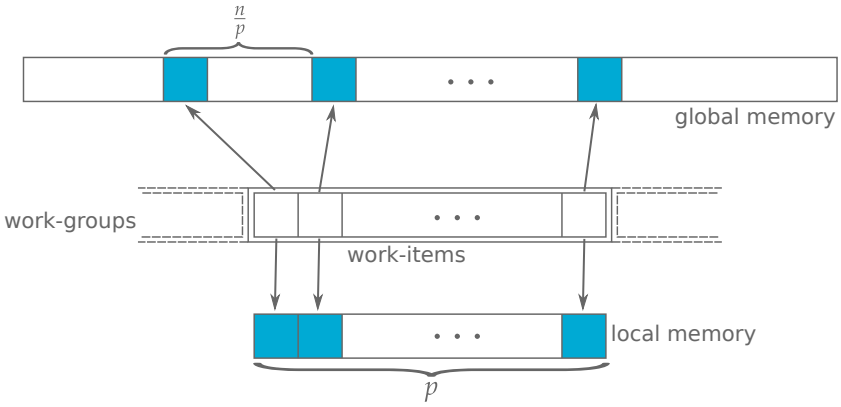

Figure 4.5: A work-group loading p elements that are to be combined in a Stockham pass with the adjusted memory pattern.

4.4 FOUR-STEP WITH LOCAL MEMORY

By using local memory, we can exploit the fast on-chip shared memory of a GPU. It is particularly useful when data that is loaded from global memory is used more than once or for exchanging intermediary results between work-items of a work-group. Although we have seen previously that with our current expressions, a work-item loads only one set of elements and no intermediary results need to be exchanged with other work-items, there are intermediary results created between FFT passes. These cannot be exchanged using the fast local memory when the input vector is too large to fit into it. In reference to *stockhamTwPass*, we saw that during a pass elements are combined by multiplying twiddle factors and then computing a DFT of the size of the current radix p over them. For a radix that is large enough, we could recursively apply an FFT method to the computation of this DFT. This would reduce the global passes over the input array, because larger radices mean that there are less factors in ρ needed to multiply to n. The resulting FFT of size p could be computed by a work-group and use local memory. Unfortunately, for

```
1  fourStep = λ(vs ⇒
2      //Compute FFTs of size n1.
3      join ∘ transpose ∘ map(FFT_n1) ∘ transpose ∘ split(n2) ∘
4      //Transpose the n2 × n1 matrix-split.
5      join ∘ transpose ∘ split(n1) ∘
6      //Element-wise twiddle.
7      join ∘ transpose ∘ map(map(∗)) ∘
8          map(λ((first, second) ⇒ zip(first, second)))
9            (zip(array_2(n1, n2, genDFTSlice),
10               transpose ∘ split(n1) ∘
11               //Compute the FFTs of size n2.
12               join ∘ transpose ∘ map(FFT_n2) ∘ transpose ∘ split(n1) (vs))
```

Listing 4.7: The Four-Step method in high-level primitives.

large p, work-groups would have to load many values that are not in contiguous memory regions as depicted in Figure 4.5. Because the values that are combined are spread over the input array for the original Stockham pattern as-well as the adjusted one, the work-items in a work-group cannot access adjacent elements. Therefore, the lack of coalescing can slow down the implementation significantly.

These problems can be avoided in the Four-Step method. We previously provided the expression $fourStep$ in Theorem 3.2. In the following, we will look at how lowered versions of $fourStep$ assign work to work-items and at the resulting memory access patterns. Listing 4.7 shows a single lambda for the $fourStep$, which can not be lowered in its entirety because it involves multiple passes, meaning that an operation on the entire array has to be finished before the next one can be performed. This means that a global barrier on accesses to the vector has to be created, i. e. multiple OpenCL kernels have to be generated. This is not yet supported by Lift. Hence, we decompose the expression into lambdas representing the different passes. The algorithm is based on the idea of splitting the one-dimensional input vector into an array representation of a matrix and then computing FFTs over its columns. By choosing the parameters n_1 and n_2 such that the columns are small enough, we are able to compute the FFTs in local memory. For particularly large inputs, we choose only one parameter to be small enough and apply the same method recursively to the FFTs that still do not fit into local memory.

Listing 4.8 shows a possible way to lower the expression. Every lambda is going to be compiled into a single kernel. To reduce the amount of passes that are needed, we fuse the twiddle multiplication either with the first column FFTs or with the transposition to form a single pass. Therefore, only three lambdas are needed. However, the memory access pattern that occurs, as depicted in the example in Figure 4.6, is not ideal for GPUs. For the first pass, the elements that form a vector over which an FFT is computed are not contiguous

$$
\begin{aligned}
&firstPass = \\
&\quad \lambda(vs \Rightarrow join \circ mapWorkgroup(FFT_{n_2}) \circ transpose \circ split(n_1)\ \ (vs)) \\
&\\
&transpWithTwPass = \\
&\quad \lambda(vs \Rightarrow join \circ transpose \circ mapGlobal_1(mapGlobal_0(*))\ \circ \\
&\qquad map((first, second) \Rightarrow zip(first, second)) \\
&\qquad\quad (zip(array_2(n_1, n_2, genDFTSlice),\ \ split(n_2)\ \ (vs)))) \\
&\\
&thirdPass = \lambda(vs \Rightarrow join \circ transpose \circ split(n_1)\ \circ \\
&\qquad\qquad join \circ mapWorkgroup(FFT_{n_1}) \circ split(n_1)
\end{aligned}
$$

Listing 4.8: Lowered Four-Step, broken up into multiple compilable lambdas. FFT_{n_i} stands for a full Fast Fourier Transform that is executed by a work-group and uses local memory.

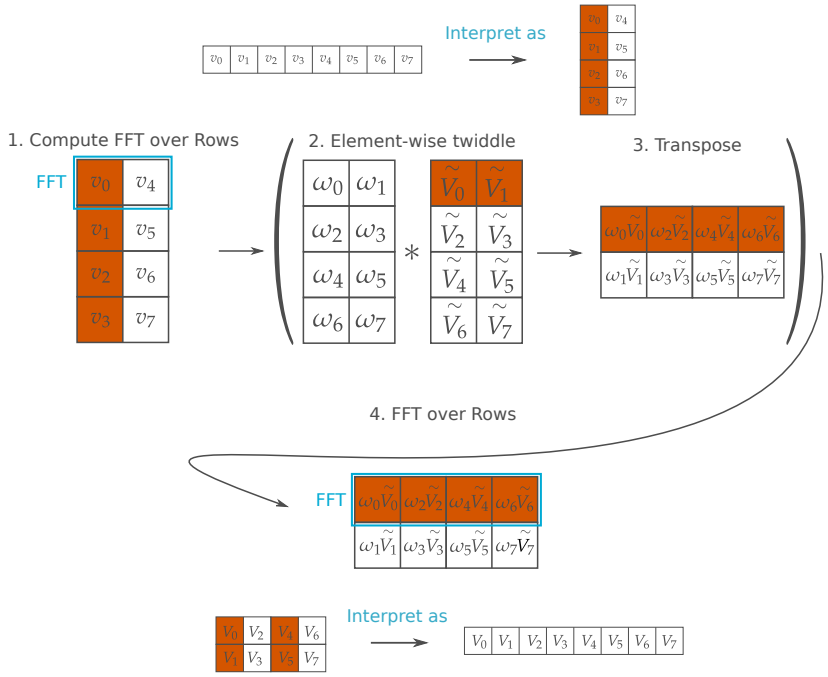

Figure 4.6: Example Four-Step for a split with $n_1 = 4$ and $n_2 = 2$, for eight elements. Highlighted are contiguous regions in global memory.

in memory and therefore cannot be read in a coalesced manner, by a work-group. This problem occurs because the initial transposition that is necessary to form the matrix-split of the vector is only a view in terms of Lift and therefore does not actively apply changes to the underlying memory. Note that FFTs over the rows of the array and not the columns are shown, contrary to the definition of the Four-Step FFT in matrix terms. The reason for this is that the Figure depicts the array that is created and transformed with Lift expressions. For mapping a function over the columns of the array representation of a matrix, the array has to be transposed and then the function can be mapped over the rows instead. The output of the first pass can be adapted to enable coalescing in the following two passes. These are the combination of transpose and twiddle factor multiplications, and the second pass of FFT computations over the transposed matrix. By pre- and appending additional passes that transpose the array, such that the location of the elements change in memory, coalescing in the first pass can be achieved. Effectively, we go from three passes to five:

1. transpose the matrix-split $v_{n_2 \times n_1}$ in memory
2. DFT_{n_2} over the columns of $v_{n_2 \times n_1}$, which are now contiguous memory regions
3. multiply twiddle factors and transpose in memory
4. DFT_{n_1} over the columns of $v_{n_1 \times n_2}$
5. transpose the matrix-split $v_{n_2 \times n_1}$ in memory to bring it into correct final order

This is a method that is used by clFFT. It introduces a trade-off between the runtime of the transposition pass and the gained performance from coalesced memory accesses.

We see that we have to compute three transpositions over the entire input vector. It follows that it is important to implement the transposition passes efficiently. A well-known problem with transposing matrices on GPUs is that a naive implementation in which every adjacent global work-items copy adjacent elements of a row into their columns prevents the hardware from coalescing the write accesses [7, 35]. A common approach therefore is to implement transpositions using so-called *tiling*. A two-dimensional array is split into tiles that are loaded into local memory by the work-items of a work-group. In the fast local memory that does not depend on coalesced memory accesses for high performance, the uncoalesced writes and reads are performed in order to coalesce the memory accesses to global memory. Using high-level primitives, tiling can be expressed in Lift [34].

$$
\begin{aligned}
&tile(theight,\ twidth,\ M) = \\
&\quad map(map(transpose) \circ split(twidth) \circ transpose) \circ \\
&\qquad split(theight)\ (M)
\end{aligned}
$$

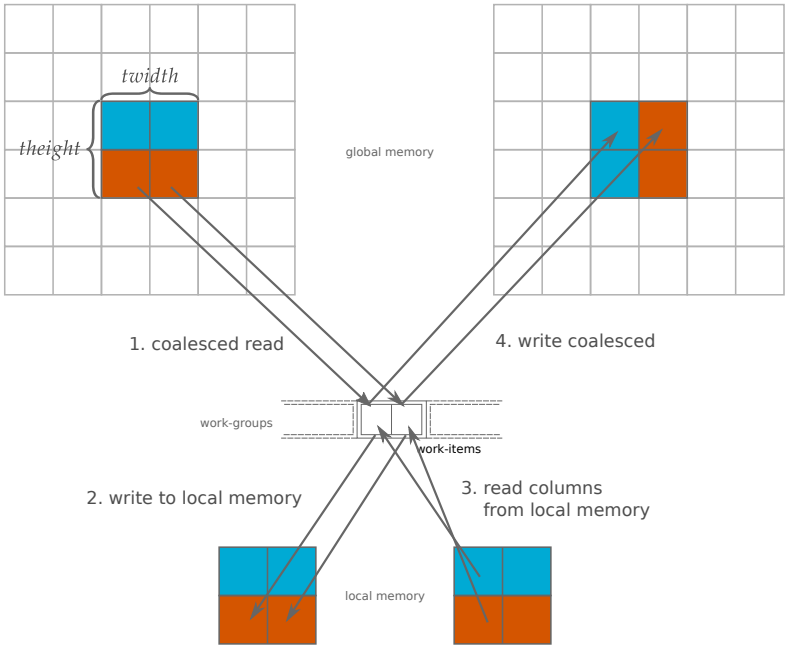

Figure 4.7: Transposing a matrix using local memory.

creates tiles of size $theight \times twidth$ over the two-dimensional array M. These tiles are assigned to work-groups with *mapWorkgroup* and then loaded into local memory. Because the third pass is used to multiply twiddle factors as well as transpose an array, the expression for the third pass is more complicated than a pure transpose pass with tiles. Figure 4.7 depicts how work-groups access the memory regions to read and write tiles. For the third pass, when reading from global memory, twiddle factors are multiplied and the results stored in local memory. The work-groups shown are organized in only one dimension. In the final expression, the matrix of tiles and the tiles themselves are linearized to form one-dimensional arrays so that only work-groups and work-items in one dimension are assigned work. The final lowered expression is shown in Listing 4.9. In the expression, *untile* is a placeholder for the inverse of *tile*. The separate steps that are performed are explained in the comments.

BANK CONFLICTS ACCESSING COMPLEX ELEMENTS Shared memory in GPUs is organized in so-called *banks*. Accesses by work-items in a warp that access shared memory simultaneously, can be parallelized under certain conditions, increasing the memory throughput and possibly improving overall program performance. To fully parallelize the shared memory accesses of a warp containing 32 work-items, no two work items are allowed to access the same bank and in case two or more work-items access the same bank, these accesses have to be executed sequentially. The attempt of n work-items accessing the same bank simultaneously is called a *(n-way) bank conflict*. Modern Nvidia GPUs for example use 32 banks which are of 4 bytes

```
1  transpWithTwPass =
2      //create the two-dimensional matrix of tiles again to then
           untile it
3      λ(vs ⇒ join ∘ untile ∘ split(n1/theight) ∘
4          //assign tiles to work-groups in one dimension
5          mapWorkgroup0(
6              //write the transposed tile to its new location in
                   global memory
7              split(theight) ∘ mapLocal0(toGlobal(complexId)) ∘
8                  //create two-dimensional tile from 1d tile,
                       transpose and linearize again
9                  join ∘ transpose ∘ split(twidth) ∘
10                     //linearize two-dimensional tile and mutliply
                           and load into local memory
11                     mapLocal0(toLocal(id ∘ ∗)) ∘ join) ∘
12                 //create and transpose the positions of tiles,
                       then linearize the matrix of tiles
13                 join ∘ transpose ∘ tile(theight, twidth) ∘
14                 //assign twiddles to matrix elements
15                 map(λ(twvals ⇒ zip(twvals.1, twvals.2)))
16                     (zip(array2(n1, n2, genDFTSlice), split(n2) (vs)))
```

Listing 4.9: The combined pass for twiddle factor multiplication with transposition in the Four-Step method.

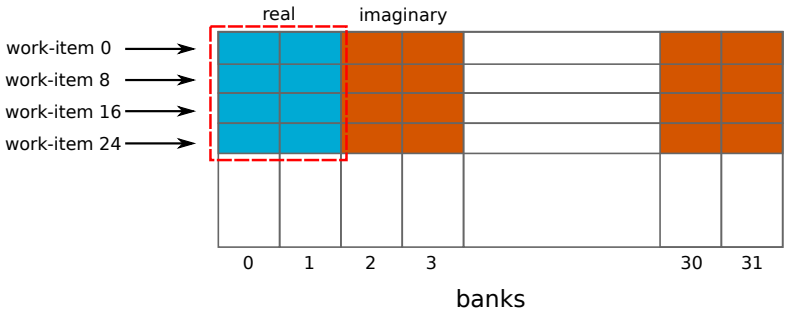

Figure 4.8: A bank conflict that occurs when accessing real parts of a complex values in double precision.

in size. A warp contains 32 work-items. These sizes are very useful in combination with coalesced global memory accesses. When adjacent work-items of a warp access adjacent elements with a size of 4 bytes in global memory to copy them into local memory, the hardware can coalesce the accesses (provided that the accesses are properly aligned) and writing to local memory automatically avoids bank conflicts. Unfortunately, this is not the case for elements with a size of 8 or even 16 bytes. Figure 4.8 depicts a 4-way bank conflict that occurs when loading real parts of complex values. Four different threads try to load a real value from the same bank. This occurs throught the entire warp.

This case occurs in the Four-Step when accessing double precision complex values in local memory, during the passes that compute FFTs. We can reduce the amount of bank conflicts from a 4-way to a 2-way bank conflict by loading the real and imaginary parts of the values separately. The risk with this method is, that the global memory accesses might not be coalesced, because the real and imaginary parts are stored interleaved in global memory. Instead of writing $mapLocal(toLocal(id))$ to load elements into local memory, we use

$$
\begin{aligned}
&mapLocal(\\
&\quad \lambda(c \Rightarrow \\
&\qquad tuple(toLocal(id_{double})(c.1),\ toLocal(id_{double}(c.2)))))
\end{aligned}
$$

where $tuple$ is a data-layout primitive that simply creates a tuple based on two values.

5 EVALUATION

In this chapter, we evaluate the code that is generated from the low-level lambdas that we derived from high-level expressions. On an Nvidia GPU, we compare the different FFT methods against each other. At first, we look at the difference between FFT computations with the initial Stockham pass derivation and the FFT computations with the adjusted memory access pattern. Then, we compare the adjusted Stockham FFT to the Cooley-Tukey FFT for three sets of same radices. We follow this up by evaluating the impact of using precomputed twiddle factors. Afterwards, we show the effects of using the Four-Step method. We then conclude the evaluation by comparing the best results of the different FFTs to clFFT and cuFFT. Furthermore, using an AMD GPU, we compare the best results measured with it to clFFT as well.

5.1 HARDWARE SETUP

We use two different GPUs to conduct our experiments. One is an Nvidia Kepler K20c Graphics Card with Nvidia's driver in version 384.145. The GPU's maximum clock frequency is 705 MHz and it contains 13 SMXs (Streaming Multiprocessors). 2496 streaming processors are equally distributed over the SMX (192 cores per SMX). The on-chip local memory has a size of 48 KiB. There are 4.631 GiB of off-chip global memory available and a global memory cache line holds 128 bytes.

The other GPU is a Radeon RX Vega 64 Graphics Card with AMD's driver version 2574.0 (HSA1.1,LC). The GPU's maximum clock frequency is 1630 MHz and it contains 64 CUs (Compute Units). 4096 stream processors are equally distributed over the CUs (64 stream processors per CU). The on-chip local memory has a size of 64 KiB. There are 7.892 GiB of off-chip global memory available and a global memory cache line holds 64 bytes.

All experiments are conducted using Ubuntu 16.04.5 LTS with the kernel version 4.15.0-30-generic x86_64. The used OpenCL platforms are NVIDIA CUDA version 1.2 CUDA 9.0.424 for the Nvidia GPU and AMD Accelerated Processing version 2.1 AMD-APP.internal (2574.0) for the AMD GPU.

5.2 EXPERIMENTAL SETUP

For every experiment, we use double precision data types with a size of 8 bytes to represent the real or imaginary parts of values. The input vector that is used has a size of $2^{24} = 16777216$ bytes and consists of randomly generated values. The amount of arithmetic operations in the algorithms does not depend on the values of the input vector. Hence, there is no risk of highly different performance results for different values. With this, we have an input vector with a size that is a power of 2 and 4. Therefore, FFTs that are split into passes purely consisting of radices 2 or 4 are possible. These radices are considered optimal for saving arithmetic operations [5]. We generate a new kernel for every configuration of p, L in an expression that contains these parameters in data layout primitives because parametrization of those primitives is not supported in Lift yet. For example, we generate an OpenCL kernel for a Stockham pass with radix $p = 4$ and $L = 4$ if the generated kernel is supposed to compute the second pass of a Stockham FFT where the first two radices are 4 and generate a new OpenCL kernel for a Stockham pass $p = 2$ and $L = 2$ if the generated kernel is supposed to compute the second pass of a Stockham FFT where the first two radices are 2.

Then, we use ATF [33], a generic auto-tuning framework, to tune the amount of work-items and work-groups that execute the kernel in order to find a fast configuration. Every kernel is tuned for either one hour or until the best runtime has not improved for 1000 tested configurations. The found configurations are then used to execute the kernels that compose an FFT in order to compute the FFT over the input. The FFT computations are repeated 20 times and the runtimes are measured using the OpenCL API and then averaged.

5.3 ADJUSTED STOCKHAM PASS ACCESS PATTERN

We explained that programming in a way in which the hardware can coalesce memory accesses is often important to achieve high-performance on GPUs. Therefore, we changed the pattern in which a Stockham pass loads and stores values. We generate the kernels for a Stockham FFT for which all radices are set to 2 in order to show the impact that this change had on the Kepler GPU. 24 passes of the Stockham method are needed to compute an FFT over our input vector. Figure 5.1 shows the runtimes measured, separated by kernel. The runtimes of both version are grouped together by the number of the passes. We see that the performance of the kernels with the adjusted access pattern are relatively constant. This is what we would expect, because with the same radix in every pass, the amount of arithmetic operations stays the same as well. However, the kernels using the original access pattern vary in performance, with the run-

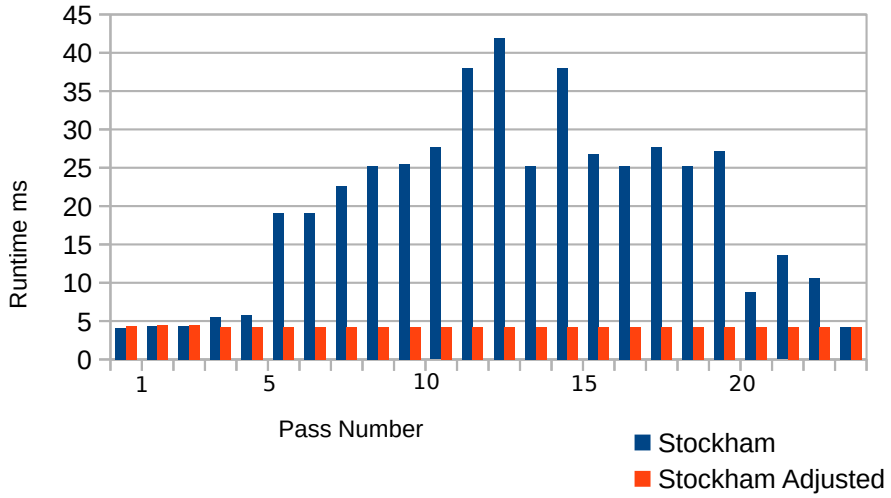

Figure 5.1: Kepler K20c: Comparison original vs. adjusted access pattern in a Stockham pass for $p = 2$.

times growing towards the middle of the passes. Because of this, we will concentrate on the Stockham pass with the adjusted pattern in the following experiments.

5.4 COMPARISON STOCKHAM AND COOLEY-TUKEY

Throughout this thesis we used the Stockham method of computing FFTs as our main example, but developed expressions for Cooley-Tukey passes, as well. We compare the performance of both methods for so-called *same-radix* FFTs where the radix of every pass is the same. The radices we use are 2, 4 and 8. We are not able to run our experiments with larger radices because the kernels that are generated attempt to allocate too much private memory, i. e. not enough registers are available in the GPU. Using same-radix FFTs is useful because if the runtimes of different passes within a same-radix FFT are faster than others we are able to use these insights to create a *mixed-radix* FFT. Figure 5.2 shows the runtimes measured for the different FFTs according to the radices. We see that the performance for Stockham FFTs is slightly better for the best measured results. However, the Cooley-Tukey method is useful for FFTs with the radices 4 and 8. Our conjecture is that the memory access patterns that result from utilizing the Bit-Reversal step allow for a better exploitation of the cache hierarchy and therefore increase the performance. Note that the increase in performance is large enough to hide the time it takes to compute the Bit-Reversal pass.

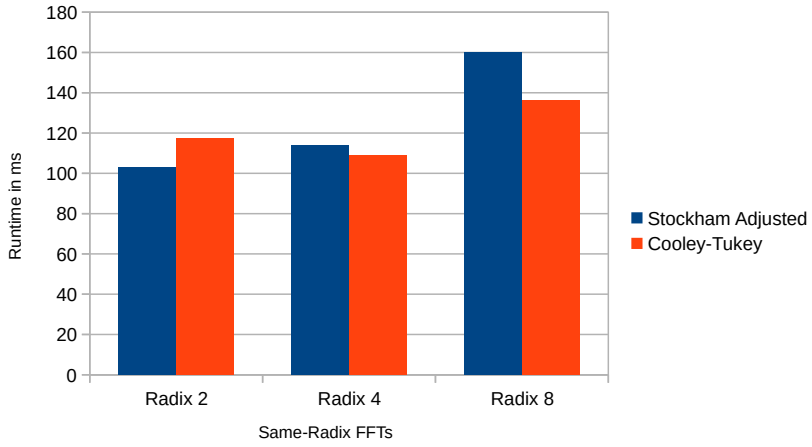

Figure 5.2: Kepler K20c: Comparison of different generated same-radix Stockham and Cooley-Tukey FFTs.

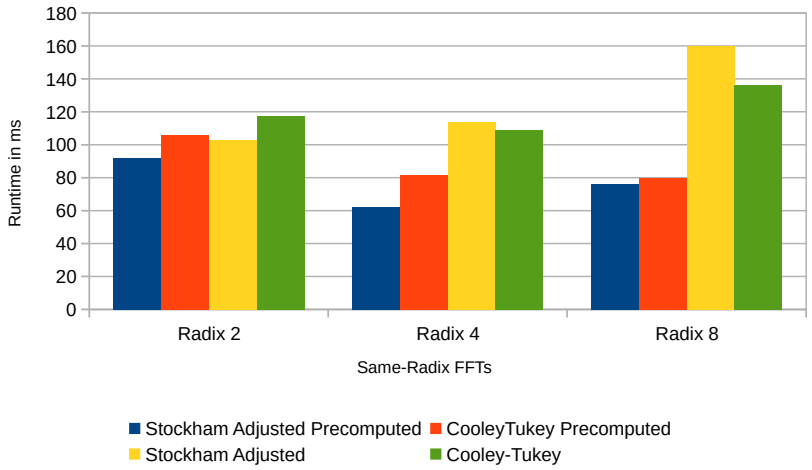

Figure 5.3: Kepler K20c: Performance of FFTs with precomputed values in comparison to on-line *cos* and *sin* computations.

5.5 PRECOMPUTING TWIDDLE FACTORS

To avoid expensive twiddle factor computations, we use the *carray* primitive to precompute values that can be combined to form all the needed twiddle factors. Similar to our previous experiment, we compute the same-radix FFTs for radices 2, 4 and 8 with the Stockham and Cooley-Tukey method. This time, we use the kernels generated from the expressions using *carray* to express the arrays holding twiddle factors. Figure 5.3 shows the results in comparison to the FFT implementations that do not use precomputed values. We notice a significant performance gain. Now the Stockham FFT with radix-4 is the fastest. With bigger radices a single work-item has to do more work. By reducing the time that is spent on computing the twiddle factors a single work-item is able to combine more values in a shorter amount of time.

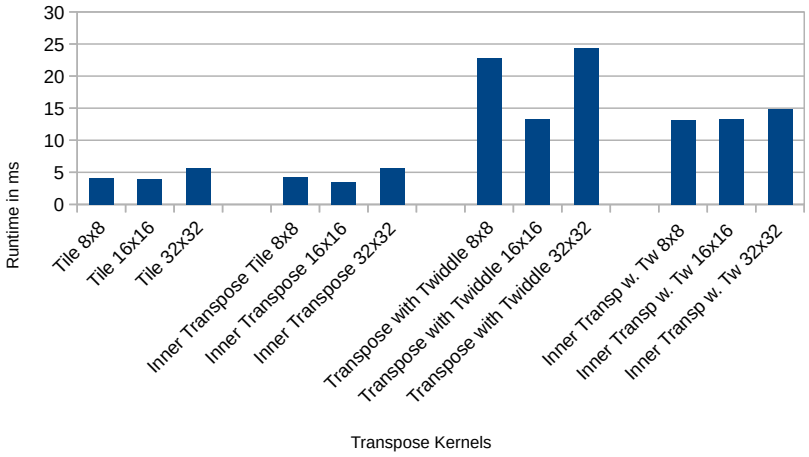

Figure 5.4: Kepler K20c: Performance of transpose passes for different tile sizes.

5.6 FOUR-STEP FFTS

Next, we test our implementation of the Four-Step method. The local memory of the Kepler GPU is 48 KiB. Hence, $48 \cdot 1024/16 = 3072$ complex double precision values can be loaded by a single work-group. Splitting the input vector into a two-dimensional matrix according to the Four-Step method with matrix rows of length 3072 for 16777216 elements results in a matrix of height 5461.3. Even if this was an integer number, the height would be too large to compute FFTs over the columns of the matrix. Hence, we need to apply the Four-Step method recursively to split the dimension that is too large even further. For our experiments we create a $256 \times 256 \times 256$ matrix, keeping the size of computed FFTs the same for every pass. We transposed the matrices in memory to achieve coalesced memory accesses during the first pass. Figure 5.4 shows the performance of transpose passes for different tile sizes, with twiddle factor multiplications and without. Passes that are named *Tile* transform the matrices in the first and second dimension of the three-dimensional matrix. Passes that are named *Inner Transpose* transform the matrices in the second and third dimension. We see that a tile size of 16×16 leads to the best performance in every case. We use this insight to measure complete Four-Step FFTs. The results for different radices are depicted in Figure 5.5. It is interesting to note that the best performance results from computing two passes with radix 8, which is theoretically worse, because more arithmetic operations need to be performed when multiplying a DFT_8 in comparison to two DFT_4. However, the effect on the memory access pattern is more important than the additional computational work.

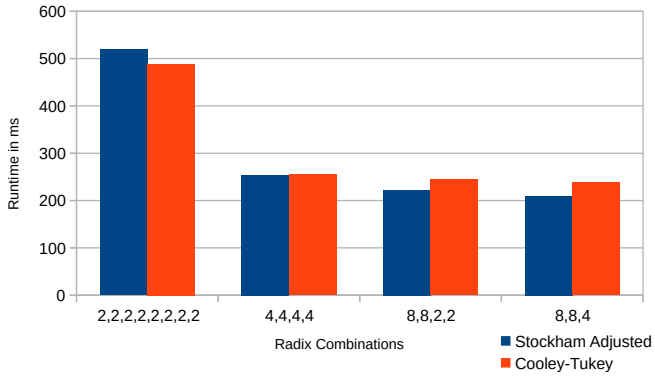

Figure 5.5: Kepler K20c: Performance of Four-Step for different radices.

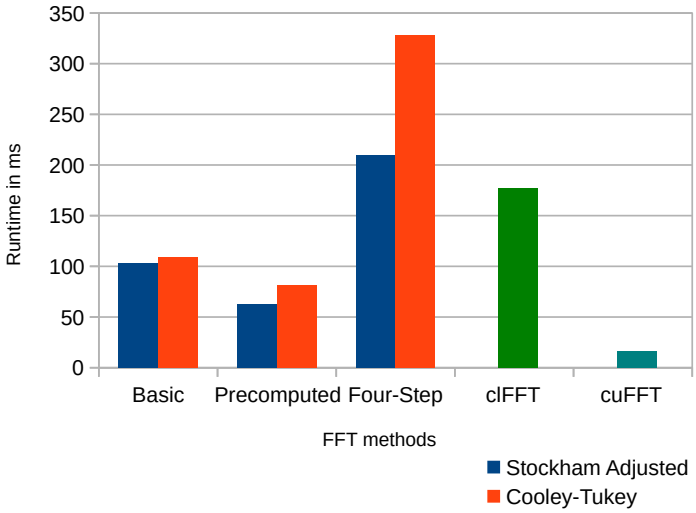

Figure 5.6: Kepler K20c: Best results from generated FFTs vs clFFT vs cuFFT.

5.7 COMPARISON WITH CLFFT AND CUFFT

Finally, we compare the best results of the different generated FFTs to professional FFT libraries. The cuFFT library is written in CUDA and therefore works with Nvidia GPUs only. Although the clFFT library is written in OpenCL, its FFT routines are optimized for AMD GPUs. To make a fair comparison, we measure the clFFT performance on the Kepler as well as the Radeon GPU. clFFT generates OpenCL kernels. The correctness of these kernels depends on specific amounts of work-groups and work-items, which prevents us from tuning them similarly to our own kernels. Figures 5.6 and 5.7 show the results for the two hardware architectures. On the Kepler K20c, cuFFT is roughly 4 times faster than our fastest generated FFT, but we beat the clFFT library by almost a factor of 3. On the Radeon RX Vega, the clFFT implementation is still slightly faster than our fastest generated FFT. Note that clFFT uses the Four-Step method with Stockham FFTs over the matrix rows and columns and that our best results come from a Stockham FFT. The clFFT implementation uses fast hand-tuned im-

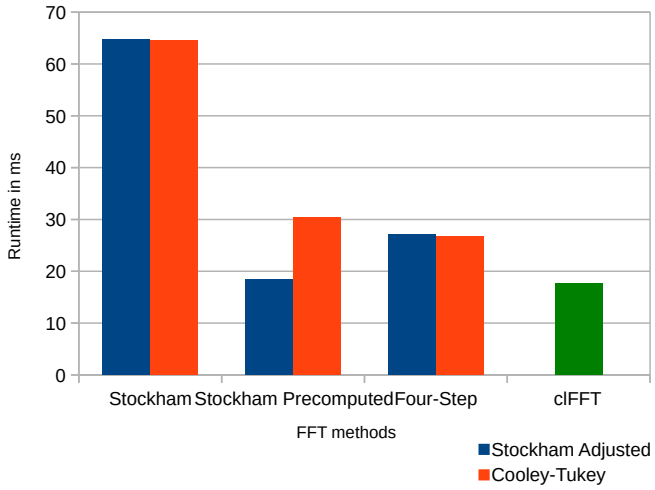

Figure 5.7: Radeon RX Vega 64: Best results from generated FFTs vs clFFT.

plementations of Butterfly computations for the radices 2, 4 and 8 that utilize less registers and perform less arithmetic operations. Furthermore, FFTs in local memory are computed purely in place, while our generated code creates input and output buffers, heavily limiting the range that parameters can take (e. g. the size of the dimensions when splitting a vector into a matrix). Nonetheless, clFFT is only 1.05 times faster than our generated code.

6 CONCLUSION

In this thesis, we looked at three different methods of expressing Fast Fourier Transforms: the Stockham, Cooley-Tukey and Four-Step method. We used formal definitions of these Fast Fourier Transforms that were given in matrix-vector notations in order to define parameterized high-level expressions to represent them. Then, we decomposed these expressions into high-level data parallel building blocks based on Lift's high-level primitives. Many of these building blocks could be reused for different FFT methods which suggests that even more FFT methods can be composed based on them. While the matrix values that are used to compute FFTs over an input vector are usually computed based on the indices of affected vector elements, we were able to assign these matrix values to the input vector by transforming the array representations of the matrices and the vector utilizing only high-level primitives. This lead to correct algorithmic descriptions for computing the different FFTs. The fact that we were able to express the building blocks without introducing new primitives is testimony of the power that the Lift system provides.

In the subsequent chapter, we analyzed how the work described by the high-level building blocks can be assigned to a Graphics Processing Unit and introduced optimizations to our original expressions. We used semantics preserving rewrite rules to rewrite high-level expressions into low-level expressions containing low-level primitives that are closely related to OpenCL. With these primitives, we were able to assign work to work-items using global and private memory regions in a way that every element in global memory has to be loaded only once by a single work-item. With the concept of coalescing in mind, we changed the memory access pattern of the Stockham pass. Thanks to our fine grained decompositions into high-level primitives, we were able to do this by simply changing parameters and omitting individual primitives. We then looked at how to avoid expensive *cos* and *sin* computations during runtime. As a solution, we introduced the data layout primitive *carray* to precompute the elements of an array during Lift's compilation stage. To simulate an array access, these precomputed elements could then be combined into an element at an accessed position by loading from constant memory. With this, we were able to avoid *cos* and *sin* computations during runtime entirely, substituting them for fast operations such as bit shift. In an attempt to decrease the amount of passes over global memory data, we described how to use the Four-Step method to split the input vector into parts small enough to fit into local memory. We

explained how to achieve coalesced memory accesses when reading into and writing from local memory and how to express the use of separate local memory buffers for real and imaginary data for the possibility to avoid bank conflicts.

Finally, we evaluated the OpenCL kernels that are generated using the Lift system. We saw that our optimization of changing the access pattern increased performance significantly. Precomputing twiddle factors using *carray* lead to further improvements of performance. With these optimizations in combination with the theoretically optimal radix 4, we generated code for which we measured runtimes that are almost 3 times faster than the runtimes measured for clFFT on a Kepler K20c GPU and nearly as fast as the runtimes measured on a Radeon RX Vega GPU.

Hence, we were able to provide a flexible implementation of Fast Fourier Transforms based on small data parallel building blocks from which we can generate code with a performance that is competitive to clFFT.

A

APPENDIX

A.1 CORRECTNESS PROOFS OF FFT EXPRESSIONS

A.1.1 *Modpsort is Well-Defined*

LEMMA A.1. *The expression modpsort (Definition 2.2) is well defined. Let vs be an array representation of vector v of size n with elements v_l, where $0 \leq l < n$ and $n = pm$. It holds that*

$$modpsort(p, vs) = [(\Pi^T_{p,L})_0, (\Pi^T_{p,L})_1, \ldots, (\Pi^T_{p,L})_{n-1}]$$

expresses the array representation of v.

Proof.

$$\begin{aligned}
&modpsort(p, vs)\\
=&join \circ transpose \circ split(p)\ (vs)\\
=&join \circ transpose \left(\begin{bmatrix} [v_0, v_1, \ldots, v_{p-1}], \\ [v_p, v_{p+1}, \ldots, v_{2p-1}], \\ \vdots \\ [v_{n-p}, v_{n-p+1}, \ldots, v_{n-1}] \end{bmatrix} \right)\\
=&join \left(\begin{bmatrix} [v_0, v_p, \ldots, v_{n-p}], \\ [v_1, v_{p+1}, \ldots, v_{n-p+1}], \\ \vdots \\ [v_{p-1}, v_{2p-1}, \ldots, v_{n-1}] \end{bmatrix} \right)\\
=&[v_0, v_p, \ldots, v_{n-p}, v_1, v_{p+1}, \ldots, v_{(n-p)+1}, v_{p-1}, v_{2p-1}, \ldots, v_{n-1}]\\
=&vs'
\end{aligned}$$

Where vs' is a representation of a vector v'. We see that

$$[v_j, v_{p+j}, \ldots, v_{(n/p-1)p+j}] = [v_j, v_{p+j}, \ldots, v_{(n-p)+j}]$$

represents the vector

$$v(j : p : n-1), \quad \forall j : 0 \leq j < p$$

With that, it follows that vs' is an array representation of:

$$\begin{pmatrix} v(0 : p : n-1) \\ v(1 : p : n-1) \\ \vdots \\ v(p-1 : p : n-1) \end{pmatrix} = \Pi^T_{p,L} v$$

□

A.1.2 *A Statement About genB*

LEMMA A.2. *With $B_{p,L}$ as defined, we say that j represents a row of block matrices, k represents a column of block matrices and l represents the index of an element on the diagonal of a block matrix, where $0 \leq j,k < p$ and $0 \leq l < L$. For the expression genB (Definition 3.13) it holds that:*

$$genB(j,k,l,p,p,L/p) \text{ is a representation of } \left(B_{p,L}\right)_{(j\frac{L}{p}+l),(k\frac{L}{p}+l)}.$$

Proof.

$$\begin{aligned}
&\left(B_{p,L}\right)_{(j\frac{L}{p}+l),(k\frac{L}{p}+l)} \\
&= \left(DFT_p \otimes I_{\frac{L}{p}}\ diag\left(\Omega^0_{p,\frac{L}{p}}, \Omega^1_{p,\frac{L}{p}}, \ldots, \Omega^{p-1}{}_{p,\frac{L}{p}}\right)\right)_{(j\frac{L}{p}+l)(k\frac{L}{p}+l)} \\
&= \left(\omega_p^{jk} \cdot I_{L/p} \cdot \Omega^k_{p,L/p}\right)_{l,l} \\
&= \left(\omega_p^{jk} \cdot diag\left(\omega_L^0, \omega_L^1, \ldots, \omega_L^{L/p-1}\right)^k\right)_{l,l} \\
&= \omega_p^{jk} \cdot \omega_L^{kl} \\
&= \exp((-2\pi i/p) \cdot jk) \cdot \exp((-2\pi i/L) \cdot kl) \\
&= \exp((-2\pi i/p) \cdot jk + (-2\pi i/L) \cdot kl) \\
&= \exp((-2\pi i \cdot \frac{L}{p}/L) \cdot jk + (-2\pi i/L) \cdot kl) \\
&= \exp(-2\pi i(\frac{L}{p} \cdot jk + kl)/L) \\
&= \exp(-2\pi i(j\frac{L}{p} + l)k/L) \\
&= \omega_L^{(j\cdot(L/p)+l)k}
\end{aligned}$$

By definition of *omega* (Definition 2.10) this is what is represented with:

$$\begin{aligned}
&omega\left(\left(j \cdot \frac{L}{p} + l\right) \cdot k, L\right) \\
=&genB\left(j,k,l,p,p,\frac{L}{p}\right)
\end{aligned}$$

□

MULTB IS WELL-DEFINED

LEMMA A.3. *Let vs be an array representation of the vector v with L elements v_j, where $0 \leq j < L$. For the expression multB (Definition 3.18) it holds that*

$$multB(p, L, vs) = [(B_{p,L}v)_0, (B_{p,L}v)_1, \ldots, (B_{p,L}v)_{L-1}]$$

expresses an array representation of $B_{p,L}v$.

Proof.

$$\begin{aligned}
&multB(p, L, vs)\\
&= prepBmult^{-1} \circ map(matrixMult)\\
&\quad \circ zip(reorB(p, L)) \circ prepBmult(p)\ (vs)\\
&= prepBmult^{-1} \circ map(matrixMult)\\
&\quad \circ zip(reorB(p, L)) \circ transpose \circ split(L/p)\ (vs)\\
&= prepBmult^{-1} \circ map(matrixMult)\\
&\quad \circ zip(reorB(p, L)) \left(\begin{bmatrix} [v_0, v_{L/p}, \ldots, v_{L-L/p}], \\ [v_1, v_{L/p+1}, \ldots, v_{L-L/p+1}], \\ \vdots \\ [v_{L/p-1}, v_{2L/p-1}, \ldots, v_{L-1}] \end{bmatrix} \right)\\
&= prepBmult^{-1} \circ map(matrixMult)\\
&\quad \circ zip(transpose \circ map(transpose)\ (B(p, L)))\\
&\qquad \left(\begin{bmatrix} [v_0, v_{L/p}, \ldots, v_{L-L/p}], \\ [v_1, v_{L/p+1}, \ldots, v_{L-L/p+1}], \\ \vdots \\ [v_{L/p-1}, v_{2L/p-1}, \ldots, v_{L-1}] \end{bmatrix} \right)
\end{aligned}$$

From the way that $B_{p,L}$ (Definition 2.3) is constructed and Lemma A.2 it follows that

$$(transpose \circ map(transpose)(B(p, L)))(j, k, r) = omega((kL/p + j) \cdot r, L)$$

where $0 \leq j, k < p$ and $0 \leq l < L/p$. And $omega((kL/p + j) \cdot r, L)$ represents $\omega_L^{(kL/p+j)\cdot r}$.

Hence, we continue our chain of equations with

$$
\begin{aligned}
&= prepBmult^{-1} \circ map(matrixMult) \circ zip(\\
&\begin{bmatrix}
\begin{bmatrix} [\omega_L^{(0\cdot(L/p)+0)\cdot 0}, \ldots, \omega_L^{(0\cdot(L/p)+0)\cdot(p-1)}], \\ \vdots \\ [\omega_L^{((p-1)\cdot(L/p)+0)\cdot 0}, \ldots, \omega_L^{((p-1)\cdot(L/p)+0)\cdot(p-1)}] \end{bmatrix}, \\
\vdots \\
\begin{bmatrix} [\omega_L^{(0\cdot(L/p)+L/p-1)\cdot 0}, \ldots, \omega_L^{(0\cdot(L/p)+L/p-1)\cdot(p-1)}], \\ \vdots \\ [\omega_L^{((p-1)\cdot(L/p)+L/p-1)\cdot 0}, \ldots, \omega_L^{((p-1)\cdot(L/p)+L/p-1)\cdot(p-1)}] \end{bmatrix}
\end{bmatrix}, \\
&\begin{bmatrix} [v_0, v_{L/p}, \ldots, v_{L-L/p}], \\ [v_1, v_{L/p+1}, \ldots, v_{L-L/p+1}], \\ \vdots \\ [v_{L/p-1}, v_{2L/p-1}, \ldots, v_{L-1}] \end{bmatrix})
\end{aligned}
$$

and see that zipping the matrices leads to the following tuple in row m with $0 \leq m < L/p$.

$$
\left\{ \begin{bmatrix} [\omega_L^{(0\cdot(L/p)+m)\cdot 0}, \ldots, \omega_L^{(0\cdot(L/p)+m)\cdot(p-1)}], \\ \vdots \\ [\omega_L^{((p-1)\cdot(L/p)+m)\cdot 0}, \ldots, \omega_L^{((p-1)\cdot(L/p)+m)\cdot(p-1)}] \end{bmatrix}, \right.
$$

$$
\left. [v_m, v_{L/p+m}, \ldots, v_{L-L/p+m}] \right\}
$$

The correctness of *matrixMult* has been proven in [37] and therefore mapping *matrixMult* over the tuples results in L/p elements that are one-dimensional arrays containing p elements. The element at position m has the following form:

$$
\left[\sum_{r=0}^{p-1} \omega_L^{(0\cdot(L/p)+m)\cdot r} v_{m+rL/p}, \ldots, \sum_{r=0}^{p-1} \omega_L^{((p-1)\cdot(L/p)+m)\cdot r} v_{m+rL/p} \right] \quad \text{(A.1)}
$$

The elements in this array are the elements in vector $B_{p,L}v$ at the positions $p \cdot (L/p) + m$ as shown in the following:

$$
\begin{aligned}
B_{p,L} &\overset{Def.\ 2.3}{=} (DFT_p \otimes I_{L/p}) diag(I_{L/p}, \Omega_{p,L/p}, \ldots, \Omega_{p,L/p}^{p-1}) \\
&\overset{Def.\ 2.1 + DFT_p}{=} \begin{pmatrix} \omega_p^{0\cdot 0} I_{L/p} \Omega_{p,L}^0 & \cdots & \omega_p^{0\cdot(p-1)} I_{L/p} \Omega_{p,L/p}^{p-1} \\ \vdots & & \vdots \\ \omega_p^{(p-1)\cdot 0} I_{L/p} \Omega_{p,L}^0 & \cdots & \omega_p^{(p-1)\cdot(p-1)} I_{L/p} \Omega_{p,L/p}^{p-1} \end{pmatrix}
\end{aligned}
$$

The diagonal block matrix $\omega_p^{j\cdot r} I_{L/p} \Omega_{p,L}^r$, with $0 \leq j, r < p$ has the form:

$$\omega_p^{j\cdot r} I_{L/p} \Omega_{p,L}^r = \begin{pmatrix} \omega_p^{j\cdot r} \cdot \omega_L^{r\cdot 0} & & \\ & \ddots & \\ & & \omega_p^{j\cdot r} \cdot \omega_L^{r\cdot (L/p)} \end{pmatrix}$$

Hence, we know that the element in row $jL/p + k$ and column $rL/p + k$ of $B_{p,L}$ is $\omega_p^{j\cdot r} \cdot \omega_L^{r\cdot k}$, for k with $0 \leq k < L/p$. It follows that

$$(B_{p,L} v)_{jL/p+k} = \sum_{r=0}^{p-1} \omega_p^{j\cdot r} \cdot \omega_L^{r\cdot k} v_{rL/p+k} \tag{A.2}$$

By definition of ω, we know that $\omega_m = \exp^{-2\pi i/m}$. Therefore, after applying power rules, we can express equation (A.3) with:

$$(B_{p,L} v)_{jL/p+k} = \sum_{r=0}^{p-1} \omega_L^{(j(L/p)+k)\cdot r} v_{rL/p+k} \tag{A.3}$$

Now with our knowledge about the structure of the array after mapping *matrixMult* (Equation (A.1)), we see that after applying *transpose*, the element at position $0 \leq q < p$ of the resulting array has the following form:

$$\left[\sum_{r=0}^{p-1} \omega_L^{(q\cdot(L/p)+0)\cdot r} v_{0+rL/p}, \ldots, \sum_{r=0}^{p-1} \omega_L^{(q\cdot(L/p)+(L/p)-1)\cdot r} v_{(L/p)-1+rL/p} \right]$$

With Equation A.3 we see that these elements are consecutive in the result vector $B_{p,L} v$. Furthermore, the last value in one element and the first in the following

$$\sum_{r=0}^{p-1} \omega_L^{(q\cdot(L/p)+(L/p)-1)\cdot r} v_{(L/p)-1+rL/p} = (B_{p,L} v)_{(q+1)\cdot(L/p)-1}$$

and

$$\sum_{r=0}^{p-1} \omega_L^{((q+1)\cdot(L/p)+0)\cdot r} v_{0+rL/p} = (B_{p,L} v)_{(q+1)\cdot(L/p)}$$

lie beside each other in $B_{p,L} v$ as well. Therefore, after applying *join*, the resulting array is an array representation of $B_{p,L} v$. □

A.1.3 *Kronapply is Well Defined*

LEMMA A.4. *Let vs be an array representation of vector v with n elements v_j, where $0 \leq j < n$. Let f be a function that computes a square matrix multiplication with the $r \times r$ matrix M and a vector of size r, where $n = rm$. For the expression kronapply (Definition 3.11) it holds that:*

$$kronapply(m, f, vs) = vs'$$

where vs′ is an array representation of v′, with

$$v'_{r\times m} = Mv_{r\times m}$$

Proof. From the definition of an $(r \times m)$ matrix-split through Lemma 3.1, we know that the k's column of $v_{r\times m}$ has the elements

$$\begin{pmatrix} v_k & v_{k+m} & v_{k+2m} & \dots & v_{k+n-m} \end{pmatrix}^T$$

in all columns $0 \leq k < m$. By definition of matrix multiplication, for a $r \times r$ matrix M we get that

$$Mv_{r\times m} = v'_{r\times m}$$

has the elements

$$M \begin{pmatrix} v_k & v_{k+m} & \dots & v_{k+n-m} \end{pmatrix}^T = \begin{pmatrix} v'_k & v'_{k+m} & \dots & v'_{k+n-m} \end{pmatrix}^T$$

in its k's column. With the definition of the matrix-split we get that

$$v' = \begin{pmatrix} v'_0 & v'_1 & \dots v'_{n-1} \end{pmatrix}^T$$

By applying the high-level primitives that make up *kronapply* we get the following:

$$\begin{aligned}
&kronapply(m, f, vs) \\
&\overset{Def.\ 3.11}{=} join \circ transpose \circ map(f) \circ transpose \circ split(m)\ (vs) \\
&\overset{Def.\ 3.4+Def.\ 3.6}{=} join \circ transpose \\
&\qquad \circ map(f) \left(\begin{bmatrix} [v_0, v_m, \dots, v_{n-m}], \\ [v_1, v_{1+m}, \dots, v_{1+(n-m)}], \\ \vdots \\ [v_{m-1}, v_{m-1+m}, \dots, v_{m-1+n-m}] \end{bmatrix} \right) \\
&\overset{Def.\ 3.1}{=} join \circ transpose \left(\begin{bmatrix} f([v_0, v_m, \dots, v_{n-m}]), \\ f([v_1, v_{1+m}, \dots, v_{1+(n-m)}]), \\ \vdots \\ f([v_{m-1}, v_{m-1+m}, \dots, v_{m-1+n-m}]) \end{bmatrix} \right)
\end{aligned}$$

And we see that the arrays in the k's row hold the values of the k's column of $v_{r\times m}$. We apply our assumption about f and get:

$$
\begin{aligned}
&\overset{assump.}{=} join \circ transpose \left(\begin{bmatrix} [v'_0, v'_m, \ldots, v'_{n-m}], \\ [v'_1, v'_{1+m}, \ldots, v'_{1+(n-m)}], \\ \vdots \\ [v'_{m-1}, v'_{m-1+m}, \ldots, v'_{m-1+n-m}] \end{bmatrix} \right) \\
&\overset{Def.\ 3.6}{=} join \circ \left(\begin{bmatrix} [v'_0, v'_1, \ldots, v'_{m-1}], \\ [v'_m, v'_{1+m}, \ldots, v'_{m-1+m}], \\ \vdots \\ [v'_{n-m}, v'_{1+(n-m)}, \ldots, v'_{m-1+n-m}] \end{bmatrix} \right) \\
&\overset{Def.\ 3.5}{=} [v'_0, v'_1, \ldots, v'_{m-1}, v'_m, \ldots, v'_{m-1+n-m}]
\end{aligned}
$$

which is the array representation of v' by the definition of an array representation. □

A.1.4 *Correctness of Stockperm*

Proof. By definition of *stockperm*

$$\begin{aligned} &stockperm(p, L, vs) \\ &\quad \overset{Def.\ 3.12}{=} kronapply(n/L,\ modpsort(p),\ vs) = ys \end{aligned}$$

From Lemma A.1 we know that $modposrt(p)$ represents a mod-p sort permutation and therefore a square matrix multiplication of the size of the array passed to *modpsort*.

Hence, we can use Lemma A.4 and get that ys is an array representation of y, where

$$y_{L \times \frac{n}{L}} = \Pi^T_{p,L} v_{L \times \frac{n}{L}}.$$

With Lemma 3.1 it follows that

$$y = (\Pi^T_{p,L} \otimes I_{n/L}) v \overset{Def.\ 2.6}{=} Q_{p,L} v.$$

□

A.1.5 *Correctness of Stockcombine*

Proof. By definition of *stockcombine*

$$\begin{aligned} &stockcombine(p, L, vs) \\ &\quad \overset{Def.\ 3.19}{=} kronapply(n/L,\ multB(p, L),\ vs) = ys \end{aligned}$$

From Lemma A.3 we know that *multB* represents a multiplication with a Butterfly matrix $B_{p,L}$ and therefore a square matrix multiplication of size $L \times L$.

Hence, we can use Lemma A.4 and get that ys is an array representation of y, where

$$y_{L \times \frac{n}{L}} = B_{p,L} v_{L \times \frac{n}{L}}.$$

With Lemma 3.1 it follows that

$$y = (B_{p,L} \otimes I_{n/L}) v \overset{Def.\ 2.7}{=} S_{p,L}$$

□

A.1.6 *TwiddleMult is Well Defined*

LEMMA A.5. *Let vs be an array representation of vector v of size n with elements v_j, where $0 \leq j < n$ and $n = n_1 \cdot n_2$. It holds that*

$$twiddleMult(n_1, n_2, vs) = join(ys)$$

where ys is the array representation of

$$DFT_n(0 : n_2 - 1,\ 0 : n_1 - 1). * v_{n_2 \times n_1}.$$

Proof. By Definition 3.24

$$\begin{aligned}
&twiddleMult(n_1, n_2, vs)\\
&= join \circ transpose \circ map(map(*)) \circ map(zip) \circ\\
&\quad zip(array_2(n_2, n_1, genDFTSlice)) \circ\\
&\qquad transpose \circ split(n_1)\ (vs)
\end{aligned}$$

$$\begin{aligned}
&\overset{Def.\ 3.4+Def.\ 3.6}{=} join \circ transpose \circ map(map(*)) \circ map(zip) \circ\\
&\quad zip(array_2(n_2, n_1, genDFTSlice))\\
&\qquad \left(\begin{bmatrix} [v_0, v_{n_1}, \ldots, v_{n-n_1}], \\ [v_1, v_{n_1+1}, \ldots, v_{n-n_1+1}], \\ \vdots \\ [v_{n_1-1}, v_{2n_1-1}, \ldots, v_{n-1}] \end{bmatrix}\right)\\
&\overset{Def.\ 3.24}{=} join \circ transpose \circ map(map(*)) \circ map(zip) \circ\\
&\quad zip\left(\begin{bmatrix} [\omega_n^{0\cdot 0}, \omega_n^{0\cdot 1}, \ldots, \omega_n^{0\cdot(n_2-1)}], \\ [\omega_n^{1\cdot 0}, \omega_n^{1\cdot 1}, \ldots, \omega_n^{1\cdot(n_2-1)}], \\ \vdots \\ [\omega_n^{(n_1-1)\cdot 0}, \omega_n^{(n_1-1)\cdot 1}, \ldots, \omega_n^{(n_1-1)\cdot(n_2-1)}], \end{bmatrix}\right)\\
&\qquad \left(\begin{bmatrix} [v_0, v_{n_1}, \ldots, v_{n-n_1}], \\ [v_1, v_{n_1+1}, \ldots, v_{n-n_1+1}], \\ \vdots \\ [v_{n_1-1}, v_{2n_1-1}, \ldots, v_{n-1}] \end{bmatrix}\right)
\end{aligned}$$

Therefore, by Definition 3.3 of *zip*, the rows in the matrices are combined to tuples. The tuple resulting from the k's rows, whith $0 \leq k < n_1$, is

$$\left\{[\omega_n^{k\cdot 0}, \omega_n^{k\cdot 1}, \ldots, \omega_n^{k\cdot(n_2-1)}], [v_k, v_{n_1+k}, \ldots, v_{n-n_1+k}]\right\}$$

Where the second element of the tuple represents the column of the matrix-split $v_{n_2 \times n_1}$ according to Lemma 3.1. Applying $map(zip)$ to

all these tuples results in n_1 elements of the following form (follows from Definiton 3.1 *map* and Definition 3.3 *zip*):

$$\left[\{\omega_n^{k\cdot 0}, v_k\}, \{\omega_n^{k\cdot 1}, v_{n_1+k}\}, \ldots, \{\omega_n^{k\cdot(n_2-1)}, v_{n-n_1+k}\}\right]$$

Hence, a two-dimensional array of tuples is created. For $map(map(*))$ we apply Definition 3.1 *map* twice with complex multiplication and continue our chain of equations:

$$\begin{aligned}
&= join \circ transpose \\
&\left(\begin{bmatrix}
[\omega_n^{0\cdot 0} v_0, \omega_n^{0\cdot 1} v_{n_1}, \ldots, \omega_n^{0\cdot(n_2-1)} v_{n-n_1}], \\
[\omega_n^{1\cdot 0} v_1, \omega_n^{1\cdot 1} v_{n_1+1}, \ldots, \omega_n^{1\cdot(n_2-1)} v_{n-n_1+1}], \\
\vdots \\
[\omega_n^{(n_1-1)\cdot 0} v_{n_1-1}, \omega_n^{(n_1-1)\cdot 1} v_{2n_1-1}, \ldots, \omega_n^{(n_1-1)\cdot(n_2-1)} v_{n-1}]
\end{bmatrix}\right) \\
&\overset{Def.\ 3.6}{=} join \\
&\left(\begin{bmatrix}
[\omega_n^{0\cdot 0} v_0, \omega_n^{1\cdot 0} v_1, \ldots, \omega_n^{(n_1-1)\cdot 0} v_{n_1-1}], \\
[\omega_n^{0\cdot 1} v_{n_1}, \omega_n^{1\cdot 1} v_{n_1+1}, \ldots, \omega_n^{(n_1-1)\cdot 1} v_{2n_1-1}], \\
\vdots \\
[\omega_n^{0\cdot(n_2-1)} v_{n-n_1}, \omega_n^{1\cdot(n_2-1)} v_{n-n_1+1}, \ldots, \omega_n^{(n_1-1)\cdot(n_2-1)} v_{n-1}]
\end{bmatrix}\right) \\
&= join(ys)
\end{aligned}$$

from the element-wise product, the definition of DFT_n and Lemma 3.1, it follows that ys is the array representation of

$$DFT_n(0 : n_2 - 1,\ 0 : n_1 - 1) .* v_{n_2 \times n_1}.$$

□

A.2 LOW-LEVEL COOLEY-TUKEY EXPRESSION

```
1  dotproduct =
2      λ((xs, ys) ⇒ toGlobal(mapSeq(id)) ∘ reduceSeq(+, 0) ∘
3          toPrivate(mapSeq(∗)) ∘ zip(xs, ys)
4
5  matrixMult =
6      λ((M, xs) ⇒ join ∘ mapSeq(λ(Mrow ⇒ dotproduct(xs, Mrow))))
7
8  reorB = transpose ∘ map(transpose) (array3(p, p, L/p, genB))
9
10 cooleytukeyPass =
11     λ(vs ⇒ join ∘ transpose ∘ map(join ∘ transpose) ∘ split(L/p)∘
12         //Butterfly computations in only one dimension of WI.
13         mapGlobal0(matrixMult) (
14             //Multiplied reordered Butterfly matrix.
15             zip(pad(0, (n − L)/p, wrap, reorB),
16                 join ∘ map(transpose ∘ split(L/p)) ∘
17                     split(L/p) (vs))))
```

Listing A.1: A completely lowered *cooleytukeyPass* expression.

BIBLIOGRAPHY

[1] Inc. Advanced Micro Devices. *AMD APP SDK OpenCL Optimization Guide*. Aug. 2015.

[2] David H Bailey. "A high-performance FFT algorithm for vector supercomputers." In: *The International Journal of Supercomputing Applications* 2.1 (1988), pp. 82–87.

[3] Charles K Birdsall and A Bruce Langdon. *Plasma physics via computer simulation*. CRC press, 2004.

[4] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[5] James Cooley and John Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." In: *Mathematics of Computation* 19.90 (1965), pp. 297–301.

[6] Nvidia Corporation. *CUDA cuFFT*. 2018. URL: https://developer.nvidia.com/cufft.

[7] Xiang Cui, Yifeng Chen, and Hong Mei. "Improving performance of matrix multiplication and FFT on GPU." In: *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*. IEEE. 2009, pp. 42–48.

[8] Steffen Ernsting and Herbert Kuchen. "Algorithmic skeletons for multi-core, multi-GPU systems and clusters." In: *International Journal of High Performance Computing and Networking* 7.2 (2012), pp. 129–138.

[9] Franz Franchetti, Markus Puschel, Yevgen Voronenko, Srinivas Chellappa, and José MF Moura. "Discrete Fourier transform on multicore." In: *IEEE Signal Processing Magazine* 26.6 (2009).

[10] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. "FFT program generation for shared memory: SMP and multicore." In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 115.

[11] Matteo Frigo. "A fast Fourier transform compiler." In: *Acm sigplan notices*. Vol. 34. 5. ACM. 1999, pp. 169–180.

[12] Matteo Frigo and Steven G Johnson. *The fastest fourier transform in the west*. Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE, 1997.

[13] Matteo Frigo and Steven G Johnson. "The design and implementation of FFTW3." In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[14] Naga Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. "High Performance Discrete Fourier Transforms on Graphics Processors." In: *Proceedings of the 2008 ACM/ IEEE conference on Supercomputing*. IEEE, Jan. 2008.

[15] Y-Z Guo, M Li, M Lu, Z Wen, K Wang, G Li, and J Wu. "Classifying G protein-coupled receptors and nuclear receptors on the basis of protein power spectrum from fast Fourier transform." In: *Amino Acids* 30.4 (2006), pp. 397–402.

[16] Bastian Hagedorn. "An Extension of a Functional Intermediate Language For Parallelizing Stencil Computations And Its Optimizing GPU Implementation Using OpenCL." MA thesis. University of Münster, 2016.

[17] Bastian Hagedorn, Michel Steuwer, and Sergei Gorlatch. "A Transformation-Based Approach to Developing High-Performance GPU Programs." In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer. 2017, pp. 179–195.

[18] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High performance stencil code generation with Lift." In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM. 2018, pp. 100–112.

[19] Adam Harries, Michel Steuwer, Murray Cole, Alan Gray, and Christophe Dubach. "Compositional Compilation for Sparse, Irregular Data Parallelism." In: *Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU)*. 2016.

[20] Lee Howes and Aaftab Munshi. "The OpenCL specification, version 2.0." In: *Khronos Group* (2015).

[21] Advanced Micro Devices Inc. *AMD clFFT*. 2018. URL: https://github.com/clMathLibraries/clFFT.

[22] S Lennart Johnsson, Michel Jacquemin, and Ching-Tien Ho. *High radix FFT on Boolean cube networks*. Thinking Machines Corporation, 1989.

[23] S Lennart Johnsson and Robert L Krawitz. "Cooley-tukey FFT on the connection machine." In: *Parallel Computing* 18.11 (1992), pp. 1201–1221.

[24] David G Korn and Jules J Lambiotte. "Computing the fast Fourier transform on a vector computer." In: *Mathematics of computation* 33.147 (1979), pp. 977–992.

[25] Mario Leyton and José M Piquer. "Skandium: Multi-core programming with algorithmic skeletons." In: *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE. 2010, pp. 289–296.

[26] D Brandon Lloyd, Chas Boyd, and Naga Govindaraju. "Fast computation of general Fourier transforms on GPUs." In: *Multimedia and Expo, 2008 IEEE International Conference on*. IEEE. 2008, pp. 5–8.

[27] Paul A Lynn and Wolfgang Fuerst. *Introductory Digital signal processing with computer applications*. John Wiley and Sons, 1989.

[28] Michael Mathieu, Mikael Henaff, and Yann LeCun. "Fast training of convolutional networks through ffts." In: *arXiv preprint arXiv:1312.5851* (2013).

[29] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. "Optimising purely functional GPU programs." In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 49–60.

[30] CUDA Nvidia. *Programming guide*. 2010.

[31] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. "A survey of general-purpose computation on graphics hardware." In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 80–113.

[32] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530.

[33] Ari Rasch, Michael Haidl, and Sergei Gorlatch. "ATF: A Generic Auto-Tuning Framework." In: *High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference on*. IEEE. 2017, pp. 64–71.

[34] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. "Performance Portable GPU Code Generation for Matrix Multiplication." In: *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM. 2016, pp. 22–31.

[35] Greg Ruetsch and Paulius Micikevicius. "Optimizing matrix transpose in CUDA." In: *Nvidia CUDA SDK Application Note* 18 (2009).

[36] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[37] Michel Steuwer. "Improving programmability and performance portability on many-core processors." PhD thesis. University of Münster, 2015.

[38] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. "Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High- Performance OpenCL Code." In: *SIGPLAN Not.* 50.9 (Aug. 2015), pp. 205–217. ISSN: 0362-1340.

[39] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "Skelcl-a portable skeleton library for high-level gpu programming." In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 1176–1182.

[40] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation." In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM. 2016, p. 15.

[41] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation." In: *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. IEEE. 2017, pp. 74–85.

[42] Charles Tong and Paul N Swarztrauber. "Ordered fast Fourier transforms on a massively parallel hypercube multiprocessor." In: *Journal of Parallel and Distributed Computing* 12.1 (1991), pp. 50–59.

[43] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil. "Software pipelined execution of stream programs on GPUs." In: *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE. 2009, pp. 200–209.

[44] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992. ISBN: 0-89871-285-8.

DECLARATION

Hiermit versichere ich, dass die vorliegende Arbeit über *Implementing and Optimizing Fast Fourier Transforms in LIFT* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Münster, September 2018

Bastian Köpcke

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in einer Datenbank einverstanden.

Münster, September 2018

Bastian Köpcke

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.

Final Version as of October 10, 2018 (`classicthesis` version 4.5).