Generating Efficient FFT GPU Code with Lift

Bastian Köpcke
University of Münster
Münster, Germany
bastian.koepcke@wwu.de

Michel Steuwer
University of Glasgow
Glasgow, Scotland, United Kingdom
michel.steuwer@glasgow.ac.uk

Sergei Gorlatch
University of Münster
Münster, Germany
gorlatch@wwu.de

Abstract

The Fast Fourier Transform is a well-known algorithm used in many high-performance applications, ranging from signal processing to convolutional neural networks.

In this paper, we encode FFTs by building high-level abstractions based on a set of functional parallel patterns in the Lift language. Abstractions are derived from and closely resemble mathematical definitions for FFTs. We leverage the Lift performance-portable code generator to generate high performing GPU code for FFTs. No FFT-specific patterns are required for this, showing the expressive power of the generic parallel patterns used in Lift.

Our experimental results show that our approach achieves performance better than AMD's OpenCL implementation clFFT on an Nvidia GPU. Nvidia's highly optimized cuFFT implementation still performs better on their GPUs.

CCS Concepts • **Software and its engineering → Parallel programming languages**; **Compilers**.

Keywords FFT, GPU, Code Generation, Lift

ACM Reference Format:
Bastian Köpcke, Michel Steuwer, and Sergei Gorlatch. 2019. Generating Efficient FFT GPU Code with Lift. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '19), August 18, 2019, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3331553.3342613

1 Introduction

The *Fast Fourier Transform* is a well-known algorithm used in many high-performance applications, ranging from signal and image processing to more recent usage in convolutional neural networks [21].

Traditionally FFTs were implemented in high-performance libraries such as FFTW [8]. As there are many possible implementations of FFTs, each with their own trade-offs and performance characteristics, it is challenging to provide an optimal implementation for each input size and target device. Modern libraries for GPUs, such as cuFFT [12], are very sophisticated and contain different variations carefully tuned for the target GPU architectures.

Spiral [7, 13] is a successful example of a code generator taking advantage of domain knowledge to generate high-performance code from domain-specific patterns. Spiral originally generated efficient FFT implementations for multiple CPU architectures [13], while recent efforts expand it towards GPUs and FPGAs [7]. Spiral achieves high performance with an FFT-specific program representation and domain-specific transformation rules that allow the implementation space to be explored systematically.

In this paper, we investigate the generation of fast implementations of FFT on GPUs starting from a functional high-level pattern-based program representation. Our goal is to explore the expressiveness of well-known functional patterns of parallel computing, like *map* and *reduce*. Furthermore, we explore the capabilities of the pattern-based code generator Lift [17] for achieving high performance of FFTs.

Programs expressed as compositions of parallel patterns are transformed into a GPU-specific representation of low-level patterns from which high-performance OpenCL code is generated by the Lift code generator. The transformation process is driven by a set of semantic-preserving rewrite rules expressing implementation and optimization choices. We extend the area of Lift applications which has already been shown to achieve high performance [19] in multiple domains including dense and sparse linear algebra, as well as stencil computations.

The general-purpose philosophy of Lift allows the composition of algorithms from different domains in larger applications that are compiled and optimized together. Encoding FFTs in general-purpose high-level patterns enables new optimization choices in the Lift code generator. It provides the possibility to automatically decide to use FFT in bigger applications based on domain-specific rewrite rules. Parts of an application for which there exist semantically equivalent encodings based on FFT can be substituted. For example, it is possible to use stencils, as well as FFT, to implement convolutional layers in convolutional neural networks [21]. The implementation choice between stencils, which have

FHPNC '19, August 18, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6814-8/19/08…$15.00
https://doi.org/10.1145/3331553.3342613

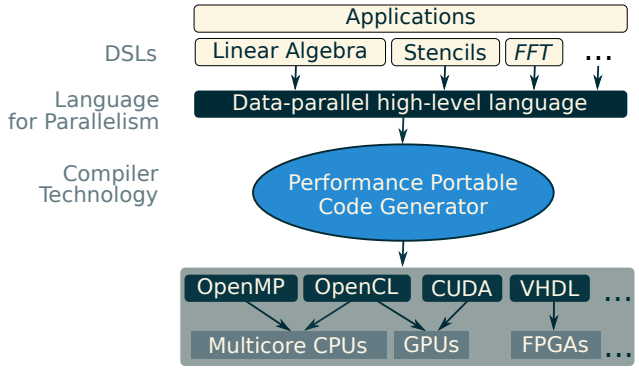

Figure 1. Overview of the Lift parallel language and code generator.

been encoded in Lift previously [10], and FFT, can easily be expressed as a rewrite rule. Overly specialized code generators such as Spiral do not have this ability.

Our paper makes the following contributions:

- We encode FFTs in the functional language Lift by building high-level pattern-based abstractions that are closely related to the mathematical notations of FFTs (Section 3).
- We describe how efficient code is generated from these high-level abstractions using Lift's semantic preserving rewrite rules (Section 4).
- We experimentally evaluate our approach showing that we achieve performance close to or even better than the optimized OpenCL implementation clFFT by AMD (Section 5).

2 Background

2.1 Lift: Generating Efficient GPU Code with Patterns and Rewrite Rules

The Lift system [19] comprises a functional high-level data-parallel language and code generator that aim at providing performance portability for high-performance programs. The Lift language is based on high-level patterns which express what to compute rather than how to compute it. As depicted in Figure 1, high-performance programs for different target architectures are generated from expressions in the Lift language by a performance-portable code generator. For GPUs, Lift programs are compiled into OpenCL kernels.

```
1 def dotproduct = λ((v, w) =>
2     reduce(+, 0) <:  map(*) <: zip(v, w))
3
4 def matVecMult = λ((M, v) =>
5     map(λ(rowM => dotproduct(rowM, v))) <: M)
```

Listing 1. "High-level Lift implementation of a matrix-vector multiplication."

Listing 1 shows an example of a high-level Lift expression for matrix-vector multiplication of matrix `M` and vector `v`. It is implemented using the Lift functions `matVecMult` and `dotproduct`. Lift functions are created using lambda-expressions, with parameters on the left-hand side of the arrow `=>` and the function body on the right-hand side. In the function body, an expression is created by applying predefined high-level patterns or lambda-expressions to values. For readability, we use the operator `<:`, which is synonym to function application, *e.g.*, `map(*) <: zip(v, w)` ≡ `map(*, zip(v, w))`. In the example, the parameter `M` – a two-dimensional array representing a matrix – is passed to a *map* which is instantiated with another Lift function. The use of the *map* pattern encodes that for every row of the matrix the dotproduct with the vector v is computed. The function `dotproduct` encodes multiplying two arrays element-wise first and then summing the results up with *reduce*.

During code generation in Lift, a high-level program is rewritten into a low-level representation consisting of low-level, hardware-specific patterns. Low-level patterns encode how computations and data are mapped to a target device, *e.g.*, a GPU. Lift's rewrite system ensures that the low-level program has the same semantics as the high-level program. In its low-level form, the program is enriched with the information about how the computation is mapped onto the thread and memory hierarchies of the target device. When the target architecture is a GPU, the low-level patterns are closely related to the OpenCL programming model, such that OpenCL code can be generated quite straightforwardly. This approach was shown to generate high-performance code for various domains, including sparse linear algebra, General Matrix Multiplication (GEMM), and stencil computations.

In this paper, we express FFTs using the following data-parallel patterns defined in Lift. One of the findings of this paper is that it is not necessary to add further patterns in order to express the variety of FFT versions. This emphasizes the expressiveness of patterns that are well-known in the functional and algorithmic skeleton communities.

$$
\begin{aligned}
\textbf{\textit{map}} &: (\alpha \to \beta) \to [\alpha]_n \to [\beta]_n \\
\textbf{\textit{reduce}} &: \alpha \to (\alpha \to \beta \to \alpha) \to [\beta]_n \to \alpha \\
\textbf{\textit{zip}} &: [\alpha]_n \to [\beta]_n \to [(\alpha \times \beta)]_n \\
\textbf{\textit{split}} &: (m : Int) \to [\alpha]_n \to [[\alpha]_m]_{n/m} \\
\textbf{\textit{join}} &: [[\alpha]_m]_n \to [\alpha]_{m \cdot n} \\
\textbf{\textit{transpose}} &: [[\alpha]_m]_n \to [[\alpha]_n]_m \\
\textbf{\textit{array}} &: (n : Int) \to (Int \to Int \to \alpha) \to [\alpha]_n
\end{aligned}
$$

Each high-level pattern is given with its type. Array types are written as $[\alpha]_n$, where α is the type of the elements, and the size n of the array is tracked in the type. For tuple types, we write $(\alpha \times \beta)$ with component types α and β. Finally, $\alpha \to \beta$ is the function type expecting a value of type α and returning a value of type β.

In the following, we briefly describe the semantics of all patterns: *map* applies a given function to each element of an array, creating a new array of the same size containing the results. *reduce* combines all the elements in an array by traversing it and applying a binary reduction operator; the result is stored in an accumulator variable initialized by the first argument. *zip* combines two input arrays of the same size into a single array of tuples ($\alpha \times \beta$). *split* partitions the input array of size n into chunks of m elements, resulting in a two-dimensional array. *join* is the inverse operation of the *split* pattern. *transpose* swaps the dimensions of a two-dimensional array and reorders the array elements accordingly. The *array* generator pattern produces an array by invoking a function for each element passing the element's position and the size of the array to it. For higher dimensions, *array* is defined analogously. Additionally, Lift allows the definition of *user functions* written in C and embedded in the generated OpenCL code. They are used to implement operators, *e.g.*, multiplication of `float` values. User functions only accept and return values that are not of array type.

A key contribution of this paper is to show how to express FFTs in this generic framework and eventually achieve high performance. We accomplish this by taking advantage of Lift's rewrite rules to map a high-level pattern-based representation to a low-level representation, from which efficient GPU code is generated automatically.

2.2 Fast Fourier Transform

Fast Fourier Transforms (FFTs) are fast implementations of the *Discrete Fourier Transform (DFT)*. The DFT is computed as a matrix-vector product $D_n \cdot v$ of a complex-valued matrix D_n of size $n \times n$ and vector v of size n, where the values $d_{j,k}$ of D_n are defined by the following formula:

$$d_{j,k} = \omega_n^{j \cdot k} \qquad \text{for } 0 \leq j, k < n \text{ and } \omega_n^{j \cdot k} = e^{(-2\pi i/n) \cdot jk}$$

Here, e is Euler's number and i the imaginary number. In other words, *"computing the DFT of size n over vector v"* is a synonym for the matrix-vector product of D_n and v.

The general idea of FFT is to decompose the matrix D_n into a product of multiple matrices A_p. These matrices are sparse with a regular pattern of non-zero values, thus allowing for efficient multiplication implementations. All A_r are iteratively multiplied with the input vector, such that

$$D_n \cdot v = A_t \cdot (\cdots (A_1 \cdot v)) \tag{1}$$

Implementations of the matrix-vector multiplications

$$A_p \cdot v_{p-1} \qquad \text{where } v_{p-1} = A_{p-1} \cdot v_{p-2} \text{ and } v_0 = v$$

exploit the sparsity of A_p to reduce the time complexity to $O(n \ \log n)$ compared to the complexity of $O(n^2)$ required for the multiplication of D_n and v. In the following, we will call a single matrix-vector multiplication of $A_p \cdot v_{p-1}$ a *pass over the vector* v_{p-1}. Different variations of FFT decompose D_n into different passes A_p.

Figure 2 shows the passes of the two most popular FFT variations – Cooley-Tukey and Stockham. Both variations perform passes that permute the vector or combine smaller DFT results together. A pass which combines smaller DFT results either with or without permuting the vector is called a combine pass. The *Cooley-Tukey* FFT begins with the bit-reversal pass that permutes the vector, such that there are no further permutations necessary in later passes. For this, A_1 is a permutation matrix. The bit-reversal pass is followed by passes $A_p \cdot v_{p-1}$ for $2 < p \leq t$ that combine previous results into new intermediary results until the complete DFT is computed. In contrast to this, the *Stockham* FFT consists of combine passes only. It permutes the vector, followed by computing the intermediary results, in every single pass.

The result of a combine pass is a vector consisting of several subvectors that hold the result of computing the DFT of size L_q over specific elements of the input vector v, *e.g.*, $D_2 \cdot v(0, 4)$ in Figure 2. We call such a result an intermediary DFT result. For the intermediary DFT results, a DFT of size r_q over elements of the previous pass is computed and multiplied with values, that are called twiddle factors (shown in Figure 2 by the lines without arrowheads). r_q is called the radix of combine pass q, with $1 \leq q < u$. All radices factorize the size n of the input vector v, *i.e.*, $n = r_u \cdots r_2 \cdot r_1$. Hence, u is the number of combine passes in an FFT version. The size L_q of the intermediary DFT results depends on the radix r_q of the current pass and the radices of previous passes, *i.e.*, $L_q = r_q \cdots r_2 \cdot r_1$. In the example FFTs in Figure 2, every radix equals 2 and, therefore, $u = 3$ and $n = 2 \cdot 2 \cdot 2 = 8$.

Figure 3 shows how FFT variations are expressed as different matrix-vector products. The notation uses the Kronecker product operator. The Kronecker product of a $p \times s$ matrix A and a $m \times n$ matrix B, is a matrix of size $pm \times sn$:

$$A \otimes B = \begin{pmatrix} a_{0,0} \cdot B & \cdots & a_{0,s-1} \cdot B \\ \vdots & & \vdots \\ a_{p-1,0} \cdot B & \cdots & a_{p-1,s-1} \cdot B \end{pmatrix}$$

We call a matrix that is contained in another matrix a block matrix, *e.g.*, the $a_{j,k}B$ form block matrices. We note that there appear only three different kinds of matrices in Stockham (2) and Cooley-Tukey (3), namely Π_{r_q, L_q}, B_{r_q, L_q} and I_{r_q, L_q}. Multiplying the butterfly matrices B_{r_q, L_q} with a vector multiplies twiddle factors and computes DFTs of size r_q. The butterfly matrices are defined as follows:

$$B_{r_q, L_q} = (D_{r_q} \otimes I_{L_q/r_q}) \cdot diag(I_{L_q/r_q}, \Omega_{r_q, L_q/r_q}, \ldots, \Omega_{r_q, L_q/r_q}^{r_q - 1}) \tag{4}$$

Matrix $\Omega_{r_q, L_q/r_q}$, is a diagonal matrix of twiddle factors. The other matrices, I_{n/L_q} and Π_{r_q, L_q}, are identity matrices of size n/L_q and permutation matrices, respectively. To form passes, these matrices are combined in terms of matrix-matrix multiplication, the Kronecker product ($\otimes$) and matrix transposition. Operation $*$ denotes element-wise multiplication.

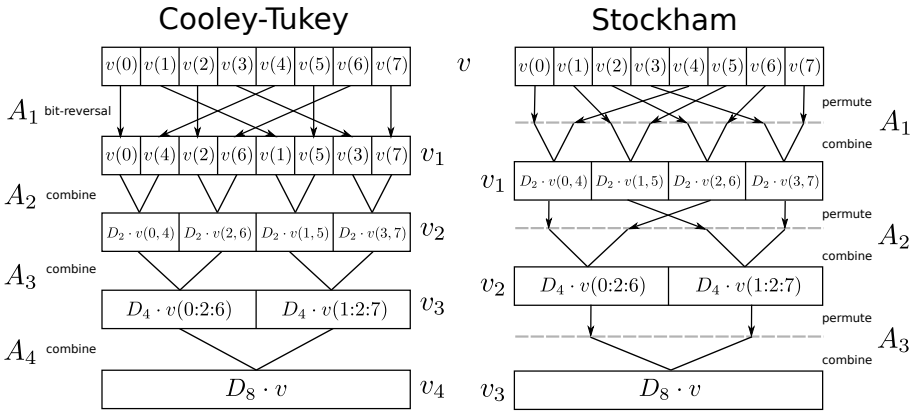

Figure 2. Example of the dataflow in the Cooley-Tukey and Stockham FFTs over an input array of size 8 (inspired by [11]).

$$\text{Stockham: } D_n v = \overbrace{(B_{r_u,L_u} \otimes I_{\frac{n}{L_u}}) \cdot (\Pi^T_{r_u,L_u} \otimes I_{\frac{n}{L_u}})}^{\text{Stockham pass}} \cdots \overbrace{(B_{r_1,L_1} \otimes I_{\frac{n}{L_1}}) \cdot (\Pi^T_{r_1,L_1} \otimes I_{\frac{n}{L_1}})}^{\text{Stockham pass}} \cdot v \quad (2)$$

$$\text{Cooley-Tukey: } D_n v = \overbrace{(I_{\frac{n}{L_u}} \otimes B_{r_u,L_u})}^{\text{Cooley-Tukey pass}} \cdots \overbrace{(I_{\frac{n}{L_1}} \otimes B_{r_1,L_1})}^{\text{Cooley-Tukey pass}} \cdot \overbrace{((I_{\frac{n}{L_u}} \otimes \Pi_{r_u,L_u}) \cdots (I_{\frac{n}{L_1}} \otimes \Pi_{r_1,L_1}))^T}^{\text{bit-reversal pass}} \cdot v \quad (3)$$

Figure 3. FFT variations in matrix-vector notation from [20].

In this section, we have shown that the DFT of a vector is performed by computing a matrix-vector product. The general idea for FFTs is that they split this matrix-vector product into multiple passes consisting of computing matrix-vector products with sparse matrices. These passes are implemented efficiently, so that FFTs are faster than DFTs. There are many variations of FFTs including the Cooley-Tukey and Stockham FFT, which are expressed in matrix-vector notation using operators such as matrix-vector multiplication and the Kronecker product. Next, we are going to use the parallel patterns of Lift to encode FFTs with the goal of generating efficient GPU code.

3 Encoding FFTs with High-Level Lift Patterns

In this section, we encode FFTs in Lift. We aim for building abstractions with Lift that are as close as possible to the mathematical notation introduced in Section 2. We show how to encode FFTs at the example of Stockham FFT. The ideas used for encoding the Stockham FFT are then reused to implement the Cooley-Tukey FFT.

A Stockham FFT pass is decomposed into multiple smaller matrix-vector multiplications. To echo this structure in Lift, we define Stockham pass as a function that is composed out of multiple function calls each representing a smaller matrix-vector multiplication. A key idea is that we encode domain-specific knowledge about the structure of the matrix operands which eventually will allow us to generate efficient implementations.

As described earlier, the amount of passes depends on the choice of radices. Larger radices lead to less passes. For a compiled program, this choice also has an impact on whether the hardware is able to coalesce memory accesses and how much work is performed in a single thread. With growing radices the matrix and vector in the smaller matrix-vector multiplications becomes larger increasing the amount of arithmetic operations.

The hierarchical FFT is another FFT variant that treats the input as a matrix and has a fixed number of passes. It recursively computes FFTs instead of smaller matrix-vector multiplications and elements are guaranteed to be read in a coalesced manner. As a drawback of this variant, matrix transpositions over the entire input have to be computed resulting in more work that is performed in a single pass. This can slow down execution especially for larger inputs. We give a brief overview on how we encoded this variant in Lift as well. Throughout this section, we will point out the close correspondence between the mathematical formulas of Section 2 and the Lift programs presented in this section.

3.1 Encoding Stockham FFT in Lift

As seen in Section 2.2, Stockham FFT consists of a series of passes which are matrix-vector multiplications. We encode this series of passes by creating Lift functions for individual passes. These functions are composed to encode a Stockham FFT, as follows:

```
1 def fftStockham(rs, v) = {
2   def n = r_1 * r_2 * ... * r_u;
3   stockPass(r_u, n, n)
4     <<: ... <<: stockPass(r_1, r_1, n) <<: v
5 }
```

Function `fftStockham` takes as arguments an input vector `v` represented as an array and an array `rs = [r_1, r_2, ..., r_u]` of radices. It composes Lift functions that are created by calling `stockPass` with different arguments. For pass q, it calls `stockPass` with the radix r_q and computes L_q by multiplying radices. In matrix-vector notation, a Stockham pass is encoded as two consecutive matrix-vector multiplications:

$$\text{stockPass } v = \underbrace{(B_{r,L} \otimes I_{\frac{n}{L}})}_{\text{combine}} \cdot \underbrace{(\Pi^T_{r,L} \otimes I_{\frac{n}{L}})}_{\text{permute}} \cdot v$$

```
1 def stockPass(r, L, n) = λ(v =>
2   stCombine(r, L, n)) <<: stPerm(r, L, n) <<: v
```

For a specific pass, parameters r, L and n are fixed while v is an arbitrary vector of size n. We encode this in function `stockPass` which creates a Lift function that expects a single argument v. The created Lift function composes two other Lift functions that encode the matrix-vector multiplications with permute and combine matrices. In matrix-vector notation, the permute matrix is created by the Kronecker product with an identity matrix as the right operand and a transposed permutation matrix as the left operand:

$$\text{permute } v = (\Pi^T_{r,L} \otimes I_{\frac{n}{L}}) \cdot v$$

```
1 def stPerm(r, L, n) = λ(v =>
2   kronIdR(n/L, multPiT(r)) <<: v)
```

The function `kronIdR` encodes a general matrix-vector multiplication with a matrix created by the Kronecker product operator, where the right operand is an identity matrix. Function `multPiT` encodes the matrix-vector multiplication of a transposed permutation matrix $\Pi^T_{r,L}$.

Similarly, in matrix-vector notation, the combine matrix is created by the Kronecker product operator with an identity matrix as the right operand and a butterfly matrix as the left operand:

$$\text{combine } v = (B_{r,L} \otimes I_{\frac{n}{L}}) \cdot v$$

```
1 def stCombine(r, L, n) = λ(v =>
2   kronIdR(n/L, multB(r, L)) <<: v)
```

3.1.1 The Kronecker Product (⊗)

A general implementation of the Kronecker product would require multiplying every element of the left operand to every element of the right operand. We avoid performing these multiplications by creating specialized functions that encode knowledge about the structure of the matrix operands.

In the matrix-vector notation of Stockham FFT – in the combine and the permute matrices – the right operand of the Kronecker product is an identity matrix. Using this knowledge we create a function which computes the Kronecker product with an identity matrix as the right operand.

```
1 def kronIdR(m, matMult)= λ(v =>
2   join <<: transpose <<: map(matMult)
3     <<: transpose <<: split(m) <<: v)
```

Function `kronIdR` is based on predefined high-level Lift patterns. Figure 4 shows an example of a matrix-vector multiplication, where the matrix is created by the Kronecker product operator with an identity matrix on the right. It shows that multiplying a matrix $A \otimes I_m$ with a vector is similar to multiplying the matrix A with smaller vectors. These vectors hold elements from v that are a fix size apart from each other. The distance between elements depends on the size of the identity matrix. In Lift, this is encoded by creating arrays of elements that form smaller vectors and applying *map* with a function that encodes the multiplication of A, over them.

3.1.2 Multiplying a Transposed Permutation Matrix

We encode multiplying a transposed permutation matrix as follows:

$$\text{multPiT } v = \Pi^T_{r,L} \cdot v$$

```
1 def multPiT(r) = λ(v =>
2   join <<: transpose <<: split(r) <<: v)
```

By definition, multiplying a transposed permutation matrix with a vector v permutes the elements in the vector such that elements in v that are $r-1$ elements apart from each other, appear next to each other in the permuted vector. Function `multPiT` splits an array into chunks of size r and transposes the resulting two-dimensional array. This groups elements that are $r-1$ elements apart from each other. After joining the two-dimensional into a one-dimensional array, the indices of the one-dimensional array's elements are sorted like the vector that results from multiplying a transposed permutation matrix.

3.1.3 Multiplying a Butterfly Matrix

In order to encode the matrix-vector multiplication of a butterfly matrix $B_{r,L}$, we need to represent the matrix as an array in Lift. $B_{r,L}$ is sparse, *i.e.*, has many zero entries. Our goal is to avoid multiplying by zero. We achieve this by deriving an array representation of butterfly matrices that contains

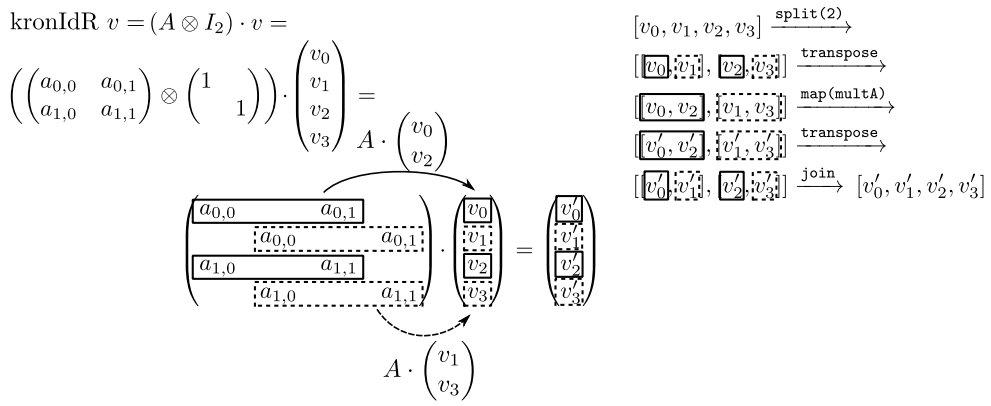

Figure 4. Matrix-vector multiplication of a Kronecker product with an identity matrix on the right.

non-zero elements only and by multiplying only elements from this representation with vector elements.

```
1 def multB(r, L) = λ(v =>
2   join <<: transpose <<: map(complexMatVecMult)
3     <<: zip(transpose <<: split(L/r) <<: v,
4             RepOfB(r, L))
```

Figure 5 shows an example of a matrix-vector multiplication with a butterfly matrix. It shows that multiplying $B_{r,L}$ with a vector is similar to multiplying smaller matrices $B^{(j)}_{r,L}$ with smaller vectors. The smaller matrices are made up out of elements in $B_{r,L}$ and do not contain any zero entries. The smaller vectors are made up out of elements in v. We represent butterfly matrices as three-dimensional arrays where the inner two-dimensional arrays represent the matrices $B^{(j)}_{r,L}$. This way, the representation of the butterfly matrix does not contain any zero-entries. Similarly to a matrix-vector multiplication where the matrix is a Kronecker product with an identity matrix on the right, we group the values of the smaller vectors into arrays. The matrices $B^{(j)}_{r,L}$ together with the vector that they are multiplied with, are passed to Lift function `complexMatVecMult` which encodes complex matrix-vector multiplication.

The array representation of butterfly matrices is encoded in `RepOfB` using Lift's three-dimensional array pattern $array_3$:

```
1 def RepOfB(r, L) = array3(r, r, L/r, genRepB)
2
3 def genRepB(j, k, l, numMats, height, width) =
4   omega((k * numMats + j) * l, height * numMats
        )
5
6 def omega(k, n) =
7   { cos(-2 * π * k / n), sin(-2 * π * k / n) }
```

Pattern $\texttt{array}_3$ expects the sizes of the array in three dimensions and a user function. User function `genRepB` computes a specific array element depending on the element's indices in the representation of the butterfly matrix, by calling `omega`. Function `omega` returns a complex number. Complex numbers are represented as tuples of a real and imaginary element. In `omega`, functions `cos` and `sin` are called. These are realized by the corresponding OpenCL function calls. Depending on the GPU, the computation of trigonometric functions can be very slow. We address this with an optimization in Section 3.3.

3.2 Encoding Cooley-Tukey FFT in Lift

The major difference between encoding the Stockham and the Cooley-Tukey FFT is that the latter begins with a bit-reversal pass. Because of the similarities between the FFT versions, we show only how to encode the bit-reversal pass. In matrix-vector notation, the bit-reversal pass is a product of permutation matrices that is transposed:

$$\text{bitRev}\ v = \underbrace{((I_{\frac{n}{L_u}} \otimes \Pi_{r_u, L_u}) \cdots}_{\text{permute}} \underbrace{(I_{\frac{n}{L_1}} \otimes \Pi_{r_1, L_1}))^T}_{\text{permute}} \cdot v$$

```
1 def bitRev(rs) = λ(v =>
2   ctPerm(r1, ru * ... * r1)
3   <<: ...
4   <<: ctPerm(ru−1, ru * ru−1)
5   <<: ctPerm(ru, ru) <<: v)
```

To encode that a transposed bit-reversal matrix is multiplied, the order in which the radices `rs` are used is reverted [20]. Function `ctPerm` implements the Kronecker products $I_{n/L_q} \otimes \Pi_{r_q, L_q}$ with an identity matrix as the left operand. For this, the implementation makes use of function `kronIleft`, that is derived analogously to `kronIright`. Function `kronIleft` expects as a parameter an implementation of the multiplication of the matrices Π_{r_q, L_q} with a vector. We make use of the equality $\Pi_{r,L} = \Pi^T_{L/r,L}$ and use `multPiT` to encode:

$$\text{ctPerm}\ v = (I_{n/L} \otimes \Pi_{r,L}) \cdot v$$

```
1 def ctPerm(r, L) = λ(v =>
2   kronIleft(n/L, multPiT(L/r)) <<: v)
```

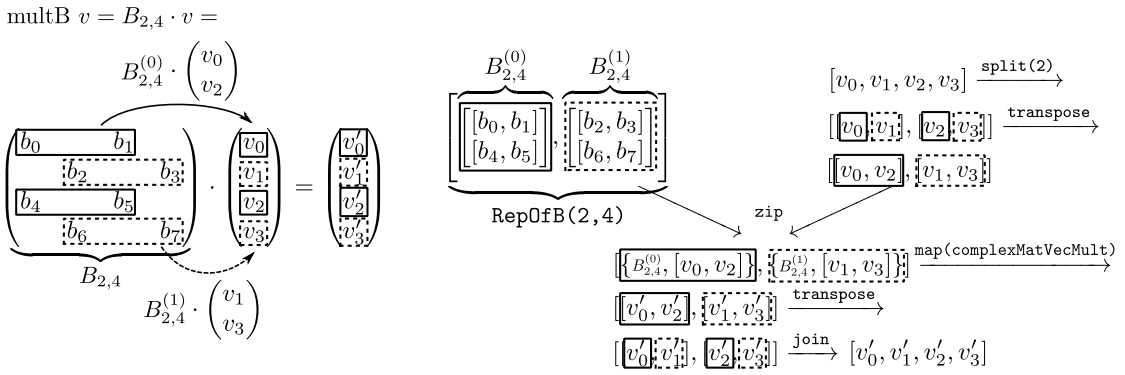

Figure 5. Multiplying a butterfly matrix with a vector.

3.3 Optimizing Butterfly Multiplication with Twiddle Factors

For the butterfly matrix multiplication in Section 3.1.3, we ignored that in matrix-vector notation, the butterfly matrix is made up out of a Kronecker product multiplied with a diagonal matrix holding twiddle factors:

$$\text{multDFTTwiddle } v = (D_r \otimes I_{L_q/r_q}) \cdot \underbrace{diag(I_{L_q/r_q}, \Omega_{r_q, L_q/r_q}, \ldots, \Omega^{r_q-1}_{r_q, L_q/r_q})}_{\text{twiddle factors}} \cdot v$$

```
1 def multDFTTwiddle(r, L) = λ(vs =>
2   join <<: transpose
3     <<: map(multDFT(r) o elemWiseMult)
4     <<: zip(transpose <<: split(L/r) <<: v,
5             RepOfTwiddle)
6
7 def RepOfTwiddle(r,L) =
8   array_2(L/r, r, genTwiddle)
```

Function `multDFTTwiddle` encodes multiplying a butterfly matrix and therefore looks very similar to `multB`. The major difference is that `multDFTTwiddle` encodes two separate operations. First, arrays holding twiddle factors and arrays of grouped vector elements are multiplied element-wise. Then, a matrix-vector multiplication is applied to the results. We represent the matrix of twiddle factors by creating a two-dimensional array `RepOfTwiddle` in which every inner one-dimensional array represents a different $\Omega^j_{r,L/r}$. The user function `genTwiddle` computes an element depending on its indices in the two-dimensional array. Function `multDFT(r)` encodes the matrix-vector multiplication $D_r \cdot v$ by passing an array representing D_r and an array representing a vector to function `complexMatVecMult`, encoding complex matrix-vector multiplication.

Often, this decomposition is used in practice, because `multDFT(r)<<:v` can be optimized by hand. However, generating a straightforward implementation leads to additional arithmetic operations, because twiddle factors have to be multiplied on top of the matrix-vector multiplication. A user function expressing a hand-optimized multiplication of a small D_r with a vector, has to return a value of array type which is currently not possible in the Lift system. We are able to make use of this decomposition with another common optimization in which a table of twiddle factors is precomputed, thereby we avoid to call library functions that compute sine and cosine, which can be very expensive depending on the hardware support. This optimization is used by FFTW and the clFFT OpenCL library, for example. By expressing the above decomposition of the butterfly matrix multiplication, we enable the precomputation of the elements in `RepOfTwiddle` during code generation.

3.4 Encoding Hierarchical FFT

We now investigate how to reuse the previously introduced abstractions to express the hierarchical FFT variant [2].

For the non-hierarchical FFTs and for larger choices of radices, the matrix-vector multiplication becomes inefficient in terms of arithmetic operations as well as the amount of registers used. Additionally, the global memory access patterns depend on the chosen radices, influencing the number of coalseced memory accesses. An alternative to performing a matrix-vector multiplication to compute a DFT for large radices is to recursively compute an FFT based on smaller radices that are better suited for matrix-vector multiplication. For these hierarchical FFTs there is a trade-off with guaranteed coalesced memory accesses and less passes but with additional copies and the necessity for matrix transpositions.

Figure 6 depicts an example of the hierarchical FFT variant:

1. Group the input vector into chunks of size n_1, transpose the resulting matrix and write the result into memory. This forms a $n_1 \times n_2$ matrix which has a row-major layout in memory.
2. Map a function encoding a FFT variant over every row of the matrix. Again write the results are into memory.
3. Multiply a matrix consisting of twiddle factors element-wise with the result from step 2. Transpose the resulting matrix to form a $n_2 \times n_1$ matrix and write it into memory again.

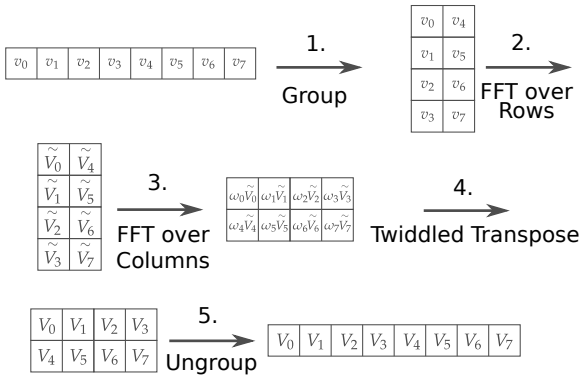

Figure 6. Steps of the hierarchical FFT

4. Repeat step 2 on the transposed matrix.
5. Transpose the matrix a last time, write into memory and ungroup the rows to create the FFT result vector.

These five steps are executed in sequential order. Therefore, we encode every step in a separate Lift function in order to later generate a different OpenCL kernel for every step. Steps that transpose the matrix are implemented with tiling an optimization described in [16].

In this section, we have shown how to build and compose abstractions that closely resemble the mathematical notation for FFTs, in Lift. A Lift function that encodes a complete FFT pass is composed of smaller Lift functions or parallel patterns. In the next section, we show how OpenCL code is generated from this function.

4 Generating Optimized OpenCL code

In this section, we explain how OpenCL code is generated from the high-level programs introduced in Section 3. Crucially, we describe how the abstractions built to closely reflect the mathematical notation of FFTs are compiled away.

Generating Stockham OpenCL Code In Lift, a Stockham pass is represented as a composition of functions. The Lift compiler inlines all of these functions and works directly on the underlying pattern-based representation. Functions in Lift facilitate abstractions without introducing a cost for the generated code. After inlining all functions, a Stockham FFT pass is represented in Lift as shown in Listing 2.

Not all patterns in Lift produce OpenCL code directly. Some patterns, such as *join* or *split* only influence how successive patterns read their input from memory. Therefore, these *data layout patterns* [19] only influence index computations in the generated OpenCL code. Code is only directly generated for the *map* and *reduce* patterns. For pattern *map*, OpenCL code is only generated when for the nested function code is generated as well. No code is generated when the function nested in *map* only consists of data layout patterns, like it is the case for the *map* in line 12. To generate

```
1  def stockPass(r, L, n) = λ(v =>
2   join <<: transpose <<: map(λ(v =>
3     join <<: transpose <<: map(λ((v, RepOfB_j) =>
4       map(λ(rowB_j =>
5         reduce(+,{0,0}) o map(*) o zip(rowB_j,v))
            )
6       <<: RepOfB_j))
7     <<:zip(transpose<<:split(L/r,v),
8           array_3(r,r,L/r, genRepB))))
9   <<: transpose <<: split(n/r)
10  <<: join <<: transpose
11  <<: map(join o transpose o split(r))
12  <<: transpose <<: split(n/L) <<: v )
```

Listing 2. Inlined high-level Stockham FFT pass in Lift.

code, an expression is rewritten into a low-level form, where the *map* patterns have been rewritten into OpenCL specific variations such as *mapGlobal*, that directly correspond to a specific `for`-loop in OpenCL, distributing the work across global work-items[1].

Naively rewriting the above code into a low-level form would result in an OpenCL kernel containing four nested loops that correspond to the four *map* patterns in lines 2–6 which describe the computation of multiple matrix-vector multiplications of complex values. The matrix-vector multiplication is shown in line 6. It is encoded using the `zip`, `map(*)`, and `reduce(+,{0,0})` patterns.

To avoid the four-level deep nesting, we apply the following rewrite rule that rewrites two nested *map* patterns into an expression with a single *map*:

```
1  map(λ(x =>
2    g <<: map(f) <<: zip(x, y)) <<: xs
3                    ⟼
4  λ(x => map(g) <<: split(n) <<: map(f)
5    <<: zip(x, pad(0,(m-1)*n,wrap,y)) )
6    <<: join <<: xs
```

This rule is usable when a function is applied to a each element of a two-dimensional array, where each row has been zipped with a fix array. It captures the intuition that instead, the function can be applied to a flattened two-dimensional array that is zipped with an array in which the elements of the fix array are repeated. On the left-hand side of the rule, a *zip* is nested in the outer *map*. *Zip*'s argument *y* is repeated for each row of the outer array. In the rewritten expression, the repetition is achieved using *pad* (introduced in [10]) and *wrap* that enlarges arrays by repeating its values.
The reduction and map patterns used in the matrix-vector multiplication can be fused to generate only a single loop using the following rule (from [17]):

```
1  reduce(f,id) <<: map(g) <<: xs
2                  ⟼
3  reduceSeq(λ((acc,x) =>
4    f(acc, g(x))),id) <<: xs
```

[1] *Work-item* is OpenCL's terminology for a thread.

```
1  def stockPass(r, L, n) = λ(v =>
2   join <<: transpose
3   <<: λ(v => map(join o transpose) <<: split(n)
4    <<: mapGlobal(λ((v, RepOfB_j) =>
5      mapSeq(λ(rowB_j =>
6        reduceSeq(multAndSum,{0,0})
7        o zip(rowB_j,y)), RepOfB_j)
8     )) <<:zip( transpose<<:split(L/r) <<: v,
9                pad(0,(m-1)*n,wrap,
10                 array_3(r,r,L/r, genRepB)) )
11  ) <<: join <<: transpose <<: split(n/r)
12  <<: join <<: transpose
13  <<: map(join o transpose o split(r))
14  <<: transpose <<: split(n/L) <<: v )
```

Listing 3. Rewritten Stockham FFT pass using low-level patterns

```
1  kernel void fftPass12Radix2(
2       const global complex *restrict v,
3       global complex *out) {
4   for (int id=get_global_id(0);id<8388608;
5      id += get_global_size(0)) {
6    for (int row = 0; row < 2; row += 1) {
7    complex acc = {0.0, 0.0};
8     for (int col = 0; col < 2; col += 1) {
9      acc = multAndSum(acc, {
10       v[((id / 2048) + (4096 * col)
11              + (8192 * (id % 2048)))],
12       genRepB_j((id%2048),row,col,2048,2,2)});}
13    out[((id/2048) + (4096*(id % 2048))
14                     + (8388608*row))] = acc
                           ;}}}
```

Listing 4. OpenCL code for a Stockham pass with radix $r = 2$, $L = 2048$ and $n = 16777216$

```
1  def fftOver(n, nestedFFT) = λ(v =>
2    join <<: mapWorkgroup(
3      toGlobal(mapLocal(id)) o  nestedFFT o
4      toLocal(mapLocal(id)) )
5        <<: split(n) <<: v)
```

Listing 5. Rewritten low-level Lift expression for steps *2. FFT over Rows* and *4. FFT over Columns*.

It is interesting to note that the resulting reduction operator is not associative and, therefore, the reduction has to be performed sequentially.

Together with Lift's lowering rewrite rules (as described in [17]) which rewrite *map* patterns into OpenCL-specific counterparts, we obtain the rewritten Lift expression in Listing 3.

The expression now consists only of data layout patterns and the patterns *mapGlobal*, *mapSeq*, and *reduceSeq* which directly correspond to OpenCL code snippets. Generating code is, therefore, straightforward with the exception of handling the data layout patterns. For these, a compiler-internal data structure called *views* is created that is used to compute the array indices. Detail of the code generation process are described in [19]. Listing 4 shows the OpenCL code generated from the Lift program shown above.

In Listing 4, the OpenCL kernel consists of three nested `for`-loops in lines 4–8 that correspond to the *mapGlobal*, *mapSeq* (both line 5) and *reduceSeq* (line 7) in the rewritten Lift program. The indices in the OpenCL kernel have been influenced by the many data layout patterns visible in the Lift program. To generate reasonably concise indices, Lift performs a number of arithmetic simplifications. In [19], it is shown that these simplifications are often crucial to achieve high performance. In the context of FFT, Lift's capability to compile complex sequences of data layout patterns into concise and efficient indices is crucial to generate efficient OpenCL kernels like the one shown in Listing 4.

Generating Hierarchical FFT OpenCL Code We apply the same rewriting process to passes of the hierarchical FFT and the non-hierarchical FFT functions within them with different rewrite rules to make use of local memory. There are two groups of steps that are similar: *1. Group*, *3. Twiddled Transpose*, *5. Ungroup* and *2. FFT over Rows*, *4. FFT over Columns*. The computations in the steps in the first group are similar as these are implemented using a matrix transposition optimized with tiling in local memory. The transposition in the first step ensures that coalesced memory accesses are possible in the following step, the same is true for the third step. The Lift functions for step 2 and 4 are similar as they both encode applying the nested FFT. The functions are rewritten into a lower level form explicitly encoding that the nested FFTs are computed by work-groups. Listing 5 shows the rewritten expression for these steps.

In line 5 the input vector is interpreted as a matrix by splitting it into rows of size n. The pattern *mapWorkgroup* in line 2 indicates that work-groups apply a function to the rows of said matrix. The *toLocal* and *toGlobal* patterns represent writing the result of the evaluation of an expression into local or global memory respectively. Therefore, the function in *mapWorkgroup* first copies elements of the matrix row into OpenCL local memory by mapping the *id* function to every element using *mapLocal* wrapped into *toLocal*. Then, it computes the FFT using the work-items of the work-group and writes the result back from local memory into global memory. We note that the local memory is much more limited in size than global memory and for large input vectors it can be impossible to find matrix dimensions $n_1 \times n_2$ such that both dimensions fit into local memory completely. In this case, we choose one dimension o be small enough to fit into local memory and recursively apply the hierarchical FFT to the dimension that is too large. Hence, the input vector is not treated as a matrix but as a cube with dimensions $n_1 \times n_2 \times n_3$.

5 Experimental Evaluation

In this section, we experimentally evaluate the performance of the GPU code generated for FFTs using our Lift approach. We are interested in comparing the runtimes of large FFTs – where the large numbers of GPU cores are fully exploited – to professionally developed FFT libraries for GPUs.

Hardware Setup We use two different GPUs to conduct our experiments. One is an Nvidia Kepler K20c GPU with driver version 384.145. The GPU's maximum clock frequency is 705 MHz and it contains 13 SMXs (Streaming Multiprocessors) with overall 2496 streaming processors. There are 4.631 GiB of off-chip global memory available.The other GPU is a Radeon RX Vega 64 GPU with driver version 2574.0 (HSA1.1,LC). The GPU's maximum clock frequency is 1630 MHz and it contains 64 CUs (Compute Units) with overall 4096 stream processors. There are 7.892 GiB of off-chip global memory available.

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.

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 has a size of $2^{24} = 16777216$ bytes and consists of randomly generated values. The number of arithmetic operations in the algorithms does not depend on the values of the input vector. Hence, there is no risk of large variation in the performance results for different values. The Lift functions encoding FFT passes are parameterized over a statically known radix and a statically known size of intermediary DFTs computed by the previous pass. We generate a specialized kernel for every FFT pass and their parameters. The choice of parameters depends on the input size which therefore has to be known statically as well. This restriction means that Lift generated FFT kernels can not handle arbitrary input sizes and that new kernels must be generated for a new input size. However, in practice the size of the input vector is usually known statically, and library generators such as Spiral or an OpenCL kernel generator such as clFFT generate specialized code for specific input sizes as well.

We use ATF [15] – a generic auto-tuning framework – to tune the number of work-items and work-groups that execute the kernel in order to find a better-performing configuration. Every kernel is tuned for either one hour or until the best runtime has not improved for the last 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.

Choosing Radices As mentioned in Section 3 the choice of radices has an important impact on the hardware's ability to coalesce memory accesses and the amount of passes needed to compute an FFT. The hierarchical FFT variant offers the possibility to avoid these problems. Nonetheless, the choice of radices impacts the performance of the hierarchical FFT variant as well. The nested non-hierarchical FFT passes access local memory where bank conflicts need to be kept low. A poor choice of radices increases the amount of bank conflicts. We generate code for different choices of radices for non-hierarchical and hierarchical FFTs. The input size chosen for the evaluation is a power of 2 and 4 and with this, the non-hierarchical FFT passes are based on radices of 2 or 4. These radices are considered optimal in terms of the number of arithmetic operations [5]. Additionally, we evaluate the performance for code generated for radix 8 that reduces the amount of passes needed and increase the computational load of threads. For the hierarchical FFT the input vector is too large to fit entirely into local memory. Therefore, we apply the hierarchical FFT recursively as described in Section 4 by choosing to group the vector in step 1 of the hierarchical FFT in groups of 256. Radices might consist of 2s, 4s only or of 8s with 2s or 4s. Radices larger than 8 cannot be tested at the moment because the generated OpenCL kernels for these choices use too many registers.

Figure 7 shows the runtime measurements for the non-hierarchical and hierarchical variations of FFT on the Kepler K20c GPU. For the non-hierarchical FFTs, only using radix 4 leads to the best performance results for both Stockham and Cooley-Tukey. Although, radix 8 is not an optimal choice in theory, kernels using only this radix perform better than the theoretically optimal radix 2. This is probably due to the fact that the additional work for the larger radices is hidden by more efficient memory accesses. For the hierarchical FFTs, the best results occur when choosing the radices $8^{12} \times 4$. We assume that this is due to the fact that for faster memory accesses to local memory, more arithmetic operations hide memory accesses more efficiently. For radices 2^8, the amount of bank conflicts is probably much larger than for the other versions which explains the big gap in performance.

Performance of Non-Hierarchical vs Hierarchical The hierarchical FFT offers a trade-off. On the one hand it guarantees coalesced memory accesses and a fixed set of steps, therefore less synchronization, but on the other hand it requires additional copies and memory accesses. When comparing the experimental results in figure 7 for non-hierarchical and hierarchical FFTs the non-hierarchical FFTs are much faster (around 3× for for the fastest versions; please note the different scales on the y-axis). One of the main reasons is that the chosen radices for non-hierarchical FFTs allow for coalesced memory accesses. Therefore, one of the main advantages of the hierarchical FFTs disappears and the extra copies to local memory are only additional work. As the kernel runtimes are

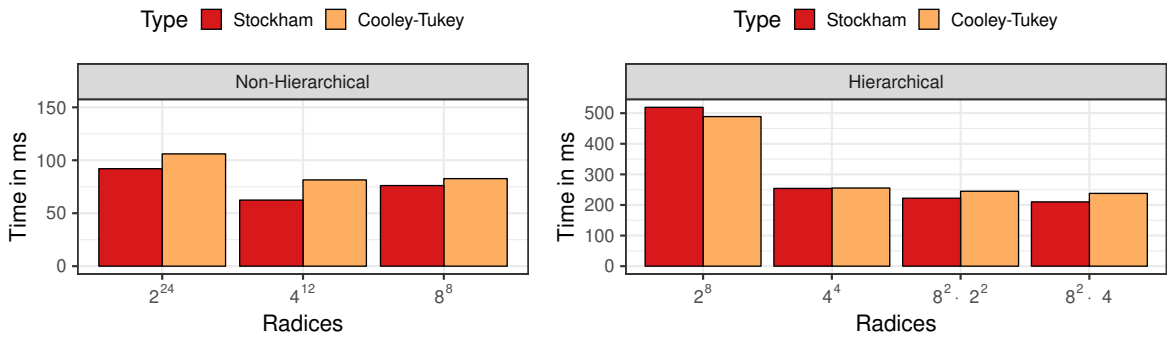

Figure 7. Runtime in milliseconds of Lift generated code for different radices.

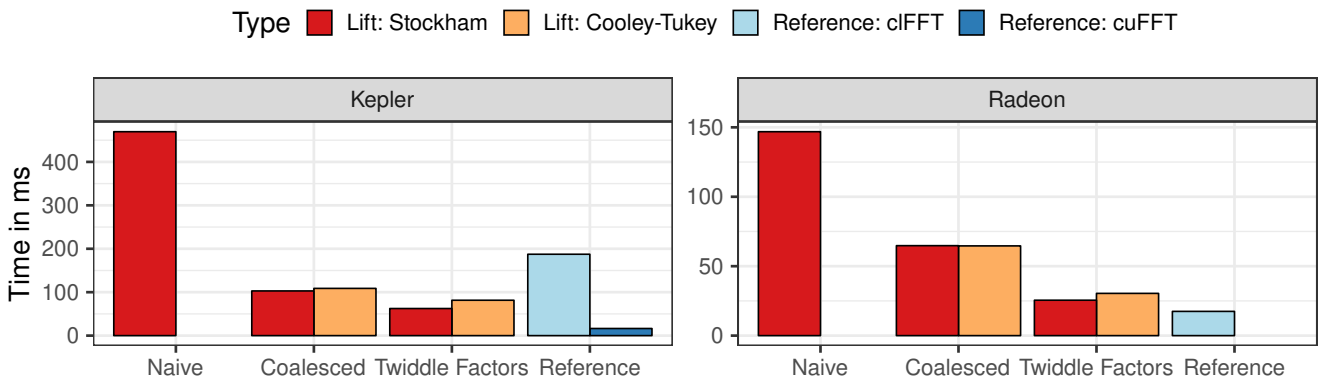

Figure 8. Runtime in milliseconds of Lift generated code (red and orange) vs. AMD's clFFT (light blue) and Nvidia's cuFFT (dark blue) on Kepler K20c and Radeon RX Vega 64 GPUs.

sufficiently long the overhead of individual OpenCL kernel launches required for each step of the hierarchical FFT are neglectable. With these results, we do not investigate the hierarchical FFT further and will focus on the performance of non-hierarchical FFTs compared to high performance library implementations.

Performance Comparison with the Reference Libraries clFFT and cuFFT We compare the performance of a number of Lift generated OpenCL kernels with two-highly tuned reference library implementations. The cuFFT library by Nvidia only works with Nvidia GPUs. The clFFT library by AMD is written in OpenCL. Its FFT routines are mainly optimized for AMD GPUs, but the library works with any OpenCL device. The correctness of clFFT implementations depends on launching specific numbers of work-groups and work-items, which prevents us from tuning them similarly to our own kernels.

For Lift, we measured the performance for three different versions: 1) A *naive* version of the Stockham pass, as discussed in Section 4; 2) versions Stockham and Cooley-Tukey where we have ensured in the code generation via rewriting that all memory accesses are *coalesced*; 3) versions of Stockham and Cooley-Tukey where the multiplication of the Butterfly matrix has been optimized with *twiddle factors* as described in Section 3.3.

Figure Figure 8 shows the runtime measurements for both GPUs. We can see for both devices that the naive version of the Stockham FFT pass generated by Lift performs poorly. This is due to the accesses into the global memory which are not coalesced. For the Cooley-Tukey implementation, the naive version naturally results in coalesced memory accesses. It is well-known that coalescing memory accesses is an important optimization for Nvidia's and AMD's GPUs, as it allows the hardware to combine memory accesses that are simultaneously issued by multiple work-items into a single memory access. On the Kepler K20c, there is a 4.5× performance benefit by ensuring coalesced memory accesses (2.2× on the Radeon RX Vega).

The optimization using twiddle factors is beneficial on both GPUs with a performance benefit of 1.6× on Nvidia and 2.5× on AMD. The Stockham FFT implementation benefits slightly more than Cooley-Tukey.

Compared to the reference library implementation, the Lift generated code performs well compared to the OpenCL implementation but less well compared to cuFFT. On the Nvidia GPU, Lift comfortably beats clFFT (by 3.0×) but cuFFT is still significantly faster (3.8×). On the AMD GPU, the best Lift generated version using the twiddle factor optimization comes close to achieving about 0.68× of the performance of the optimized clFFT library.

The performance gap between Lift and clFFT on the AMD GPU is due to highly tuned functions for small FFTs in the clFFT implementation that update the output array in-place and significantly reduce the amount of arithmetic operations in a kernel. In Lift, this optimization would require significant changes to how user functions work, if a domain specific FFT primitive is to be avoided. At the moment, a user function can only return scalar result values, but would have to be able to describe in place computations with arrays as the return type. Due to the functional nature of Lift this is not a change that can be introduced straightforwardly and will be subject to future research.

We cannot give a clear explanation for why cuFFT is so much faster than Lift, because of the proprietary nature of this library. However, function names and parameters shown in the CUDA profiler suggest that specialized functions for radices of large sizes (i.e., 32 or even bigger) are used. This problem might be addressed by changing how user functions work as well.

6 Related Work

Code Generation Approaches Spiral [7] is a code generation system that automatically generates high-performance code for a range of computational kernels which includes Fast Fourier Transforms. Code for CPUs, GPUs and FPGAs has been generated. Spiral is based on a DSL that was developed to be linear transform specific. Recently, it has been extended to express other applications as well. In Spiral, code is generated for specific computational kernels that are defined using the DSL. In contrast, Lift supports a broad range of applications allowing the generation of high-performance code for combinations of computational kernels.

Delite [3] is a framework for implementing DSLs that are mapped to a small set of patterns that statically optmized and compiled. Static device-specific optimizations are implemented for separate platforms, leading to a lack of performance portability. Lift tackles this problem by encoding optimizations in an extensible system of rewrite rules.

SkelCL [18] and SkePU [6] are algorithmic skeleton libraries promoting high-level programming with parallel patterns. SkePU is a recent C++ framework that focuses on multi-core and multi-GPU. Similar to Delite, the optmizations introduced in the source-to-source translation have to be implemented for separate platforms leading to a challenge of performance portablility.

Other code generation approaches such as Halide [14] or TVM [4] have been shown to be efficient in their respective fields. In Halide the algorithmic descriptions of image processing applications are decoupled from their implementations. Descriptions of optimizations are provided to determine how to generate code. TVM is a compiler stack for deep learning applications. Abstract schedules consisting of TVM specific primitives are automatically optimized and compiled into efficient code for different back-ends. Similarly to Spiral, both these frameworks are domain specific and not as generic as Lift.

High Performance FFT Libraries Nvidia's cuFFT library [12] offers professionally optimized Cooley-Tukey based FFT implementations for many combinations of radices using CUDA. AMD's clFFT library [1] generates OpenCL kernels for specific FFT sizes. The performance of both libraries is platform specific and hardware changes make new manual optimizations necessary.

FFT Algorithms for GPUs Previous work on FFT for GPUs has been published in [9, 11] which presents FFT implementations based on the Stockham FFT expressed in a low-level language focusing on optimizing memory accesses and minimizing arithmetic operations in a hardware specific way. In Lift, FFT variations are expressed at a high-level, allowing exploring parallelism and memory optimizations separately and maintaining performance portability.

7 Conclusions

In this paper, we have shown how to encode FFTs by building high-level abstractions based on a set of generic parallel patterns in the Lift language. This approach results in abstractions that are derived from and closely resemble mathematical definitions for FFTs. We have shown how the Lift performance-portable code generator compiles these abstractions away. Our experimental results have confirmed that our approach achieves performance better than AMD's OpenCL implementation clFFT on an Nvidia GPU. Nvidia's highly optimized cuFFT implementation still performs better on their GPUs.

Acknowledgments

The first author was supported by an HPC-Europa3 travel grant. We would like to thank NVIDIA for providing hardware that was used in this research. Finally, we would like to thank the entire Lift team for their development efforts.

References

[1] AMD. 2018. clFFT. https://github.com/clMathLibraries/clFFT
[2] David H Bailey. 1990. FFTs in external or hierarchical memory. *The journal of Supercomputing* 4, 1 (1990), 23–35.
[3] Kevin J Brown, Arvind K Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A heterogeneous parallel framework for domain-specific languages. In *PACT*.

IEEE, 89–100.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* (2018), 1–15.

[5] James Cooley and John Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965).

[6] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming* 46, 1 (2018).

[7] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. 2018. SPIRAL: Extreme performance portability. *Proc. IEEE* 106, 11 (2018), 1935–1968.

[8] Matteo Frigo and Steven G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *ICASSP*. IEEE, 1381–1384.

[9] Naga Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. 2008. High Performance Discrete Fourier Transforms on Graphics Processors. In *SC*. IEEE/ACM.

[10] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with Lift. In *CGO*. ACM.

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

[12] Nvidia. 2018. CUDA cuFFT. https://developer.nvidia.com/cufft

[13] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[14] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[15] Ari Rasch, Michael Haidl, and Sergei Gorlatch. 2017. ATF: A Generic Auto-Tuning Framework. In *HPCC/SmartCity/DSS*. IEEE, 64–71.

[16] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. 2016. Performance portable GPU code generation for matrix multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM, 22–31.

[17] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *ICFP*. ACM.

[18] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL - A portable skeleton library for high-level GPU programming. In *HIPS*. IEEE.

[19] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: a functional data-parallel IR for high-performance GPU code generation. In *CGO*. IEEE.

[20] Charles Van Loan. 1992. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics.

[21] Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. In *ICLR*.