Code Generation from Functional to Imperative: Combining Destination-Passing Style and Views

Zhitao Lin

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science

School of Computer Science
McGill University
Montréal, Québec, Canada

August, 2022

Abstract

Programming in low-level imperative languages provides good performance but is error-prone. In contrast, high-level functional programming is usually free from low-level errors but performance suffers from costly abstractions. To benefit from both worlds, approaches like Lift compile from high-level functional programs to high-performance imperative code. However, problems such as removing high-level abstraction costs and handling lazy operations are not tackled ideally, which makes such compilation remains an open challenge.

This thesis presents an approach to compiling a high-level array-based functional Intermediate Representation (IR) into high-performance imperative code. It combines the existing work on Destination-Passing Style (DPS) with the Lift *views* system by extending the notion of view to destinations. Destination views can be seen as lazy operations that work in reverse and affect how data is being produced into memory, rather than how data is being consumed.

This approach produces imperative code that existing techniques cannot produce. The evaluation shows that the code produced outperforms the existing DPS approach on several real-world workloads when targeting sequential CPU code. The thesis also demonstrates how destination views can be used to generate high-performance stencil code on Graphics Processing Units (GPUs), by encoding the 2.5D tiling optimization in a functional style.

Abrégé

La programmation dans des langages impératifs de bas niveau offre de bonnes performances mais est sujette aux erreurs. En revanche, la programmation fonctionnelle de haut niveau est généralement exempte d'erreurs de bas niveau, mais les performances souffrent d'abstractions coûteuses. Pour bénéficier des deux mondes, des approches comme Lift compilent des programmes fonctionnels de haut niveau en code impératif hautes performances. Cependant, des problèmes tels que la suppression des coûts d'abstraction de haut niveau et la gestion des opérations paresseuses ne sont pas résolus de manière idéale, ce qui fait qu'une telle compilation reste un défi ouvert.

Cette thèse présente une approche pour compiler une IR fonctionnelle basée sur un tableau de haut niveau en code impératif haute performance. Il combine les travaux existants sur DPS avec le système Lift *views* en étendant la notion de vue aux destinations. Les vues de destination peuvent être considérées comme des opérations paresseuses qui fonctionnent en sens inverse et affectent la manière dont les données sont produites en mémoire, plutôt que la manière dont les données sont consommées.

Cette approche produit un code impératif que les techniques existantes ne peuvent produire. L'évaluation montre que le code produit surpasse l'approche DPS existante sur plusieurs charges de travail réelles lors du ciblage du code CPU séquentiel. La thèse montre également comment les vues de destination peuvent être utilisées pour générer du code stencil hautes performances sur les GPUs, en encodant l'optimisation du carrelage 2.5D dans un style fonctionnel.

Acknowledgements

Words cannot express my gratitude to my supervisor Christophe Dubach. This endeavor would not have been possible without his enduring patience and invaluable feedback. Thanks should also go to my labmates for their help and support. Last but not least, I would like to express my deepest appreciation to my mother and my girlfriend for their support of my studies over all these years. Their belief is my motivation during this process.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

3Add	Three Vectors Addition.
ANF	A-Normal Form.
BA	Bundle Adjustment.
Cross	Cross Product.
DPS	Destination-Passing Style.
GMM	Gaussian Mixture Model.
GPU	Graphics Processing Unit.
HPC	High Performance Computing.
HT	Hand Track.
IR	Intermediate Representation.
MM	Matrix Multiplication.
PRE	Partial Redundancy Elimination.

Chapter 1

Introduction

With increasing performance and efficiency in computer hardware, compute-intensive applications such as deep learning have been flourishing in the last decade. Nevertheless, the rapid growth in the computer hardware performance that Moore's Law promised may now come to an end [18]. As a result, it is increasingly important for software developers to deliver efficient implementations for those compute-intensive workloads. Low-level imperative languages, such as C and OpenCL, are suitable tools for fulfilling this goal. However, programming in these languages is error-prone, hard to maintain and requires expert knowledge due to the intertwining low-level details, such as explicit memory management and the lack of high-level operations and data structures. The functional approach, on the other hand, is a more productive way to compose maintainable and efficient programs as low-level details are abstracted away. But the presence of high-level abstraction may lead to runtime overheads that slow down the program. Thus, generating low-level imperative code from a high-level functional IR seems to be a feasible solution for the dilemma between the functional and the imperative and benefits from both of them [37, 16, 19].

1.1 Challenges

This thesis presents a similar approach to Lift [37] for generating low-level imperative code from a high-level array-based functional IR.

This thesis focuses on three major problems related to compiling functional code to imperative code. First, functional programs leave memory allocations and operations implicit, while they are explicit in low-level languages such as C. Secondly, high-level array-based functional languages express computations through the composition and nesting of high-level primitives, such as *Map* and *Reduce*. For example, $Reduce(+, 0, arr)$ is used for summing the array *arr* and corresponds to a for-loop in imperative code. These high-level primitives must be translated to imperative loops with explicit array indexing operations. Finally, high-level primitives may result in intermediate data structures and slow down the performance.

The first challenge can be mitigated using DPS [27, 35] to perform memory management in a functional way. DPS is very common in many imperative languages, where the caller is responsible for allocating space in the memory for the result of its callee.

The second issue can be resolved by lowering each array operations primitive into an equivalent for-loop construct with index operations explicit. This approach is taken by Lift [37] for instance.

Intermediate data structures are a critical concern, and this is where the main contribution of this thesis lies. This thesis uses the concept of *views* [37] to address this problem so that certain primitives, such as *Zip* or *Split*, do not directly produce their results in memory. Instead, they behave as *lazy* operators, and subsequent operations will have a modified *view* of the data. Meanwhile, the coupling of views with DPS offers a unique opportunity: views can be considered for both source and *destination*. As we will see, this brings extra optimization opportunities where concatenation, for instance, can be handled efficiently.

The evaluationas shows the presented approach outperforms the prior work on DPS both in terms of runtime performance and memory consumption on sequential code. As a GPU

use-case, the thesis also shows how 2.5D tiling, a common optimization performed for High Performance Computing (HPC), can be expressed elegantly in a functional style.

1.2 Goals

This thesis strives to provide a compiler for generating efficient imperative code from the high-level array-based functional IR. Also, it endeavors to provide a step-by-step formalization of the lowering process from the functional to the imperative and to generate efficient imperative code.Moreover, it aspires to combine the concept of views and DPS to introduce extra optimization opportunities that the prior works like Lift cannot provide.

1.3 Contributions

In summary, this thesis makes the following contributions:

- It presents a formalized approach that translates an array-based functional IR into efficient imperative code using a multi-level IRs design that incorporates functional and imperative details.
- It describes an approach to remove unnecessary intermediate data structures from array-based functional primitive by combining DPS and views.
- It introduces the concept of destination views by combining DPS and views and offers extra optimization opportunities for stencil computation.
- It provides a mechanism for automatically exploring the variants of programs at the function level and trade-off the suitable variants for code generation.
- It shows that the generated C code is competitive against hand-optimized implementations and outperforms an existing DPS code generator.

1.4 Publication

Part of this thesis has been published in:

> **Zhitao Lin**, and Christophe Dubach. "From functional to imperative: combining destination-passing style and views." Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY). 2022.

Under the guidance and supervision of Christophe Dubach, Zhitao Lin devised the methods, implemented the prototype, performed the experiments, and wrote the paper. Christophe Dubach also edited and polished the paper.

1.5 Outline

The rest of the thesis is organized into the following chapters:

- Chapter 2 - Background: presents an overview of the concepts used in the rest of the thesis, including lambda calculus, Lift IR and its high-level pattern primitives, DPS, and Views as a technique for removing intermediate data structures.
- Chapter 3 - Related Work: presents a review of the related work on DPS and intermediate data structures removal, both of which are popular research topics over the decades and are important components that form the foundation of this thesis.
- Chapter 4 - IR design with Destination and Views: describes the design decisions for the multi-levels IRs that enable the combination of DPS and views. It first provides an overview of the entire approach and two motivating examples. Then, it formalizes the grammar, types, and primitives used in the IRs throughout the thesis.
- Chapter 5 - Lowering to Imperative Level and Code Generation: It first shows a normalization transformation to ensure that the evaluation order is explicit. Then it proceeds to the transformations for lowering the functional IR to the imperative level IR

while providing memory management in DPS and simplifying views. Finally, it covers the final step for generating imperative code from the imperative level IR.

- Chapter 6 - Automatic Exploration: provides a mechanism for automatically exploring the variants of programs at the function level and trade-off the suitable variants for code generation. For that, it defines the search space for programs in the functional IR, introduces the rewrites rules, and provides two automatic exploration strategies.

- Chapter 7 - Evaluation: focuses on the evaluation of the generated code compared with the prior work on DPS. It first introduces the experimental setup. Then the focus shifts to the result from comparing the code generation strategies proposed in this thesis and the references in 11 real-world benchmarks. Finally, a use-case for the 2.5D tiling optimization is presented to demonstrate the proposed approach can be used for generating OpenCL code for GPU and encoding optimizations.

- Chapter 8 - Conclusion: The concluding chapter covers the summary of this thesis. Then, it proceeds to a critical analysis of the work that has been done, and finally, several avenues where this work can follow in the future.

Chapter 2

Background

This section presents an overview of the concepts used in the rest of the thesis, including lambda calculus, Lift IR and its high-level pattern primitives, DPS, and the concept of views as a technique for removing intermediate data structures.

2.1 Lambda Calculus

The IRs presented in this thesis are based on lambda calculus. The lambda calculus is a formalized system introduced in the 1930s by Alonzo Church for expressing calculation via function abstraction and application. It is universal to express any computable function and thus is equivalent to any Turing machine. The formalism of lambda calculus lays a strong foundation for the functional programming language family [26, 28].

Definition The syntax for lambda calculus is defined recursively and consists of expressions:

$$E := x \mid \lambda x.E \mid E\ E \qquad \textit{variables, function abstraction, function application}$$

A function abstraction $\lambda x.e$ is a definition of an anonymous function that takes argument x as input and returns e. For example, $\lambda x.x$ is the identity function that returns whatever the

input is. An application $f(e)$ stands for the application of function f to input e. Here, the brackets is only used to avoid ambiguity in parsing. Function application is evaluated by substitution of the value of the argument. For example:

$$(\lambda x.x)(y) = [y\backslash x]x = y$$

Here, $[y\backslash x]e$ means substituting variable y to variable x in e.

Free and bound variables The function $\lambda x.t$ binds variable x in expression t, so that x can be used in t. Variables introduced along with the function operator λ are said to be bound in the scope of the function body. All other variables are described as free. For instance, $\lambda x.xy$ has x as bound variable and y as free variable. Free variables of an expression, $FV(e)$, is defined as follows:

$$\begin{aligned} FV(x) &:= x, \textit{ where } x \textit{ is a variable} \\ FV(\lambda x.e) &:= FV(e) - x \\ FV(e\ s) &:= FV(e) \cup FV(s) \end{aligned}$$

Typed Lambda Calculus Typed lambda calculus is a typed formalism with lambda expressions to represent functions and applications [24]. Types are syntactic nature that attached to expressions like $e : \tau$ for expression e with type τ. Typed lambda calculi allow proving certain program properties to prevent unwanted behaviours, such as memory access violations. A simply typed lambda calculus has basic types, such as $IntT$ and $FloatT$, and only one type constructor $\rightarrow$ for constructing function types $\sigma \rightarrow \tau$ where σ and τ are two types. As a result, the identity function specifically for integers in a simply typed

lambda calculus is annotated as:

$$\lambda x.x : IntT \rightarrow IntT$$

System F System F [34, 15] is a more advanced variant of typed lambda calculus. In addition to the simply typed lambda calculus, System F formalizes parametric polymorphsim [40] in the language, *i.e.*, a mechanism for using universal quantification over types.

In a simply typed lambda calculus, there are variables ranging over expressions, functions and function applications. Corresponding, in System F, there are *type variable*s ranging over types, *type function*s annotated as Λ, and *type-function application*s.

A *type-function type* is also introduced in system F, which has the syntax of $\forall\alpha.\tau$, where α is a type variable, and τ is a type where X can be used.

Going back to the identity function example, now a more generic identity function is formalized as:

$$\Lambda\alpha.\lambda x^{\alpha}.x : \forall\alpha.\alpha \rightarrow \alpha$$

With a type-function application with an integer type as an argument, the aforementioned identity function for integer specifically can be retrieved.

For those who are more familiar with imperative programming, Java generic method is a more intuitive example for the usage of type function. In Java, assuming T is a type variable, an identity function can be defined as:

```
1 static <T> T Id(T a) {
2   return a;
3 }
```

System F-sub($F_<$) The IRs presented by this thesis relied on System F-sub($F_<$) [6] which is an extension to System F. In addition to System F, $F_<$ combines parametric polymorphism with subtyping. The support of subtyping is through the introduction of a new type constant

Top, which is the supertype of all types, and subtypes bound on type-function types: $\forall(\alpha <: \tau).\tau'$ where α is a type variable that should be the subtype of τ and can be used in τ'.

The benefit of subtyping hierarchies is that the language can now precisely restrict what subtypes can be used in a particular function. For instance, assuming that $IntT$ and $FloatT$ are subtypes of $ScalarT$, with $F_<$ we can have an identity function that only receives an integer or a float:

$$\forall(\alpha <: ScalarT).\lambda x^{\alpha}.x$$

2.2 Lift IR

The implementation and design of Lift IR [36, 37] greatly influences this thesis. Lift attempts to leverage the benefit of a functional array-based programming language to express high-performance data-parallel computation in various domains, including OpenCL [37], stencil computations [17], OpenMP [31] and FFT [22]. Given the generic and expressiveness of the high-level primitives from Lift, as we will see, the functional level IR adopted by thesis shares many common design elements from Lift.

Lift IR is based on system F as introduced in the previous section and consists of the nesting and combination of high-level primitives. There are two kinds of primitives: algorithmic pattern primitives that represent high-level array-based operations, such as Map and $Reduce$, and data layout pattern primitives that do not perform any computation but only reorganize the data layout, such as $Split$ and Zip.

Algorithmic Pattern Primitives Map and $Reduce$ are two major algorithmic pattern primitives in Lift that perform computation on arrays.

Map is a higher-order function that constructs an array by applying the input function to each element of the input array (α, β, n and m are type variables as used in the rest of this section):

$$Map : (\alpha \to \beta) \to [\alpha]_n \to [\beta]_n$$

For instance, $Map(f, [x_0, x_1, x_2, x_3]) = [f(x_0), f(x_1), f(x_2), f(x_3)]$ where $[x_0, .., x_n]$ is an array constructor. A common-used example for using *Map* is to obtain the square of every element in an array:

$$Map(\lambda x.x * x, arr)$$

Reduce is a higher-order that takes all the elements from an input array, and combines them as well as an initial value using a binary operation to produce a single value. It has the following signature:

$$Reduce : (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha]_n \rightarrow \beta$$

The first argument for *Reduce* is a function *i.e.*, a binary operation that takes two arguments: the first is for the current value, and the second is for the accumulated value. The second argument for *Reduce* is an initial value. The third argument for *Reduce* is an array. *Reduce* applied the input function to the initial value and all elements from the third argument to produce a single value. For instance, $Reduce(f, z, [x_0, x_1, x_2, x_3]) = f(f(f(f(z, x_0), x_1), x_2), x_3)$. A common-used case of *Reduce* is to sum up all elements of an array:

$$Reduce(+, 0, arr)$$

Data Layout Pattern Primitives The data layout pattern primitives are those operations that only change the data layout of the input but do not perform any computation. Here goes an overview of the mainly used data layout pattern primitives in Lift:

- *Split* splits an array into multiple sub-arrays with the following signature:

$$Split: \quad m{:}Nat \rightarrow [\alpha]_n \rightarrow [\alpha]_{n/m}$$

The first argument m is a natural number that represents the length of resulted sub-arrays. The second argument is the array to split. For instance, $Split(2, [x_0, x_1, x_2, x_3]) = [[x_0, x_1], [x_2, x_3]]$.

- $Join$ has the opposite effect to $Split$ by joining sub-arrays into one array:

$$Join : [[\alpha]_n]_m \rightarrow [\alpha]_{n*m}$$

For instance, $Join([x_0, x_1], [x_2, x_3]) = [x_0, x_1, x_2, x_3]$.

- $Tuple$ bundles two inputs together and creates a tuple:

$$Tuple : \alpha \rightarrow \beta \rightarrow (\alpha, \beta)$$

For instance, $Tuple(x_0, x_1) = (x_0, x_1)$ where (x_0, x_1) represents a tuple.

- Zip receives a tuple of arrays and returns an array of tuples by putting the elements in two arrays together:

$$Zip : ([\alpha]_n, [\beta]_n) \rightarrow [(\alpha, \beta)]_n$$

For instance, $Zip(Tuple([x_0, x_1], [y_0, y_1])) = [(x_0, y_0), (x_1, y_1)]$

- Fst and Snd projects a component of a tuple:

$$Fst : (\alpha, \beta) \rightarrow \alpha \qquad Snd : (\alpha, \beta) \rightarrow \beta$$

For instance, $Fst(x_0, x_1) = x_0$ and $Snd(x_0, x_1) = x_1$.

- At projects a component of an array:

$$At : \quad m{:}Nat \rightarrow [\alpha]_n \rightarrow \alpha$$

For instance, $At(0, [x_0, x_1]) = x_0$.

- *Slide* applies a moving window to the input data and can be used for stencil computation where m is for window size and k is for step length:

$$Slide: \quad m{:}Nat \to k{:}Nat \to [\alpha]_n \to [[\alpha]_m]_{(n-m+k)/k}$$

For instance, $Slide(3, 1, [x_0, x_1, x_2, x_3]) = [[x_0, x_1, x_2], [x_1, x_2, x_3]]$ with 3 as the window size and 1 as the step.

2.3 Destination-passing Style

DPS is commonly used in many imperative programs, where the caller is responsible for the memory allocation of the result of its callee. By applying this style, a function will receive a pointer where the function result is stored as one of the parameters. Therefore, the callee does not need to allocate space for storing its result; instead, the responsibility will be taken by the caller. For instance, most of C standard library's functions, *e.g.*, `strcpy`, follows this style by receiving a destination for the resulted string:

```
char* strcpy(const char* source, char* destination);
```

The usage of `strcpy` copies the first argument to the second argument. And the first argument is the source, while the latter is the destination for storing the result.

Code written in DPS would have its return value passed explicitly as an extra argument, *i.e.*, the destination. For example, a function `sum` that sums up three input arguments can be implemented as:

```
1 int sum(int a, int b) {
2         return a + b;
3 }
```

In DPS, a destination `d` is used to storing the function result, and the program becomes:

```
1 void sum(int a, int b, int *d) {
2         *d = a + b;
3 }
```

Using DPS also provides a formalization for handling the memory management problem when compiling high-level functional code to low-level imperative code, as we will see in the rest of the thesis.

2.4 Views

Lift [37], which greatly influences this thesis, introduces the concept of view to remove intermediate data structures. In Lift, a view is a compiler internal data structure for storing information of array access. Functions that only change the data layout are lazy and produce views instead of eagerly materializing the result into memory. In other words, the data reshaping operations only happen when the data is accessed in a computation.

Two steps are needed to leverage the Lift views to eliminate intermediate data structures: the *view construction* and the *view consumption*.

View Construction In this step, the compiler traverses the IR following the data flow and constructs a view for each node that influences the array access pattern. For example, in the program $Map(\lambda x.Fst(x), Zip(a, b))$: Zip creates a $ZipView$. Map produces an $ArrayAccessView(i)$ to access each element produced by the $ZipView$, where i represents which element of the array is to be accessed. For the input function of the Map, Fst is used to access the first element for the variable x, which creates a $TupleAccessView(0)$ for accessing the first element of a tuple view. Finally, the variable a and b also represent two views

for the actual memory accessing. The resulted views construction will be:

$$TupleAccessView(0, ArrayAccessView(i,$$
$$ZipView(MemoryView(a), MemoryView(b)))$$

View Consumption In this step, an array stack for array access patterns and a tuple stack for tuple access patterns are used for calculating the accumulated effect of views. By examining the constructed view graph, the patterns will be added to the stacks and then used in a first-in-last-out manner.

Going back to the previous example: $TupleAccessView(0)$ pushes 0 to the tuple stack to represent that there is a tuple access to the first element of a tuple view. $ArrayAccessView$ pushes i to the array stack to represent an array access to the ith element of an array view. Then the $ZipView$ pops 0 from the tuple stack and use the information to determine which view should be used next which is $MemoryView(a)$ in this case. Finally, the remaining array access i in the array stack will be used to emit a final index to the memory, which is $a[i]$.

While the Lift view system is simple and effective in removing most of the intermediate data structures, it has two drawbacks. First, the Lift view is implemented by internal data structures in the compiler, and it seems hard for the end-users to control what should be materialized and what should be considered as a view. Secondly, the Lift view is only limited to the source view, and there are other kinds of views called *destination views* that affect the pattern for writing to memory instead of reading. The destination view is crucial for expressing certain efficient imperative programs in a functional style that is impossible when only the source view is available, which is one of the main contributions of this thesis.

This thesis proposes another view system that is inspired by the Lift view. There are two major improvements compared with the Lift view system. First, this thesis expresses views at the language level, so end-users have fine-grained control over them. Secondly,

this thesis combines views with DPS so that the concept of destination view is introduced for generating efficient code (more discussion in the rest of the thesis).

Chapter 3

Related Work

This chapter reviews the related work on code generation from high-level abstractions, Destination-Passing Style and removing intermediate data structures, all of which are popular research topics over the decades and are important components that form the foundation of this thesis.

3.1 Code Generation from High-level Abstractions

Generating low-level imperative code from high-level abstractions allows programs to be written in high-level abstractions so that the end-users do not need to worry about the low-level details. At the same time, it provides good performance by generating low-level imperative code where overheads from high-level abstractions can be eliminated.

The first example is SAC [16] (Single Assignment C), a functional language designed for array-intensive applications and generates C code with support for parallel execution on multiprocessor systems. The fundamental idea in the language design is to keep the language as close as possible to C but still support high-level array-based programming. SAC treats arrays as first citizens and provides a data-parallel skeleton operation With-Loop, which is similar to array comprehension in Haskell. The distinguishing feature of With-Loop is that it can be used for expressing shape-invariant operations, *i.e.*, operations on arrays

whose shape is statically unknown. Optimizations around With-Loop provide the capability of the fusion between With-Loops and avoid creating intermediate data structures.

Another interesting project is Accelerate [7], a domain-specific high-level embedded language in Haskell. Since it is an embedded language, the operations do not directly issue any computation in Haskell. Instead, it builds term trees to represent the computation via higher-order combinators such as *Map*, *Fold*, *Scan*, etc. Based on the term trees, NVIDIA's CUDA code will be generated.

With a very similar idea of using higher-order combinators for expressing computation, Futhark [19] is a purely functional array-based language that seeks to generate OpenCL code. Its core language consists of three higher-order parallel combinators: *Map*, *Reduce* and *Scan*. One of the main contributions of the Futhark languages is that it supports in-place updates on arrays and provides a type system that guarantees that in-place update operations are safe.

The implementation and design of Lift [37] have greatly influenced this thesis. Lift generates efficient OpenCL code from a functional array-based programming language. It has been applied in various domains including OpenCL [37], stencil computations [17], OpenMP [31], FFT [22] and matrix multiplication [38]. As we have seen in section 2.2, compared with Futhark, Lift IR consists of more fine-grained combinators, including *Map*, *Reduce*, *Transpose*, *Slide*, etc. For lowering and optimization, Lift uses a rewrite rule system [36] to derive lower-level constructs from the high-level functional primitives. Lift's fundamental ideas and compilation process can be traced back to Lime [10], an embedded programming language in Java that generates the OpenCL API calls.

3.2 Destination-Passing Style (DPS)

DPS is a programming convenience where the caller is responsible for providing a destination for the callee.

This technique is first applied to remove the dependence constrains on Lisp program for parallel execution [23]. Especially when a statement in the tail of a recursive function uses the result of the recursive call, there will be a dependence constraint where the statement must execute after the subsequence iterations finish. DPS is used to solve this constraint by creating data structures with *holes*, *i.e.*, uninitialized fields, to be used as destinations. Thus, the subsequence iterations can produce and store their result to the destinations. With a similar idea of using DPS to provide the facilities to transform a non-tail-recursive function into a tail-recursive function, a formally functional representation [27] of data structures with a *hole* has been presented. It includes the introduction of hole abstraction, hole application and the usage of linear types [47]. *Tail-recursive modulo cons* [13, 48] is a special case for transforming those non-tail-recursive functions where the recursive call is not in the tail position but a part of data constructors, such as *cons*, into tail-recursive ones. For that, recently, a modular transformation [5] has been proposed with the usage of DPS and the formalization of the hole [27].

Differently, $\widetilde{F}$ (pronounced as F smooth) [35] investigates the application of DPS for memory management in an array-based functional language. As the main inspiration for this thesis, $\widetilde{F}$ has provided a compiler for translating an F#-like language to imperative C code leveraging the principle of DPS. Nevertheless, its highest level IR consists of relatively low-level primitives, such as *Build* and $Ifold$, which correspond to the imperative level IR of this thesis. Instead, this thesis uses *Map*, *Reduce*, and other high-level primitives at the highest level, which comes with two advantages: first, the high-level primitives introduce an extra level of abstraction that allows many rewritings to be applied to these primitives, such as $Map(f) \circ Map(g) = Map(f \circ g)$ and $Map(f, Concat(a, b)) = Concat(Map(f, a), Map(f, b))$. Secondly, while lazy primitives can be easily represented by the low-level primitive *Build*, using *Build* directly discards the necessary information to lower those primitives as destination views that affect where data should be materialized. For example, $Zip(in)$ can be rewritten as $Build(\lambda i.Tuple(in.0[i], in.1[i]))$. But when using the latter representation, it is

hard for the compiler to detect whether this *Build* is corresponding to the *Zip*. As we will see in section 5.3, the information hidden in the high-level primitives, such as *Zip* or *Split*, is needed for lowering them as destination views. *Build*, however, is too low-level to retrieve the required information.

DPS is also comparable to the acceptor-passing translation introduced in DPIA [3] which, just like this thesis, aims to provide a formalized translation from a functional array-based language to low-level imperative code. Acceptor in the acceptor-passing translation essentially corresponds to the destination in DPS. Using data layout primitives to change where to write, as a destination view, is also realized in DPIA through rewrite rules such as (the semantics of *Join* and *Split* have been shown in section 2.2):

$$A := Split(e) \Rightarrow SplitAcc(A) := e$$

On the right-hand side of the $\Rightarrow$, A is an acceptor, and $:=$ assigns the result from $Split(e)$ to the acceptor A. On the left-hand side of the $\Rightarrow$, the program writes the result from e to $SplitAcc(A)$ instead. Here, *SplitAcc* is actually a *Join* so that, instead of splitting the input e, it joins the acceptor A. This process is similar to how destination views are handled, as we will see in 5.3, but DPIA only discusses such reversed effect for four primitives: *Tuple*, *Zip*, *Split*, and *Join*. This thesis, however, supports more primitives as well as their composition as destination views. It also provides fine-grained control for whether a primitive should have such a reverse effect, which is not provided in DPIA.

In summary, rather than being interested in the non-tail-recursive function transformation or the memory management aspect of DPS, this thesis is more focused on the formalization of using DPS to compile a Lift-like language to the imperative code and explores the benefit of introducing the concept of destination to the view system to generate efficient imperative code.

3.3 Intermediate Data Structure

The problem of removing intermediate data structures has been well studied over the years. One of the most prominent techniques is desforestation [46] for eliminating the intermediate data structures of a so-called "*treeless*" form program. Later on, there are more techniques proposed for this very purpose in different contexts. This section provides a thorough review of these techniques and their influences on the method proposed by this thesis.

Shortcuts to deforestation The shortcut to deforestation [14] provides a considerably simpler solution for the same purpose by providing a fusion system between the two combinators: the list catamorphism $foldr$ for consuming lists and the list constructor $build$ for producing lists. The key of this fusion is the following fusion rule that enables the elimination of the intermediate lists: $\forall f\ g\ z.\ foldr\ f\ z\ (build\ g) = g\ f\ z$.

Similar to the $foldr/build$ fusion, the $destroy/unfoldr$ fusion system [41] are proposed with two combinators: $destroy$ and $unfoldr$ for production and consumption respectively. By carefully selecting their implementation, the key of this $destroy/unfoldr$ fusion lies on the rule: $destroy\ g\ (unfoldr\ f\ e) = g\ f\ e$. Compared with $foldr/build$, the $destroy/unfoldr$ fusion has the advantage of handling the combinator that involves accumulating parameters such as $foldl$ and zip-like functions. Examples for expressing accumulating parameters and zip-like functions can be found in the paper [41], which are too lengthy to discuss here.

Stream fusion [8, 9] is proposed to allow the separation of reading, writing, and computation on each element, which overcomes some shortages of the previous work on handling zip, concatenation, and nested lists. The term "stream" may seem confusing since it is sometimes overloaded to represent an infinite sequence, while this term used here is for a terminating sequence. Data type *Stream* takes a step function for producing the next element and an initial state. Given a list, function *stream* constructs a *Stream* as a result. In contrast, function *unstream* takes an *Stream* as input and returns a list. With these con-

structs, the fusion either happens between *Streams* themselve or is based on the rule of $stream\ (unstream\ s) = s$, assuming that $stream \circ unstream$ is the identity on *Streams*.

Recently, Strymonas [21] makes a step further to completely remove the overhead, such as extra function calls or closures, caused by those step functions introduced in the stream fusion system via staging. Staging is a technique to write programs that generate programs and to ensure that the generated programs are well-formed, well-scoped and well-typed. It also supports the combinator *concatMap* and the corner cases arising from the composition of the nested array and zip-like function.

While the aforementioned approaches are effective in removing intermediate data structures, they pay little attention to the combinators' effect on the destination, which is one of the main contributions of this thesis.

View System in Lift Lift [37], greatly influences this work, and introduces the concept of view to remove the intermediate data structures. The implementation details of view system in Lift is already discussion in section 2.4. Compared with the prior work, the view system is a more straightforward way to remove the intermediate data structures in the Lift-like language because programs in such a language are composed with a concise set of array-based primitives, such as *Map*, *Split*, and *Zip*.

However, unlike the prior work where the fusion process is embedded in language level and based on rewrite rules, the process of removing the intermediate data structures in the view system is implemented through the usage of compiler internal data structures. The reason is that it fails to discover, or at least doesn't elaborate, the fact that most of the primitives can be rewritten by *Build*. *Build* is an array constructor and is different from the *build* used in $build/foldr$ fusion [14]. Instead, *Build* has the following signature, where n is for the length of the resulted array and $(Int \rightarrow t)$ is the type of the index function that takes an index as input and returns an element at the index position:

$$Build : \forall n.\forall t.n \rightarrow (Int \rightarrow t) \rightarrow [t]_n$$

Once those primitives are rewritten by *Build*, the key rewrite rule for fusion will be:

$$Build(n, f)[i] \Rightarrow f(i)$$

In addition, the Lift view system doesn't include the concept of destination views which, as we will be seen in the result of the thesis, is necessary for generating efficient code.

The *pull*/*push* Array Using the source and destination view, presented in this thesis, to remove intermediate data structures is more synergistic with the pull and push array approach. A pull array is a delayed representation of an array [11]. It is defined as, which is essentially identical to the definition of the *Build* mentioned before:

$$data\ Pull\ a = Pull\ Length\ (idx \rightarrow a)$$

The data type *Pull* first takes a *Length* standing for the length of the array it is going to produce. And then, it takes an index function for producing the array.

Since it is constructed by an indexing function, the removal of intermediate arrays comes for free. The guarantee of pull array fusion can be traced back to Feldspar [4] and repa [20].

Differently, a push array contains a function for writing the value into memory, and is a more efficient way to represent arrays for array concatenation and writing multiple elements per loop iteration. It is defined as the following in Haskell:

$$data\ Push\ a = Push\ ((idx \rightarrow a \rightarrow CM()) \rightarrow CM())\ Length$$

where *CM* is a compile monad for generating code.

Push array first introduced in Obsidian [43] and is extended in Feldspar [4], Nikola [25], meta-repa [1] and strymonas [21]. The combination of pull and push arrays by defunction-

alizing push arrays [42] suggests the necessity of the coexistence of pull and push arrays in the same library.

In fact, the notions of the source view and destination view, which will be presented in detail in chapter 4, are very similar to pull/push array. However, the approach proposed in this thesis overcomes some of their limitations. First, views are not array-specific but can be generalized to other data structures, *e.g.*, tuples. Secondly, this thesis focuses on the compilation of functional code. With the use of DPS, a destination view will be transformed into a function that returns locations on memory for writing and maintains a direct mapping to imperative code. Push arrays, however, remain too abstract for a straightforward translation to imperative code. Thirdly, expressing nested array operations that involve high-dimensional arrays via push arrays, such as *Split* and *Transpose*, seems complicated. Nevertheless, as we will see later, destination views can express them without obstacles.

3.4 Summary

This chapter has reviewed the related work on DPS and removing intermediate data structures. Especially for those prior work on removing intermediate data structures, less attention has been paid to the effect on the destination side. While the push arrays can be used to express how an array is written into the memory, they cannot be generalized to other data structures, *e.g.*, tuples, and to express nested array operations easily. The next chapters will show how the combination of DPS and *views*, one of the methods for removing intermediate data structures, can overcome the aforementioned drawbacks and provide a framework as a whole to generate efficient code from a Lift-like function IR.

Chapter 4

IR Design with Destination and Views

This chapter describes the design decisions for the multi-levels IRs that enable the combination of DPS and views. It first provides an overview of the entire approach as well as two motivating examples. Then, it formalizes the grammar, types, and primitives that will be used in the IRs throughout the thesis.

4.1 Overview and Example

Figure 4.1 presents an overview of the proposed approach. The input program is expressed at the functional level by composing and nesting high-level array-based functional primitives. Then, the functional level IR is lowered to the imperative level using rewriting, where it is first transformed into a DPS style, followed by the removal of *views*. As we will see, this removal of views allows for the elimination of intermediate data structures. Finally, high-performance imperative code is produced using either a sequential C backend or a parallel OpenCL backend to target GPUs.

The rest of this section provides two examples, vector addition and concatenation, that illustrate the compilation process. These examples will show how views are used to remove intermediate data and produce efficient code.

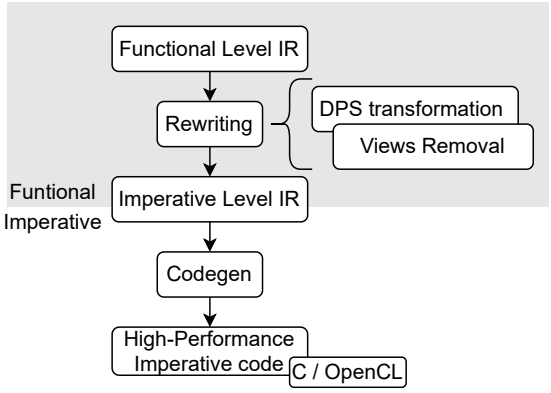

Figure 4.1: Overview of the compilation process from a functional IR to imperative code.

4.1.1 Vector Addition

At the functional level, vector addition is expressed as:

$$\lambda a, b.Map(\lambda x.(x.0 + x.1), Zip(Tuple(a, b)))$$

The first issue to consider is how to ensure that no intermediate data structure is created by Zip and $Tuple$ to generate efficient code. To achieve this, it is expected that Zip and $Tuple$ are removed so that the operands for addition are direct array accesses to a and b.

Zip and $Tuple$ can be eliminated because they are *views* in the language (more details in section 4.2). Views are operations that are lazy where data should not be materialized, *i.e.*, not written to memory. As we will see later, views are removed automatically during lowering, preventing the use of intermediate data structures.

After removing Zip and $Tuple$, the program is still too abstract for generating imperative code. DPS is used to introduce a *destination*: a memory location where data can be written to. With d as the destination, after applying a DPS transformation, the program becomes:

$$\lambda a, b, d.Build(n, \lambda i.Assign(a[i] + b[i], d[i]))$$

Here, *Build* takes a length and an index function to build an array that corresponds to a for-loop in the target C language. As a result, the program now has a straightforward translation to imperative code:

```
1 for (int i = 0; i < n; i++)
2     d[i] = a[i] + b[i];
```

4.1.2 Concatenation

The previous example has shown how the Zip and $Tuple$ simply inform where the addition(+) reads its input so that any unnecessary intermediate data structures are eliminated. In the previous example, these views are called *source views*, since they only affect where and how the next operation should read its input. Now the focus lies on a different type of views: *destination views*.

To illustrate the concept and necessity for destination views, a simple concatenation example is used. First, it is expressed using source views (as indicated by the S superscript) and later we will see the advantage of using destination views.

This example concatenates two arrays, a and b, with length n and m respectively. In the program below, $Map(Id)$ is used to materialize the two concatenated arrays into memory using the identity function:

$$\lambda a, b.Map(Id, Concat^S(Tuple^S(a, b)))$$

After conversion to DPS, the program becomes:

$$\lambda a, b, d.Build(n + m, \lambda i.If(i < n, Assign(a[i], d[i]), Assign(b[i - n], d[i])))$$

And a direct translation to imperative code is:

```
1 for(int i = 0; i < n + m; i++) {
2     if(i < n) d[i] = a[i];
3     else      d[i] = b[i - n];}
```

In a single for-loop, arrays a and b are concatenated and written into destination d. However, this code might suffer from bad performance since a test is performed at every iteration. A more efficient implementation could be:

```
1 for (int i = 0; i < n; i++) d[i] = a[i];
2 for (int i = 0; i < m; i++) d[i + n] = b[i];
```

This second implementation, however, can not be directly derived from the functional expression. The core of the issue is that the location of the *writes* must be modified, rather than the location of the read as in the first example. To solve this problem, the concept of destination view needs to be introduced. A destination view influences, lazily, where an operation should *write* its result.

If we use *Concat* (D superscript) as a destination view, the functional program for concatenation becomes:

$$\lambda a, b.Concat^D(Tuple^D(Map(Id, a), Map(Id, b)))$$

At the imperative level, two *Build* are produced to represent two for-loops. By employing DPS and removal of the destination views, which we will see later in the thesis, the arrays a and b are directly written into the right location:

$$\lambda a, b, d.Build(n, \lambda i.Assign(a[i], d[i]));$$
$$Build(m, \lambda i.Assign(b[i], d[i + n]))$$

Here, the semicolon operator ***;*** is a syntatic sugar for a let expression. For example, $e1; e2$ is equivalent to $let\ tmp\ =\ e1\ in\ e2$ where tmp is never used in $e2$.

The code generator will produce the second C code seen earlier which should lead to better performance. This example has illustrated the benefit of using destination views.

$$
\begin{array}{lr}
E := x \mid 0.0 \mid P \mid & \textit{variables, literals, primitives} \\
\quad \lambda x.E \mid E(E) \mid & \textit{function abstraction, application} \\
\quad \Lambda x.t^T \mid E\langle T\rangle \mid & \textit{type-function abstraction, type application} \\
\quad \textit{Let } x = E \textit{ in } E \mid x = E; E \mid & \textit{let expression} \\
P := \textit{Map} \mid \textit{Reduce} \mid \textit{Tuple} \mid \ldots & \textit{primitive identifiers}
\end{array}
$$

$$
\begin{array}{lr}
T := X \mid \textit{ValueT} \mid \textit{MetaT} & \textit{type variables, value types, meta types} \\
\textit{ValueT} := \textit{DataT} \mid X^T \mapsto T & \textit{data types, type-function types} \\
\quad T \xrightarrow{\textit{EffectT}} T \mid & \textit{function types}
\end{array}
$$

$$
\begin{array}{lr}
\textit{DataT} := \textit{ScalarT} \mid & \textit{scalar types} \\
\quad [\textit{DataT}]_{\textit{NatT}} & \textit{array types} \\
\quad (\textit{DataT}, \textit{DataT}) & \textit{tuple types} \\
\textit{ScalarT} := \textit{IntT} \mid \textit{FloatT} \mid \textit{DoubleT} \mid \ldots & \textit{scalar types}
\end{array}
$$

$$
\begin{array}{lr}
\textit{MetaT} := \textit{NatT} \mid \textit{EffectT} & \textit{natural numbers, effect types} \\
\textit{EffectT} := S \mid D \mid E \mid & \textit{source view, destination view, eager types} \\
\quad \emptyset \mid \textit{EffectT} \ominus \textit{EffectT} & \textit{no-effect, composed effect types}
\end{array}
$$

Figure 4.2: Abstract Grammar for the Functional IR

4.2 Functional Level and Views

The functional level IR shares many common design elements with Lift [37]. It consists of the composition and nesting of high-level functional array-based primitives and is optimized via a set of rewriting rules. One of the main differences with Lift is that the concept of view is embedded as effect types at the functional level IR, which provides end-users with fine-grained controls over whether primitives should be materialized or not. This section presents the expressions, types, primitives available at the function level and the effects types related to views.

4.2.1 Core Language

Expressions and Types Figure 4.2 describes the grammar for the functional IR, which is based on typed lambda calculus with subtyping in the style of System F-sub ($F_<$) [6]. $F_<$ is

discussed in section 2.1 as an extension to System F [34, 15] that supports both parametric polymorphsim and subtyping which allows for an intuitive type hierarchy.

A set of primitives, such as Map and Reduce, provided by the IR are handled specially in the later transformations and lowering. They are essentially implemented as pre-defined type-function abstractions. More details on these primitives will be discussed later.

Data and Function Types are both value types, *i.e.*, they can be used as input for or returned by a function. While it is extensible, the IR support three kinds of data types by default; they are scalar types, array types and tuple types. Notation $[T]_{NatT}$ stands for an array with $NatT$ elements of data type T where $NatT$ is a meta type. Similarly, (T, U) represents a tuple with data type T and U. A function type is written as $T \overset{EffectT}{\rightarrow} T$ where the first T is the input type and the second T is for the return type, and they are both subtypes of $ValueT$. As part of the effect system, which will be explored later, an effect type is embedded in the function type and placed above the arrow.

Type-function types, as discussed in section 2.1, are used to represent generic functions. A type-function type is written as $X^T \mapsto T$ where the superscript on a type variable indicates its supertype, *e.g.*, X^T means X must be a subtype of T. Also, type variable X is expected to appear in the $ValueT$ on the right-hand side of $\mapsto$.

MetaT are used to represent meta information. It includes natural numbers and effect types. Natural numbers are used, for instance, in arrays: an array type takes a natural number to represent the length of the array. The effect type is a part of the effect system that will be discussed later.

Typing Rules The following judgments are used in the type system where Γ stands for a well-formed context:

1. $\Gamma \vdash \tau$ *type* states that τ is a type.
2. $\Gamma \vdash \tau' <: \tau$ states that τ' is a subtype of τ.
3. $\Gamma \vdash x : \tau$ states that x has type τ.

$$\frac{X <: \tau \in \Gamma}{\Gamma \vdash X <: \tau}\ \textsc{TVar} \qquad \frac{\Gamma \vdash \tau\ type}{\Gamma \vdash \tau <: \tau}\ \textsc{Refl} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \quad \Gamma \vdash \tau_2' <: \tau_2}{\Gamma \vdash (\tau_1' \to \tau_2') <: (\tau_1 \to \tau_2)}\ \textsc{SubAbs} \qquad \frac{\Gamma \vdash \tau_1' <: \tau_1 \quad \Gamma, X <: \tau_1' \vdash \tau_2' <: \tau_2}{\Gamma \vdash ((X <: \tau_1') \mapsto \tau_2') <: ((X <: \tau_1) \mapsto \tau_2)}\ \textsc{SubTAbs}$$

Figure 4.3: Subtyping rules of the functional IR

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash x : \tau' \quad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash x : \tau}\ \textsc{Subsumption}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1(e_2) : \tau_1}\ \textsc{App}$$

$$\frac{\Gamma; X <: \tau_1 \vdash e : \tau_2}{\Gamma \vdash \Lambda(X <: \tau_1).e : (X <: \tau_1) \mapsto \tau_2}\ \textsc{TAbs} \qquad \frac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma \vdash e : (X <: \tau_1) \mapsto \tau_2}{\Gamma \vdash e\langle \tau_1 \rangle : [X \backslash \tau_1']\tau_2}\ \textsc{TApp}$$

Figure 4.4: Typing Rules of the Functional IR

Figure 4.3 shows the subtyping rules for type system of the functional IR. As we can see, the subtyping judgment, $\Gamma \vdash \tau <: \tau'$, is a reflexive and transitive relation. The SubAbs and SubTAbs rules define the subtyping rule for function type as well as type-function type.

Figure 4.4 shows the typing rules for the functional IR. Subsumption allows a function to take an argument of a subtype of its input type. App and Abs are rules for function abstraction and application. TApp and TAbs are rules for type-function abstraction and application of System F with the support of subtyping. The typing rule makes sure that all well-formed expressions in the functional IR are correct.

4.2.2 Introduction to Views

This subsection provides an overview of the concept of views and how we can leverage it toward efficient code generation. Views provide fine-grained controls to end-users on whether an intermediate data structure needs to be materialized or not. Most primitives that only affect data layout, such as *Concat*, are expressed as views. Views are divided

into two categories: source and destination views. Source views determine the location for reading, while destination views alter the location for writing.

In section 4.1, the example of vector addition and array concatenation has been used to show the necessity of supporting source and destination views at the same time to cater to the end-users. To provide more intuition behind views and the categorization, let's now reconsider the array concatenation example $Concat(a, b)$, and simply assume that a is an array of $[x_0, x_1]$, b is an array of $[x_2, x_3]$ and x_0 to x_3 are integers.

When *Concat* is treated as a source view, the result of $Concat^S(a, b)$, which is $[x_0, x_1, x_2, x_3]$, will never be materialized as it is merely a *view*, which is guaranteed in the later transformations. Hence, when the result of $Concat^S(a, b)$ is using by other primitives, if the i-th element is accessed and $i < 2$, then one of the elements from a will be returned. Similarly, if $i >= 2$, then the element to be returned is from b.

When *Concat* is treated as a destination view, however, it is expected to alter the destination of a and b so that they can be written into a single array directly during evaluation. Thus, different from a source view, the result of $Concat^D$ is materialized, *i.e.*, an array of $[x_0, x_1, x_2, x_3]$ will actually be written in the memory. Nevertheless, no intermediate data structure creation is involved in this process, since a and b write directly to that array.

4.2.3 Effect Types and Effect System

Effect Types As part of the effect system, the effect types in figure 4.2 are used to indicate the effect of a function, as they will be embedded in a function type such as $T \overset{EffectT}{\rightarrow} T$. There are four subtypes of *EffectT*:

- *Source view type* (***S***): represents a function that generates a source view
- *Destination view type* (***D***): represents a function generates a destination view
- *Eager type* (***E***): represents a function that conducts computation and writes the result to a given destination.

$$\frac{x : \tau; \epsilon \in \Gamma}{\Gamma \vdash x : \tau; \epsilon}\ \textsc{Var} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2; \epsilon}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\epsilon} \tau_2; \emptyset}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\epsilon_3} \tau_1; \epsilon_1 \qquad \Gamma \vdash e_2 : \tau_2; \epsilon_2}{\Gamma \vdash e_1(e_2) : \tau_1; \epsilon_1 \ominus \epsilon_2 \ominus \epsilon_3}\ \textsc{App}$$

$$\epsilon_1 \ominus \epsilon_2 = \begin{cases} error & \text{if } \epsilon_1 = S \text{ and } \epsilon_2 = D \\ \epsilon_2 & \text{otherwise} \end{cases} \qquad \textsc{Compose}$$

Figure 4.5: Main Rules for the Effect System

- *No-effect type* ($\emptyset$) is used when a function is not a view and its result will not be materialized, *e.g.*, a function that returns another function as a result.

Preventing Invalid Programs While the division of source and destination view is beneficial, as motivated in section 4.1, their composition may produce invalid programs. For instance, $Split^D(Join^S(arr))$ is invalid when $Split^D$ is a destination view and $Join^S$ is a source view, since $Split^D$ aims to alter the location where $Join^S$ is written to, but the latter is a source view and will not be materialized in the first place.

Embedding view information as effect types allows the effect system [30] to prevent such invalid programs. The typing judgments of the effect system have the form of:

$$\Gamma \vdash e : \tau; \epsilon$$

stating that expression e has type τ and effect type ϵ.

Figure 4.5 shows the related rules to forbid invalid programs. The function abstraction rule (Abs) assigns the effect ϵ to the function body and is annotated at the arrow. The function application rule (App) presents how the information comes together: the effects from evaluating ϵ_1 and ϵ_2 composed with (via $\ominus$) ϵ_3 obtained from the function type. As shown in rule Compose in Figure 4.5, $\ominus$ is a left-associative and non-commutative operator,

propagates the right operand and is used to detect errors. Following these rules, a program like $Split^D(Join^S(arr))$ will be detected invalid since $S \ominus D$ raises an error.

Left-associativity is an important property for $\ominus$. Given how effect types are composed together in rule App and the fact that primitives are evaluated from innermost to outermost, effect types composed by $\ominus$ should be interpreted from left to right. To better illustrate the intuition behind this, let's look at the invalid expression $Map^E(f, Split^D(Join^S(a)))$ with the effect type of $S \ominus D \ominus E$ as an example. If $\ominus$ is right-associative, according to rule Compose, there will be: $(S \ominus (D \ominus E)) = S \ominus E$, which is incorrect since no error can be detected from the intermediate result $S \ominus E$. Instead, if $\ominus$ is left-associative, the effect type will become: $((S \ominus D) \ominus E)$ where $S \ominus D$ is encountered and an error is raised. From the perspective of the evaluation order, $Join^S$ is first evaluated, then $Split^D$ and finally Map^E. Therefore, the first composed effect types to be examined should be $S \ominus D$ from $Split^D(Join^S(a)))$ instead of $D \ominus E$ from $Map^E(f, Split^D(..))$. As we can see, only when $\ominus$ is left-associative, this examination order can be fulfilled.

4.2.4 Functional Primitives

The primitives used at the functional IR level are shown in Figure 4.6. Laziness or eagerness depends on a type parameter provided at construction. For example, if *Split* takes a source view type, it becomes a source view primitive. It is worth noting that some effects don't apply to some primitives. *Split*, for example, can only be a source view or destination view primitive. Since it only changes the data layout and is inherently lazy, it is unnecessary to provide an eager *Split*. However, it does not mean that this functional IR cannot have a *Split* that actually splits an array into memory. With the *Materialize* transformation that will be soon introduced, the end-users can express an eager *Split*.

$$Id : t^{ScalarT} \mapsto t \xrightarrow{E} t \tag{4.1}$$

$$UnaryOp : t^{ScalarT} \mapsto t \xrightarrow{E} t \tag{4.2}$$

$$BinaryOp : t^{ScalarT} \mapsto t \to t \xrightarrow{E} t \tag{4.3}$$

$$Map : v_0^{S|D|E} \mapsto v_1^{S|D|E} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto n^{NatT} \mapsto (t \xrightarrow{v_1} u) \to [t]_n \xrightarrow{v_0} [u]_n \tag{4.4}$$

$$Reduce : t^{DataT} \mapsto u^{DataT} \mapsto n^{NatT} \mapsto (u \to t \xrightarrow{E} u) \to u \to [t]n \xrightarrow{E} u \tag{4.5}$$

$$Tuple : v^{S|D} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto t \xrightarrow{v} u \xrightarrow{v} (t, u) \tag{4.6}$$

$$Zip : v^{S|D} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto n^{NatT} \mapsto ([t]_n, [u]_n) \xrightarrow{v} [(t, u)]_n \tag{4.7}$$

$$Split : v^{S|D} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto m^{NatT} \mapsto [t]_n \xrightarrow{v} [[t]_m]_{n/m} \tag{4.8}$$

$$Join : v^{S|D} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto m^{NatT} \mapsto [[t]_m]_n \xrightarrow{v} [t]_{n*m} \tag{4.9}$$

$$Concat : v^{S|D} \mapsto t^{DataT} \mapsto n^{Nat} \mapsto m^{Nat} \mapsto ([t]_n, [t]_m) \xrightarrow{v} [t]_{n+m} \tag{4.10}$$

$$Transpose : v^{S|D} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto m^{NatT} \mapsto [[t]_m]_n \xrightarrow{v} [[t]_n]_m \tag{4.11}$$

$$Permute : v^{S|D} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto u^{IntT} \mapsto (u \to u) \to [t]_n \xrightarrow{v} [t]_n \tag{4.12}$$

$$If : v^{S|D} \mapsto u^{DataT} \mapsto BoolT \to u \to u \xrightarrow{v} u \tag{4.13}$$

$$Slide : v^{S|E} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto m^{NatT} \mapsto k^{IntT} \mapsto [t]_n \xrightarrow{v} [[t]_m]_{(n-m+k)/k} \tag{4.14}$$

$$Slice : v^{S} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto m^{NatT} \mapsto IntT \to [t]_n \xrightarrow{v} [t]_m \tag{4.15}$$

$$Repeat : v^{S|D} \mapsto n^{NatT} \mapsto t^{DataT} \mapsto t \xrightarrow{v} [t]_n \tag{4.16}$$

$$At : v^{S|D} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto [t]_n \to IntT \xrightarrow{v} t \tag{4.17}$$

$$Fst : v^{S|D} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto (t, u) \xrightarrow{v} t \tag{4.18}$$

$$Snd : v^{S|D} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto (t, u) \xrightarrow{v} u \tag{4.19}$$

Figure 4.6: Primitives at the highest, functional level. If a function has a no-effect type $\emptyset$, the arrow is left empty.

For convenience, superscripts S, D, and E are used on the primitive showing its effect. As a result, for $Split$, the following will hold, where $\langle T \rangle$ stands for a type application:

$$Split^S = Split\langle S \rangle$$

$$Split^D = Split\langle D \rangle$$

Syntactic sugars, including $e[idx]$ for $At(e, idx)$, $.0$ for Fst and $.1$ for Snd, will be used in the rest of the thesis.

4.2.5 Materialization of View

Lazy primitives generate either source or destination views that will be removed when lowering to the imperative level. However, it is sometimes beneficial to materialize their result to improve data locality. This ability is also provided in Repa [20], with the *Manifest* function to materialize arrays. In contract, Lift [37] and $\widetilde{F}$ [35] remove intermediate data structures by default, and it may not always be the best option.

In this thesis, the end-user can $Materialize$ a view by triggering a transformation shown in Figure 4.7. If $Materialize$ takes an array as input, it will produce a Map that materializes each element of the array, similar to if it takes a tuple. If it takes a scalar, a call to Id is produced. Id is an identity function that returns its input. It will be lowered as, for instance, a C assignment to materialize its input. While Id is limited to scalars, combining with Map and $Tuple$ enables the ability to materialize any data structures supported in this IR.

$Materialize$ delivers the eager versions of source view primitives. The following shows an example for materializing the result from $Split^S$ and $Join^S$, where a lambda expression

$$[[expr : ArrayT]]_M = Map^E(\lambda x.[[x]]_M, expr)$$
$$[[expr : TupleT]]_M = Tuple^D([[expr.0]]_M, [[expr.1]]_M)$$
$$[[expr : ScalarT]]_M = Id(expr)$$

Figure 4.7: M (*Materialize*) Transformation

is used to capture them to provide the corresponding eager version:

$$Split^E(m) = \lambda src.\ [[Split^S(m, src)]]_M \tag{4.20}$$
$$Join^E = \lambda src.\ [[Join^S(src)]]_M \tag{4.21}$$

4.3 Imperative Level

Unlike high-level functional code, low-level imperative code has memory operations and array accesses explicit, which imposes a challenge for lowering since those concepts are absent. This section presents the imperative constructs, similar to $\widetilde{F}$ [35], enabling the expression of imperative concepts in a functional style. With the imperative constructs, the goal is to provide a straightforward translation to imperative code.

Types Two types are introduced specifically for the imperative constructs so that the syntax on 4.2 can be extended for the imperative level:

$T := ...$	*other types presented in Figure 4.2*
LocT[T]	*Location Type*
VoidT	*Void Type*

Location type ($LocT[T]$) is introduced to represent the location of data in memory, similar to a pointer in C. As a result, a destination from DPS can be defined as an expression of a location type, which will be used later for lowering the IR. Also, void type ($VoidT$) is

introduced to represent the absence of value. As indicated by the name, it is similar to the void type in C for expressing a function that does not provide a result to its caller via the return statement.

Primitives There are two significant differences between the functional primitives and the imperative primitives. First, array accesses are explicit in the latter, which allows for a straightforward translation to imperative for-loops. Secondly, the latter operates on location types, allowing operations on the source and the destination to share the same set of primitives and be symmetric.

The primitives for memory operations are listed below and should be self-explanatory:

$$Alloc : t^{DataT} \mapsto LocT[t] \tag{4.22}$$

$$Assign : t^{ScalarT} \mapsto LocT[t] \to LocT[t] \to VoidT \tag{4.23}$$

$$Free : t^{DataT} \mapsto LocT[t] \to VoidT \tag{4.24}$$

$$LocOf : t^{DataT} \mapsto t \to LocT[t] \tag{4.25}$$

Build and $Ifold$ expose array accesses explicitly. *Build*, similar to pull array [11], takes a length and an index function. When *Build* is lazy, *i.e.*, the source view primitive $Build^S$ or the destination view primitive $Build^D$, its signature is:

$$Build^v : v^{S|D} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto (IntT \xrightarrow{v} LocT[t]) \xrightarrow{v} LocT[[t]_n] \tag{4.26}$$

However, when *Build* is eager, written as $Build^E$, $VoidT$ will be returned since *Assign* is used within its body to write its result into a destination. Its signature becomes:

$$Build^E : n^{NatT} \mapsto (IntT \xrightarrow{E} VoidT) \xrightarrow{E} VoidT \tag{4.27}$$

$Ifold$ is similar to a reduction that accumulates the result into a single value:

$$Ifold : t^{DataT} \mapsto u^{DataT} \mapsto n^{NatT} \mapsto (LocT[u] \to IntT \xrightarrow{E} VoidT) \xrightarrow{E} LocT[u] \to VoidT \tag{4.28}$$

When translating to imperative code, each $Build^E$ and $Ifold$ will generate a for-loop.

At, Fst, Snd, $Tuple$, If, $BinaryOp$, $UnaryOp$ and Id from the functional level are all overloaded to operate on location types.

$$At : v^{S|D} \mapsto t^{DataT} \mapsto n^{NatT} \mapsto LocT[[t]_n] \to IntT \xrightarrow{v} LocT[t] \tag{4.29}$$

$$Fst : v^{S|D} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto LocT[(t,u)] \xrightarrow{v} LocT[t] \tag{4.30}$$

$$Snd : v^{S|D} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto LocT[(t,u)] \xrightarrow{v} LocT[u] \tag{4.31}$$

$$Tuple : v^{S|D} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto LocT[t] \xrightarrow{v} LocT[u] \xrightarrow{v} LocT[(t,u)] \tag{4.32}$$

$$Id : t^{ScalarT} \mapsto LocT[t] \xrightarrow{E} LocT[t] \tag{4.33}$$

$$UnaryOp : t^{ScalarT} \mapsto LocT[t] \xrightarrow{E} LocT[t] \tag{4.34}$$

$$BinaryOp : t^{ScalarT} \mapsto LocT[t] \to LocT[t] \xrightarrow{E} LocT[t] \tag{4.35}$$

In the rest of this thesis, all these primitives (and their respective syntactic sugar) are overloaded. For example, if At is applied to a data type then it is the functional primitive from figure 4.6. If it is applied to a location type, then it is the imperative one presented above.

The intuition behind such modification lies in the fact that data in imperative code always have corresponding addresses for their location in memory. Moreover, it allows the same primitives to be used for manipulating destinations as well. The same syntax $[Idx]$, $.0$ and $.1$ will be overloaded by eq.(4.29), eq.(4.30) and eq.(4.31) respectively.

4.4 Summary

This chapter has introduced the multi-levels IRs used in this thesis. The types and primitives are dedicated to using the concept of destination and views. Due to this particular design, the imperative level, with exposed indices as well as those memory operators, already has a very straightforward translation to the imperative code. The remaining puzzle is how to lower from the functional level IR to the imperative level IR, which will be discussed in the next chapter.

Chapter 5

Lowering to Imperative Level and Code Generation

In the previous section, the functional and imperative level constructs are presented. However, for lowering the functional level IR to the imperative level IR there remains a big gap. To bridge the gap, this chapter first shows a normalization transformation to make sure that the evaluation order is explicit. Then it introduces the $\mathcal{DPS}$ transformation, inspired by $\widetilde{F}$, to lower the functional IR into the imperative constructs while providing memory management and simplifying views at the same time. At the end of this chapter, it covers the final step for generating imperative code from the imperative level IR.

5.1 Explicit Evaluation Order

At the functional level, programs are written by the nesting and combination of function primitives with view information embedded. As the first step for lowering to the imperative level, the program will be normalized based on the given view information to explicitly reveal the evaluation order.

Before diving into the normalization details, let's have a look at the three vectors addition program, where the intermediate array from the addition of v_1 and v_2 is desired to be materialized (just for demonstration purpose):

$$Map^E(+, Zip^S(Tuple^S(v_0, Map^E(+, Zip^S(Tuple^S(v_1, v_2))))))$$

The program is actually equivalent to the following if the inner Map^E, highlighted in blue, is taken out by using a let expression and a variable named tmp:

$$let\ tmp = Map^E(+, Zip^S(Tuple^S(v_1, v_2)))\ in\ Map^E(+, Zip^S(Tuple^S(v_0, tmp)))$$

They are equivalent because the result of the inner Map^E is expected to be evaluated and materialized before being used as the input to the outer Map^E, highlighted in red.

The above example shows the fact that by using let expression, before lowering into the imperative level, we can make sure that the evaluation order is explicit. It is beneficial since as we will see when lowering into the imperative level, Map will be lower into $Build$. When the evaluation order is not explicit, $Build$ may execute the same code in the loop body in every iteration. For example, when rewriting the previous example to $Build^E$ by using the following rewrite rule, where $[x \backslash in[i]]e$ means subtituting x to $in[i]$ in expression e:

$$Map^E(\lambda x.e, in) \Rightarrow Build^E(\lambda i.[x \backslash in[i]]e)$$

The three vectors addition program becomes:

$$Build^E(\lambda i.v_0[i] + Build^E(\lambda j.v_1[j] + v_2[j])[i])$$

Clearly, the $Build^E$ corresponding to the inner Map^E is used in the outer $Build^E$ loop body and will be executed redundantly in each iteration. The nesting of one $Build^E$ may signif-

$$p^S(e_0, .., e_n) \text{ where } \exists k \in [0, n].\ e_k \text{ is a destination view or eager primitive call} \Rightarrow \quad (5.1)$$
$$\text{let } tmp = e_k \text{ in } [e_k \backslash tmp] p^S(e_0, .., e_n)$$
$$p^E(e_0, .., e_n) \text{ where } \exists k \in [0, n].\ e_k \text{ is a destination view or eager primitive call} \Rightarrow \quad (5.2)$$
$$\text{let } tmp = e_k \text{ in } [e_k \backslash tmp] p^E(e_0, .., e_n)$$
$$p(e_0, .., e_n) \text{ where } \exists k \in [0, n].\ e_k \text{ match } (\text{let } p = a \text{ in } b) \Rightarrow \quad (5.3)$$
$$\text{let } p = a \text{ in } [e_k \backslash b] p(e_0, .., e_n)$$

Figure 5.1: Rewrite Rules to Demonstrate the Explicit Evaluation Order

icantly slow down the runtime performance because the whole temporary array is being recomputed for every access.

Rewrite Rules Figure 5.1 shows the rewrite rules to demonstrate the explicit evaluation order of the functional level programs before feeding them into the later lowering process. The metavariable $\boldsymbol{p}$ is used to present a primitive available at the functional level IR. Similarly, $\boldsymbol{p^S}$ is more specific for a source view primitive, while $\boldsymbol{p^E}$ is for an eager primitive. $p(e_0, .., e_n)$ stands for a function call to the primitive p with expressions e_0 to e_n as arguments. Also, existential quantification is used in the rewrite rules, such as $\exists k \in [0, n].P$ where P is a predicate using k.

With the definition of the syntax, eq.(5.1) states that given a source view primitive p^S and expressions e_0 to e_n as inputs, if there exists a natural number k that belongs to $[0, n]$ and an expression e_k is either a function call to a destination view primitive or an eager primitive, then e_k is hoisted in a let expression and forced to be evaluated first. For instance, the vector addition example $Tuple^S(v_0, Map^E(...))$ becomes $\text{let } tmp = Map^E(...) \text{ in } Tuple^S(v_0, tmp)$. Similarly, eq.(5.2) hoists function calls to destination view primitives or eager primitives out of eager primitives. Eq.(5.3) hoists let expressions as input to other primitives upward, ensuring that let expressions are not nested within other primitives.

Revisiting the Vector Addition Example Going back to the previous example again, with eq.(5.1) applied, the program becomes:

$$Map^E(+, Zip^S(let\ tmp = Map^E(+, Zip^S(Tuple^S(v_1, v_2)))\ in\ Tuple^S(v_0, tmp)))$$

By applying eq.(5.3) to hoist the let expression, the program has the evaluation order fully explicit and avoids recomputing the entire temporary array multiple times:

$$let\ tmp = Map^E(+, Zip^S(Tuple^S(v_1, v_2))\ in\ Map^E(+, Zip^S(Tuple^S(v_0, tmp)))$$

Guarantee of Evaluation Order *Let* expressions are used to maintain the evaluation order after rewrite rules in Figure 5.1 are applied. In the rest of the thesis, this explicit evaluation order formed by *Let* expressions is guaranteed to be not modified by any other rewrite rule so that the code generator can generate imperative following this order.

5.2 $\mathcal{DPS}$ Transformation

The $\mathcal{DPS}$ transformation is introduced to lower the high-level IR into a still functional, but more imperative, form where the imperative constructs will be used. This section focuses on the source view primitives and eager primitives, while the problem for handling destination views is left to the section 5.3.

The $\mathcal{DPS}$ transformation takes an expression s and a destination d as input. s is the source and has a data type, while d is the destination and has a location type.

$$[[s, d]]_{\mathcal{DPS}}$$

Lowering Core Language Figure 5.2 shows the $\mathcal{DPS}$ transformation for the core functional constructs.

$$[[\lambda p.b : t^{DataT}), \varnothing]]_{\mathcal{DPS}} = \lambda p.\lambda d : LocT[t].[[b, d]]_{\mathcal{DPS}} \quad (5.4)$$

$$[[\lambda p.b, d]]_{\mathcal{DPS}} = \lambda p.[[b, d]]_{\mathcal{DPS}} \quad (5.5)$$

$$[[p^D(s) : t, \varnothing]]_{\mathcal{DPS}} = \quad d = Alloc(t); [[p^D(s), d]]_{\mathcal{DPS}} \quad (5.6)$$

$$[[p^E(s) : t, \varnothing]]_{\mathcal{DPS}} = \quad d = Alloc(t); [[p^E(s), d]]_{\mathcal{DPS}} \quad (5.7)$$

$$[[p = init; b, d]]_{\mathcal{DPS}} = \quad p = [[init, \varnothing]]_{\mathcal{DPS}}; [[b, d]]_{\mathcal{DPS}} \quad (5.8)$$

$$[[ParamUse(p) : DataT, \varnothing]]_{\mathcal{DPS}} = LocOf(ParamUse(p)) \quad (5.9)$$

Figure 5.2: Rules for core functional constructs in $\mathcal{DPS}$ transformation. @ binds the matched pattern to a variable that can be reused. $a; b$ stands for a let expression $x = a; b$, where x is not used in b, which happens when, for example, x is type of $VoidT$.

The $\mathcal{DPS}$ transformation starts from the top-level lambda which represents the program, $[[\lambda p.b, \varnothing]]_{\mathcal{DPS}}$. This will immediately trigger eq.(5.4) which will create a destination for the program. In plain C, starting with the example:

```
int foo(float* p)
```

The $\mathcal{DPS}$ transformation will turn it into the following where an extra parameter d is introduced for storing the function result:

```
void foo(float* p, int* d)
```

Eq.(5.5) is used when a destination is already provided or in the case where the lambda body is not of type $DataT$. In eq.(5.6, 5.7), p^D and p^E stands for destination view primitives and eager primitives respectively. When function calls to them are encountered, allocations will be nested for storing their results. Any let expression is handled by eq.(5.8) which applies the transformation on the initial value $init$ and the body b separately. Since the program is lowered to use imperative constructs that operate on location types, eq.(5.9) is needed for getting the location of a parameter usage if it is of data type.

$$[[Id(s,d)]]_{\mathcal{DPS}} = Assign(Id([[s,\varnothing]]_{\mathcal{DPS}}), d) \tag{5.10}$$

$$[[UnaryOp(op,s),d]]_{\mathcal{DPS}} = Assign(UnaryOp(op,[[s,\varnothing]]_{\mathcal{DPS}}), d) \tag{5.11}$$

$$[[BinaryOp(op,s_0,s_1),d]]_{\mathcal{DPS}} = Assign(BinaryOp(op,[[s_0,\varnothing]]_{\mathcal{DPS}},[[s_1,\varnothing]]_{\mathcal{DPS}}), d) \tag{5.12}$$

$$[[Map^E(\lambda x.e,s) : [t]_n, d]]_{\mathcal{DPS}} = Build^E(n, \lambda idx.[[[x\backslash[[s,\varnothing]]_{\mathcal{DPS}}[idx]]e, d[idx]]]_{\mathcal{DPS}}) \tag{5.13}$$

$$[[Reduce(\lambda acc.\lambda cur.e, init, s), d]]_{\mathcal{DPS}} = Ifold(\lambda acc.\lambda idx. \tag{5.14}$$
$$\quad [[[cur\backslash[[s,\varnothing]]_{\mathcal{DPS}}[idx]]e, d]]_{\mathcal{DPS}}, [[init, d]]_{\mathcal{DPS}}))$$

$$[[Slide^E(size,step,src) : [[t]_{size}]_n, d]]_{\mathcal{DPS}} = \tag{5.15}$$
$$wd = Alloc([t]_{size});$$
$$[[[[Slide^S(size,step,src)[0][0..size-step]]]_M, wd[step..size]]]_{DPS}$$
$$Build^E((n-size+step)/step, \lambda i.$$
$$\quad [[[[wd[step..size]]]_M, wd[0..size-step]]]_{\mathcal{DPS}};$$
$$\quad [[[[Slide^S(size,step,src)[i][size-step..size]]]_M, wd[size-step..size]]_{DPS};$$
$$\quad [[wd, d[i]]]_{\mathcal{DPS}})$$

Figure 5.3: $\mathcal{DPS}$ transformation for eager primitives. $[x\backslash y]e$ means substituting x by y in the expression e. $[a..b]$ stands for an array slice.

Lowering Eager Primitives Figure 5.3 shows the rules for handling eager primitives in the $\mathcal{DPS}$ transformation. Eager primitives that operate on scalar types, as shown in eq.(5.10 -5.12), can be directly written into memory using *Assign*.

Map^E, is lowered into $Build^E$ so that a destination $d[idx]$ can be provided to its input function body, as shown in eq.(5.13). The same applies to *Reduce* where an $Ifold$ is generated. Since the $\mathcal{DPS}$ transformation is applied recursively, it will eventually produce an *Assign* when traversing the body of a $Build^E$ or $Ifold$.

For $Slide^E$, instead of directly writing its result, a rolling window wd is introduced for the better locality as shown in eq.(5.15). To provide more intuition for this rule, the corresponding C code template for $[[Slide^E(3,1,src),d]]_{\mathcal{DPS}}$ is:

```
1 int wd[3];
2 wd[1] = src[0]; wd[2] = src[1];  // Slide^S
```

$$[[Map^S(\lambda in.e, s) : [t]_n, \varnothing]]_{\mathcal{DFS}} = Build^S(n, \lambda idx.[[[in \backslash [[s, \varnothing]]_{\mathcal{DFS}}[idx]]e, \varnothing]]_{\mathcal{DFS}}) \quad (5.16)$$

$$[[Tuple^S(s_0, s_1), \varnothing]]_{\mathcal{DFS}} = Tuple^S([[s_0, \varnothing]]_{\mathcal{DFS}}, [[s_1, \varnothing]]_{\mathcal{DFS}}) \quad (5.17)$$

$$[[Zip^S(s) : [(t, u)]_n, \varnothing]]_{\mathcal{DFS}} = Build^S(n, \lambda idx. \quad (5.18)$$
$$Tuple^S([[s, \varnothing]]_{\mathcal{DFS}}.0[idx], [[s, \varnothing]]_{\mathcal{DFS}}.1[idx]))$$

$$[[Split^S(n, s) : [[t]_m]_n, \varnothing]]_{\mathcal{DFS}} = Build^S(n, \lambda idx_0.$$
$$Build^S(m, \lambda idx_1.[[s, \varnothing]]_{\mathcal{DFS}}[n * idx_0 + idx_1])) \quad (5.19)$$

$$[[Join^S(s : [t]_m]_n), \varnothing]]_{\mathcal{DFS}} = Build^S(n * m, \lambda idx.[[s, \varnothing]]_{\mathcal{DFS}}[idx/m][idx\%m]) \quad (5.20)$$

$$[[Concat^S(s{:}([t]_n, [t]_m), \varnothing]]_{\mathcal{DFS}} = Build^S(n+m, \lambda idx.If^S(\quad (5.21)$$
$$[[idx < n, \varnothing]]_{\mathcal{DFS}}, [[s.0[idx], \varnothing]]_{\mathcal{DFS}}, [[s.1[idx - n], \varnothing]]_{\mathcal{DFS}}))$$

$$[[Slide^S(size, step, s) : [t]_n, \varnothing]]_{\mathcal{DFS}} = Build^S(n, \lambda idx_0.$$
$$Build^S(size, \lambda idx_1.[[s, \varnothing]]_{\mathcal{DFS}}[idx_0 * step + idx_1])) \quad (5.22)$$

$$[[Repeat^S(n, s), \varnothing]]_{\mathcal{DFS}} = Build^S(n, \lambda idx.[[s, \varnothing]]_{\mathcal{DFS}}[idx]) \quad (5.23)$$

$$[[s[idx], \varnothing]]_{\mathcal{DFS}} = [[s, \varnothing]]_{\mathcal{DFS}}[idx] \quad (5.24)$$

$$[[s.0, \varnothing]]_{\mathcal{DFS}} = [[s, \varnothing]]_{\mathcal{DFS}}.0 \quad (5.25)$$

$$[[s.1, \varnothing]]_{\mathcal{DFS}} = [[s, \varnothing]]_{\mathcal{DFS}}.1 \quad (5.26)$$

Figure 5.4: $\mathcal{DFS}$ transformation for source view primitives. Operations, such as $a + b$ or $a < b$, are $BinaryOp^E$.

```
for(int i = 0; i < n - 2; i++) { // Build^E
  wd[0] = wd[1]; wd[1] = wd[2];
  wd[2] = src[i + 2];            // Slide^S
  for(int j = 0; j < 3; j++)
    d[i][j] = wd[j];}
```

Since the rolling window wd is a small array, the chance is high that it is allocated in registers. As a result, reusing the elements stored in the rolling window avoids repeatedly requesting data from the cache or main memory, and thus improves the data locality.

$$Build^S(n, \lambda idx.s)[idx'] \Rightarrow [idx \backslash idx']s \tag{5.27}$$
$$Tuple^S(s_0, s_1).0 \Rightarrow s_0 \tag{5.28}$$
$$Tuple^S(s_0, s_1).1 \Rightarrow s_1 \tag{5.29}$$
$$If^S(cond, s_0, s_1)[idx] \Rightarrow If^S(cond, s_0[idx], s_1[idx]) \tag{5.30}$$
$$If^S(cond, s_0, s_1).0 \Rightarrow If^S(cond, s_0.0, s_1.0) \tag{5.31}$$
$$If^S(cond, s_0, s_1).1 \Rightarrow If^S(cond, s_0.1, s_1.1) \tag{5.32}$$
$$(p = a; b)[idx] \Rightarrow p = a; b[idx] \tag{5.33}$$
$$(p = a; b).0 \Rightarrow p = a; b.0 \tag{5.34}$$
$$(p = a; b).1 \Rightarrow p = a; b.1 \tag{5.35}$$

Figure 5.5: Rewrite Rules for Removing Source Views

Lowering Source View Primitives The lowering rules for source view primitives are shown in Figure 5.4. After these rules have been applied, the only primitives that are left are $Build^S$, $Tuple^S$, If^S, At^S, Fst^S, Snd^S.

Figure 5.5 shows how it is possible to simplify access to the result of a $Build^S$ or $Tuple^S$ view. In the system presented, user-defined functions always operate on scalars and are eager. Therefore, any array or tuple views will always be accessed sooner or later. This implies that there will never be any $Build^S$ or $Tuple^S$ left when applying the $\mathcal{DPS}$ transformation followed by these simplifications rules.

Once the simplifications have been applied, we will only be left with If^S, At^S, Fst^S, Snd^S which can be directly translated to the equivalent C constructs. Therefore, no for-loop or tuple creation will ever appear for any source view originally presented at the highest functional level. This property guarantees that no unnecessary intermediate data structures will be created by source views to slow down the generated code's performance.

5.2.1 Revisiting the Vector Addition Example

Using the $\mathcal{DPS}$ transformation, the complete derivation of the vector addition example from section 4.1 is:

$$[[\lambda a, b.Map^E(\lambda x.(x.0 + x.1), Zip^S(Tuple^S(a, b))), \varnothing]]_{\mathcal{DPS}}$$

$$\xrightarrow{\text{Eq(5.4, 5.5)}} \lambda a, b, d.[[Map^E(\lambda x.(x.0 + x.1), Zip^S(Tuple^S(a, b))), d]]_{\mathcal{DPS}}$$

$$\xrightarrow{\text{Eq.(5.16)}} \lambda a, b, d.Build^E(n, \lambda i.[x \backslash [[Zip^S(Tuple^S(a, b)), \varnothing]]_{\mathcal{DPS}}[i]][[x.0 + x.1, d[i]]]_{\mathcal{DPS}})$$

$$\xrightarrow{\text{Eq.(5.17, 5.18)}}$$

$$\lambda a, b, d.Build^E(n, \lambda i.[x \backslash [[Build^S(n, \lambda j.Tuple^S(a[j], b[j])), \varnothing]]_{\mathcal{DPS}}[i]][[x.0 + x.1, d[i]]]_{\mathcal{DPS}})$$

$$\xrightarrow{\text{Eq.(5.27, 5.28, 5.29)}} \lambda a,b,d.Build^E(n,\lambda i.[[a[i] + b[i], d[i]]]_{\mathcal{DPS}})$$

$$\xrightarrow{\text{Eq.(5.12)}} \lambda a, b, d.Build^E(n, \lambda i.Assign(a[i] + b[i], d[i]))$$

5.3 Handling Destination Views

The previous section has introduced the $\mathcal{DPS}$ transformation and how it handles eager and source view primitives. This section focuses on destination views.

Before diving into any detail, the following example illustrates how a destination view primitive should be handled by the $\mathcal{DPS}$ transformation:

$$[[Split^D(m, e) : [[t]_m]_n, d]]_{\mathcal{DPS}} = [[e, Build^D(n * m, \lambda i.d[i/n][i\%m])]]_{\mathcal{DPS}}$$

On the left side, e is split and written into the destination d. However, since $Split^D$, a destination view primitive, is lazy, the splitting process should not occur. Instead, as shown on the right side, e should be written to the memory as if it is being splitted. But how?

If the result of e is, say, a 1D array, then d should be a location for a 2D array. The imperative destination constructs, $Build^D(n * m, \lambda i.d[i/n][i\%m])$, lazily joins a 2D array location into a 1D array location. By doing so, when writing the result of e using the joined 1D array

location, it maps to the actual 2D array location. Therefore, to achieve this transformation, the key is to *invert* the splitting of the source into a joining on the destination.

5.3.1 Lowering Destination View Primitives

As shown in the above example, when encountering a destination view primitive, $\mathcal{DPS}$ transformation has the following form, where p^D stands for a destination view primitive:

$$[[p^D(e), d]]_{\mathcal{DPS}} = [[e, [[p^D]]_{\mathcal{I}}(d)]]_{\mathcal{DPS}}, where\ d \neq \varnothing \tag{5.36}$$

While the $\mathcal{I}$ transformation has not been presented yet, the above rule provides a taste of how a destination view primitive is handled. On the right side of the equation, the first argument becomes e, which is the operation used as the input to the destination view primitive. The destination for this operation e is modified to take the destination view into account. The second argument, *i.e.*, the result of the modification, becomes $[[p^D]]_{\mathcal{I}}(d)$, where the function returned by $[[p^D]]_{\mathcal{I}}$ is applied to the original destination d. The rationale and rules for $\mathcal{I}$ transformation will be discussed shortly.

5.3.2 $\mathcal{I}$ Transformation

As explained in the above example, $\mathcal{I}$ transformation takes a function, usually a destination primitive, as input, and returns a lowered *inverse function*.

Figure 5.6 shows the rules for $\mathcal{I}$ transformation, demonstrating how inverse functions are defined and lowered. In the above example, eq.(5.37) is used to return a function with the effect of $Join$ on a destination (using $Build^D$). Eq.(5.39) uses $Unrepeat^D$ as the inverse function for $Repeat^D$. Its signature is $LocT[[t]_n] \rightarrow LocT[t]$.

Eq.(5.43, 5.44, 5.45) are used when At^D, Fst^D and Snd^D are encountered. These primitives take input but only return a part of it. Introducing $None$ allows the description of a destination that only a part of the source can be written into. For instance, when using

$$[[Split^D : [t]_{n*m} \rightarrow [[t]_m]_n]]_{\mathcal{I}} = \lambda d.Build^D(n * m, \lambda i.d[i/m][i\%m]) \tag{5.37}$$

$$[[Join^D : [[t]_m]_n \rightarrow [t]_{m*n}]]_{\mathcal{I}} = \lambda d.Build^D(n, \lambda i.Build(m, \lambda j.d[i * m + j])) \tag{5.38}$$

$$[[Repeat^D(n)]]_{\mathcal{I}} = \lambda d.Unrepeat^D(n, d) \tag{5.39}$$

$$[[Concat^D : ([t]_n, [t]_m) \rightarrow [t]_{n+m}]]_{\mathcal{I}} = \tag{5.40}$$
$$\lambda d.Tuple^D(Build^D(n, \lambda i.d[i]), Build^D(m, \lambda i.d[i + n]))$$

$$[[Zip^D : ([t]_n, [u]_n) \rightarrow [(t, u)]_n]]_{\mathcal{I}} = \tag{5.41}$$
$$\lambda d.Tuple^D(Build^D(n, \lambda i.d[i].0), Build^D(n, \lambda i.d[i].1))$$

$$[[Transpose^D : ([[t]_n]_m) \rightarrow [[t]_m]_n]]_{\mathcal{I}} = \lambda d.Build^D(n, \lambda i.Build^D(m, \lambda j.d[j][i])) \tag{5.42}$$

$$[[At^D(i)]]_{\mathcal{I}} = \lambda d.Build^D(\lambda j.If^D(j == i, d, None)) \tag{5.43}$$

$$[[Fst^D]]_{\mathcal{I}} = \lambda d.Tuple^D(d, None) \tag{5.44}$$

$$[[Snd^D]]_{\mathcal{I}} = \lambda d.Tuple^D(None, d) \tag{5.45}$$

$$[[Map^D(\lambda in.e) : [t]_n \rightarrow [u]_n]]_{\mathcal{I}} = \lambda d.Build^D(n, \lambda i.[[e]]_{\mathcal{I}}(d[i])) \tag{5.46}$$

$$[[p_0^D \circ p_1^D]]_{\mathcal{I}} = [[p_1^D]]_{\mathcal{I}} \circ [[p_0^D]]_{\mathcal{I}} \tag{5.47}$$

Figure 5.6: Rules for $\mathcal{I}$ transformation. $\circ$ stands for a function composition.

$Tuple^D(d, None)$ as the destination where we try to write $Tuple(a, b)$, only the first element a is materialized. This is achieved by making sure that no corresponding code is generated by any assignment to $None$.

Eq.(5.43) also introduces condition check If^D for At^D to ensure that only the i-th elements from the source can be materialized. This additional condition check can be further removed using rewrite rules, such as:

$$Build^E(\lambda j.If(j == i, Assign(s[i], d), Assign(s[j], None))) \Rightarrow Assign(s[i], d) \tag{5.48}$$

This rule is based on the property that $Assign(s[j], None)$ is an assignment to $None$ and does not generate any code.

Map as Destination View Primitive Combining eq.(5.46) and eq.(5.47) allows the usage of Map^D as destination view primitive. Specially, eq.(5.47) handled the cases where function composition appears in the input function to Map^D, such as: $Map^D(\lambda x.Split^D(Join^D(x)))$.

However, it is impossible to define the inverse function when the input function in a Map^D uses free variables since they do not come from the Map's input array. Normalization rules that hoist the free variables and constants from a Map^D function body can be applied automatically to remove these cases. For a free variable in $\lambda x.e$ with n occurrences in total, the following normalization rule can be applied to remove these cases:

$$Map^D(\lambda x.e, in), \exists y.y \in FV(\lambda x.e) = \tag{5.49}$$
$$Map^D(\lambda p.[x \backslash Fst^D(p)][y \backslash Snd^D(p)]e, Zip^D(Tuple^D(in, Repeat^D(n, y))))$$

In addition, when the effect type of f is an $EagerT$, *i.e.*, eager primitives are used in the function body, its inverse function will, again, become impossible to define. Thus, such Map^D is handled directly by $\mathcal{DPS}$ transformation:

$$[[Map^D((\lambda in.e) : t \xrightarrow{E} u)(s), d]]_{\mathcal{DPS}} = \tag{5.50}$$
$$Build^E(\lambda idx.[[[in \backslash s[idx]^D]e, d[idx]]]_{\mathcal{DPS}}), where\ d \neq \varnothing$$

Here, Map^D as a destination view does not affect where an eager primitive to write but what to write. This special rule will be leveraged by the example in section 7.4.

Tuple as destination view Special consideration must be taken for $Tuple^D$ when defining its inverse function since it takes two data inputs. When $Tuple^D$ is encountered by the $\mathcal{DPS}$ transformation the following rule is used instead of the generic one:

$$[[Tuple^D(e_0, e_1), d]]_{\mathcal{DPS}} = [[UnFst^D(e_0), d]]_{\mathcal{DPS}}; [[UnSnd^D(e_1), d]]_{\mathcal{DPS}}, where\ d \neq \varnothing \tag{5.51}$$

Within this rule, $UnFst^D$ and $UnSnd^D$ have the following signatures:

$$UnFst^D : t^{DataT} \mapsto u^{DataT} \to LocT[t] \xrightarrow{D} LocT[(t,u)] \tag{5.52}$$

$$UnSnd^D : t^{DataT} \mapsto u^{DataT} \to LocT[u] \xrightarrow{D} LocT[(t,u)] \tag{5.53}$$

$UnFst$ takes a location type of t and returns a location type of a tuple type with t as the first data type. The same applied to $UnSnd$, except that the input location type is used for the second data type of the returned tuple type. As indicated by their names, $UnFst^D$'s inverse function is Fst^D and $UnSnd^D$'s inverse function is Fst^D:

$$[\![UnFst^D]\!]_{\mathcal{I}} \Rightarrow \lambda d.Fst^D(d) \tag{5.54}$$

$$[\![UnSnd^D]\!]_{\mathcal{I}} \Rightarrow \lambda d.Snd^D(d) \tag{5.55}$$

Now, it is not hard to see that eq.(5.51) can be derived into:

$$\begin{aligned}
&[\![Tuple^D(e_0,e_1), d]\!]_{\mathcal{DPS}} \\
&\quad = [\![UnFst^D(e_0), d]\!]_{\mathcal{DPS}}; [\![UnSnd^D(e_1), d]\!]_{\mathcal{DPS}} \\
&\quad = [\![e_0, [\![UnFst^D]\!]_{\mathcal{I}}(d)]\!]_{\mathcal{DPS}}; [\![e_1, [\![UnSnd^D]\!]_{\mathcal{I}}(d)]\!]_{\mathcal{DPS}} \\
&\quad = [\![e_0, Fst(d)]\!]_{\mathcal{DPS}}; [\![e_1, Snd(d)]\!]_{\mathcal{DPS}}, where\ d \neq \varnothing
\end{aligned} \tag{5.56}$$

Since d must have a location type of a tuple type, the above rule makes sure that e_0 writes to $d.0$ and e_1 writes to $d.1$.

A similar process also needs to be applied when $Tuple^D$ is used inside a Map^D. Just like the way how $\mathcal{DPS}$ transformation handles $Tuple^D$, the first and second input from the tuple needed to be separated and chained by a let expression. The following normalization rule

$$Build^D(\lambda idx.d)[idx'] \Rightarrow [idx \backslash idx']d \tag{5.58}$$

$$Tuple^D(d_0, d_1).0 \Rightarrow d_0 \tag{5.59}$$

$$Tuple^D(d_0, d_1).1 \Rightarrow d_1 \tag{5.60}$$

$$If^D(cond, d_0, d_1)[idx] \Rightarrow If^D(cond, d_0[idx], d_1[idx]) \tag{5.61}$$

$$If^D(cond, d_0, d_1).0 \Rightarrow If^D(cond, d_0.0, d_1.0) \tag{5.62}$$

$$If^D(cond, d_0, d_1).1 \Rightarrow If^D(cond, d_0.1, d_1.1) \tag{5.63}$$

$$Unrepeat^D(n, d)[idx] \Rightarrow Unrepeat^D(n, Build^D(\lambda k.d[k][idx])) \tag{5.64}$$

$$Unrepeat^D(n, d).0 \Rightarrow Unrepeat^D(n, Build^D(\lambda k.d[k].0) \tag{5.65}$$

$$Unrepeat^D(n, d).r \Rightarrow Unrepeat^D(n, Build^D(\lambda k.d[k].1) \tag{5.66}$$

$$Assign(s, Unrepeat^D(n, d)) \Rightarrow \tag{5.67}$$
$$\quad Assign(s, d[0]); Build^E(n-1, \lambda i.Assign(d[0], d[i+1])$$

$$Assign(s, If^D(cond, d_0, d_1)) \Rightarrow If^D(cond, Assign(s, d_0), Assign(s, d_1)) \tag{5.68}$$

$$Assign(s, None) \Rightarrow Pass() \tag{5.69}$$

Figure 5.7: Rewrite Rules for Removing Destination Views

is needed for the separation:

$$[[Map^D(\lambda in.Tuple^D(e_0, e_1), s), d]]_{\mathcal{DFS}} = \tag{5.57}$$
$$[[Map^D(\lambda in.UnFst^D(e_0), s), d]]_{\mathcal{DFS}}; [[Map^D(\lambda in.UnSnd^D(e_1), s), d]]_{\mathcal{DFS}}$$

Simplifying Destination Views The destination view primitives have now been lowered by the $\mathcal{I}$ transformation. Figure 5.7 shows how it is possible to simplify the accesses and assignments to the lowered constructs. Similar to the simplification of the source views, assignments are set to only operate on scalar, thus, the $Build^D$ and $Tuple^D$ will always be accessed and removed by these rules. Also, combining the rules in eq.(5.64-5.67), $Unrepeat^D$ will be simplified and eventually disappear. After simplification, only At^D, Fst^D and Snd^D and If^D will remain, which have direct translations to imperative code.

5.3.3 Revisiting the Array Concatenation Example

Using the $\mathcal{DPS}$ transformation, the complete derivation of the concatenation example from section 4.1 is:

$$
\begin{aligned}
&[[\lambda a, b.Concat^D(Map^E(Id, a), Map^E(Id, b)), \varnothing]]_{\mathcal{DPS}} \\
&\xrightarrow{\text{Eq.(5.4, 5.5)}} \lambda a, b, d.[[Concat^D(Tuple^D(Map^E(Id, a), Map^E(Id, b)), d]]_{\mathcal{DPS}} \\
&\xrightarrow{\text{Eq.(5.36, 5.40)}} \lambda a, b, d.[[Tuple^D(Map^E(Id, a), Map^E(Id, b)), \\
&\qquad\qquad Tuple^D(Build^D(n, \lambda i.d[i]), Build^D(m, \lambda i.d[i+n]))]]_{\mathcal{DPS}} \\
&\xrightarrow{\text{Eq.(5.51, 5.59, 5.60)}} \lambda a, b, d.[[Map^E(Id, a), Build^D(n, \lambda i.d[i])]]_{\mathcal{DPS}}; \\
&\qquad\qquad [[Map^E(Id, b), Build^D(m, \lambda i.d[i+n])]]_{\mathcal{DPS}} \\
&\xrightarrow{\text{Eq.(5.13, 5.58)}} \lambda a, b, d.Build^E(n, \lambda i.[[Id(a[i]), d[i]]]_{\mathcal{DPS}}); \\
&\qquad\qquad Build^E(m, \lambda i.[[Id(b[i]), d[i+n]]]_{\mathcal{DPS}}) \\
&\xrightarrow{\text{Eq.(5.10)}} \lambda a, b, d.Build^E(n, \lambda i.Assign(a[i], d[i])); Build^E(m, \lambda i.Assign(a[i], d[i+n]))
\end{aligned}
$$

5.4 Code Generation

Having seen the lowering process, this section now looks at techniques for generating low-level imperative code such as C and OpenCL code.

5.4.1 A-Normal Form

For code generation, programs from the imperative level are transformed into A-Normal Form (ANF) [12] to make sure that the evaluation order is explicit in the program by placing all intermediate results into let bindings.

Here is an example of a program translated into ANF:

$$\lambda a.\lambda b.\lambda c.a + b + c \quad \Rightarrow \quad \lambda a.\lambda b.\lambda c.let\ k = (b + c)\ in\ a + k$$

Having the program in ANF is useful for generating imperative code. For instance, the example above can be seen as an imperative program if we consider that the argument and the body of a let expression always correspond to two statements. By rewriting them into semicolon expressions, as seen in section 4.1.2, the resulted program is very similar to targeted imperative code:

$$\lambda a.\lambda b.\lambda c.k = b + c; a + k$$

The following shows the rules for performing the A-reduction to transform the imperative level IR into ANF.

$$let\ p_0 = (let\ p_1 = a_1\ in\ b_1)\ in\ b_0 \Rightarrow let\ p_1 = a_1\ in\ (let\ p_0 = b_1\ in\ b_0) \tag{5.70}$$

$$p(e_0, .., e_n)\ where\ e_k(k \in [0, n])\ is\ a\ function\ call \Rightarrow \tag{5.71}$$
$$let\ x = e_k\ in\ [e_k \backslash x] p(e_0, .., e_n)$$

Eq.(5.70) makes sure that let expressions are not nested in the ANF. In eq.(5.71), $p(e_0, .., e_n)$ stands for a primitive invocation with $e_0, .., e_n$ as its inputs. When any of the input is a function call, eq.(5.71) will be used to lift the function call to the argument part of the let expression so that the function call will be evaluated first.

A-reduction for transforming the programs into ANF is similar to the normalization pass in section 5.1 for exposing the explicit evaluation order at the functional level. However, the purpose of these two transformations is different. A-reduction is performed at the imperative level to make code generation and optimizations easier, as we will see shortly. The normalization pass in section 5.1 is to avoid redundant computation of temporary arrays when lowering *Map* to *Build*.

```
1 def deallocate(expr: Expr, variablesToFree: Seq[ParamUse] = Seq()): Expr =
2   expr match {
3     case Let(p, b, a: AllocExpr) => Let(p, deallocate(b, variablesToFree :+ p), a)
4     case Let(p, b, a) => Let(p, deallocate(b, allocs), deallocate(a))
5     case other =>
6       val frees = variablesToFree.maps(Free(_))
7       val init = other.visitAndRebuild(e => deallocate(e))
8       frees.foldRight(init, acc => cur => Let(ParamDef(VoidType()), cur, acc))
9 }
```

Figure 5.8: Pesudocode for Deallocation

5.4.2 Passes and Optimizations

After the transformation into ANF, the following passes and optimizations are applied:

Memory Deallocation When lowering to the imperative level, allocations are inserted to provide destinations to eager primitives as seen. However, they need to be deallocated when the results are no longer needed to prevent memory leaks. It is easy to determine when a destination is not used at compilation time, given that no complex control flow exists. The only primitive that creates control flow is If, but since both branches have the exact same type, the same allocation will be created regardless of which branch is executed at runtime. The primitive $Free$ is defined by eq.(4.24) to release the allocated memory.

$Alloc$ is always placed in the argument part of a let expression so that the code in the let expression body can use the destination provided by $Alloc$. Given this property, the algorithm for memory deallocation traverses the IR tree, memorizes the $Alloc$s along the way, and adds $Free$s to the tail of a let expression chain according to the encountered $Alloc$s.

Figure 5.8 shows the implementation of `deallocate` in a Scala-like pseudocode (the compiler is implemented in Scala). The input to `deallocate` is expected to be an expression in ANF. Line 3 and 4 are for collecting allocations that need to be free along the tree. Line 5 - 8 inserts the `Frees` to the tail of a chain of let expressions. Line 6 creates a sequence of `Free` expressions based on the collected allocations. Since `other` itself is an expression, such as a $Build^E$ or a $Ifold$, line 7 continues the deallocation process by using the method

`visitAndRebuild` to traverse and rebuild the expression `other` and its children based on the given function `e => deallocate(e)`. Finally, line 8 bundles `init` and `frees` together with `foldRight` via let expressions.

Loop-Invariant Code Motion is an important subset of Partial Redundancy Elimination (PRE). It aims to find code in a loop body that produces the same value in every iteration. At the dataflow level, *Ifold* and $Build^E$ generate for-loops which may contain loop-invariant code. Therefore, in the rest of this paragraph, a loop refers to the input function of $Build^E$ or $Ifold$. An expression is a loop invariant if this value remains the same while running the loop. Loop invariants can be lifted out of the loop body to avoid redundant computation.

Whether an expression has the same effect in each iteration of a loop body is, unfortunately, undecidable [2]. However, a conservative approximation exists for the IR in ANF. A loop invariant, rewritten as $inv(e)$ where e is an expression, can be described precisely by the following rules:

$$\frac{c \text{ constant}}{inv(c)} \text{ Constant} \qquad \frac{e \text{ only defined outside loop}}{inv(e)} \text{ Outside} \qquad \frac{let\ e = inv(a)\ in\ b}{inv(e)} \text{ Let}$$

$$\frac{inv(e)}{inv(\lambda p.e)} \text{ Abs} \qquad \frac{p(inv(e_0), .., inv(e_k)) \quad p \text{ is not } Alloc}{inv(p(e_0, .., e_k))} \text{ Primitive App}$$

Rule Constant states that all constants are loop invariants. Rule Outside assures that a loop invariant e does not appear in a let expression like $let\ e = a\ in\ b$ or an assignment like $Assign(s, e)$ inside the loop body. Rule Let and Abs define loop invariants out of let expressions and function abstractions. Rule Primitive App allows treating a primitive call as a loop invariant as long as all arguments are also loop invariants and the primitive p is not an *Alloc*. The reason for precluding primitive calls to *Alloc* from being loop invariants is that they have the side-effect of allocating space in the memory, and it is no guarantee that hoisting them will not affect the semantics of the program.

Once loop invariants are detected, the following rewrite rules can be used to hoist them out of the loop body:

$$Build^{E}(n, \lambda i.e) \text{ where } inv(a) \text{ is used in } e \Rightarrow let\ x = a\ in\ Build^{E}(n, \lambda i.[a\backslash x]e)$$

$$Ifold(n, \lambda acc.\lambda i.e) \text{ where } inv(a) \text{ is used in } e \Rightarrow let\ x = a\ in\ Ifold(n, \lambda acc.\lambda i.[a\backslash x]e)$$

5.4.3 Imperative Code Generation

C and OpenCL are chosen as the target languages for evaluation. Data types such as tuples and arrays are mapped into structs and arrays. A struct of two members of type T and U with the name of `DEF_TUPLE_T_U` will be created for a tuple of type (T, U). A location to an array is represented by a pointer. For the C backend, multi-dimensional arrays are represented as pointers of pointers because it matches the nested array types and is easier to implement. Another alternative for representing multi-dimensional arrays is through a flattened memory which may lead to better performance, but it will be left as future work for the C backend. However, it is worth mentioning that the flattened memory representation is already adopted in the OpenCL backend as it is a common practice for writing GPU kernels.

The $Ifold$ and *Build* primitives generate for-loops. The If primitive is turned into an if-else statement. Since *Assign* is guaranteed to only operate on scalar types in the IR, it simply turned into a C assignment (=). By default, arrays and tuples are allocated to the heap via `malloc` and scalars are allocated to the stack.

When targetting a GPU, OpenCL code is generated instead of plain C code. In this case, special primitive *MapGlb* with the same signature as *Map* will be used for expressing a for-loop that executes parallelly. And the code templates like `get_global_id(0)` and `get_global_id(1)` will be generated when *MapGlb* is encountered.

5.5 Summary

This chapter has covered the lowering process from the functional level IR to the imperative level IR through the $\mathcal{DPS}$ transformation as well as the code generation for C and OpenCL.

The next chapter presents the automatic exploration techniques for tuning programs at the functional level for generating code with good performance.

Chapter 6

Automatic Exploration

The previous sections have described how views influenced the lowering process in the $\mathcal{DPS}$ transformation and resulted in different generated code. For instance, treating a *Concat* as a source view or a destination view leads to different imperative code. The functional level IR in this thesis presents end-users a fine-grained control on the effects of all primitives, *i.e.*, whether they should be source views, destination views or eager primitives, but one may find it is hard to make those decisions. Firstly, a single primitive with different effects may end up with different generated code and has various runtime performances, not to mention the compositions of several primitives. Secondly, it is error-prone to make these decisions. For instance, having a source view primitive as an input to a destination view primitive will lead to an invalid program. Therefore, it is necessary to provide a mechanism for automatically exploring the variants of programs at the function level and trade-off the suitable variants for code generation.

6.1 Search Space

Before discussing the exploration strategies adopted in this thesis, let's first define the search space. It includes the definition of the input program for the exploration strategies as well as the rewrite rules to apply.

6.1.1 Effectless Program

The input program should consist of primitives without any view information so that decisions can be automated. For convenience, those programs are defined as *effectless program*. An effectless program should be written at the functional level IR, but involves no effect type, *i.e.*, no view information. Primitives for effectless programs can be directly obtained from figure 4.6 on page 34 by removing the effect types embedded in the function types. As a result, the effectless program is very similar to the original Lift IR. For example, eq.(6.1) shows the signature for the *Map* with effect types, and eq.(6.2) shows the *Map* without effect types:

$$Map : v_0^{S|D|E} \mapsto v_1^{S|D|E} \mapsto t^{DataT} \mapsto u^{DataT} \mapsto n^{NatT} \mapsto (t \xrightarrow{v_1} u) \to [t]_n \xrightarrow{v_0} [u]_n \tag{6.1}$$

$$Map : t^{DataT} \mapsto u^{DataT} \mapsto n^{NatT} \mapsto (t \xrightarrow{v_1} u) \to [t]_n \to [u]_n \tag{6.2}$$

6.1.2 Exploration Starting Point

For the automatic exploration, the given effectless program will be first transformed to use the primitives from figure 4.6 by adding effect types. For example, *Map* in eq.(6.2) will become eq.(6.1). However, for most primitives, including *Map*, decisions of whether they should be views or eager need to be made. At the starting point of the exploration, all primitives are expected to be eager by default, *i.e.*, to be materialized. This is because, during the exploration, eager primitives can be turned into views gradually, and finally terminate at a point where as many primitives as possible become views, as we will see.

The following strategy is used to ensure that all primitives are materialized by default:

- For those eager primitives, *e.g.*, *Id*, *UnaryOp*, *BinaryOp* and *Reduce*, there is nothing to be done.
- For the rest of the primitives, including *Map* and other lazy primitives, *e.g.*, *Split* and *Zip*, they will be first treated as source view primitives and then captured by primitive

MayMaterialize to provide their eager versions. The primitive *MayMaterialize* has the signature of $t^{DataT} \mapsto t \xrightarrow{E} t$. Once we decide to materialize the input, *MayMaterialize* can be rewritten by the *Materialize* transformation introduced in section 4.2.5, by using the rule: $MayMaterialize(in) \Rightarrow [[in]]_M$

After applying this strategy, the program at the starting point will only consist of primitive M, eager primitives, and source view primitives. As we will see soon, those primitives from the starting point program will be rewritten gradually to use more source and destination views, and all possible variants can be reached at the end.

6.1.3 Defining Search Space

The search space can be explored from the starting point based on the following properties.

For each *MayMaterialize* primitive, there are two options: keeping it so that its input will be materialized, or removing it should that its input will be treated as a view.

For each source view primitive, there are also two options: turning it to a destination view or keeping it unchanged. To turn it to a destination view primitive, one just needs to change the effect type it takes. For example, to change a $Split^S$ to a destination view, one just needs to let it take a D effect type instead of an S effect type. Nevertheless, turning a source view primitive into an eager primitive is unnecessary because, at the starting point, any source view primitive is already captured by a *MayMaterialize*.

For primitives that are inherently eager, *e.g.*, *Id*, *UnaryOp*, *BinaryOp* and *Reduce*, there is no option available.

In summary, if m is the number of *MayMaterialize* primitives and s is the number of source view primitives, the upper bound of the search space will be 2^{m+s}. This upper bound, however, does not exclude those invalid cases where source view primitives are used as inputs to destination view primitives.

6.1.4 Exploration Rules

The following rules show the options for removing a *MayMaterialize* primitive:

$$MayMaterialize(p^D(e*)) \Rightarrow p^D(e*) \tag{6.3}$$

$$MayMaterialize(p^E(e*)) \Rightarrow p^E(e*) \tag{6.4}$$

$$p^S(e_0, .., e_n) \text{ where } \exists k \in [0, n].\ e_k \text{ match } MayMaterialize(e') \Rightarrow [e_k \backslash e']p^S(e_0, .., e_n) \tag{6.5}$$

$$p^E(e_0, .., e_n) \text{ where } \exists k \in [0, n].\ e_k \text{ match } MayMaterialize(e') \Rightarrow [e_k \backslash e']p^E(e_0, ..e_n) \tag{6.6}$$

Eq.(6.3) and eq.(6.4) allow the removal of *MayMaterialize* primitive when it tries to materialize an eager or a destination view primitive, given that most of the time this materialization is redundent. Eq.(6.5) states that given a function call to a source view primitive p^S with expressions e_0 to e_n as arguments, if there exists a natural number k belongs to $[0, n]$ where the argument e_k patten matches with $MayMaterialize(e')$, e_k can be subtituted to e' in the function call. By doing so, *MayMaterialize* primitive is removed and e' will not be materialized. Eq.(6.6) is similar to eq.(6.5) except it targets eager primitives.

For turning a source view primitive into a destination view primitive, a special function *toDp* need to be defined first. Function *toDp* changes a source view primitive to a destination view primitive by changing the effect type it takes from S to D. For instance, $toDp(Split^S) = Split^D$. The following rules can be applied:

$$p^S(e_0, .., e_n) \text{ where } \nexists k \in [0, n].\ e_k \text{ is a source view primitive call} \Rightarrow toDp(p^S(e_0, .., e_n)) \tag{6.7}$$

Eq.(6.5) states that given a function call to a source view primitive p^S with expressions e_0 to e_n as arguments, if there does not exist a natural number k belongs to $[0, n]$ where the argument e_k is a function call to a source view primitive, p^S can be turned into a destination view primitive.

Rules in eq.(6.3 - 6.7) are designed specially so that no invalid programs, prevented by the effect system in section 4.2.3, can be introduced. In other words, no rewrite will produce a program with a destination view primitive using a source view primitive as input.

Combining the rewriting rules introduced in this subsection allows the exploration begins from the starting point, gradually removes the eager primitives, turns source view primitives into destination view primitives, and terminates at a point where no more rules can be applied.

The Termination of the Rewriting The termination of the rewriting are guaranteed by the rules in eq.(6.3 - 6.7). By applying these rules in any given program, the program will always reach a point where no more rules can be used. The rules in eq.(6.3 - 6.6) remove *MayMaterialize*, the rule in eq.(6.7) turns source views into destination views. Therefore, when no more *MayMaterialize* or source views can be applied to these rules, the rewriting will terminate. Moreover, one of the important properties of these rules is that no new *MayMateralize* or new source view will be created. It assures that these rules will not sabotage each other and prevents the rewriting from never coming to an end.

6.1.5 Example for the Vector Addition

To better understand the concepts proposed in this section, here is an example to show how the presented rewriting rules are used to explore the variants for a vector addition program.

At the functional level, without any view information, the vector addition program is implemented as:

$$\lambda a, b.Map(+, Zip(Tuple(a, b))$$

By transforming it into the starting point:

$$\lambda a, b.MayMaterialize(Map^S(+,$$
$$MayMaterialize(Zip^S($$
$$MayMaterialize(Tuple^S($$
$$MayMaterialize(a), MayMaterialize(b)))))))$$

If we apply eq.(6.5) to remove the last four primitives, the program becomes:

$$\lambda a, b.MayMaterialize(Map^S(+, Zip^S(Tuple^S(a, b))))$$

By rewriting *MayMaterialize* via the *Materialize* transformation presented in section 4.2.5, the program becomes:

$$\lambda a, b.Map^E(+, Zip^S(Tuple^S(a, b)))$$

which introduces no intermediate data structure.

There are many other variants that can be explored as well. For example, if we only remove the last two *MayMaterialize*, the program will become:

$$\lambda a, b.Map^E(+, Zip^S(Tuple^D(Id(a), Id(b))))$$

However, compared with the previous program, this program is less efficient because an intermediate structure is created by materializing the result from a tuple.

This subsection has shown an example for manually using the rewrite rules from eq.(6.3 - 6.7) to explore the program's variant. In the next section, searching strategies will be presented to automate this exploration process.

```
1     def perf(expr: Expr): Double = {
2       expr.visit({
3         case MayMaterialize(Map(f, in, EagerType(), _)) =>
4           in.asInstanceOf[ArrayTypeT].len * perf(f) + perf(in)
5         case Reduce(f, init, in, EagerType(), _) =>
6           in.t.asInstanceOf[ArrayTypeT].len * perf(f) + perf(init) + perf(in)
7           case p@Primitive(_, EagerType(), t) =>
8             perf(t) + p.children.map(perf).sum
9         case other => // do nothing
10      })
11    }
12
13    def perf(t: Type): Double = t match {
14      case ArrayType(et, len) => len.ae.evalInt * perf(et)
15      case tup@TupleType(fst, snd) => perf(fst) + perf(snd)
16      case other => 1 // scalar types
17    }
```

Figure 6.1: Pesudocode for the Performance Model

6.2 Exploration Strategies

As mentioned, after transforming the program into the starting point, the upper bound of the search space is 2^{m+s} where m is the number of M primitives and s is the number of source view primitives. The search space is exponential, and it is impossible to iterate all possibilities when m and s are big. Therefore, for tuning a given program, two strategies are presented in this section: the heuristic strategy and the random strategy.

6.2.1 Heuristic Strategy

The heuristic strategy is based on a performance model to filter out the variants with potential poor runtime performance so that the search space can be significantly narrowed. The adopted performance model estimates the runtime performance of a given expression based on the potential memory allocation.

```
1  def getBestVariants(exprs: Seq[Expr], top: Int): Seq[Expr] =
2    exprs.sortBy(perf).reverse.take(top)
3
4  def heuristicExplore(root: Expr): Seq[Expr] = {
5    val q = Queue()
6    val result = Seq()
7    q.enqueue(root)
8    result.add(q)
9    while(q.isNotEmpty) {
10     val curVariant = q.dequeue()
11     val newVariants = applyRules(curVariant, rules)
12     val bestVariants = getBestVariants(newVariants, top) // narrow down search space
13     q.enqueue(bestVariants)
14     result.add(bestVariants)
15   }
16   return getBestVariants(result, top)
17 }
```

Figure 6.2: Pesudocode for the Heuristic Exploration

Figure 6.1 shows the implementation of the performance model. $perf$ returns a score by calculating the potential memory allocation. A higher score means more allocation required and thus worse expected performance. Since *Map* and *Reduce* will generate for-loops, line 4 and line 6 multiple their lengths with the scores obtained from visiting their input functions. Line 8 is for any other eager primitives whose scores are based on their types.

Figure 6.2 provides the pesudocode for the heuristic strategy. `getBestVariants` takes a sequence of variants as input and returns the best `"top"` variants according to the performance model. Function `heuristicExplore` expects a program at the starting point as input. Line 5 - 8 sets up the data structures before the exploration begins. Line 9 presents a while-loop that stops when `q` is empty, and this is also where the exploration ends and returns. Line 10 dequeues an expression as `curVariant`. Line 11 uses function `applyRules` to obtains a sequence of new variants where rules in section 6.1.4 are applied to `curVariant`. For instance, if `n` places in `curVariant` can be rewritten by the given rules, `newVariants` will be a sequence of `n` expression. Line 12 narrows the search space by filtering out the vari-

```
1  def randomExplore(root: Expr): Seq[Expr] = {
2    val q = Queue()
3    val result = Seq()
4    q.enqueue(root)
5    result.add(q)
6    while(q.isNotEmpty) {
7      val curVariant = q.dequeue()
8      val newVariants = applyRules(curVariant, rules)
9      val randomVariants = pickRandomly(newVariants, n) //narrow down search space
10     q.enqueue(randomVariants)
11     result.add(randomVariants)
12   }
13   return pickRandomly(result, n)
14 }
```

Figure 6.3: Pesudocode for the Random Exploration

ants with potentially poor performance. Line 13 enqueue the new founded `bestVariants` to continue the exploration. Line 14 adds those best variants as possible results. Finally, when the exploration ends, line 16 returns the best `"top"` variants from `result`.

During the process, we explore the search space by narrowing it down via the performance model. Eventually, the search terminates when no more rewrites can be found. And it is guaranteed that the search will always terminate, as indicated in section 6.1.4. At the end, the best `"top"` variants are returned.

6.2.2 Random Exploration Strategy

Another way to deal with the exponential search space is to search randomly. Let's assume that function `pickRandomly(exprs,n)` randomly picks `n` expressions from the input `exprs`. Figure 6.3 shows the implementation of the random exploration. One of the main differences compared with the heuristic implementation lies in line 9, where `pickRandomly` is used to randomly pick `k` variants in each step to narrow the search space. When the search process ends, line 13 uses `pickRandomly` again to randomly pick `n` expressions as result to return.

6.3 Summary

This section aims to solve the problem of how to automatically attach view information to the input program, which is an essential component for making this work practical in real-world production. It has shown the search space, the starting point, and available rules of searching. Due to the exponential search space, a heuristic and a random strategy have been presented. The next chapter focuses on evaluating the whole code generation approach we have discussed so far. It also evaluates the two automatic strategies and compares them with the hand-tuned strategy.

Chapter 7

Evaluation

The previous chapters have introduced the whole lowering process and the automatic exploration strategies for generating imperative code. This chapter focus on the evaluation of the generated code compared with the prior work on DPS. It first introduces the experimental setup. Then, the focus shifts to the result from the comparison between the approaches proposed in this thesis, including the automatic exploration strategies, and the references in 11 real-world benchmarks. Finally, a use-case for the 2.5D tiling optimization is presented to demonstrate the proposed approach can be used for generating OpenCL code for GPU and encoding optimizations.

7.1 Experimental Methodology

Hardware and Operating System The experiment machine is equipped with an AMD 4800H CPU @ 2.9GHz and 16GB DDR4 RAM @ 3200Mhz. The operating system is Ubuntu 21.10. The generated C code is compiled by Clang version 13.0.0. The `-O3` optimization is used for compilation as the main reference $\widetilde{F}$ [35] also uses this flag in its experiments.

Measurements The evaluation focuses on the runtime performance and memory consumption. For the runtime performance, function `clock` in the C standard library is used to

measure the execution time. For the memory consumption, GNU time [1] is used to obtain the maximum memory consumption during execution. The average coefficient of variation (the ratio of the standard deviation to the mean) cross all 11 benchmarks is 2.59% in terms of runtime performance and 0.00% in terms of memory consumption.

Benchmarks There are 11 benchmarks chosen for the evaluation. There are 5 benchmarks from [35], including Three Vectors Addition (3Add), Cross Product (Cross), Bundle Adjustment (BA), Gaussian Mixture Model (GMM) and Hand Track (HT). And 6 benchmarks from PolyBench [32]: Matrix Multiplication (MM), 2MM, 3MM, jacobi1D, jacobi2D and seidel2D. These matrix multiplication and stencil workloads are chosen from PolyBench because they are representative and widely used in many areas such as machine learning and computer vision.

Approaches Evaluated In this section, the following approaches are compared:

- *All-views*: This approach tries to use lazy primitives to views to avoid using intermediate data structures. All views will be removed eventually, and no intermediate data structure will be created. However, counter-intuitively, the lack of intermediate data structures may reduce performance, due to lower data locality.

- *Hand-tuned*: The hand-tuned approach uses some primitives lazily and others are turned eager. This is controlled manually by the user to trade off performance and memory consumption.

- *Heuristic*: In the heuristic approach, code is generated by the heuristic strategy discussed in section 6.2. For each benchmark, the top 50 programs considered to have the best performance by the heuristic strategy are generated and run. The one with the shortest execution time is chosen for report in Figure 7.1 and 7.2.

[1] https://ftp.gnu.org/gnu/time/

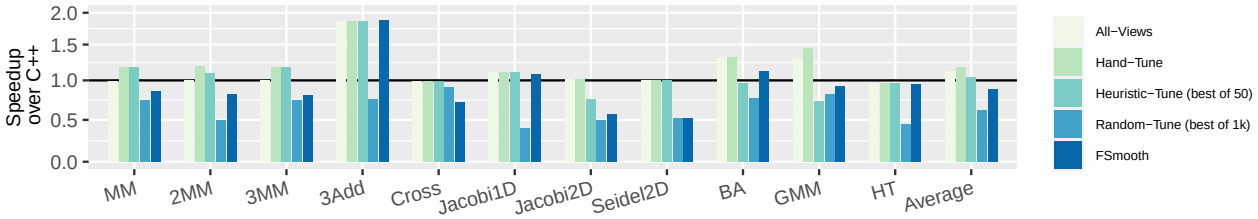

Figure 7.1: Evaluation result in terms of runtime performance. The higher, the better.

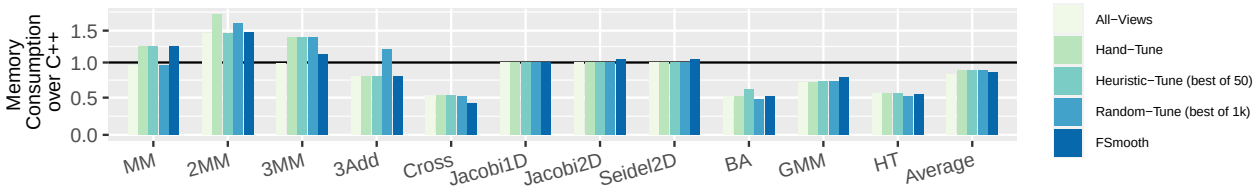

Figure 7.2: Evaluation result in terms of memory consumption. The lower, the better.

- *Random*: In the random approach, code is generated by the random strategy discussed in section 6.2. For each benchmark, 1000 programs are generated and run. The one with the shortest execution time is chosen for report in Figure 7.1 and 7.2.

- $\widetilde{F}$ [35] introduced the concept of DPS and compiles a high-level functional array-processing programs to C code. The compiler is downloaded from the authors' repository[2]. The generated C code for 3Add, Cross, BA, GMM, and HT are from the same repository. With several versions of C code generated by $\widetilde{F}$, the most optimized one in DPS is chosen. For some benchmarks, however, the $\widetilde{F}$ compiler fails to produce valid C code due to some bugs when turning on all the optimizations available. In these cases, we turn down the optimizations until the C code produced is correct.

- *Idiomatic C++*: The C++ code for 3Add, Cross, BA, GMM, and HT is downloaded from the same $\widetilde{F}$ repository, and the rest are implemented by hand.

7.2 Experimental Results

Figure 7.1 and figure 7.2 presents the evaluation result for 11 benchmarks on the CPU with idiomatic C++ as baseline. The absolute values are listed in table B.1 and table B.2. All reported numbers are obtained by executing the programs 10 times and calculating the average values.

In terms of runtime performance, compared with the idiomatic C++ approach, the all-views, hand-tuned, heuristic and random approach achieve an average of 1.11×, 1.18×, 1.07× and 0.65× speedup. $\widetilde{F}$ achieves 0.88× a speedup over the idiomatic C++ approach. In terms of memory consumption, compared with the idiomatic C++ approach, the all-views, hand-tuned, heuristic and random approach require on average of 0.82×, 0.88×, 0.87× and 0.87× space over idiomatic C++. $\widetilde{F}$ requires 0.85× space over the idiomatic C++ approach.

The evaluation shows that the hand-tuned approach outperforms $\widetilde{F}$ and idiomatic C++ code. The comparison between the all-views approach and the hand-tuned approach shows the advantage of providing the fine-grained controls on the eagerness of view primitive to the end-users. The evaluation also shows that the heuristic method is effective since it is on par with the hand-tuned approach and clearly outperforms the random strategy in most of the benchmarks. More discussion for these two automatic exploration strategies will be left to section 7.3. We now take a deeper look at the result for each individual benchmark.

MM, 2MM, 3MM These workloads include matrix multiplications and additions on matrices with the size of 1024x1024. Matrix multiplication consists of a matrix transposition and dot products:

```
1 MatMul0 = λ mat0, mat1.Map^E(λ row0.Map^E(λ row1.DotProd(row0, row1), mat1)) <|
2     Transpose^S(mat0)
3 MatMul1 = λ mat0, mat1.Map^E(λ row0.Map^E(λ row1.DotProd(row0, row1), mat1)) <|
4     [[Transpose^S(mat0)]]_M
5 DotProd = λ row0, row1.Reduce(0, +) <| Map^E(*) <| Zip^S(Tuple^S(row0, row1))
```

[2] commit *adbe1c* from http://github.com/awf/Coconut

The result from the matrix transposition is materialized in the hand-tuned approach which is implemented as $MatMul1$. Compared with the all-view approach, *i.e.*, $MatMul0$, and idiomatic C++, where the matrix transposition is treated as a source view, the runtime performance is increased by about 17% in the hand-tuned approach. The downside is that it consumes more memory. The flexibility provided by this approach presents opportunities for more optimization choices. $\widetilde{F}$ is generally the worst since it fails to generate the C code with optimizations enabled for these benchmarks.

Jacobi1D, Jacobi2D, Seidel2D A stencil computation updates a multiple-dimension data grid given a fixed pattern based on the neighboring value. Prior works have shown they can be represented [17, 39] using primitives *Slide* and *Pad*. The implementation from this thesis uses the primitive *Concat* to implement *Pad*. Since *Concat* can be used as a source or destination view primitive, as seen in section 4.1, there are two different implementations.

The focus lies on Seidel2D as an example in the rest of this section. Seidel2D convolves a 4096x4096 matrix with a 3x3 all-ones matrix. After convolution, the result will have the shape of 4094x4094. Thus, padding is needed to return a matrix of the same original size.

In the first method, Seidel2D is implemented as follows, where $[-1]$ represents the last elements of an array, $::^S$ is syntax sugar for $Concat^S$, $<|$ is for application, $Conv2$ is for a 2D convolution function:

```
1 PadS = λ mat.Repeat^S(1, mat[0]) ::^S mat ::^S Repeat^S(1, mat[-1])
2 PadS2D = λ mat.PadS <| Map^S(PadS) <| mat
3 Seidel = λ mat.Map^E_2(Conv2) <| Slide^S_2 <|PadS2D(mat)
```

$PadS2D$ produces a source view of the padded matrix. $Slide_2^S$ creates a source view of the neighborhoods is implemented as [17]:

```
Slide^S_2 = λ mat.Map(Transpose^S) <| Map(Slide^S(1, 3)) <| mat
```

```
if(j<1) {
  if(i<1) d = mat[0][0];
  else { if(i-1<4095)
           d = mat[i-1][0];
         else d = mat[-1][0];}
} else {
  if(j-1<4095) {
    if(i<1) d = mat[0][j-1];
    else { if(i-1<4095)
             d = mat[i-1][j-1];
           else d = mat[-1][j-1];}
  } else {
    if(i<1) d = mat[0][-1];
    else { if(i-1<4095)
             d = mat[i-1][-1];
           else d = mat[-1][-1];}}}
```

```
d[0][0] = conv_mat[0][0];
d[0][4095] = conv_mat[0][4093];
for(int i = 0; i < 4094; i++)
  d[0][i + 1] = conv_mat[0][i]
for(int i = 0; i < 4094; i++) {
  d[i+1][0] = conv_mat[i][0];
  d[i+1][4095] = conv_mat[i][4093];
    for(int j = 0; j < 4094; j++)
      d[i+1][j+1] = conv_mat[i][j];}
d[4095][0] = conv_mat[4093][0];
d[4095][4095] = conv_mat[4093][4093];
for(int i = 0; i < 4094; i++)
  d[4095][i + 1] = conv_mat[4093][i];
```

Figure 7.3: Comparison between stencil computation code generated by source views and by destination views. The left side shows the piece code for $PadS2D(mat)[i][j]$ being written into the destination d where if-else statements are used widely. The right side shows the code for padding with $Concat^D$, where a convoluted matrix with the size of 4094x4094 is written into the destination d with the size of 4096x4096.

As seen, these two source views will end up being fused with the operation performed in the Map^E resulting in no intermediate data structures.

However, since $Concat^S$ is used, this will produce loops with checks for boundary conditions in 2D conditions occurring at every iteration as already seen in the motivation example. The body of the inner loop is given on the left-side of Figure 7.3 to illustrate the complicated conditions generated which will have an impact on performance.

The second method, however, leverages destination views to remove the condition checks in the body of the loop generated:

```
PadD = λ row.Repeat^D(1, row[0]^D) ::^D row ::^D Repeat^D(1, row[-1]^D)
Seidel=λ mat.PadD <| Map^E(PadD ∘ Map^E(Conv2)) <| Slide^S_2(mat)
```

Unlike the previous method, $Concat^D$ are destination views so that padding occurs when its source, a convoluted matrix, writes to memory. The generated code on the right of Figure 7.3 suggests a superior implementation for padding that obviates the need for conditions.

While again the focus of the work has been on exposing control to the user, it is not too hard to see how one could transform automatically the first version of the program into the second one to enable the automatic exploration of program variants.

Because the second method will not suffer from time-consuming conditions and has better runtime performance, it is chosen to report the numbers in Figure 7.1 and 7.2. In contrast, $\widetilde{F}$ generates C code following the first approach and is unable to produce code from the second approach without any if-then-else in the loop. This is due to the lack of support for a destination view, or lazy destination at the IR level and the impossibility to express this implementation at the functional level in $\widetilde{F}$.

Three Vectors Addition 3Add adds three vectors with 2^{24} elements each into one. Both hand-tuned and all-views method have the following implementation that uses source views to ensure that no intermediate array is produced:

```
3Add = λ v_0, v_1, v_2. Map^E(+) <| Zip^S <| Tuple^S(v_0) <| Map^S(+) <|
    Zip^S(Tuple^S(v_1, v_2))
```

Correspondingly, $\widetilde{F}$ also has a ruled-based loop fusion strategy applied to achieve the same effect. In contrast, the C++ code does not fuse the two vector additions and thus results in an intermediate array created slowing down the runtime performance.

Cross Cross is used between two vectors with three elements each. The programs for all-views and hand-tuned are the same as the follows:

```
Cross = λ v0,v1.ElemWiseVecSub(
  ElemWiseVecMul(Permute^S(λ i.(i+1)%3, v0), Permute^S(λ i.(i+2)%3, v1)),
  ElemWiseVecMul(Permute^S(λ i.(i+2)%3, v0), Permute^S(λ i.(i+1)%3, v1)))

ElemWiseVecSub = λ v0,v1. Map^S(-) <| Zip^S(Tuple^S(v0, v1))
```

```
6 ElemWiseVecMul = λ v0,v1. Map^S(*) <| Zip^S(Tuple^S(v0, v1))
```

Permute is used for permuting the input vectors and will be removed as a source view. Therefore, the generated C code by the all-views and hand-tuned strategies produces no intermediate results. Similarly, $\widetilde{F}$ uses array comprehension for implementation so that no intermediate data structures are created. However, the C code generated by $\widetilde{F}$ maintains a struct *array_number_t* with a length and a pointer to represent an array, which slows down the runtime performance.

Hand-Tracking HT [44] is a computer vision task that tracks a real hand and fits the model into the depth information observed by the depth sensor. One of the main functions of hand-tracking is used for benchmarking. The implementation for all-views and hand-tuned strategy is the same and is shown in appendix A.3. Because there is no particular opportunity to exploit different strategies for the views or for fusing, all approaches have comparable performance.

Bundle Adjustment BA [45] is an important problem in computer vision. It refines a visual reconstruction and estimates jointly optimal 3D structure and camera internal parameters to estimate a projection of a 3D point on a camera. It calculates the projected coordinates of a 3D point by a camera and a model of the radial distortion of the lens.

Figure 7.1 shows the runtime difference for running the *Project* function ten million times. The implementation of bundle adjustment for both all-views and hand-tuned strategy is attached in appendix A.1. Due to the nature of the benchmark, there is no particular opportunity to exploit different strategies for the views or for fusing. The *Project* function only has opportunities for removing the source that comes from *Slice*, *VectorAdd*, and *VectorMul*, which is also provided in $\widetilde{F}$ and easy to implement in idiomatic C++. Since all-views and hand-tuned also have the same implementation in these workloads, the runtime performance for all the approaches is very close.

Gaussian Mixture Model GMM [33] is an unsupervised machine learning method commonly used in computer vision applications and data clustering tasks. Both idiomatic C++ and $\widetilde{F}$ can remove all intermediate arrays. Thus, their runtime performance is similar. Materializing some computation results that are used multiple times from the hand-tuned method speeds up the code compared with all-view's one. In the hand-tuned approach, as shown in appendix A.2, the function call to $Qtimesv$ is materialized so that it speeds up the runtime performance by around 39% over all-views. This, again, highlights the advantage of being able to have fine-grained control over the views.

7.3 Evaluation Result for Automatic Exploration

As we have seen earlier in figure 7.1, the heuristic strategy is on par with the hand-tuned strategy and outperforms the random strategy. For a better understanding of the performance distribution when tuning with different combinations of source views, destination views, and eagernesses, Figure 7.4 shows the runtime performance of the 1000 randomly generated code for each benchmark. Histograms with kernel density estimation are used to visualize the distribution of the runtime performance. Since some code generated by the random approach is extremely inefficient, a 10 seconds timeout is set for MM, 2MM, 3MM, BA, GMM and HT; a 20 seconds timeout is set for Jacobi2D and Seidel2D. The execution time for any code that executes beyond the timeout will be treated as 10 or 20 seconds.

As seen in figure 7.4, like looking for a needle in a haystack, only a few variants can finish within the timeout and be evaluated. By inspecting the generated code, most variants cannot finish as too many intermediate data structures are being created. It is not surprising given that the search space is exponential, as mentioned in section 6.1, and the chance is low for the random strategy to hit those efficient variants with fewer primitives materialized.

While the random strategy cannot always give desirable variants, the red dashed vertical lines in figure 7.4, representing the runtime performance of the code generated by the

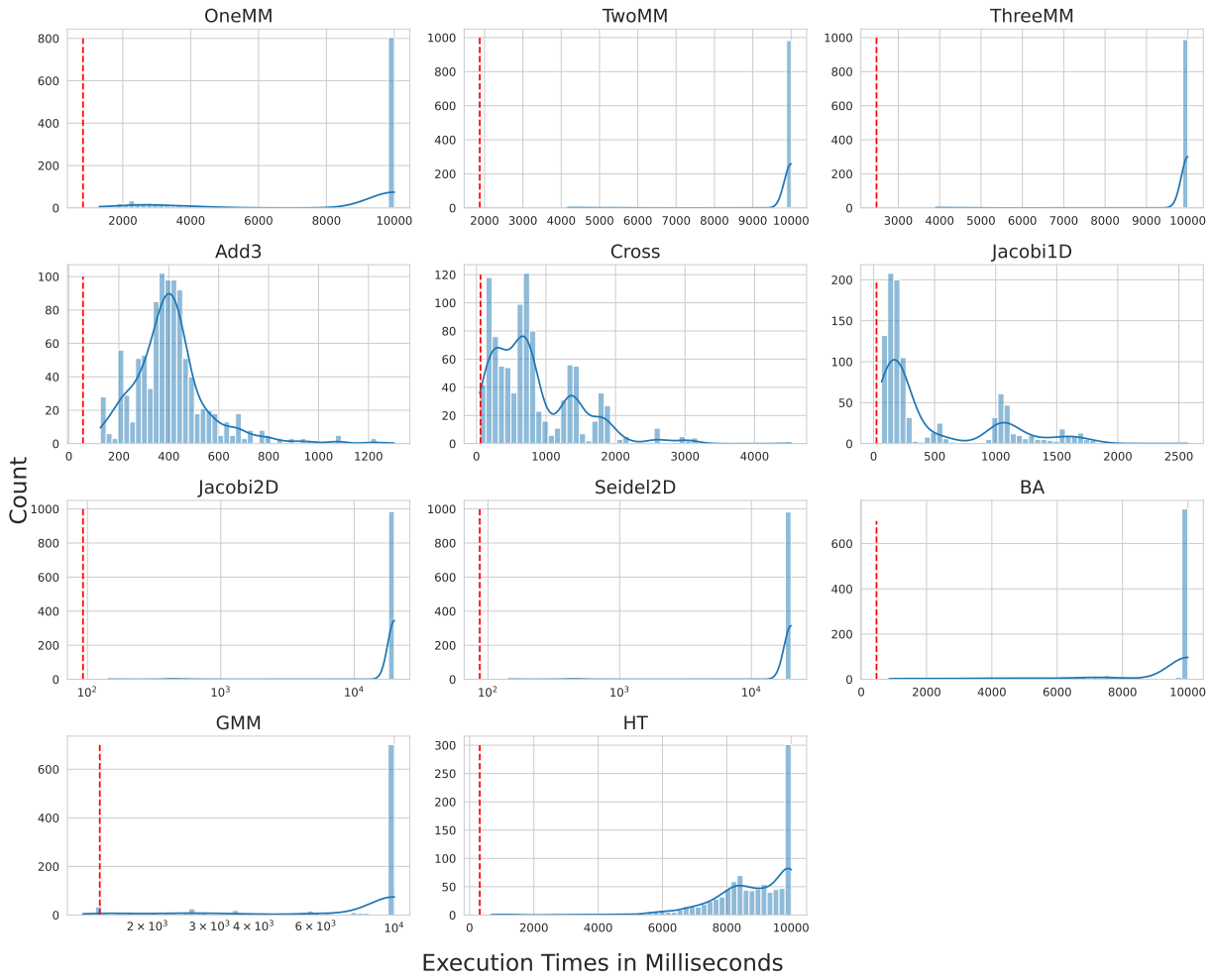

Figure 7.4: Histograms for the evaluation result of random strategy. The red dashed vertical line stands for the execution time of the code from the heuristic approach.

heuristic strategy, again indicates the effectiveness of the heuristic strategy compared with the random strategy.

7.4 Jacobi3D 2.5D Tiling OpenCL Use-case

This use-case targets OpenCL and is here to demonstrate another advantage of using destination views explicitly. A vanilla Jacobi3D without considering padding is (Conv3 is a function that stands for a 3D convolution):

$$\texttt{Jacobi3D} = \lambda\ \texttt{src.}\mathbf{Map}^{\mathbf{E}}_{\mathbf{3}}\texttt{(Conv3,}\mathbf{Slide}^{\mathbf{S}}_{\mathbf{3}}\texttt{(size,step,src))}$$

2.5D tiling is an optimization for iterative 3D stencils, which iterates over two spatial dimensions in parallel and sequentially in the third dimension [49, 29]. The sequential iteration in the third dimension reduces the number of reads from memory since most of the elements loaded from memory can be reused at the next iteration.

This optimization has been explored and exploited in the context of the Lift compiler showing that significant performance improvement is possible [17, 39]. However, to support this optimization, Lift introduced a new primitive *Mapseqslide* to create a moving window iterating the third dimension and applying the input function to the window:

```
1 Jacobi3D = λ src.Map_2^E(Mapseqslide(Conv3,size,step)) <| Transpose^S <| Map^S(Transpose^S)
2     <| Slide_2^S(size,step) <| Map^S(Transpose^S) <| Transpose^S(src)
```

While this approach works, it is far from being ideal since a new primitive needs to be introduced. Instead, using the concept of destination view, it is possible to express this primitive as a combination of the existing primitives:

$$Mapseqslide(f, step, size, src) = Map^{D}(f, Slide^{E}(size, step, src))$$

In this case, the $Slide^{E}$ is eager, generating a template C code that contains a rolling window to reuse elements across iterations, as shown in eq.(5.15). Moreover, the Map^{D}, a destination view, only affects the way the slide writes to memory.

By applying the rolling windows and using Map^{D} as destination view, the corresponding piece of generated OpenCL code for the Jacobi3D benchmark is shown in Figure 7.5.

Evaluation Jacobi3D7pt, Jacobi3D15pt and Jacobi3D27pt are chosen for evaluation, each with a different number of neighborhood points. With the input size as 128x128x128 experiments are run on an NVIDIA 940M GPU and use an OpenCL backend where arrays are represented flattened in memory.

```
float wd[3][3][3];
int gx = get_global_id(0);
int gy = get_global_id(1);
for(int i = 0; i < 3; i++) {
  for(int j = 0; j < 3; j++) {
    wd[i][j][1] = src[gx+i][gy+j][0];
    wd[i][j][2] = src[gx+i][gy+j][1]; }}
for(z=0; z<126; z++){
  for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++) {
      wd[i][j][0] = wd[i][j][1];
      wd[i][j][1] = wd[i][j][2];
      wd[i][j][2] = src[gx+i][gy+j][z+2]; }}
  conv3(wd, &d[z]);}
```

Figure 7.5: Generated OpenCL code for Jacobi3D with rolling windows applied

Neighborhood Points	7pt	15pt	27pt
Speedup	1.07x	1.09x	1.63x

Table 7.1: Evaluation Result for 2.5D Tiling

Table 7.1 shows that using 2.5D tiling results in a speedup of 1.07×, 1.09×, and 1.63× for the 7pt, 15pt and 27pt workloads respectively, over the implementation without applying this optimization.

The preliminary result on 2.5D tiling is similar to the prior result [39] that needs to introduce a specialized primitive for this use-case.

7.5 Summary

This chapter has provided a detailed evaluation of several approaches for generating imperative code, which shows that the all-views and hand-tuned strategy outperforms the idiomatic C++ and the state of the art, which is $\widetilde{F}$, thanks to the use of destination views and the ability to provide a fine-grained control on whether a primitive is eager or a view to

end-users. Also, the evaluation result of the heuristic strategy has demonstrated that it can generate code that is on par with the hand-tuned approaches, and is way better than the random strategy. Furthermore, as an OpenCL use-case, the section has presented how to encode the 2.5D titling optimization in a composable way and reach a similar performance to the prior works.

Chapter 8

Conclusion

This thesis has presented an approach to compiling a high-level array-based functional IR to high-performance imperative code. It has combined the existing techniques on DPS with the Lift views system by extending the notion of view to destinations. The evaluation has shown that this approach can generate imperative code comparable with state-of-the-art.

8.1 Summary of Contributions

This thesis first has provided the necessary background knowledge for understanding this thesis in chapter 2, which includes the discussion of lambda calculus, Lift IR and its high-level primitives, the definition of DPS and the application of views in Lift. Then, chapter 3 has shown the related works on the functional approaches for generating high-performance code, the usage of DPS and the removal of intermediate data structures.

Based on the background and the related works, chapter 4 has proposed the functional level and the imperative level IRs design, including the types, the primitives and the effect system to carry view information. We have also seen how it is possible to control when an expression should be materialized in memory directly or should be a view at the functional level. Following closely, chapter 5 has shown the lowering process from the functional IR to imperative code step by step, which involves the combination of DPS and views to provide

memory management and remove all intermediate data structures. As seen in chapter 6, two automatic exploration strategies have been provided for searching the efficient programs automatically.

As seen in chapter 7, introducing destination views is crucial to allow the expression of programs that result in highly optimized C code. The evaluation has shown that C code generated by the approaches from this thesis is on par with idiomatic C++ code, and outperforms the *DPS* reference work in terms of execution time and memory consumption, thanks to the introduction of destination views and the ability to provide a fine-grained control on whether a primitive is eager or a view to end-users. It has also shown the effectiveness of the heuristic strategy for automatic exploration. Moreover, it has enabled the expression of optimizations such as 2.5D tiling in a composable manner, leading to high-performance code on GPUs.

8.2 Guidance on Using Views

As indicated in chapter 6, it is non-trivial to make optimal choices when using views. Thus, this section aims to provide general insights on properly using views proposed in this thesis for end-users to develop computation kernel and resolve performance related issues. The first suggestion is to rely on the automatic exploration approaches provided in chapter 6. As shown by the experiments, the heuristic-based strategy can provide programs with good performance most of the time. The second suggestion is to manually implement the program with views as the same as how the hand-tuned programs are implemented in chapter 7. Based on the experiment results, it is benefitical for source views to be used as many as possible to avoid intermediate data structures unless the following two situations happen: first, when the result of a primitive is used multiple times or locality is critical, the result should be materialized. Secondly, when the result of a primitive needs to be written

into different locations, *e.g.*, array concatenation, destination view should be used to avoid condition checks.

8.3 Critical Analysis

In hindsight, several deficiencies in this work could be tackled differently. The first problem is how the view information is embedded in the IR. The method proposed by this thesis is through an effect system so that primitives take an extra type parameter for an effect type and embed the effect type in the function type. Therefore, examining the effect types can determine whether a primitive is eager, source view, or destination view primitive and prevent invalid programs. However, introducing an effect system increases the complexity of this work, especially for understanding. An alternative approach is embedding the view information to the data type instead of the function type, which may achieve a similar outcome in storing the view information. While the latter approach sounds easier to understand, it might introduce additional complexity in implementation, especially when defining the signatures of the primitives used in the IRs, as care must be taken for the presence of the view information. Having the view information handled by a standalone effect system is neater and easier in terms of the implementation. Hence, there is a trade-off between the understandability and the simplicity of the implementation.

The second issue comes from the way destination views are handled in the $\mathcal{DPS}$ transformation. This process might be convoluted because it involves the concept of the inverse function. For every supported destination primitives, one rule needs to be added to the $\mathcal{I}$ transformation. Also, there are some special primitive, such as Map^D and $Tuple^D$ needed to be handled specially. Nevertheless, with the concept of destination views, it seems there is no better way to diminish the inherent complexity. An alternative design may resort to the push array that can be used to express various high-level primitives. While each supported

primitive still requires one rule for encoding by push arrays, the lowering of the push array is uniform without exceptional cases that need special attention.

Thirdly, *BinaryOp* and *SingleOp* are always eager primitives, though it is not always an ideal design decision. For example, the 3Add workload is implemented as:

$$Map^{E}(+, Zip^{S}(Tuple^{S}(\boldsymbol{Map^{S}}(+, Zip^{S}(Tuple^{S}(a, b))), c)))$$

The problem here is that while $\boldsymbol{Map^{S}}$ is a source view so that an intermediate array can be avoided, its input function is eager, as + is a syntax sugar for the eager primitive *BinaryOp*. It is not intuitive since *Map* and its input function should always have the same effect type. The root of this problem is that *BinaryOp* and *SingleOp* cannot be used as source view primitives, which is necessary as long as their results are eventually materialized by *Id*. Therefore, a better design choice is to relax the restriction on *BinaryOp* and *SingleOp* so that they can be source view or even destination view primitives.

8.4 Future Work

Several avenues are open to this work in the future. First, as an improvement, it can have more exploration on the application of DPS in a parallel environment. It includes the support for vectorization so that this language's computation and materialization, *i.e.*, assignment, should not only center on scalars. When targeting GPU, it also requires the awareness of the memory hierarchy in the GPU architecture, for which the $\mathcal{DPS}$ transformation needs to decide which memory area allocations should be placed in.

Meanwhile, it will also be interesting if it can support dynamic arrays, which are arrays with lengths that are only available at runtime. With a proper array size inference algorithm and more expressive natural number types, dynamic memory allocations can be provided by DPS. Having dynamic arrays allows more operations that involve arrays with uncertain sizes at the compile-time, such as *Filter*, and recursive functions with various input sizes at each function call.

Besides, while destination views certainly help achieve better performance for some workloads, such as the use-case of *Concat* and the stencil computation, it does not always lead to a better result. In following work, other use-cases can be used for evaluation and for providing more insights into destination views. Fast Fourier transform is one of the use-cases that may be worth exploring since it involves a data permutation where computation results need to be duplicated multiple times. Using destination view allows expressing a computation that only computes once but is written to multiple locations directly. *Filter* is another use-case that may be interesting, as the computation that is performed on the filtered result can be encoded by Map^D as destination view and eventually fuse with the for-loop generated *Filter* where the predicate is evaluated.

Appendix A

Benchmark Implementations

This appendix shows the implementions of BA, GMM and HT in the functional level IR. The implementations for other benchmarks are already shown in chapter 7.

A.1 Bundle Adjustment

```
1  VectorAdd = λ v0, v1. Map^S(+, Zip^S(Tuple^S(v0, v1)))
2  VectorSum = λ v. Reduce(+, 0, v)
3  VectorMulScalar = λ v, s. Map^E(λ x.x * s, v)
4  SqNorm = λ v. VectorSum(Map^S(λ x.x^2, v))
5  DotProd = λ v0, v1. VectorSum(Map^S(λ x.(x.0 * x.1), Zip^S(Tuple^S(v0, v1)))
6
7  RadialDistort = λ radParams, proj.
8      let rsq = SqNorm(proj) in
9      VectorMulScalar(proj, radPrams[0] * rsq + 1 + radPrams[1] * rsq * rsq)
10
11 RodriguesRotatePoint = λ rot, x.
12     let sqtheta = SqNorm(rot) in
13     If(sqtheta == 0,
14         let theta = sqrt(sqtheta) in
15         let costheta = cos(theta) in
16         let sintheta = sin(theta) in
17         let thetaInv = 1 / theta in
18         let w = VectorMulScalar(rot, thetaInv)
19         VectorAdd(
20             VectorAdd(VectorMulScalar(x, costheta), VectorMulScalar(Cross(w, x),
                   sintheta)),
```

```
            VectorMulScalar(w, DotProd(w, x) * 1 - costheta)
        ),
        VectorAdd(x, Cross(rot, x))
    )

Project = λ cam, X.
    let Xcam = RodriguesRotatePoint(cam[0..3], X - cam[3..6]) in
    let distorted = RadialDistort(cam[9..11], VectorMulScalar(Xcam[0..2],
        VectorMulScalar(1 / Xcam[2]))) in
    VectorAdd(cam[7..9], VectorMulScalar(distorted, cam[6]))

BundleAdjustment = λ a, b, cam. Project(cam, RadialDistort(a, b))
```

A.2 Gaussian Mixture Model

```
VectorPow2 = lambbda v: Map^S(λ e. e**2, v)
VectorExp = lambbda v: Map^S(λ e. exp(e), v)
VectorSubScalar = λ v, s. Map^S(λ e. e - s, v)
VectorMax = λ v. Reduce(λ acc, cur. If^D(acc > cur, Id(acc), ID(cur)), -DBLMAX, v)
  // -DBLMAX is the smallest double literal in the target machine

LogSumExp = λ x.
    let mx = VectorMax(x) in
    log(VectorSum(VectorExp(VectorSubScalar(x, mx)))) + mx

Tri = λ n. (n * (n + 1)) / 2

Qtimesv = λ q, l, v.
  Map^E(λ i.
    Reduce(λ j, acc.
        let k = j - tri(i - 1) in
        If^D(k > 0 && k < i, acc + l[j] * v[k], Id(acc)),
      Counter^S(l.length)) + exp(q[i]) * v[i],
    Counter^S(v.length))

GMM = λ n, xs, alphas, means, qs, ls.
  VectorSum(
    Map^S(λ x. LogSumExp(
      Map^S(λ e.
        e.0 + VectorSum(e.1) - SqNorm(Qtimesv(e.1), e.2, VectorSub(x, e.3)) / 2,
        Zip^S(Tuple^S(alphas, qs, ls, means)))), xs)
  ) - xs.length * LogSumExp(alphas) +
```

```
28  VectorSum(Map^S(λ x. SqNorm(VectorExp(x.0)) + SqNorm(x.1), Zip^S(Tuple^S(qs, ls))))/2
```

A.3 Hand Track

```
1  AngleAxisToRotationMatrix = λ n, angleAxis.
2    If^D(n < 0.001,
3      (Repeat^D(1, Id(1)) ::^D Repaet^D(1, Id(0)) ::^D Repeat^D(1, Id(0))) ::^D
4      (Repeat^D(1, Id(0)) ::^D Repaet^D(1, Id(1)) ::^D Repeat^D(1, Id(0))) ::^D
5      (Repeat^D(1, Id(0)) ::^D Repaet^D(1, Id(0)) ::^D Repeat^D(1, Id(1)))
6      ,
7      let n = sqrt(SqNorm(angleAxis)) in
8      let x = angleAxis[0] / n in
9      let y = angleAxis[1] / n in
10     let z = angleAxis[2] / n in
11     let s = sin(n) in
12     let c = cos(n) in
13     (Repeat^D(1, x * x + (1 - x * x) * c)  ::^D
14      Repeat^D(1, x * y * (1 - c) - z * s)  ::^D
15      Repeat^D(1, x * z * (1 - c) + y * s)) ::^D
16     (Repeat^D(1, x * y * (1 - c) + z * s)  ::^D
17      Repeat^D(1, y * y * (1 - y * y) * c)  ::^D
18      Repeat^D(1, y * z * (1 - c) - x * s)) ::^D
19     (Repeat^D(1, x * z * (1 - c) - y * s)  ::^D
20      Repeat^D(1, z * y * (1 - c) + x * s)  ::^D
21      Repeat^D(1, z * z * (1 - z * z) ))
22   )
```

Appendix B

Evaluation Data

	Runtime Performance (milliseconds)						
	No-Views	All-Views	Hybrid	$\widetilde{F}$	Heuristic	Random	C++
BA	14436.22	492.62	492.62	579.67	676.04	853.30	651.69
GMM	-	821.82	742.37	1160.02	1461.06	1309.31	1070.79
HT	13173.52	315.98	315.98	320.01	315.98	676.58	304.37
3Add	480.41	56.46	56.46	56.25	56.46	138.60	105.15
Cross	1919.06	58.55	58.55	81.22	58.55	63.63	57.97
MM	-	982.15	827.35	1128.18	827.35	1303.76	972.76
2MM	-	2044.05	1711.276	2485.62	1874.57	4170.60	2048.52
3MM	-	2920.47	2473.8	3582.34	2473.8	3906.92	2910.66
Jacobi1D	1148.08	23.68	23.68	24.09	23.68	65.17	26.148
Jacobi2D	-	69.38	69.38	122.32	92.66	142.76	70.28
Seidel2D	-	86.56	86.56	165.95	86.56	167.54	86.39

Table B.1: Absolute values for runtime performance evaluation. A dash '-' is used when the execution time is longer than 1 minute. Strategy *No-Views*, not discussed before, does not leverage views to remove intermedate data structures, and thus its generated code has bad performance. Some strategies may end up generating the same code, for which the same execution time will be reported.

	Memory Consumption (kbytes)						
	No-Views	All-Views	Hybrid	$\widetilde{F}$	Heuristic	Random	C++
BA	2176	1988	1988	2028	2376	1864.0	3868
GMM	-	12268	12296	13420	12360	12356.0	17112
HT	2189608	2140	2140	2112	2140	2024	3872
3Add	1705224	525800	525800	525884	525800	787840	658196
Cross	-	1984	1984	1544	1984	1904	3708
MM	-	26520	34428	34388	34428	26676	27744
2MM	-	75432	91720	75848	75432	84104	51700
3MM	-	58920	83480	67600	83592	83544	60016
Jacobi1D	1574612	264044	264044	265356	264148	264144	276876
Jacobi2D	-	1050936	1050936	1102812	1050888	1051004	1051944
Seidel2D	-	1050984	1050984	1102836	1050972	1051376	1051952

Table B.2: Absolute values for memory consumption evaluation. A dash '-' is used when the execution time is longer than 1 minute and the memory consumption is not available.

Bibliography

[1] Ankner, J., and Svenningsson, J. D. An EDSL approach to high performance Haskell programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, Association for Computing Machinery, pp. 1–12.

[2] Appel, A. W. *Modern Compiler Implementation in C*. Cambridge university press.

[3] Atkey, R., Steuwer, M., Lindley, S., and Dubach, C. Data parallel idealised algol. arXiv:1710.08332.

[4] Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., and Persson, A. The Design and Implementation of Feldspar: An Embedded Language for Digital Signal Processing. In *Implementation and Application of Functional Languages*, J. Hage and M. T. Morazán, Eds., vol. 6647. Springer Berlin Heidelberg, pp. 121–136.

[5] Bour, F., Clément, B., and Scherer, G. Tail Modulo Cons. arXiv:1312.5602.

[6] Cardelli, L., Martini, S., Mitchell, J. C., and Scedrov, A. An extension of system F with subtyping. In *Information and Computation*, Springer-Verlag, pp. 750–770.

[7] Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., and Grover, V. Accelerating haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming - DAMP '11*, ACM Press, p. 3.

[8] Coutts, D., Leshchinskiy, R., and Stewart, D. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on*

Functional Programming, ICFP '07, Association for Computing Machinery, pp. 315–326.

[9] COUTTS, D., STEWART, D., AND LESHCHINSKIY, R. Rewriting haskell strings. In *International Symposium on Practical Aspects of Declarative Languages*, Springer, pp. 50–64.

[10] DUBACH, C., CHENG, P., RABBAH, R., BACON, D. F., AND FINK, S. J. Compiling a high-level language for GPUs: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, Association for Computing Machinery, pp. 1–12.

[11] ELLIOTT, C., FINNE, S., AND DE MOOR, O. Compiling embedded languages. In *Journal of Functional Programming*, vol. 13, Cambridge University Press, pp. 455–481.

[12] FLANAGAN, C., SABRY, A., DUBA, B. F., AND FELLEISEN, M. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 237–247.

[13] FRIEDMAN, D. P., AND WISE, D. S. Unwinding stylized recursions into iterations. In *Comput. Sci. Dep., Indiana University, Bloomington, IN, Tech. Rep*, vol. 19.

[14] GILL, A., LAUNCHBURY, J., AND PEYTON JONES, S. L. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture - FPCA '93*, ACM Press, pp. 223–232.

[15] GIRARD, J.-Y. The system F of variable types, fifteen years later. In *Theoretical Computer Science*, vol. 45, Elsevier, pp. 159–192.

[16] GRELCK, C., AND SCHOLZ, S.-B. SAC—A Functional Array Language for Efficient Multi-threaded Execution. In *International Journal of Parallel Programming*, vol. 34, pp. 383–427.

[17] Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., and Dubach, C. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ACM, pp. 100–112.

[18] Hennessy, J. L., and Patterson, D. A. A new golden age for computer architecture. In *Communications of the ACM*, vol. 62, pp. 48–60.

[19] Henriksen, T., Serup, N. G. W., Elsman, M., Henglein, F., and Oancea, C. E. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, Association for Computing Machinery, pp. 556–571.

[20] Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., and Lippmeier, B. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, Association for Computing Machinery, pp. 261–272.

[21] Kiselyov, O., Biboudis, A., Palladinos, N., and Smaragdakis, Y. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, Association for Computing Machinery, pp. 285–299.

[22] Köpcke, B., Steuwer, M., and Gorlatch, S. Generating efficient FFT GPU code with Lift. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, FHPNC 2019, Association for Computing Machinery, pp. 1–13.

[23] Larus, J. R. Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES.

[24] Loader, R. *Notes on Simply Typed Lambda Calculus*. University of Edinburgh.

[25] Mainland, G., and Morrisett, G. Nikola: Embedding compiled GPU functions in Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, Association for Computing Machinery, pp. 67–78.

[26] Michaelson, G. *An Introduction to Functional Programming through Lambda Calculus*.

[27] Minamide, Y. A functional representation of data structures with a hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, Association for Computing Machinery, pp. 75–84.

[28] Morris Jr, J. H. Lambda-calculus models of programming languages. PhD Thesis, Massachusetts Institute of Technology.

[29] Nguyen, A., Satish, N., Chhugani, J., Kim, C., and Dubey, P. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, IEEE Computer Society, pp. 1–13.

[30] Nielson, F., Nielson, H. R., and Hankin, C. Type and Effect Systems. In *Principles of Program Analysis*. Springer, pp. 283–363.

[31] Pizzuti, F. Implementing an OpenMP backend for the Lift compiler. MSc Thesis, The University of Edinburgh.

[32] Pouchet, L.-N., and Yuki, T. https://sourceforge.net/projects/polybench/. PolyBench/C.

[33] Reynolds, D. A. Gaussian mixture models. In *Encyclopedia of Biometrics*, vol. 741, Berlin, Springer.

[34] Reynolds, J. C. Towards a theory of type structure. In *Programming Symposium*, Springer, pp. 408–425.

[35] Shaikhha, A., Fitzgibbon, A., Peyton Jones, S., and Vytiniotis, D. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, Association for Computing Machinery, pp. 12–23.

[36] Steuwer, M., Fensch, C., Lindley, S., and Dubach, C. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, Association for Computing Machinery, pp. 205–217.

[37] Steuwer, M., Remmelg, T., and Dubach, C. Lift: A functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, IEEE Press, pp. 74–85.

[38] Steuwer, M., Remmelg, T., and Dubach, C. Matrix multiplication beyond auto-tuning: Rewrite-based GPU code generation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, Association for Computing Machinery, pp. 1–10.

[39] Stoltzfus, L., Hagedorn, B., Steuwer, M., Gorlatch, S., and Dubach, C. Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift. In *ACM Transactions on Architecture and Code Optimization*, vol. 16, pp. 52:1–52:25.

[40] Strachey, C. Fundamental Concepts in Programming Languages. In *Higher-Order and Symbolic Computation*, vol. 13, pp. 11–49.

[41] Svenningsson, J. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, Association for Computing Machinery, pp. 124–132.

[42] Svensson, B. J., and Svenningsson, J. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing - FHPC '14*, ACM Press, pp. 43–52.

[43] Svensson, J., Sheeran, M., and Claessen, K. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Symposium on Implementation and Application of Functional Languages*, Springer, pp. 156–173.

[44] Taylor, J., Stebbing, R., Ramakrishna, V., Keskin, C., Shotton, J., Izadi, S., Hertzmann, A., and Fitzgibbon, A. User-specific hand modeling from monocular depth sequences. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 644–651.

[45] Triggs, B., McLauchlan, P. F., Hartley, R. I., and Fitzgibbon, A. W. Bundle adjustment—a modern synthesis. In *International Workshop on Vision Algorithms*, Springer, pp. 298–372.

[46] Wadler, P. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, Springer, pp. 344–358.

[47] Wadler, P. Linear types can change the world! In *Programming Concepts and Methods*, vol. 3, p. 5.

[48] Wadler, P. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pp. 45–52.

[49] Zhang, G., and Zhao, Y. Modeling the performance of 2.5 d blocking of 3D stencil code on GPUs. In *IEEE High Performance Extreme Computing Conference, HPEC*.