Array abstractions for GPU programming

Martin Dybdal

July 13, 2017

Abstract

The shift towards massively parallel hardware platforms for high-performance computing tasks has introduced a need for improved programming models that facilitate ease of reasoning for both users and compiler optimization.

A promising direction is the field of functional data-parallel programming, for which functional invariants can be utilized by optimizing compilers to perform large program transformations automatically. However, the previous work in this area allow users only limited ability to reason about the performance of algorithms. For this reason, such languages have yet to see wide industrial adoption.

We present two programming languages that attempt at both supporting industrial applications and providing reasoning tools for hierarchical data-parallel architectures, such as GPUs.

First, we present TAIL, an array based intermediate language and compiler framework for compiling a large subset of APL, a language which have been used in the financial industry for decades. The TAIL language is a typed functional intermediate language that allows compilation to data-parallel platforms, thereby providing high-performance at the fingertips of APL programmers.

Second, we present FCL, a purely functional data-parallel language, that allows for expressing data-parallel algorithms in a fashion where users at a low-level can reason about data-movement through the memory hierarchy and control fusion will and will not happen. We demonstrate through a number of micro benchmarks that FCL compiles to efficient GPU code.

Resumé

Overgangen til massivt parallel hardware, som platform for opgaver der kræver high-performance computing, har introduceret et behov for forbedrede programmeringsmodeller, der bedre faciliterer ræsonering for både brugere og optimerende oversættere.

En lovende retning er forskningsområdet der omhandler funktionelle data-parallelle programmeringssprog. For sådanne sprog kan funktionelle invarianter benyttes af oversættere til automatisk at udføre optimerende programtransformationer. Tidligere arbejde indenfor dette område har dog ofte kun tilladt brugere begrænset mulighed for at ræsonere om effektiviteten af deres algoritmer, netop på grund af disse automatiske optimeringer. Derfor har sådanne sprog endnu ikke set stor udbredelse i industrielle sammenhænge. Vi præsenterer to programmeringssprog der forsøger både at understøtte industrielle anvendelser og giver mulighed for ræsonering omkring effektivitet, når algoritmerne afvikles på hierarkiske data-parallele hardware arkitekturer, som for eksempel GPU'er.

Vi præsenterer først TAIL, et array baseret mellemsprog og oversættersystem, der gør det muligt at oversætte en stor delmængde af APL, et sprog der har været brugt i den finansielle industri i flere årtier. Sproget TAIL er et typestærkt funktionelt mellemsprog, der tillader oversættelse til data-parallel platforme og dermed giver APL programmører adgang til at generere effektiv kode for opgaver der kræver high-performance computing.

Dernæst præsenterer vi FCL, et funktionelt data-parallelt sprog, der tillader brugeren at udtrykke data-parallele algoritmer i en form hvor brugere kan ræsonere om hvornår data overføres mellem de forskellige hukommelseslag. Dette gøres blandt andet ved at give brugeren kontrol over hvornår array kombinatorer vil eller ikke vil blive fusioneret. Endelig demonstrerer vi at et antal mindre benchmarks skrevet i FCL kan oversættes til effektive GPU programmer.

Contents

Chapter 1

Introduction

The society is increasingly relying on large-scale computing for infrastructure as well as for industrial and scientific developments. In recent years, the number and scale of data processing tasks have increased dramatically. Simultaneously, hardware for high-performance computing have become cheaper than ever, which has made previously untractable problems tractable.

However, while hardware for high-performance computing have become a commodity, taking efficient advantage of the new hardware architectures requires specialists with intimate knowledge of the particular hardware platforms. Decreasing development time and cost for high-performance computing systems will enable organisations to move into new areas faster, cheaper and with a lower barrier for entry. In some domains quick responses are often necessary, and shortening development time thus becomes even more important. The financial sector is an example, where sudden changes in markets necessitate timely reactions embodied as updated financial models and software.

Graphical processing units (GPUs) are a cost-effective technology for many problems in high-performance computing. Traditional CPUs are equipped with automatically managed caches and control logic such as branch predictors or speculative execution, providing low latency processors that can quickly switch between tasks. GPUs, in contrast, are designed for tasks where high throughput is more important than low latency, and trades in the transistors used for control logic and caches for additional compute units (ALUs). GPUs are thus equipped with thousands of individual compute cores and a complex hierarchical memory systems managed by the user.

To take advantage, a programmer needs thousands of simultaneous threads, *data parallel* algorithms with enough similar, but independent tasks, where control-divergence and synchronisation between threads are limited, as well as careful utilisation of the deep memory hierarchies.

This dissertation explores how the use of *purely functional programming languages* can enable improvements in programmer productivity and performance of data-parallel programs written for GPUs. Purely functional programming languages delivers a programming model, where programs are constructed

from operators acting on arrays in bulk. By being pure, the languages provides referential transparency of operations, which allows equational reasoning by users as well as compilers. This in turn enables a wealth of optimisations to be performed.

1.1 Landscape of functional array languages for GPUs

The HIPERFIT research center was establish at University of Copenhagen in the aftermath of the 2008 financial crisis, with the following research goals: "HIPERFIT seeks to contribute to effective high-performance modelling by domain specialists, and to functional programming on highly parallel computer architectures in particular, by pursuing a research trajectory informed by the application domain of finance, but without limiting its research scope, generality, or applicablity, to finance" [15].

To initiate efforts in the direction stated by the HIPERFIT research goals, the first project I did as part of my Ph.D. was to evaluate the state of the art in functional data-parallel languages for GPUs, by implementing a few algorithms from financial engineering. We surveyed the languages Accelerate[28], Nikola[77], NESL/GPU [11], Copperhead [26], and Thrust [83]. Our conclusions from the study was that, current functional languages for data-parallel GPU programming were limited in several ways.[1]

We found that the existing data-parallel languages for GPU programming, provided limited control when implementing high-performance algorithms, which is necessary for certain algorithms to implemented efficiently. For instance, data-layout and data-access patterns are important consideration for obtaining good performance on GPU systems. Being unable to reason about the data-layout or data-access after applying a built-in operation, for example a transposition, made it hard to optimise algorithms.

We also found that certain loop structures could not be expressed, in many of the languages. For instance some algorithms involves construction of small arrays, or parts of arrays small enough to be constructed in a single thread. The languages did provide the necessary sequential loop contructs, necessary for expressing such algorithms.

Languages often tend to provide a single version of each operation for example, transposing an array, performing matrix multiplication, et cetera. However, when arrays are small it may be faster to let all threads perform identical operations, instead of parallelising a 3-by-3 matrix inversion. The surveyed languages did not provide the flexibility of choosing between these different algorithmic strategies.

Performance reasoning and control is important to high-performance computing programmers. Brad Chamberlain from Cray Inc., developer of Chapel [30], presented his vision for an ideal language for high-performance computing in a recent talk at DIKU [30]. He argued that computational scientists and

[1] A preliminary version of this study is available in my Master Thesis [25]. The full comparison was unfortunately never published.

financial engineers wants to express their problems in architecture independent terms, but an ideal language should still allow full control to the HPC programmer whose job is to ensure performance of the final application.

Finally, in several cases we found it necessary to manually flatten algorithms, as the only language supporting the flattening transformation was NESL/GPU [11]. We could also confirm, however, that the flattening transformation caused a huge memory overhead for NESL/GPU, as it used 128 *times* the amount of memory necessary for one of our case studies, where in comparison, we were able to entirely fuse away the array in the implementation of the same program in Thrust [83].

The above conclusions motivated our work. With these lessons in mind we went on to design an experiment that would allow us to make progress in the design of functional array languages for GPUs.

1.2 Thesis

To respond to the identified problems and gaps in existing research on functional data-parallel languages for GPUs, we decided to work on an compiler for the programming language APL, with the hope of eventually targeting GPUs. APL is a an *array programming language*, with a functional core and a large collection of built-in operations acting on arrays. The reason for choosing APL as our language of study, was first of all motivated by the wide usage of APL and descendants in the financial industry through more than forty years. There is thus evidence that the selection of operations provided by APL are suitable for a large range of problems, especially in the domain of finance. The financial institutions backing HIPERFIT only provided a few financial benchmarks we could study. By compiling APL we were able to seperate the concern of whether our language would be adequate for implementing financial algorithms, as this has already been proven for APL.

The second reason for selecting APL is that the built-in array operations in APL are generic and can be implemented by highly parallel algorithms, or quoting Robert Bernecky: "Unlike other languages, the problem in APL is not determining where parallelism exists. Rather, it is to decide what to do with all of it." [12].

We further speculated that APL programmers could not only be used for declarative specification of data-parallel algorithms, but also instrument the compilation process of the underlying lower level language, through annotations at the APL level. In this way the APL programmer would not only be implement his algorithms, but also be able to reason about their performance, and optimize through annotations at the APL-level or by inlining expressions written in one of the lower level language.

The main thesis of this dissertation was thus born: *efficient compilation of the high-level data-parallel language APL, can be achieved through compilation through several typed functional array languages*. This idea is illustrated in Figure 1.1.

In the following sections we will briefly sketch our various work towards this goal.

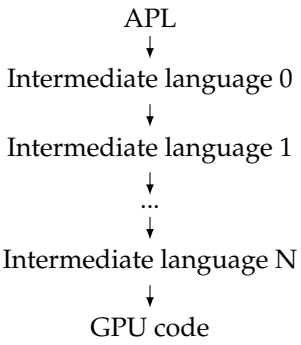

Figure 1.1: Stack of intermediate languages

1.3 Compiling APL into a typed array language

APL is a notation and programming language designed for communicating mathematical ideas and describing computational processes [63]. The language is dynamically typed, with a functional core supporting first and second-order functions and multi-dimensional arrays as the main data structure. The language makes extensive use of special characters for denoting the many different built-in functions and operators, which include prefix-sums (\), array transposition (⍉), and generalized multi-dimensional inner-product.

Traditionally, APL is an interpreted language, although there have been many attempts at compiling APL into low-level code, both in online and offline settings [50, 13, 47, 22]. However, only limited attention have been given from a programming language semantics point of view [92].

In Chapter 3 we present an APL compiler that elaborates a large subset of APL into a typed functional intermediate language, called TAIL, for which we provide a rigorous dynamic semantics and a static type system. The front-end of the APL compiler deals with much of the gory details of the APL language, including infix resolution, scalar extension resolution, resolution of function and operator overloading, and resolution of identity items for reduce operations.

The type system of TAIL provides a shape-polymorphic type system, conceptually close to the language Repa [68], with a wide set of array operations supported on both shapes and arrays. We also make use of subtyping to allow shapes to be used in array contexts. These features are necessary in the translation of APL, as APL does not make a distinction between shapes and arrays, and having access to rank-information on type level makes several optimisations possible.

Chapter 3 is an extension of previously released work [43]. The extensions include sequential loop constructs, mutable arrays with store semantics, that allows many programs requiring sequential loops and irregular array iteration patterns to be implemented. Chapter 3 further extends on our previous work

by documenting that larger benchmarks, such as an option pricer from partner company LexiFi is implementable in the subset of APL.

It should also be mentioned that TAIL has been compiled to both Accelerate (by Masters student) and Futhark (by colleagues at the HIPERFTI research center), which documents that TAIL is suitable intermediate language, for compilation to GPUs.

1.4 Hierarchical functional data-parallelism

After the TAIL project we turned the attention towards compiling TAIL to GPU code. Modern GPUs provide thousands of cores, however, they provide very limited amounts of fast memory close to processing units. When data is moved from slow memory into fast memory, it is therefore important to be able to perform enough computations to keep processors active. Otherwise, compute units will spend most of their time waiting for data-transfers to complete, and we do not take full advantage of the GPU.

Fusion optimisations are techniques for reducing number of memory transactions in program, by combining separate consecutive traversals over the same memory area into a single pass. Fusion allows programmers to write programs in logically separate functions, and later compose them into larger programs without the overhead of writing intermediate data-structures to memory.

However, as stated previously, data-parallel functional languages often perform these fusion optimisations automatically, with limited algorithmic control for the user. In practice we want the ability to reason about the performance aspects of function composition.

Instead of directly translating the built-in functions of TAIL into efficient GPU kernels, we thus started from the other end, constructing a functional language suitable for implementing the necessary GPU programs, while both providing ability to achieve fusion and the ability to reason about performance. The resulting language named FCL is by projects such as Obsidian [96], Sequoia [44] and CUB [84], and provide control of which level of the hierarchical GPU system that a particular function is executed.

FCL originated as a reimplementation of Obsidian as a standalone language, and has since been extended with additional looping constructs and other primitives, allowing further programs to be implemented. Fusion is achieved through the use of delayed array representations, which are only written to memory on the request of the user. The language is polymorphic in the level of the GPU hierarchy, allowing users to write level-agnostic programs. FCL also provides various standard optimizations and support for multi-kernel programs.

We identify several problems with the use of delayed array representations. Future work include I/O cost models for the hierarchical language, as well as type-directed translation such that algorithmic choices can depend on the level of hierarchy where a function is invoked.

1.5 Contributions

While we deviated from our original plan of providing a complete APL compiler for GPU programming, we present two separate projects working in the direction of the stated thesis. Below we list the individual contributions from the two projects.

The contributions on compiling APL to TAIL are as follows:

- We present a statically typed array intermediate language, TAIL, with support for multi-dimensional arrays and operations on such arrays. The type system supports several kinds of types for arrays, with gradual degree of refinement using subtyping. The most general array type keeps track of array ranks, using so-called shape types. One-dimensional arrays may be given a more refined type, which keeps track of vector lengths (also using shape types). To allow inference of such ranks and vector lengths, the type system also supports special refined variants of singleton scalar values and singleton vector values of statically known value. These are necessary for typing and inferring results of operations acting on shapes themselves.
- We demonstrate that TAIL is suitable as the target for an inference algorithm for an APL compiler. The type system of the intermediate language allows the compiler to treat complex operations, such as matrix-multiplication, and generalized versions thereof (inner products of higher-ranked arrays), as operations derived from the composition of other more basic operations.
- We further extend TAIL to support array updates. We present a type system and store-based semantics with support for mutable arrays, array indexing and array updates.
- We demonstrate that TAIL is useful as a language for further array-compilation, by demonstrating that the array language can be compiled effectively into a low-level array intermediate language, called Laila, which lends itself to a straightforward translation into sequential C code, OpenMP annotated C code, and GPU kernel code. The compilation is based on the concept of hybrid pull-arrays [78], which combines several variations of functional delayed arrays, and traditional materialized arrays for supporting array updates.
- We demonstrate that the typed array language can be used to compile to GPU code by compiling TAIL to the existing functional GPU language Accelerate [28, 23]. As part of a student project, it has also been demonstrated that TAIL, with good results, also can be compiled to the GPU language Futhark [55].

The contributions on hierarchical data-parallelism and FCL are as follows:

- We present FCL a statically typed hierarchical data-parallel language for GPU programming. FCL extends on the work on Obsidian, a data-parallel language for GPUs. A main consideration for both FCL and Obsidian is to allow users to reason about performance and experiment with various algorithms for solving a problem. We thus allow a programming style that allows fusion, in a way where users can reason about when fusion will happen. In contrast to Obsidian, FCL is implemented as a self-contained compiler, lifting us from some of the restrictions of prototype languages implemented as embedded DSLs.
- Data-parallel algorithms are often based on a divide-and-conquer approach, where subproblems are solved by individual parallel processors, and the results are assembled to form the final result. The Obsidian language only allows these results to be assembled through concatenation. To support a broader range of algorithms, we extend this method to allow permutations to be performed before the results are concatenated.
- Larger programs require many different GPU kernels. Obsidian users need to write and compile every kernel separately. FCL allows a programming style that supports multiple kernels, and where kernels are automatically compiled. Obsidian relied on its host language Haskell for composing programs and for executing kernels. FCL generates standalone C-programs and the necessary OpenCL kernel files.
- To support loops inside sequential code, Obsidian programmers had to rely on code generation in the meta-language and were thus limited to completely unrolled loops. This strategy is not always desirable, as it can lead to code explosion. We present a strategy that permits sequential loops to be generated and integrates with the existing model.
- We also identify weaknesses of the current approach used in FCL and Obsidian, such as limitations by the use of push/pull arrays. Push arrays following the current definition, does not allow us to lift FCL programs to operate on multiple GPU devices, as writing a socalled writer-function for use at device-level must be a bulk operation, whereas push arrays are based on writing elements individually. An alternative implementation strategy is thus needed, if we want to support multiple devices.

1.6 Structure of the dissertation

The thesis is structured as follows. In Chapter 2 we will introduce the necessary background on functional data-parallel programming languages design and implementation. Chapter 3 presents the TAIL language and compiler, and is an extended version of a previously published paper. Chapter 4 briefly covers GPU programming and hardware. Chapter 5 presents the FCL language. Chapter 6 concludes.

Chapter 2

Functional array language design

Arrays are fundamental data-structures in most programming languages for high-performance computing, as many computational problems susceptible for parallel acceleration can be written in terms of data-parallel operations on arrays. Programming languages in this domain differ in terms of the set of provided array operations, how array operations combine into whole programs, and in the ability for users to reason about programs and their performance.

We will make a distinction between the concept of an array language and that of an array library. Where both array libraries and array languages provide a great variety of built-in operations on arrays, they differ in the type of operations and how operations combine. The main focus of array libraries are to provide *independent software routines of high-performance*, often solving very specific tasks, such as solving systems of linear equations, least-squares fitting, or the underlying linear algebra operations such as matrix factorisation algorithms. For array languages, the main focus is not on performance on the individual software routines, but on *achieving high-performance when several operations are composed into a single program*.

Constructing an efficient array library comes down to selecting the routines necessary for the domain and implementing optimised algorithms for each routine individually, annotating their individual asymptotic performance behaviour. Different library implementations can be provided with each implementation optimised for a specific hardware platform. Constructing an efficient array language, on the other hand, requires a careful selection of built-in operations on arrays, that allows efficient parallel code to be generated on compositions, as well as providing the user with the ability to reason about performance of such compositions. There is a trade-off involved when it comes to selecting the built-in operations of such languages, as adding additional operations also increases the complexity of the language, which make reasoning harder [64].

This chapter will introduce the necessary concepts and background on design and implementation of functional data-parallel array languages. We will introduce the concepts by defining a small functional language with arrays that

$$
\begin{aligned}
e \ ::= \ & \texttt{true} \mid \texttt{false} \mid i \mid d \mid x \mid [\vec{e}] \mid op \\
& \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e_1\ e_2 \\
& \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{fix}\ e \\
& \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
& \mid (e_1,\ e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
i \ & \in \mathbb{Z} \\
d \ & \in \mathbb{R} \\
x \ & \in \mathsf{Var} \\
op \ & \in \mathsf{ScalarOps} \cup \mathsf{ArrayOps}
\end{aligned}
$$

Figure 2.1: Abstract syntax of AL$^{\text{core}}$

exhibits the problems that an array language implementor needs to address. Along the way, we will restrict our language with restrictions common in current typed functional array languages. By restricting the set of allowed programs, language implementors can provide better performance guarantees for the users and additional optimisation opportunities might arise.

No new ideas are introduced in this chapter; we will instead use the opportunity to discuss related work in the field.

2.1 A core functional language with arrays

We will base our discussion of array languages on a small core language, AL$^{\text{core}}$, which extends the typed lambda calculus with integers, booleans, conditionals, tuples, let-expressions (allowing recursion), and arrays.

Notation 2.1 (Sequences) *Whenever z is some object, we write $\vec{z}$ to range over sequences of such objects. When we want to be explicit about the size of a sequence $\vec{z} = z_0, \cdots, z_{(n-1)}$, we often write it on the form $\vec{z}^{(n)}$ and we write $z, \vec{z}$ to denote the sequence $z, z_0, \cdots, z_{(n-1)}$.*

Let Var be a denumerable infinite set of program variables. Let Ops be a finite set of built-in operations. We use i and n to range over integers. Figure 2.1 shows the abstract syntax of AL$^{\text{core}}$. Expressions consists of scalar integers, scalar doubles, booleans, variables, array expressions, built-in operations (op), abstraction forms, application forms, `let`-expressions, the fixed point combinator (`fix`), conditional expressions, tupling, and tuple projections.

We divide the set of built-in operations in two, a set of operations over scalar values, ScalarOps, and a set of operations on arrays, ArrayOps. We will use the same set of operations over scalars throughout the chapter, whereas we will extend and restrict the set of array operations as necessary, to illustrate issues and techniques of array language design.

We define the set of standard operations on scalar values, which consists of standard arithmetical operations overloaded to work on both integers and floating point values, as well as maximum and minimum functions, standard relational operations, and logical operations.

$$\begin{aligned}\mathsf{ScalarOps} = \{&\texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/},\ \texttt{div}, \texttt{mod},\\ &\texttt{max},\ \texttt{min},\ \texttt{==},\ \texttt{<},\ \texttt{>},\ \texttt{\&\&},\ \texttt{||}\}\end{aligned}$$

In array language design, the choice of built-in *array operations* is of utmost importance, as it is through these array operations that the structure of array computations are defined. Moreover, for the language implementor, the choice of operations determines the optimisations that users and compilers can or cannot perform. In this first section we introduce a very limited set of built-in array operations: array creation (`generate`), array indexing (`index`), and array length (`length`).

$$\mathsf{ArrayOps} = \{\texttt{generate},\ \texttt{index},\ \texttt{length}\}$$

The operation `generate` e_n e_f is an introduction form for arrays, constructing an array of length e_n, by applying e_f to each index of the array, thus returning $[e_f\ 0, e_f\ 1, \ldots, e_f\ (e_n - 1)]$. Arrays can also be introduced as literal arrays with the notation $[\vec{e}]$. Elimination forms for arrays consist of `index` e e_i for obtaining the value at index e_i in array e, and `length` e for determining the number of elements in the given array.

Together ScalarOps and ArrayOps constitute the built-in operations of AL^{core}. The choice of having only `generate`, `index`, and `length` as our only vehicles for expressing computations on arrays is made for presentation purposes. This limited set of array operations allows us to discuss the issues faced when implementing an array language. In later sections of the chapter, we will change the set of ArrayOps to allow additional programs to be written, and to make properties of algorithms, such as memory access patterns more evident.

Notation 2.2 (Recursive definitions) *To simplify the introduction of recursive functions, we introduce the common derived form:*

```
letrec x = e1 in e2
```

which should be considered equivalent to writing the following expression:

```
let x = fix (fn x => e1) in e2
```

Notation 2.3 (Top-level definitions) *When we want to present program fragments, we will use the following notation for top-level definitions:*

```
fun functionname arg0 arg1 ... argn = body
```

This notation should be considered equivalent to writing the following `let`*-expression, with the scope of the rest of the document:*

```
letrec functionname =
  fn arg0 => fn arg1 => ... fn argn => body
in ... rest of file ...
```

Example 2.1 *To exemplify, we can build a few reusable combinators. As a first example consider "*`iota`*", which creates a length n array with integers 0 through $n-1$:*

```
fun iota n = generate n (fn x => x)
```

We can also define the well-known combinator "`map`*", which applies the same function to all elements of an array:*

```
fun map f arr =
  generate (length arr)
           (fn i => f (index arr i))
```

With these definitions in place, we can build simple programs such as:

```
map (fn x => x * x) (iota 1000)
```

In this example we first create an array $[0, 1, \ldots, 999]$ and then apply a squaring-function to each element, creating a new array with elements $[0^2, 1^2, \ldots, 999^2]$.

2.1.1 Typing of AL$^{\text{core}}$

Types (τ) and type schemes (σ) for AL$^{\text{core}}$ are of the form:

$$\tau ::= \alpha \mid \texttt{int} \mid \texttt{double} \mid \texttt{bool} \mid [\tau] \mid (\tau_1, \tau_2) \mid \tau_1 \to \tau_2 \qquad \text{(types)}$$

$$\sigma ::= \forall \vec{\alpha}.\, \tau \qquad \text{(type schemes)}$$

We assign type schemes to built-in operations, *op*, by defining the relation $\mathsf{TySc}(op)$ in Figure 2.2 and Figure 2.3. We define $\mathsf{ftv}(\tau)$ as the set of free type variables in τ.

A type environment Γ, is a set of type assumptions of the form $x : \sigma$, mapping program variables to type-schemes:

$$\Gamma ::= x : \sigma, \Gamma \mid \epsilon$$

A substitution $S = [\tau_0/\alpha_0, \ldots, \tau_{n-1}/\alpha_{n-1}]$ is a mapping from type variables to types. Let S be such a substitution and let τ be a type, then $S(\tau)$ is the type obtained by simultaneously replacing every occurence of α_i in τ by τ_i, renaming type variables not bound in S if necessary. A type τ' is an *instance* of a type scheme $\sigma = \forall \vec{\alpha}.\tau$, written $\sigma \succ \tau'$, if there exists a substitution S such that $S(\tau) = \tau'$.

In Figure 2.4 we formalise the typing rules for AL$^{\text{core}}$ as the judgment form "$\Gamma \vdash e : \tau$" which can be read as "under the assumptions Γ, the expression e has type τ."

$op \in$ ScalarOps		TySc(op)
+	:	int → int → int
-	:	int → int → int
*	:	int → int → int
div	:	int → int → int
mod	:	int → int → int
+	:	double → double → double
-	:	double → double → double
*	:	double → double → double
/	:	double → double → double
max	:	int → int → int
min	:	int → int → int
==	:	int → int → bool
<	:	int → int → bool
>	:	int → int → bool
&&	:	bool → bool → bool
||	:	bool → bool → bool

Figure 2.2: Types of scalar operations in AL^{core}

$op \in$ ArrayOps		TySc(op)
index	:	$\forall\alpha.\, [\alpha] \to \texttt{int} \to \alpha$
length	:	$\forall\alpha.\, [\alpha] \to \texttt{int}$
generate	:	$\forall\alpha.\, \texttt{int} \to (\texttt{int} \to \alpha) \to [\alpha]$

Figure 2.3: Types of array operations in AL^{core}

Example 2.2 *Following these definitions we can infer the following types for the* `iota` *and* `map` *combinators introduced in Example 2.1.*

$$\begin{array}{rcl} \texttt{iota} & : & \texttt{int} \to [\texttt{int}] \\ \texttt{map} & : & \forall\alpha\beta.\, (\alpha \to \beta) \to [\alpha] \to [\beta] \end{array}$$

Notation 2.4 (Signatures) *When defining top-level functions in* AL^0*, we will annotate their types using the following notation:*

```
sig functionname : type
fun functionname a0 a1 a2 = body
```

All type variables occuring in type signatures are implicitly quantified at outermost position.

Example 2.3 *As an example using this notation, consider the following function:*

Static semantics $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash i : \texttt{int}} \ (2.1) \qquad \frac{}{\Gamma \vdash b : \texttt{bool}} \ (2.2) \qquad \frac{}{\Gamma \vdash d : \texttt{double}} \ (2.3)$$

$$\frac{\Gamma \vdash e_i : \tau \quad i \in [0; n)}{\Gamma \vdash [\overrightarrow{e^{(n)}}] : [\tau]} \ (2.4) \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \texttt{fix}\, e : \tau} \ (2.5)$$

$$\frac{\Gamma(x) = \sigma \quad \sigma \succ \tau}{\Gamma \vdash x : \tau} \ (2.6) \qquad \frac{\Gamma \vdash \mathsf{TySc}(op) \succ \tau}{\Gamma \vdash op : \tau} \ (2.7)$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \vec{\alpha} = \mathsf{ftv}(\tau) \quad (\Gamma, x : \forall \vec{\alpha}.\tau) \vdash e_2 : \tau \quad \Gamma \cap \{\vec{\alpha}\} = \emptyset}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \ (2.8)$$

$$\frac{(\Gamma, x : \tau') \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ x \texttt{=>}\ e : \tau' \to \tau} \ (2.9) \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \ (2.10)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_1 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)} \ (2.11) \qquad \frac{\Gamma \vdash e : (\tau_1, \tau_2)}{\Gamma \vdash \texttt{fst}\ e : \tau_1} \ (2.12) \qquad \frac{\Gamma \vdash e : (\tau_1, \tau_2)}{\Gamma \vdash \texttt{snd}\ e : \tau_2} \ (2.13)$$

Figure 2.4: Static semantics of AL$^{\text{core}}$

```
sig limit : [int] -> [int]
fun limit = map (fn x => max 100 (min 100 x))
```

At this point it is worth reflecting over some of the issues that will surface if we try to implement a compiler for this language. Without having introduced the evaluation semantics, we can already observe some of the larger problems that will need to be handled.

First we can observe that the presented typing rules does not put restrictions on which type of values that can be stored in arrays. A compiler would thus need to handle arrays of arrays, and arrays of functions. Furthermore, parallel expressions (`generate`) can launch other parallel computations, and the only mechanism for iteration in the language is through recursion. For many parallel architectures, it is difficult to generate efficient code if parallel operations are allowed to launch other parallel operations.

The rest of the chapter will discuss these issues and common ways to address them, often by introducing restrictions on the language. We will end this section with an overview of the rest of the chapter.

2.1.2 Values in AL$^{\text{core}}$

Figure 2.5 defines the value forms of AL$^{\text{core}}$. Values are either integers, tuples, lambda abstractions or arrays. Notice that arrays in this basic language can contain any type of value, including array values or functions.

In Figure 2.5 we define the typing of values in AL$^{\text{core}}$, with the judgment form "$\Gamma \vdash v : \tau$".

Values

$$
\begin{aligned}
v ::= {} & i \mid b \mid d \\
& \mid (v_1, v_2) \\
& \mid \texttt{fn}\ x \texttt{=>}\ e \\
& \mid [v_0, \ldots, v_{n-1}] \\
& \mid op
\end{aligned}
$$

Figure 2.5: Values of AL$^{\text{core}}$

Value typing $\boxed{\Gamma \vdash v : \tau}$

$$\frac{}{\Gamma \vdash i : \texttt{int}}\ (2.14) \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\ (2.15) \qquad \frac{}{\Gamma \vdash d : \texttt{double}}\ (2.16)$$

$$\frac{\Gamma \vdash \mathsf{TySc}(op) \succ \tau}{\Gamma \vdash op : \tau}\ (2.17) \qquad \frac{(\Gamma, x : \tau') \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ x \texttt{=>}\ e : \tau' \to \tau}\ (2.18)$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : (\tau_1, \tau_2)}\ (2.19) \qquad \frac{\Gamma \vdash v_i : \tau \quad i \in [0; n)}{\Gamma \vdash [v_0, \ldots, v_{n-1}] : [\tau]}\ (2.20)$$

Figure 2.6: Value typing of AL$^{\text{core}}$

Evaluation contexts

$$
\begin{aligned}
E ::= {} & [\cdot] \mid E\ e \mid v\ E \mid [\vec{v}, E, \vec{e}] \\
& \mid \texttt{let}\ x\ =\ E\ \texttt{in}\ e \mid \texttt{fix}\ E \\
& \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
& \mid (E, e) \mid (v, E) \mid \texttt{fst}\ E \mid \texttt{snd}\ E
\end{aligned}
$$

Figure 2.7: Evaluation contexts for AL$^{\text{core}}$

2.1.3 Dynamic semantics of AL$^{\text{core}}$

We will define the dynamic semantics of AL$^{\text{core}}$ as a small-step relation on expressions. We define the concept of *evaluation contexts* in Figure 2.7, which are ranged over by E, and denote expressions with one and only one unfilled hole indicated by the symbol $[\cdot]$. When E is an evaluation context and e is an expression, we write $E[e]$ to denote the expression resulting from filling the hole in E with e. We write $e[v/x]$ to denote the capture-avoiding substitution of f for x in expression e, renaming bound variables where necessary.

The small-step reduction rules for AL$^{\text{core}}$ are given in Figure 2.8, with the judgment form $e \Rightarrow e'/\mathbf{err}$. The small-step relation is explicit about out-of-

Dynamic semantics $\boxed{e \Rightarrow e'/\mathbf{err}}$

$$\frac{e \Rightarrow e' \quad E \neq [\cdot]}{E[e] \Rightarrow E[e']}\ (2.21) \qquad \frac{e \Rightarrow \mathbf{err} \quad E \neq [\cdot]}{E[e] \Rightarrow \mathbf{err}}\ (2.22)$$

$$\frac{}{\texttt{let}\ x\ =\ v\ \texttt{in}\ e \Rightarrow e[v/x]}\ (2.23) \qquad \frac{}{(\texttt{fn}\ x\ \texttt{=>}\ e)\ v \Rightarrow e[v/x]}\ (2.24)$$

$$\frac{}{\texttt{fix}\ (\texttt{fn}\ x\ \texttt{=>}\ e) \Rightarrow e[(\texttt{fix}\ (\texttt{fn}\ x\ \texttt{=>}\ e))/x]}\ (2.25)$$

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \Rightarrow e_1}\ (2.26) \qquad \frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \Rightarrow e_2}\ (2.27)$$

$$\frac{}{\texttt{fst}\ (v_1, v_2) \Rightarrow v_1}\ (2.28) \qquad \frac{}{\texttt{snd}\ (v_1, v_2) \Rightarrow v_2}\ (2.29)$$

$$\frac{}{\texttt{length}\ [v_0, \ldots, v_{n-1}] \Rightarrow n}\ (2.30)$$

$$\frac{i \in [0, n)}{\texttt{index}\ [v_0, \ldots, v_{n-1}]i \Rightarrow v_i}\ (2.31) \qquad \frac{i \notin [0, n)}{\texttt{index}\ [v_0, \ldots, v_{n-1}]i \Rightarrow \mathbf{err}}\ (2.32)$$

$$\frac{v_f\ i \Rightarrow^* v_i \quad n < 0}{\texttt{generate}\ n\ v_f \Rightarrow [v_0, v_1, \ldots, v_{n-1}]}\ (2.33)$$

$$\frac{n \leq 0}{\texttt{generate}\ n\ v_f \Rightarrow [\,]}\ (2.34)$$

$$\frac{i = i_0 + i_1}{i_0\ \texttt{+}\ i_1 \Rightarrow i}\ (2.35) \qquad \frac{v = i_0 \times i_1}{i_0\ \texttt{*}\ i_1 \Rightarrow i}\ (2.36) \qquad \ldots$$

$$\frac{i_0 = i_1}{i_0\ \texttt{==}\ i_1 \Rightarrow \texttt{true}}\ (2.37) \qquad \frac{i_0 \neq i_1}{i_0\ \texttt{==}\ i_1 \Rightarrow \texttt{false}}\ (2.38) \qquad \ldots$$

Figure 2.8: Dynamic semantics of AL$^{\text{core}}$

bounds errors on array indexing. A well-typed expression e is either a value, or it can be reduced into another expression, e', or the special token **err**. We use the notation $e \Rightarrow^* e'$ for the transitive and reflexive closure of $\Rightarrow$.

The parallel nature of the AL$^{\text{core}}$ language is introduced in the `generate` construct, where each array element can be computed in parallel. The available parallelism can be observed in the evaluation rule for `generate`, where the evaluation order is not given.

It would be straight forward to prove a soundness theorem, by proving type preservation and progress for our language.

2.1.4 Compiling AL$^{\text{core}}$: Not that simple a task

Although AL$^{\text{core}}$ is fairly limited, it allows for expressing a large number of array programs, through the use of `generate`, `index`, and recursion. For instance, as demonstrated in the following example, we can implement a reduction operation computing $b \oplus x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$, for an associative binary operator $\oplus$ and neutral element b.

Example 2.4 (Parallel reduction in AL$^{\text{core}}$)

```
sig halve : [a] -> ([a], [a])
fun halve arr =
  let n = length arr in
  let half = n div 2 in
   (generate (n - half)
             (fn i => index arr (2*i)),
    generate half
             (fn i => index arr (2*i + 1)))

sig zipWith : (a -> b -> c) -> [a] -> [b] -> [c]
fun zipWith f a1 a2 =
  generate (min (length a1) (length a2))
           (fn i => f (index a1 i) (index a2 i))

sig reduce : (a -> a -> a) -> a -> [a] -> a
fun reduce f b arr =
  if length arr == 0 then b
  else if length arr == 1
       then b + index arr 0
       else
         let (h1,h2) = halve ((length arr) div 2) arr
         in reduce f b (zipWith f h1 h2)
```

However the generality of AL$^{\text{core}}$ comes with a price. Being explicit about indices makes programming an errorprone process, and reasoning about programs is hard. Simple properties such as the equality:

$$\texttt{map}\ f\ (\texttt{map}\ g\ x) \equiv \texttt{map}\ (f \circ g)\ x$$

are less obvious, when expressed in terms of `generate` and `index`. The above equality is essential in array languages, as it allows intermediate arrays to be removed. In Section 2.2 we will discuss an alternative language not based on `generate` and `index`, but on primitives where such relations are easier to express. In Section 2.3 we will discuss various implementation approaches for fusion.

The use of recursion as means for iteration is also a problem for a compiler writer, as recursion is too powerful compared with facilities provided

by current data-parallel architectures. Consider the alternative formulation of `reduce`:

```
sig reduce_alt : (a -> a -> a) -> a -> [a] -> a
fun reduce_alt f b arr =
  if length arr == 0 then b
  else if length arr == 1
       then index arr 0
       else
         let (x,y) = halve ((length arr) div 2) arr
         in f (reduce_alt f b x) (reduce_alt f b y)
```

In this variant, the function is not tail-recursive, and in each iteration several smaller reductions are spawned. The possibilities for managing a recursion stack or launching new operations within a parallel program are very limited in architectures such as GPUs. It is thus common for current functional array languages to provide alternative looping constructs [59, 28, 48], or restricting recursion to tail-recursion.

As noted previously, AL^{core} allows nested parallelism and nested arrays, another non-trivial problem that would need to be handled if anyone wanted to implement the language as it is.

2.1.5 Chapter outline

The rest of the chapter is structured as follows. Section 2.2 will introduce additional constructs commonly found in array languages, such as reduction, prefix-sum and various index-space transformations. In Section 2.3 we will present the issues related to fusion, and discuss various proposed solutions. In Section 2.4 we will discuss how recursive procedures can be compiled, why it is unfeasible for modern parallel architectures and we will replace the general recursion of AL^{core} with more limited iteration constructs. In Section 2.5 we discuss the issue of nested irregular data-parallelism.

2.2 Array operations

We consider *array languages* to be languages with the characteristics of having arrays as a main and first-class data structure, with a set of built-in *generic* composable data-parallel operations, and where operator composition does not lead to performance degradation. The selection of built-in operators of such languages thus serve a crucial role, as the choice of operators decides the programs that may be written and the optimisations that may be performed.

In this section we will introduce a few widely used parallel operators found in functional array languages. The AL^{core} language introduced in the previous section provides a single operator for creating arrays, the `generate` operation, and one operator for accessing array elements, `index`. In this section we will introduce alternative operations to `generate`, that operate on elements in

bulk, allowing us to reason more clearly about programs. We will postpone the discussion of fusion to Section 2.3.

2.2.1 Element-wise operations

Our first step towards abandoning `generate` as a data-parallel operation, will be to introduce the most well-known operation on arrays from functional programming, the `map`, which, given an array $[\vec{x}]$ and a function f, produces the array $[\vec{y}]$, where $y_i = fx_i$ for all i. As we have already shown in Section 2.1, `map` and the similar `zipWith` could be directly implemented in AL$^{\text{core}}$ using `generate`.

Similarly, element-wise operations for functions of higher arity can also be provided. Languages such as NESL and Futhark provides such a generalised map:

$$\texttt{mapN} \quad : \quad \forall \vec{\alpha}\beta.\ (\alpha_0 \to \ldots \to \alpha_{n-1} \to \beta) \to [\alpha_0] \to \ldots \to [\alpha_{n-1}] \to [\beta]$$

2.2.2 Array initialisation

In AL$^{\text{core}}$ the `generate` operation was also the only array introduction form, except for literal arrays. Instead our modified language of array combinators will use the operator `iota` from APL as introduction form. Given an integer n, `iota` n produces the array `[0,1, ..., `$n-1$`]`.

2.2.3 Parallel reduction

In addition to acting element-wise on a few values at a time, we often also need operations for summarising or computing aggregates of large amounts of data, such as averaging, summations or finding the maximum value of an array. A reduction operation computes $b \oplus x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$, for an associative binary operator $\oplus$ with neutral element b.

In AL$^{\text{core}}$ we could introduce a reduction with the following type:

$$\texttt{reduce} \quad : \quad \forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$$

Associativity of the binary operator allows this operation to be implemented in parallel, through a tree structured divide-and-conquer algorithm. An illustration of the algorithm is presented in Figure 2.9. The input array is at the top of the diagram. Data flows through the edges from the top towards the bottom. A vertex represented by a black dot, corresponds to an application of the binary operation $\oplus$ on the two input edges. Most often, the requirement that the given operator is associative is made a responsibility of the programmer. However, some languages, such as OpenMP and NESL, only provides specialised versions of reduction with one of the seven operations: addition, product, maximum, minimum, bitwise or, bitwise and, and bitwise exclusive-or [17, 11, 71, 85]. These are the most common associative (and commutative) operators.

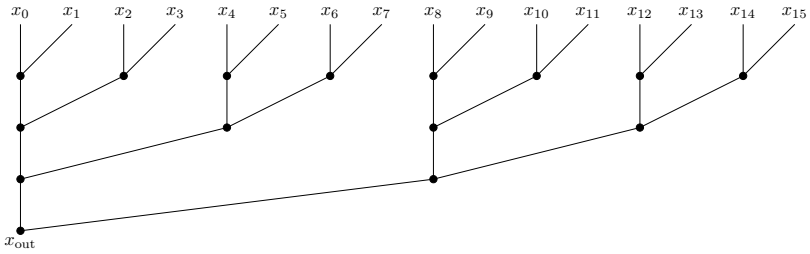

Figure 2.9: Parallel tree reduction, with associative operator.

2.2.4 Scan

Another parallel pattern used in many data-parallel algorithms allows not only access to the final accumulated value of a reduction, but also computes the accumulation of every prefix of the argument array. Given an array $[\vec{x}]$, an associative binary operator $\oplus$, and an initial value b, an *exclusive left scan* computes the array $[\vec{y}]$ where $y_0 = b$ and $y_{i+1} = b \oplus x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$, for $0 \leq i \leq n$, with the type:

$$\texttt{scanl} \quad : \quad \forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$$

The scan operation is *exclusive* as the last element of the array is not included.

Example 2.5 (Exclusive left scan)

```
scanl (+) 0 [5, 1, 4, 8, 0, 7, 1, 3]
  => [0, 5, 6, 10, 18, 18, 25, 26]
```

The first element of the result of an exclusive scan is always the neutral element, which in this case is the value 0. The alternative, an *inclusive* left scan, maintains the last array element, but does not requare a neutral element.

Example 2.6 (Inclusive scan)

```
scanl1 (+) [5, 1, 4, 8, 0, 7, 1, 3]
  => [5, 6, 10, 18, 18, 25, 26, 29]
```

Scan operations were first proposed as a programming language primitive in the context of APL [63], where the name also originates. The operation is also known as the *prefix-sum* of an array, as it computes the same value of a reduction, but for each prefix of the array. We will use the name *scan* for the remainder of the dissertation.

Several parallel scan algorithms exists [90, 91, 21, 70]. Two in-place inclusive left scans are visualised in Figure 2.10. Scans were popularised by Guy Blelloch in the early 1990s because of its wide applicability and its parallel implementation for associative and commutative binary operators [19].

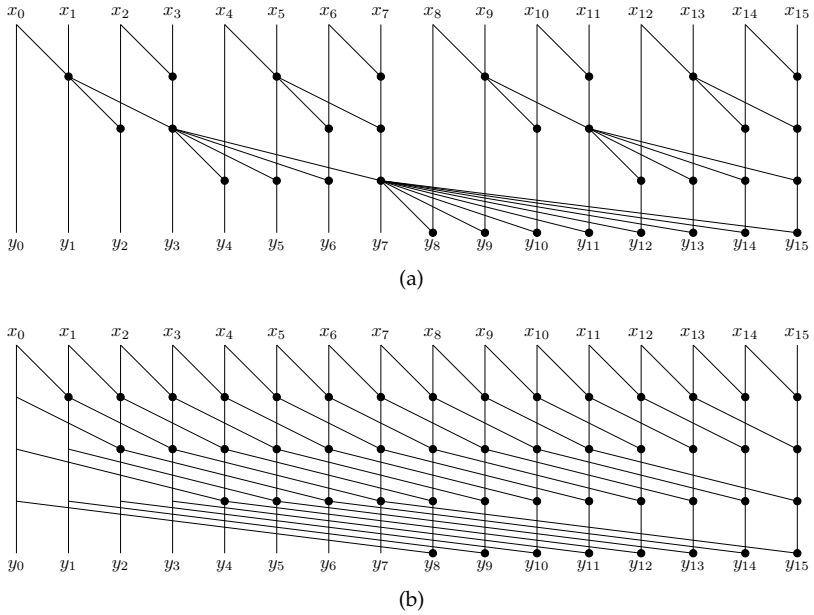

Figure 2.10: (a) The Sklansky construction for computing parallel prefix-sums. (b) The Kogge-Stone network for computing parallel prefix-sums.

2.2.5 Index-space transformations

To perform operations such as matrix transposition, array replication, and other operations rearranging the order of array elements, many languages provide dedicated operations for *index-space transformation* or *permutations*.

We can in general do permutations in two directions. Either we can specify how indexes in the old array maps to indexes in the new array (forward permutation), or we can specify how the new array is constructed by mapping the new index-space into the index-space of the old array (backward permutation).

Backward permutation

The `backpermute` function constructs a new array of a given size, and for each cell of the new array a function specifies where in the old array elements should be copied from. The function could have been directly implemented in AL^{core}:

```
sig backpermute : int -> (int -> int) -> [a] -> [a]
fun backpermute n f arr =
  generate n (fn i => index arr (f i))
```

with the assumption that $0 \leq$ `f(i)` $<$ `length(arr)`, for all $0 \leq i <$ `n`.

Example 2.7 (Replicate an array) *Repeat the same array for n repetitions.*

```
sig replicate : int -> [a] -> [a]
fun replicate p array =
  let n = length array
  in backpermute (p * n)
       (fn i => i mod n)
       array
```

Example 2.8 (Gather) *Backpermute, but where the transformation is given as an array mapping new indexes to indexes in the original array, instead of a function.*

```
sig gather : [int] -> [a] -> [a]
fun gather from input =
  let f = fn i => index from i
  in backpermute (length from) f input
```

Forward permutation

The complementary transformation, permuting an array by mapping the old index-space into the new index-space is called a forward permutation. The operation works as follows. First a new array is allocated and initialised with default values, then for each element of the input array the index-mapping is applied to determine where in the new array the value should be stored. To be deterministic, it is assumed that the index-mapping will not map several input values into the same location of the output. Alternatively, as we will present it here, an associative and commutative combination function is provided, that combines values if several values map to the same output location.

The type of `permute` that we will introduce to AL is:

$$\texttt{permute} \;:\; (\alpha \to \alpha \to \alpha) \to (\texttt{int} \to (\texttt{int}, \texttt{bool})) \to [\alpha] \to [\alpha] \to [\alpha]$$

Permute allows implementation of several useful operations, for example `filter` and `scatter` presented below.

Example 2.9 (Scatter) *Forward permutation, where the transformation is given as an array instead of a function.*

```
sig drop : int -> [a] -> [a]
fun drop n arr = backpermute n (fn i => i) arr

sig scatter : [int] -> [a] -> [a] -> [a]
fun scatter to defaults input =
  let f = fn i => (index to i, true)
      n = min (length to) (length input)
      input' = drop n input
  in permute const f defaults input'
```

Example 2.10 (Filtering) *Removing values based on some predicate can be done using a* `scan` *to compute the positions of the values to retain, and* `permute` *to create the filtered array based on these destination values.*

```
sig const : a -> (b -> a)
fun const i = fn x => i

sig filter : (a -> bool) -> [a] -> [a]
fun filter pred arr =
  let flags = map pred arr
      dest = scanl (+) 0 flags
      n = index (length dest - 1) dest
      default = map (const 0) (iota n)
  in permute
       const
       (fn i => (index i dest,
                 index i flags))
       default
       arr
```

2.2.6 Built-in vs. derived forms

We have now presented a few of the most common array operators found in functional data-parallel array languages. Most (if not all) of these operations could have been implemented using `generate`, `index`, and recursion. There are however benefits from having many of these operators built-in, as they provide information about the computation structure useful for automatic optimisers. As we shall see in the next section, this can help minimise expensive memory transactions.

Providing built-in versions of `scan`, `reduce` and so on, allows the language implementor to use highly-optimised version of these operations, written by developers with expertise in the target platform. Such templates of data-parallel operations are called *algorithmic skeletons*, and is a common implementation technique used by the Bohrium project [71], NESL for GPUs [11], Accelerate [28], Repa [68], and others.

2.2.7 Array language extended with operators: AL^{op}

To summarise, we have replaced the set of array operations, to avoid the very general `generate` operation. Our new set of core array operators consist of:

$$\begin{aligned}\mathsf{ArrayOps} = \{&\texttt{length}, \texttt{index}, \texttt{map}, \texttt{zipWith}, \texttt{iota},\\ &\texttt{reduce}, \texttt{scanl}, \texttt{backpermute}, \texttt{permute}\}\end{aligned}$$

with functions such as `filter`, `replicate`, `gather`, and `scatter` as derived forms. These should not be seen as a comprehensive set of built-in operations, in our presentation of TAIL in Chapter 3 we will introduce further constructs and cover a more complete set of array operations.

op		$\mathsf{TySc}(op)$
`length`	:	$\forall\alpha.\ [\alpha] \to \texttt{int}$
`index`	:	$\forall\alpha.\ [\alpha] \to \texttt{int} \to \alpha$
`map`	:	$\forall\alpha\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$
`zipWith`	:	$\forall\alpha\beta\gamma.(\alpha \to \beta \to \gamma) \to [\alpha] \to [\beta] \to [\gamma]$
`backpermute`	:	$\forall\alpha.\texttt{int} \to (\texttt{int} \to \texttt{int}) \to [\alpha] \to [\alpha]$
`reduce`	:	$\forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$
`scanl`	:	$\forall\alpha.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$
`permute`	:	$\forall\alpha.(\alpha \to \alpha \to \alpha) \to (\texttt{int} \to (\texttt{int}, \texttt{bool})) \to [\alpha] \to [\alpha] \to [\alpha]$

Figure 2.11: Built-in operations in AL^{op}

Operation		Type scheme
`filter`	:	$\forall\alpha.(\alpha \to \texttt{bool}) \to [\alpha] \to [\alpha]$
`gather`	:	$\forall\alpha.[\texttt{int}] \to [\alpha] \to [\alpha]$
`scatter`	:	$\forall\alpha.[\texttt{int}] \to [\alpha] \to [\alpha] \to [\alpha]$

Figure 2.12: Derived operations AL^{op}

2.3 Fusion

The overarching goal of array language research is to deliver languages where users do not need to rely on built-in problem-specific building blocks, but where high-performance can be achieved from the *composition* of completely generic operations, such as those described in the previous section, as well as making it possible to reason about the achieved performance of programs.

Input–output operations that move data between memory systems and processing units, are time consuming operations for modern microprocessor architectures. Deep memory hierarchies have been put in place to alleviate some of the potential bottlenecks. There is a small amount of fast memory close to processing units (registers), and increasingly larger but slower memories (caches, shared memory, device memory, main memory, disk drives, tapes). On such systems, it is important to use data as much as possible when they are brought all the way to the fast memory, before returning results to slower memory.

During computation, memory systems might still be active, thus another benefit of increasing the *computation per I/O operation* ratio, is that the processors can be kept active with useful work while waiting for data transfers, thus achieving *latency hiding*.

In array languages, we express algorithms in terms of combinators acting on entire arrays, not as individual data-elements. Often, these transformations can be combined using *fusion techniques*, such that several computations over

the same memory area are combined into one pass over the same memory.

The same concept exists in other programming models, such as imperative programming, where the optimisation *loop fusion* is used as an optimisation that combines several loops over the same range, into a single pass over the data [9]. Loop fusion originates from the ALPHA-language compiler [110].

In general, fusion can be beneficial at each level of a memory hierarchy. At each level there is the potential to keep the data for additional computations, before returning results back to slower memory.

Example 2.11 (Vertical fusion) *The problem of superfluous I/O operations may occur when we chain several operations calls after each other. As a simple example consider the following two invocations of* `map`*:*

```
fun f input =
  let xs = map (fn x => x / 100.0) input in
  let ys = map (fn x => x + 1.0) xs
  in ys
```

Following the dynamic semantics of AL^{core}, the array `xs` *will be written to memory, before being read during the computation of* `ys`*. In this case the construction of the intermediate array is unnecessary, and* `f` *can be computed directly:*

```
fun f_fused input =
  map (fn x => (x / 100.0) + 1.0) input
```

Vertical fusion is the process of combining array operations that are applied in sequence.

Example 2.12 (Horizontal fusion) *Another potential for optimisation is when two independent computions act on the same data in a similar pattern.*

```
fun g input =
  let xs = map (fn x => x / 100.0) input in
  let ys = map (fn x => x + 1.0) input
  in (xs, ys)
```

If we were to follow the dynamic semantics of AL^{op}, `xs` *and* `ys` *would be computed independently. This independence would result in two traversals over the input array. Horizontal fusion is the process of combining two such unrelated traversals into one sweep over the same data. In this case we would be able to optimise the function in the following way:*

```
fun g_fused input =
  map (fn x => (x / 100.0, x + 1.0)) input
```

We do not want to require that programmers need to perform such fusion optimisations by hand, as that would sacrifice the ability to construct reusable components.

The different operations allow various possibilities for fusion. Not all fuse with the same ease as `map`. To distinguish between how well operators fuse, it is common to distinguish between *producers* such as `map`, that produce a result and *consumers* such as `reduce`. In general, producers can fuse both on their inputs and their outputs, while consumers only fuse on their inputs.

In AL$^{\text{op}}$, the producers are `map`, `zipWith`, and `backpermute`, and consumers are `reduce`, `scanl`, and `permute`.

There are also reasons to limit fusion, even in cases where it is possible. Some intermediate values are needed many times, and thus recomputing them every time they are needed might be more expensive than the memory operation that stores them for later.

2.3.1 Implementation approaches

There are in general two approaches to fusion. Either rewriting will always be a local transformation, and the rewrite rules are applied in a systematic fashion, or the compiler takes a global view of the program and are able to perform larger restructurings of the program. The localised approach is often called *short-cut fusion*, whereas the global approach most commonly views the program as a graph structure, and fusion as graph rewriting.

2.3.2 Short-cut fusion

Deforestation

Deforestation is the general term for various techniques used in functional programming languages for removal of list and tree-based intermediate data structures. Philip Wadler presented the original Deforestation Algorithm [106], which uses seven rewrite rules, to remove such intermediate data structures. The algorithm requires the programmer to write functions in a so-called *treeless* form, and guarantees that all intermediate data structures will be removed, when combining functions written in this form. The algorithm is however limited, by the restrictions that the *treeless* form imposes; higher-order functions are disallowed, variables must be used linearly and nested data structures (e.g. list of lists) are not handled. Restricting programs to *treeless* form is necessary for guaranteeing termination of the algorithm.

Another and more successful approach is called `build/foldr` fusion [45]. In this approach the construction of a list is delayed by parameterising list producing functions over the list constructors ("`cons`" and "`nil`").

To construct the list in memory, a special function `build` is used to provide the list constructors:

$$\texttt{build}\ g = g\ \texttt{cons}\ \texttt{nil}$$

Alternative, if the list is to be directly consumed by a fold operation, the reduction operand and neutral element that would be given to `foldr` can be given instead of the list contructors. To hide these complexities for the user, the

library writer can always insert `build` and use the following rewrite rules to remove those unnecessary `build`-operations again [45, 67].

$$\forall k, z, g.\ \texttt{foldr}\ k\ z\ (\texttt{build}\ g) \equiv g\ k\ z$$

The `build`/`foldr` techniques also generalises to operations on tree structures [67], but are limited as left-folds (`foldl`) cannot be expressed, and for a function such as a *zip* it is not possible to deforest both input lists.

Stream fusion solves these problems, by introducing a special stream data type, represented as a state and a step-function. Calling the step function yields the values of the stream. Operations `stream` and `unstream` are provided for converting between streams and lists [38]. Library functions are written in terms of streams, but details are hidden from the user by applying `stream` and `unstream`, and a single rewrite rule removes the unnecessary intermediate streams:

$$\forall s.\ \texttt{stream}\ (\texttt{unstream}\ s) \equiv s$$

Both `build`/`foldr` and stream fusion produces values one at a time. To take advantage of SIMD operations and optimised operations operating on data in bulk, such as `memcpy`, it is necessary to allow alternate stream representations. Generalised Stream Fusion [76] allows this by representing each stream as a *bundle of streams* of varying representation. The final consumer of the stream decides which representation to use and the alternative representations can be optimised away.

Delayed array representations

Deforestation, especially stream fusion, has been successful in generating high performing sequential code, even outperforming hand tuned C in some instances [76], and is a popular choice for current string processing and vector libraries in the functional programming community, with implementations in use in Haskell, Scala and OCaml. However, these approaches work on sequential data, and only allow for limited parallelism and direct indexing requires the underlying data structure to be materialised.

Another branch of short-cut fusion approaches is based on various *delayed* representations of arrays. This approach traces back to Abrams concept of "dragalong" in APL compilation [1], where arrays "drags along" the computation of individual array elements until they are needed, this was further explored by Guibas and Wyatt, also in the context of APL [50].

In functional programming languages, the use of delayed array representations has only recently been applied. First in the representation of images in the domain-specific language Pan [41] and since in various array languages [68, 97, 78, 8].

Pull arrays The approach taken by Pan and most other libraries is to represent array values as a length and a function from indices to array elements, sometimes known as *pull arrays*. In the case of AL, this representations corresponds

Value typing in AL^{pull} $\boxed{\Gamma \vdash v : \tau}$

$$\frac{\Gamma \vdash v_n : \texttt{int} \quad \Gamma \vdash v_f : \texttt{int} \to \tau}{\Gamma \vdash \texttt{generate}\ v_n\ v_f : [\tau]} \quad (2.39)$$

Dynamic semantics in AL^{pull} $\boxed{e \Rightarrow e'/\mathbf{err}}$

$$\frac{v_{idx} \in [0, v_n)}{\texttt{index}\ (\texttt{generate}\ v_n\ v_f)\ v_{idx} \Rightarrow v_f\ v_{idx}} \quad (2.40)$$

$$\frac{v_{idx} \notin [0, v_n)}{\texttt{index}\ (\texttt{generate}\ v_n\ v_f)\ v_{idx} \Rightarrow \mathbf{err}} \quad (2.41)$$

$$\frac{}{\texttt{length}\ (\texttt{generate}\ v_n\ v_f) \Rightarrow v_n} \quad (2.42)$$

Figure 2.13: Value typing and Dynamic Semantics of AL$^{\text{pull}}$

exactly to the parameters of `generate`:

$$\texttt{generate} \quad : \quad \forall\alpha.\, \texttt{int} \to (\texttt{int} \to \alpha) \to [\alpha]$$

Implementing pull arrays in AL corresponds to have `generate` as an addtional value form for arrays:

$$\begin{aligned} v ::= &\ \ldots \\ &\mid \texttt{generate}\ v_n\ v_f \end{aligned}$$

In this way the computation of the individual array elements can be delayed until they are needed. Figure 2.13 shows the required changes to value typing, and dynamic semantics. With these rules, indexing into an array becomes as cheap as a bounds check and a function application.

Rule (2.40) is what enables fusion.

The simplicity of the implementation comes with the price of potential duplication of work, as the values of an array created using `generate` are recomputed every time they are accessed. Duplicating work will be undesirable for all but very cheap computations. It is thus necessary to limit fusion in one way or the other, which is often done using special annotations on arrays, denoting when a delayed array should be computed and written to memory. A common approach is to use a function with identity-type for these annotations:

$$\texttt{force} \quad : \quad [\alpha] \to [\alpha]$$

The semantics of `force` corresponds to converting the `generate`-based array into a materialised array:

$$\frac{}{\texttt{force}\ (\texttt{generate}\ v_n\ v_f) \Rightarrow [v_f\ 0, v_f\ 1 \ldots, v_f\ (n-1)]} \quad (2.43)$$

Pull-arrays, however, only automate vertical fusion. Horizontal fusion is still possible through the use of tuples. However, it must be performed by the user as a manual transformation, and it is not possible in all cases. Encoding computations as arrays of tuples creates other problems, as some architectures or programming interfaces does not provide an efficient implementation of such arrays of tuples. This can however be resolved through a conversion to tuples of arrays.

Finally, pull arrays do not support efficient concatenation. It is possible to define concatenation in terms of pull arrays, it will however be necessary to introduce a conditional that is executed for every element of the new array:

```
fun concat a1 a2 =
  generate (length a1 + length a2)
           (fn i =>
              if i < length a1
              then index a1 i
              else index a2 (i - length a1))
```

On single instruction multiple data hardware (SIMD) such as GPUs, a conditional such as this can be detrimental to performance, as groups of processing units execute the same instructions in lock-step, and will thus have to execute both branches, for every element of the concatenated array. This can lead to serious performance degradation if the concatenated arrays involve large computations, which is fused inside the body conditional.

Push arrays To remedy the problem of inefficient concatenation, the concept of *push array* was introduced by Svensson et al. [36]. A push array can be seen as a dual to the concept of pull arrays, and push arrays does not replace pull array, but are implemented using an additional data structure used to express data-parallel programs.

For a pull array, two important aspects are left undecided: the order the elements are generated and the destination in memory where elements are to be written. This allows a consumer of a pull array to traverse it in any order and decide where in memory the array is to be written. A push array, on the other hand, locks the iteration pattern, and only leaves the consumer to decide where in memory the elements should be written.

A push array is most often represented by a function that can construct an array, when given a so-called *writer*-function. A *writer*-function is a function that accepts an element and an index and produces an assignment statement writing the element to its corresponding index in memory (here using Haskell notation):

```
type Writer a = a -> Idx -> Program ()
```

Here `Program ()` is a computation in a code-generation monad. Push arrays are represented by a length and a function accepting such a writer-function:

```
type Push a = (Idx, Writer a -> Program ())
```

Materializing a push array is done by applying the function to a writer function, and the writer will then be invoked for each array element.

This means that values will always be generated in the order given by that writer function (or *iteration scheme*), when we materialize the array. This also means that we can not access any single element of a push array, before it has been fully materialized. This is however not the same as saying that push arrays will necessarily write all the array elements in the same order.

Push arrays are in this way more restricted. However, the fact that the iteration pattern is locked, makes efficient concatenation possible. Two push array computations can be combined in sequence or in parallel, as appending their results is only a matter of off-setting the write operations of the second array, such that they are written where the first array ends. This offsetting is accomplished by parameterising the push array computation with a writer function, storing elements to memory.

Computations on push arrays can still be fused, however, the fact that the iteration pattern of a push array is locked makes direct indexing impossible. Thus a program will often switch between using pull and push array at different points, depending on the context. Conversion from pull array to push array is basically free, converting in the other direction requires the push array to be materialised.

Alternative local approaches

There are alternative approaches with a local view of fusion. In Nessie [11], the NESL GPU compiler, only map-map fusion is implemented as a rewrite rule. Fusing maps and reductions is not possible. In Single-assignment C [48], all array operations are defined in terms of a single loop-expression called a "with-loop", which allows both array creation, mutation and array reduction. The Single-assignment C compiler employs various rewrite rules to combine two or more with-loops. Bohrium [71] is based on a simple bytecode of array operators, without any control flow. The control flow is managed by a host-interpreter that issues the bytecode operations in sequence. This sequence of bytecode operations is then JIT compiled into parallel operations. The JIT compiler will do its best to fuse as many possible operations in sequence into a single parallel operation.

2.3.3 Graph-based approaches to fusion

An alternative to the various "shortcut fusion" approaches is to take a more global approach to fusion, analysing whole programs as graph-structures and perform larger restructurings, which can have huge benefits as it makes additional fusion possible. However, large automatic restructuring also limits transparency, and can thus limit the ability for users to optimise.

In a graph representation of an array, nodes represent operations and edges represent data dependencies. Fusion decisions in such representations

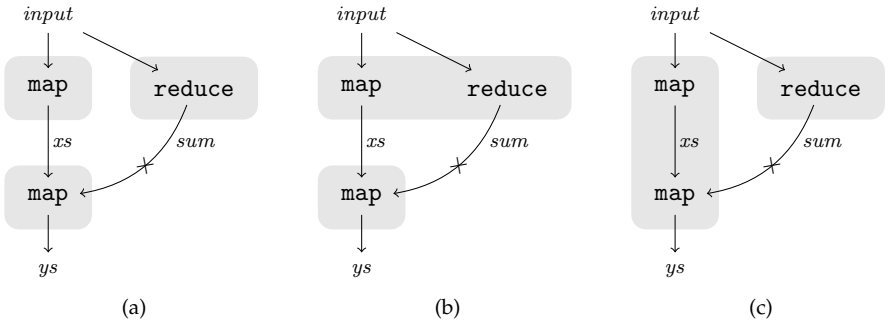

Figure 2.14: Three fusion possibilities for the example program. (a) No fusion. (b) Horizontal fusion of `map` and `reduce`. (c) Vertical fusion of the two `map`-operations.

correspond to partitioning nodes into fusable clusters, or rewriting graphs by merging nodes.

Consider the following example from Robinson et al. [88]:

```
fun normalizeInc input =
  let xs = map (fn a => a + 1.0) input in
  let sum  = reduce (+) 0.0 input in
  let ys = map (fn a => a / sum) xs
  in ys
```

This program can be described as the dataflow graph in Figure 2.14. The crossed-out edge corresponds to a noncontractable edge, as the reduction operation hinders further fusion. Grey areas corresponds to fusable clusters. In this case there are three possible graph partitionings: no fusion (Figure 2.14a), horizontal fusion of the two iterations across the input array (Figure 2.14b) or vertical fusion between the two map operations (Figure 2.14c).

The larger the program, the more possible clusterings to consider. A good clustering minimises the overall cost of the graph, where the cost is determined by the number of memory accesses. In the `normalizeInc` example the solution in Figure 2.14c is optimal for large inputs, as this choice will remove both a memory write and memory read per array element, whereas horizontal fusion in Figure 2.14b only reduces the number of memory reads by one per element.

Integer programming

Determining good clusterings is in principle NP-hard. However, a solution using integer programming has been developed that is efficient in practice [79].

Edges are annotated with weights corresponding to the memory cost of *not* fusing this edge. The optimisation goal is to find a graph partitioning that minimises the weight of inter-cluster edges. Let w_{ij} be the weight associated

with the edge between node i and j, and let x_{ij} be a 0,1-variable determining whether nodes i and j are placed in the same cluster (0) or not (1).

The integer programming objective function is thus:

$$\min \sum_{i,j} w_{ij} x_{ij}$$

To ensure that the clustering is valid, constraints on x_{ij} variables are necessary. We will not describe these contraints in detail, but just mention that they ensure that the noncontractable edges are never fused, that no cycles are introduced after contraction, and finally that all clusters are closed (i.e., if $x_{ij} = 0$, $x_{jk} = 0$ then we also have $x_{ik} = 0$).

In addition to these constraints, further constraints are necessary when size-changing operations such as filter operations are allowed [88].

Graph rewriting

An alternative to working out optimal clusterings on graphs, is to rewrite the graph based on a collection of rewrite rules, applied systematically. However, for allowing horizontal fusion, such as fusing `map` and `reduce` in Figure 2.14b, we cannot just rewrite the combined `map` and `reduce` without extending the language. In Futhark [59, 55] fusion is performed by rewriting the standard array combinators into various operators that are not exposed to the programmer. The primary (unexposed) operator being their so-called `redomap` that allows multiple maps and reductions to be fused into the same operation. This approach allows gradual refinement of the graph, where nodes are merged into `redomap`'s, instead of making all fusion decisions at once. In Futhark the above `normalizeInc` program could potentially be rewritten:

```
fun normalizeInc input =
  let (sum, xs) =
    redomap ((+),
             fn (acc, x) => let v = a + 1.0
                            in (acc+v, v),
             0.0) input in
  let ys = map (fn a => a / sum) xs
  in ys
```

The combined reduction and map operator thus both accumulates the result of the reduction, while also producing the array output from the map.

However, as we saw previously the strategy of merging the two `map` operations is more optimal, as it removes an additional memory transaction, and is probably also the choice Futhark will make in this case.

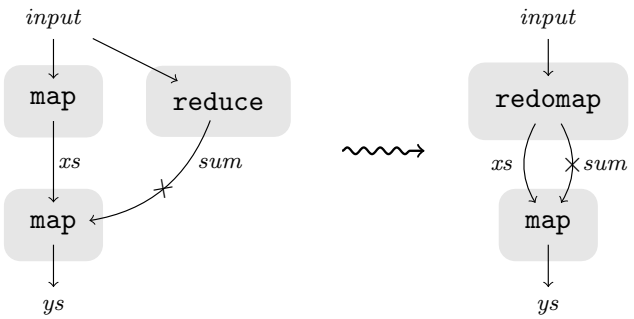

Figure 2.15: Fusion with Futharks `redomap` construct.

Auto-tuning

Another graph-rewriting approach is to apply a search algorithm to determine which rewrite rules to apply, instead of applying the rules systematically. Various attempts have been made in this direction [94, 105, 74]. These approaches often rely on a large set of rewrite rules operating on their set of array combinators. During search, the approach evaluates potential rewrite candidates, by generating code and measuring performance. The best candidate is selected as starting point for the next round. These approaches also demonstrate the need for hardware awareness during compilation, as widely different optimisations are necessary for the various hardware architectures.

2.3.4 Local versus global view of fusion

Fusion is necessary to obtain performance in array languages if composability is desired. However, the scientific comunity has yet to determine an approach that is satisfactory on all parameters.

Shortcut fusion is implemented by only a few general rules. The limited number of rules is both positive and negative. Fewer rules makes it easier for users to predict what will happen, but it also limits the number of fusion opportunities. Horizontal fusion is for instance not possible in most work on shortcut fusion. It is also sometimes necessary to write multiple versions of the same program, to accomodate various usage scenarios.

Approaches based on graph rewriting, on the other hand, makes it easy for the user, as they provide very limited control over where fusion happen. These approaches, also allows the same program to be used in various contexts, but optimised differently.

However, graph based approaches often suffer from the lack of predictability; alternative versions of the same program might perform completely different. Without reasoning tools that can explain why one approach performs better than the other, a slight change may make a different clustering seem better to the compiler. Trying to optimise an algorithm might lead to perfor-

mance penalties, as some compiler optimisation is suddenly not taken into effect. Finally, it is also unclear how well some of these approaches scale to very large programs.

There is thus a need for further research on fusion systems, to obtain a balance between predictability and ease of use.

2.4 Iteration and recursion

Recursion is the most fundamental iteration construct in functional languages, and as such, we have allowed it in AL^{core} and AL^{op}. Allowing recursive definitions has consequences when compiling for parallel architectures. Recursion is not the problem per-se; the problem is that the hardware architectures such as GPUs only supports limited recursion, with either slow or limited stack space useable for a recursion stack. The ability to launch new threads within parallel code is also limited on these devices.

A tail-recursive function is on the other hand manageable, as no stack is required, and can, through tail call elimination, be converted to loop structures. The most common approach in data-parallel languages for GPUs, however, is to replace recursion entirely by operations that simulate tail-recursion in one way or another.

In embedded domain-specific languages such as Accelerate, Nikola and Obsidian, it is common to see recursion performed on the meta-level. Outside parallel code this corresponds to recursion in the host language, and allows users to coordinate tasks. However, such recursion outside parallel code limits the compilers ability to optimise across individual kernel invocations, as these only exists on the host-level. Inside parallel code the user is restricted to construct large unrolled loops through the meta-language. This approach can lead to code-explosion inside parallel kernels and may introduce problems on architectures with limited instruction memory. Recent versions of Accelerate also provides two looping constructs, that corresponds to tail recursion on a single array. One loop construct for iteration outside parallel code (coordination of tasks) and one for iteration inside parallel code.

Similarly, Bohrium allows data-parallelism in interpreted environments, such as Python, by issuing instructions to a virtual machine without support for any control-flow [71]. The program flow is directed by the host language (Python) and the virtual machine only executes high-level data-parallel operations in bulk. These operations are combined (fused) and just-in-time compiled by the virtual machine. Thus no iteration exists in Bohrium, only in its host language.

In standalone languages other approaches to iteration and recursion have been considered. Single-assignment C (SaC) provides a generalisation of the `generate` construct in AL^{core}, called a `with`-loop, that allows multiple generator functions to be specified for individual subranges of a multidimensional array [48]. SaC allows nested loops. The QUBE language provides a similar construct [100].

NESL allows recursion through a recursion stack. However, the GPU-implementation of NESL [11] is limited to use of recursion outside parallel code, as control flow is managed on the CPU.

Another possibility that we will return to in later chapters, is the ability to recurse over the *hierarchy* of the data-parallel machine. Sequoia [44] allows this style of recursion. This allows programmer control of how problems are partitioned across the tree structured hierarchies of modern high-performance computing systems.

Instead of considering adding support for programming in tail-recursive style, we have decided to include a loop construct simulating such a tail-recursive calls;

$$\texttt{while} \;:\; (\alpha \to \texttt{bool}) \to (\alpha \to \alpha) \to \alpha \to \alpha$$

The operation "`while` c f x" repeatedly applies the step function f to the initial value x, until the stop-condition $c(x)$ returns `false`.

The `while` operation permits us to write operations such as reduce in the following style:

```
fun reduce0 f arr =
   while (fn a => length a > 1)
         (fn a =>
            let (x,y) = split ((length a) div 2) a
            in zipWith f x y)
         arr

fun reduce f b arr =
  if length arr == 0
  then b
  else index (reduce0 f arr) 0
```

2.5 Nested data-parallelism

AL$^{\text{core}}$ puts no restrictions on the type of operations that can occur inside parallel computations, and thus parallel computations can launch other parallel operations. To take advantage of both the inner and outer parallelism is however not easy when executed on SIMD-architectures such as GPUs. A simple solution would be to decide that we only allow parallelism on either the inner-most operation or the outer-most operation. Following such an approach, will however not allow us to extract all the possible parallelism in a program. To exemplify, consider the following problem of sparse matrix-vector multiplication.

Example 2.13 (Sparse Matrix-Vector Multiplication) *We can represent a sparse vector as an array of index-value pairs of type* `[int * double]`*, and represent a sparse matrix as an array of such sparse vectors* `[[int * double]]`*.*
The scalar product of a sparse vector and a dense vector, can be performed in parallel by first performing a map and then a reduction:

```
sig dotprod : [int * double] -> [double] -> double
fun dotprod vsp v =
  let ps = map (fn (c, x) => x * index v c) vsp
  in reduce (+) 0.0 ps
```

We use this function to compute the sparse matrix-vector product, by performing a dot product for each row of the sparse matrix:

```
sig smvm : [[int * double]] -> [double] -> [double]
fun smvm mat vec =
  map (fn row => dotprod row vec) mat
```

This program is an example of nested parallelism, as each invocation of `dotprod` *is in itself a parallel operation consisting of a map and a reduction. If we only took advantage of inner-most parallelism of the reduction operations, we would perform poorly on a matrix with few columns and many rows. An contrary, if we only took advantage of the outer-most parallelism, each of the dot-products would be executed sequentially, and we would perform poorly on wide arrays with few rows.*

A second problem found in the above sparse matrix-vector multiplication example is that of *irregular* arrays. A *regular* array is multidimensional of rectangular or cubic shape. An *irregular* array is an array of arrays, where the inner arrays are of variable size.

Blelloch presented an alternative strategy to parallelise outermost or inner most operations. His *flattening* transformation eliminates nested parallelism from programs. This is done through three insights: 1) using a representation of irregular nested arrays, that allows direct computations on inner arrays (segmented arrays), 2) implementation of high-performance algorithms operating on segmented arrays, 3) a converstion scheme from nested data-parallel programs to flat programs with non-nested operations acting on segmented arrays.

2.5.1 Representing irregular nested arrays

The representation used by Blelloch is based on seperating the data values from a description of where the subarray segments begins and ends. A nested array structure is represented by two parts; a flat array of all data elements and one or more *segment descriptors* denoting index and length information necessary to reconstruct the nested array structure from the flat data array.

Example 2.14 *Continuing our sparse-matrix vector example, consider the following sparse matrix representation:*

$$\begin{aligned} A = [&[(2, 0.17)], \\ &[(0, 2.0), (1, 0.4), (2, 1.0)], \\ &[(0, 0.8), (2, 9.0)]] \end{aligned}$$

The flat array of data elements are unzipped and placed in two separate data arrays:

$$\begin{aligned} A_{indices} &= [\ 3, \ 0, \ 1, \ 2, \ 1, \ 3] \\ A_{values} &= [\ 0.17, \ 2.0, \ 0.4, \ 1.0, \ 0.8, \ 9.0] \end{aligned}$$

The boundaries of the original subarrays are stored in a segment descriptor, recording the length of each segment:

$$A_{segdesc} = [1, 3, 2]$$

Various alternative representation of segment descriptors have been suggested.

2.5.2 Operations on segmented arrays

The second step of flattening nested parallelism is to lift array operations such as reductions and scans to operate on these segmented arrays. A segmented operation will operate, not on the outer array, but on each of the inner arrays. We will not go through the implementation of such operations, but just mention that they exist and illustrate the segmented reduction through an example:

Example 2.15 *Consider the nested array:* $[[0.17], [2.0, 0.4, 1.0], [0.8, 9.0]]$.
This array can be converted into the following segmented array:

$$\begin{aligned} values &= [0.17, 2.0, 0.4, 1.0, 0.8, 9.0] \\ segdesc &= [1, 3, 2] \end{aligned}$$

Applying a segmented sum operation on the segmented array computes the sum of each subarray.

$$\texttt{segReduce (+) 0.0 (}values\texttt{, segdesc)} = [0.17, 3.4, 9.8]$$

2.5.3 Flattening

Flattening can be performed either as a compiler transformation, or by introducing the necessary segmented operations, allowing users to manually flatten their programs. To illustrate, the following example shows a flattened version of the Sparse Matrix-Vector multiplication program.

Example 2.16 (Flattened Sparse Matrix-Vector multiplication)

```
sig smvm_flat : ([int], ([int], [double])) -> [double]
fun smvm_flat m vec =
  let (segdesc, (ixs, vals)) = m in
  let ps = zipWith (fn (i,x) => x * (index vec i))
                   ixs
                   vals
  in segReduce (+) 0.0 ps segdesc
```

Flattening makes it possible to take advantage of additional parallelism in nested data-parallel programs, however, with the cost of additional memory overhead associated with maintaining segment descriptors. The memory overhead is often stated as a main reason that languages based on flattening has not seen more widespread use.

Since the development of flattening transformation as part of the work on NESL, various attempts have attempted to optimise the memory usage of flattened programs through various means, such as reusing memory, alternative descriptor representations, limiting the amount of flattening necessary [69, 10], and stream processing [75].

We will not describe the full flattening transformation, and instead refer the reader to previous presentations [18]. Flattening is part of the story on data-parallel functional languages, as it is important for high-performance code-generation on irregular nested data-parallel programs, but it is outside the scope of this dissertation.

Chapter 3

TAIL: A Typed Array Intermediate Language for Compiling APL

Extended version of "Compiling a Subset of APL Into a Typed Intermediate Language" presented at ARRAY'14

In the previous chapter we saw that the choice of built-in operators determines how users and compilers can reason about programs, and which optimisations can be performed. A highly successful array language in terms of industry adoption and use for mathematical reasoning about programs is the language APL. The language has been widely successful in the financial industry, where large code bases are still operational and being actively developed. Many APL dialects have been implemented over the last 50 years [73, 107, 24, 13], and some are still used extensively within certain domains. In this chapter we want to use APL to inform us about the choice of array operations, that have been proven useful in practice.

APL is a dynamically typed array language with support for both multi-dimensional arrays and nested arrays, with a functional core consisting of a large number of array operations of both first-order and second-order, such as array transposition, generalised multi-dimensional inner-products and outer-product, prefix sums, and reductions.

Until recently, the programming language semantics community have paid only little attention to the APL language. Kenneth E. Iverson never developed a semantics for APL in terms of a formal model. Apart from recent work by Slepak et al. [92], there have been few attempts at developing formal models or type systems for APL. On the other hand, recent development in data-parallel language implementations [36, 68, 28, 58] have resulted in promising and scalable techniques for high-level programming of highly-parallel hardware, such as GPGPUs.

This chapter presents TAIL [43], a typed intermediate language suitable for implementing a large subset of APL. The chapter presents evidence that

TAIL may serve as a practical and well-defined format for generation of high-performing sequential code, and indicates the potential for translating TAIL to data-parallel GPU code. The intermediate language is conceptually close to the language Repa [68]. It treats all numeric data as multi-dimensional arrays and the type system makes the ranks of arrays explicit. Primitive operators are polymorphic in rank and in the type of the underlying data operated on. The language is sufficiently expressive that some primitive operators, such as APL's inner product operator, which works on arrays of arbitrary rank, can be compiled using more primitive operations. Following other APL compilation approaches, the compiler is based on lexical scoping and has no support for dynamic compilation (APL execute) [13, 14].

A prime goal of the project is to study the requirements for implementing a high-level array language such as APL, that have been proved useful in practice. We have therefore followed a pragmatic rather than purist approach. We are thus not attempting to encode all invariants regarding array shapes and sizes in our type system, such as systems based on dependent types [100, 98]. The purpose is rather to find a balance, which allows for complete type inference without user involvement.

APL and its derivatives, such as J [24] and K [107], are still being used extensively in certain domains, such as in the financial industry, where large code bases are still operational and being actively developed. TAIL is however only a subset of APL, and such a translation thus only provides some evidence for general applicability in computational finance. We have implemented medium sized programs in this subset of APL, and believe many problems can solved using this subset. The most notable missing feature from APL is nested irregular arrays.

The compilation framework relies on heavy inlining. It is thus unclear how well TAIL will perform on larger examples than the presented benchmarks.

Example 3.1 *As a simple example, consider the following signal processing program, derived from the APEX benchmark suite [13]:*

```
diff ← {1↓⍵-¯1⌽⍵}
signal ← {¯50⌈50⌊50×(diff 0,⍵)÷0.01+⍵}
+/ signal 9 8 6 8 7 4 4 3 2 2 1 2 4 5 6
```

This program declares two functions `diff` *and* `signal`*, both taking a single parameter,* `⍵`*. In the* `diff` *function, the expression* `¯1⌽⍵` *specifies that the parameter vector is rotated one entry to the right. This vector is then subtracted from the argument vector (point-wise), and the result of this subtraction is returned as the result with the first element dropped. The last line of the program calls the* `signal` *function on an input vector and sums (i.e., sum-reduces) the result of the call. In the compiled version of the program, which is presented in Figure 3.1, bulk operations are replaced with calls to the* `each` *function, which is equivalent to* `map` *in other functional languages. The name "each" is the standard pronounciation for the* `map`*-like operator ¨ in APL.*

Moreover, the compiler has inserted explicit integer-to-double coercions and identified the neutral element 0.0 for a reduction with addition. Array types in the target

```
let v0:<int>15 = [9,8,6,8,7,4,4,3,2,2,1,2,4,5,6] in
let v3:<int>16 = consV(0,v0) in
reduce(addd,0.00,
 each(fn v11:[double]0 => maxd(i2d(~50),v11),
  each(fn v10:[double]0 => mind(i2d(50),v10),
   each(fn v9:[double]0 => muld(i2d(50),v9),
    sum(divd,
     each(i2d,
      drop(1,zipWith(subi,v3,rotateV(~1,v3)))),
       each(fn v2:[double]0 => addd(0.01,v2),
        each(i2d,v0)))))))
```

Figure 3.1: The result of compiling the example APL program into an explicitly typed intermediate representation. Notice the presence of shape types with explicit length attributes.

language are annotated with explicit ranks; the type `[double]0`*, for instance, is the type of double-precision scalar values, which are treated in the type system as arrays of rank zero. Also notice the special one-dimensional vector types (e.g.,* `<int>16`*), which range over vectors of a specific length.*

3.0.1 Chapter outline

The rest of the chapter is structured as follows.

In Section 3.1, we present a statically typed intermediate array language with support for multi-dimensional arrays and operations on such arrays. The type system supports several kinds of types for arrays, with gradual degree of refinement. The most general array type keeps track of array ranks, using so-called shape types. One-dimensional arrays (i.e., vectors) may be given a more refined type, which keeps track of vector lengths (also using shape types). The type system also supports special refined variants of singleton scalar values and singleton vector values. We present a formal treatment of the language in two steps. First, in Sections 3.1.1–3.1.4, we present a type system and a dynamic semantics for the language without array updates. Then, in Section 3.1.5, we extend the treatment to a store-based semantics with support for mutable arrays, array indexing, and array updates.

We demonstrate that the typed array intermediate language TAIL is suitable as the target for an inference algorithm for an APL compiler. As we shall see in Section 3.2, the type system of the intermediate language allows the compiler to treat complex operations, such as matrix-multiplication, and generalized versions thereof (inner products of higher-ranked arrays) as operations derived from the composition of other more basic operations.

With this work we aim at bridging the APL community and the functional programming community. The concise syntax of APL primitives and array abstraction concepts provide a rich source for data-parallel programming tech-

niques, in particular for the APL subset that we consider, which encourages a functional style of programming [89]. As an example of how the APL community can benefit from the programming language community (besides getting APL programs to run efficiently), we demonstrate in Section 3.3 how APL programmers may use explicit type annotations, in their APL programs, to express, and statically certify, properties about defined operators and functions.

We demonstrate that the typed array intermediate language is useful as a language for further array-compilation, by demonstrating that the array language can be compiled effectively into a low-level array intermediate language, called Laila, which lends itself to a straightforward translation into sequential C code.

Finally, the effectiveness of the compilation approach is demonstrated, in Section 3.5, by performance evaluation on a number of real world application benchmarks (as well as a number of micro benchmarks) using our compilation approach and comparing with a state-of-the-art APL interpreter. We examine the possibility for future automatic parallelization, summarize the current bottlenecks, and present work on compiling TAIL to the data-parallel languages Accelerate and Futhark.

3.1 A Typed Array Intermediate Language

In this section, we present the typed array intermediate language TAIL. For presentation purposes, we first present a subset of the language that does not have support for mutable arrays in terms of array updates. Then in Section 3.1.5, we develop the formal setup for covering also mutable arrays.

We assume a denumerable infinite set of *program variables* (x). We use i and n to range over integers, d to range over doubles, and b to range over boolean values `tt` and `ff`. Whenever z is some object, we write $\vec{z}$ to range over sequences of such objects. When we want to be explicit about the size of a sequence $\vec{z} = z_0, \cdots, z_{(n-1)}$, we often write it on the form $\vec{z}^{(n)}$ and we write $z, \vec{z}$ to denote the sequence $z, z_0, \cdots, z_{(n-1)}$.

Shapes (δ), base values (a), arrays (*arr*), primitive operations (*op*), values (v), and expressions (e) are defined as follows:

$$
\begin{array}{rcll}
\delta & ::= & \langle \vec{n} \rangle & \text{(shapes)} \\
bv & ::= & i \mid d \mid b & \text{(base values)} \\
arr & ::= & [\vec{bv}]^{\delta} & \text{(arrays)} \\
op & ::= & \texttt{addi} \mid \texttt{subi} \mid \texttt{muli} \mid \texttt{mini} \mid \texttt{maxi} & \text{(operations)} \\
 & \mid & \texttt{addd} \mid \texttt{subd} \mid \texttt{muld} \mid \texttt{mind} \mid \texttt{maxd} & \\
 & \mid & \texttt{andb} \mid \texttt{orb} \mid \texttt{notb} & \\
 & \mid & \texttt{lti} \mid \texttt{ltei} \mid \texttt{gti} \mid \texttt{gtei} \mid \texttt{eqi} & \\
 & \mid & \texttt{ltd} \mid \texttt{lted} \mid \texttt{gtd} \mid \texttt{gted} \mid \texttt{eqd} & \\
 & \mid & \texttt{iota} \mid \texttt{each} \mid \texttt{reduce} \mid \texttt{i2d} \mid \texttt{b2i} & \\
 & \mid & \texttt{reshape0} \mid \texttt{reshape} \mid \texttt{rotate} & \\
 & \mid & \texttt{transp} \mid \texttt{transp2} \mid \texttt{zipWith} & \\
 & \mid & \texttt{shape} \mid \texttt{take} \mid \texttt{drop} \mid \texttt{first} & \\
 & \mid & \texttt{cat} \mid \texttt{cons} \mid \texttt{snoc} & \text{(derived ops)} \\
 & \mid & \texttt{shapeV} \mid \texttt{catV} \mid \texttt{consV} \mid \texttt{snocV} & \text{(shape ops)} \\
 & \mid & \texttt{iotaV} \mid \texttt{rotateV} & \\
 & \mid & \texttt{takeV} \mid \texttt{dropV} \mid \texttt{firstV} & \\
v & ::= & arr \mid \lambda x.e & \text{(values)} \\
e & ::= & v \mid x \mid [\vec{e}] \mid e\ e' & \text{(expressions)} \\
 & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid op(\vec{e}) &
\end{array}
$$

Notice that array expressions $[\vec{e}]$ are always one-dimensional, whereas array values $[\vec{bv}]^{\delta}$ may be multi-dimensional with their dimensionality specified by the shape δ. We often write i, d, and b to denote scalar values $[i]^{\langle\rangle}$, $[d]^{\langle\rangle}$, and $[b]^{\langle\rangle}$, respectively.

3.1.1 A Type System with Shape Polymorphism

We assume denumerable infinite sets of *type variables* (α) and *shape variables* (γ).

$$
\begin{array}{rcll}
\kappa & ::= & \texttt{int} \mid \texttt{double} \mid \texttt{bool} \mid \alpha & \text{(base types)} \\
\rho & ::= & i \mid \gamma \mid \rho + \rho' & \text{(shape types)} \\
\tau & ::= & [\kappa]^{\rho} \mid \langle \kappa \rangle^{\rho} \mid \mathrm{S}_{\kappa}(\rho) \mid \mathrm{SV}_{\kappa}(\rho) & \text{(types)} \\
 & \mid & \tau \to \tau' & \\
\sigma & ::= & \forall \vec{\alpha}\vec{\gamma}.\tau & \text{(type schemes)}
\end{array}
$$

Types are segmented into base types (κ), shape types (ρ), types (τ), and type schemes (σ). Shape types (ρ) are considered identical upto associativity and commutativity of $+$ and upto evaluation of constant shape-type expressions involving $+$.

Types (τ) are either multidimensional arrays, $[\kappa]^{\rho}$, with explicit rank ρ, one-dimensional vectors, $\langle \kappa \rangle^{\rho}$ with explicit length ρ, singleton integers and booleans ($\mathrm{S}_{\kappa}(\rho)$) or single-element integer and boolean vectors ($\mathrm{SV}_{\kappa}(\rho)$), both with value ρ, or function types. As special notation, we often write κ to denote the scalar array type $[\kappa]^{0}$.

A *type substitution* (S_t) maps type variables to base types. A *shape substitution* (S_s) maps shape variables to shape types. A *substitution* (S) is a pair (S_t, S_s) of

a type substitution and a shape substitution. Applying a substitution S to some object B, written $S(B)$, has the effect of simultaneously applying S_t and S_s to objects in B (being the identity outside their domain). A type τ' is *an instance of* a type scheme $\sigma = \forall\vec{\alpha}\vec{\gamma}.\tau$, written $\sigma \geq \tau'$, if there exists a substitution S such that $S(\tau) = \tau'$.

A type τ is a *subtype* of another type τ', written $\tau \subseteq \tau'$, if the relation can be derived according to the following rules:

Subtyping $\boxed{\tau \subseteq \tau'}$

$$\frac{}{\tau \subseteq \tau} \text{ (3.1)} \qquad \frac{\tau_1 \subseteq \tau_2 \quad \tau_2 \subseteq \tau_3}{\tau_1 \subseteq \tau_3} \text{ (3.2)} \qquad \frac{}{\langle\kappa\rangle^\rho \subseteq [\kappa]^1} \text{ (3.3)}$$

$$\frac{}{\mathrm{SV}_\kappa(\rho) \subseteq \langle\kappa\rangle^1} \text{ (3.4)} \qquad \frac{}{\mathrm{S}_\kappa(\rho) \subseteq [\kappa]^0} \text{ (3.5)}$$

The subtyping relation allows known-sized vectors (one-dimensional arrays) to be treated as shape vectors with the number of dimensions being statically known. For instance, we shall see that the constant integer vector expression $[1, 2, 3]$ is given the type $\langle\texttt{int}\rangle^3$, but that the expression can also be given the type $[\texttt{int}]^1$ using the subtyping relation. Similarly, when asking for the shape of an integer vector of type $\langle\texttt{int}\rangle^\gamma$, we obtain a one-element integer vector containing the value γ. For typing this value, we can use the singleton vector type $\mathrm{SV}_{\texttt{int}}(\gamma)$, which is a subtype of $\langle\texttt{int}\rangle^1$, the type of one-element integer vectors.

Each operator, *op*, is given a unique type scheme, σ, as specified by the relation $\mathrm{TySc}(op) = \sigma$ defined in Figure 3.2 and Figure 3.3. For all operators *op*, such that $\mathrm{TySc}(op) = \forall\vec{\alpha}\vec{\gamma}.\tau_1 \to \ldots \to \tau_n \to \tau$, where τ is not a function type, we say that the *arity* of the operator *op*, written *arity*(*op*), is n.

Type assumptions Γ map variables to type schemes:

$$\Gamma \quad ::= \quad \Gamma, x : \sigma \mid \bullet$$

The type system allows inferences among sentences of the form $\Gamma \vdash e : \tau$, which are read: "under the assumptions Γ, the expression e has type τ."

Shape typing $\boxed{\vdash \delta : \rho}$

$$\frac{}{\vdash \langle\vec{n}^{(i)}\rangle : i} \text{ (3.6)}$$

Base value typing $\boxed{\vdash bv : \kappa}$

$$\frac{}{\vdash i : \texttt{int}} \text{ (3.7)} \qquad \frac{}{\vdash d : \texttt{double}} \text{ (3.8)}$$

$$\frac{}{\vdash \texttt{tt} : \texttt{bool}} \text{ (3.9)} \qquad \frac{}{\vdash \texttt{ff} : \texttt{bool}} \text{ (3.10)}$$

Array typing $\boxed{\vdash arr : \tau}$

$$\frac{\vdash \delta : \rho \quad \vdash bv_i : \kappa}{\vdash [\vec{bv}]^\delta : [\kappa]^\rho} \text{ (3.11)} \qquad \frac{\vdash bv_i : \kappa}{\vdash [\vec{bv}]^{\langle n\rangle} : \langle\kappa\rangle^n} \text{ (3.12)}$$

$$\frac{}{\vdash \texttt{tt} : \mathrm{S}_{\texttt{bool}}(1)} \text{ (3.13)} \qquad \frac{}{\vdash \texttt{ff} : \mathrm{S}_{\texttt{bool}}(0)} \text{ (3.14)}$$

APL	*op*(s)		TySc(*op*)
	addi,...	:	$\texttt{int} \to \texttt{int} \to \texttt{int}$
	addd,...	:	$\texttt{double} \to \texttt{double} \to \texttt{double}$
⍳	iota	:	$\texttt{int} \to [\texttt{int}]^1$
¨	each	:	$\forall\alpha\beta\gamma.(\alpha \to \beta) \to [\alpha]^\gamma \to [\beta]^\gamma$
/	reduce	:	$\forall\alpha\gamma.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha]^{1+\gamma} \to [\alpha]^\gamma$
/	compress	:	$\forall\alpha\gamma.[\texttt{bool}]^\gamma \to [\alpha]^\gamma \to [\alpha]^\gamma$
/	replicate	:	$\forall\alpha\gamma.[\texttt{int}]^\gamma \to [\alpha]^\gamma \to [\alpha]^\gamma$
\	scan	:	$\forall\alpha\gamma.(\alpha \to \alpha \to \alpha) \to [\alpha]^\gamma \to [\alpha]^\gamma$
⍴	shape	:	$\forall\alpha\gamma.[\alpha]^\gamma \to \langle\texttt{int}\rangle^\gamma$
⍴	reshape0	:	$\forall\alpha\gamma\gamma'.\langle\texttt{int}\rangle^{\gamma'} \to [\alpha]^\gamma \to [\alpha]^{\gamma'}$
⍴	reshape	:	$\forall\alpha\gamma\gamma'.\langle\texttt{int}\rangle^{\gamma'} \to \alpha \to [\alpha]^\gamma \to [\alpha]^{\gamma'}$
⌽	reverse	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^\gamma$
⌽	rotate	:	$\forall\alpha\gamma.\texttt{int} \to [\alpha]^\gamma \to [\alpha]^\gamma$
⍉	transp	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^\gamma$
⍉	transp2	:	$\forall\alpha\gamma.\langle\texttt{int}\rangle^\gamma \to [\alpha]^\gamma \to [\alpha]^\gamma$
↑	take	:	$\forall\alpha\gamma.\texttt{int} \to \alpha \to [\alpha]^\gamma \to [\alpha]^\gamma$
↓	drop	:	$\forall\alpha\gamma.\texttt{int} \to [\alpha]^\gamma \to [\alpha]^\gamma$
⊃	first	:	$\forall\alpha\gamma.\alpha \to [\alpha]^\gamma \to \alpha$
	zipWith	:	$\forall\alpha_1\alpha_2\beta\gamma.(\alpha_1 \to \alpha_2 \to \beta) \to [\alpha_1]^\gamma \to [\alpha_2]^\gamma \to [\beta]^\gamma$
,	cat	:	$\forall\alpha\gamma.[\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1}$
,	cons	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1}$
,	snoc	:	$\forall\alpha\gamma.[\alpha]^{\gamma+1} \to [\alpha]^\gamma \to [\alpha]^{\gamma+1}$

Figure 3.2: Operator type schemes for standard operations.

APL	*op*(s)		TySc(*op*)
⍴	shapeV	:	$\forall\alpha\gamma.\langle\alpha\rangle^\gamma \to \mathsf{SV}_{\texttt{int}}(\gamma)$
↑	takeV	:	$\forall\alpha\gamma.\mathsf{S}_{\texttt{int}}(\gamma) \to [\alpha]^1 \to \langle\alpha\rangle^\gamma$
↓	dropV	:	$\forall\alpha\gamma\gamma'.\mathsf{S}_{\texttt{int}}(\gamma) \to \langle\alpha\rangle^{(\gamma+\gamma')} \to \langle\alpha\rangle^{\gamma'}$
,	consV	:	$\forall\alpha\gamma.\alpha \to \langle\alpha\rangle^\gamma \to \langle\alpha\rangle^{(1+\gamma)}$
,	snocV	:	$\forall\alpha\gamma.\langle\alpha\rangle^\gamma \to \alpha \to \langle\alpha\rangle^{(1+\gamma)}$
⊃	firstV	:	$\forall\alpha\gamma.\mathsf{SV}_\alpha(\gamma) \to \mathsf{S}_\alpha(\gamma)$
⍳	iotaV	:	$\forall\gamma.\mathsf{S}_{\texttt{int}}(\gamma) \to \langle\texttt{int}\rangle^\gamma$
⌽	rotateV	:	$\forall\alpha\gamma.\langle\alpha\rangle^\gamma \to \langle\alpha\rangle^\gamma$
,	catV	:	$\forall\alpha\gamma\gamma'.\langle\alpha\rangle^\gamma \to \langle\alpha\rangle^{\gamma'} \to \langle\alpha\rangle^{(\gamma+\gamma')}$

Figure 3.3: Operator type schemes for operations on shapes.

$$\frac{}{\vdash n : \mathrm{S}_{\mathtt{int}}(n)} \text{ (3.15)} \qquad \frac{\vdash v : \mathrm{S}_{\kappa}(n)}{\vdash [v]^{\langle 1 \rangle} : \mathrm{SV}_{\kappa}(n)} \text{ (3.16)}$$

Value typing $\boxed{\Gamma \vdash v : \tau}$

$$\frac{\vdash \mathit{arr} : \tau}{\Gamma \vdash \mathit{arr} : \tau} \text{ (3.17)} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \text{ (3.18)}$$

Expression typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \mathrm{S}_{\kappa}(n)}{\Gamma \vdash [e] : \mathrm{SV}_{\kappa}(n)} \text{ (3.19)} \quad \frac{\Gamma(x) \geq \tau}{\Gamma \vdash x : \tau} \text{ (3.20)} \quad \frac{\tau \subseteq \tau' \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \text{ (3.21)}$$

$$\frac{\Gamma \vdash e_i : \kappa \quad i = [0; n[}{\Gamma \vdash [\vec{e}^{(n)}] : [\kappa]^1} \text{ (3.22)} \qquad \frac{\Gamma \vdash e_i : \kappa \quad i = [0; n[}{\Gamma \vdash [\vec{e}^{(n)}] : \langle \kappa \rangle^n} \text{ (3.23)}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \text{ (3.24)} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \forall \vec{\alpha}\vec{\gamma}.\tau \vdash e_2 : \tau' \quad \mathrm{fv}(\vec{\alpha}\vec{\gamma}) \cap \mathrm{fv}(\Gamma, \tau') = \emptyset}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau'} \text{ (3.25)}$$

$$\frac{\begin{array}{c}\mathrm{TySc}(\mathit{op}) \geq \tau_0 \to \ldots \to \tau_{(n-1)} \to \tau \\ \mathrm{Arity}(\mathit{op}) = n \quad \Gamma \vdash e_i : \tau_i \quad i = [0; n[\end{array}}{\Gamma \vdash \mathit{op}(\vec{e}^{(n)}) : \tau} \text{ (3.26)}$$

Notice that operators are required to be fully applied. In examples, however, when an operator *op* is not applied to any arguments (e.g., it is passed to a higher-order function), it is eta-expanded into the form $\lambda x_1.\cdots\lambda x_n.\mathit{op}(x_1, \cdots, x_n)$, where $n = \mathrm{Arity}(\mathit{op})$.

As indicated by the operator type schemes, certain limitations apply. For instance, in accordance with APL, the `each` operator operates on each base value of a multi-dimensional array. One may consider providing a `map` operator with the following type scheme:

$$\mathtt{map} \quad : \quad \forall\alpha\beta\gamma.([\alpha]^{\gamma} \to [\beta]^{\gamma}) \to [\alpha]^{1+\gamma} \to [\beta]^{1+\gamma}$$

However, it is not possible, with the present type system, to express that the function returns arrays with the same extent for all arguments; the only guarantee the type system can give us is that result arrays have the same rank (number of dimensions). More expressive type systems, based on dependent types, such as those found in AgdaAccelerate [98] and QUBE [101, 99], allow for expressing more accurately, the assumptions of the higher-order operators.

Similarly, one may consider providing a `reduce'` operator with the following type scheme:

$$\begin{array}{rcl}\mathtt{reduce'} & : & \forall\alpha\gamma.([\alpha]^{\gamma} \to [\alpha]^{\gamma} \to [\alpha]^{\gamma}) \to [\alpha]^{\gamma} \\ & & \to [\alpha]^{1+\gamma} \to [\alpha]^{\gamma}\end{array}$$

The idea here is that the operator operates on entire subparts of the argument array. In a system supporting only `map` and `reduce'` (and not `each` and `reduce`), nested instances of `map`s can be used instead of `each` and nested

instances of `map`s with an inner `reduce'` can be used instead of `reduce`. Additionally, one may consider providing a sequential `fold` operator that does not require associativity of the argument function:

$$\begin{aligned}\texttt{fold} \;:\; & \forall\alpha\beta\gamma.([\alpha]^{\gamma} \to [\beta]^{\gamma'} \to [\beta]^{\gamma'}) \to [\beta]^{\gamma'} \\ & \to [\alpha]^{1+\gamma} \to [\beta]^{\gamma'}\end{aligned}$$

As we shall see in the next section, the semantics of `reduce` is that it reduces the argument array along its last dimension, following the traditional APL semantics [63, 73].

The implementation of the APL compiler uses a hybrid approach of type inference and local context querying for resolving array ranks, scalar extensions, and identity items (neutral elements) during intermediate language program generation. The inference is based on a simple unification algorithm using conditional unification for the implementation of the subtyping inference.

3.1.2 Example Programs

We now present a few example programs that utilize the various operators. The dot-product of two integer arrays can be defined in the language as follows:

$$\begin{aligned}\texttt{dotpi} \;&:\; (\forall\gamma.[\texttt{int}]^{1+\gamma} \to [\texttt{int}]^{1+\gamma} \to [\texttt{int}]^{\gamma}) \\ &=\; \lambda\texttt{x}.\lambda\texttt{y}.\texttt{reduce(addi,0,zipWith(muli,x,y))}\end{aligned}$$

Notice that this function also works with integer matrices and integer arrays of higher dimensions. In case the extents of the argument arrays do not match up, the `zipWith` expression—and therefore the `dotpi` call—will result in a runtime error, as further specified in the presentation of the dynamic semantics below. We can generalize the above function to be useful in a broader sense:

$$\begin{aligned}\texttt{dotp} \;&:\; (\forall\gamma\alpha.(\alpha \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha) \to [\alpha]^{1+\gamma} \to [\alpha]^{1+\gamma} \to [\alpha]^{\gamma} \\ &=\; \lambda\texttt{add}.\lambda\texttt{mul}.\lambda\texttt{n}.\lambda\texttt{x}.\lambda\texttt{y}.\texttt{reduce(add,n,zipWith(mul,x,y))}\end{aligned}$$

3.1.3 Dynamic Semantics

Evaluation contexts, ranged over by E, take the following form:

$$\begin{aligned}E \;::=\; & [\cdot] \mid [\vec{v}E\vec{e}] \mid E\ e \mid v\ E \\ \mid\; & \texttt{let}\ x = E\ \texttt{in}\ e \mid op(\vec{v}E\vec{e})\end{aligned}$$

When E is an evaluation context and e is an expression, we write $E[e]$ to denote the expression resulting from filling the hole in E with e. The dynamic semantics is presented as a small step reduction semantics, which is explicit about certain kinds of errors that are not easily checked statically. Intuitively, a well-typed expression e is either a value or it can be reduced into another expression or the special token ***err***. Errors that are treated explicitly include

negative values passed to `iota` and illegal axis specifications as arguments generalized transposition (see below), `transp2`.

We first define a few helper functions for computations on shapes and for converting between flat indexing and multi-dimensional indexing. We assume a reverse operation (rev) on shapes and an operation, named product, that takes a shape of an array and returns the number of elements in the flattened version of the array. An expression $\text{fromIdx}_\delta \delta'$ takes a shape δ of an array and a multi-dimensional index δ' into the array and returns a corresponding index into the flattened version of the array.

$$\begin{array}{lcl} \text{fromIdx}_{\langle\rangle}\langle\rangle & = & 0 \\ \text{fromIdx}_{\langle n,\vec{n}\rangle}\langle i,\vec{i}\rangle & = & i * p + \text{fromIdx}_{\langle\vec{n}\rangle}\langle\vec{i}\rangle \\ \quad \text{where } p = \text{product}(\vec{n}) & & \end{array}$$

An expression $\text{toIdx}_\delta i$ takes a shape δ and an index i into the flattened version of the array and returns the corresponding multi-dimensional index into the array.

$$\begin{array}{lcl} \text{toIdx}_{\langle\rangle} 0 & = & \langle\rangle \\ \text{toIdx}_{\langle n,\vec{n}\rangle} i & = & \langle i \text{ div } p, \vec{i}\rangle \\ \quad \text{where} \quad p & = & \text{product}(\vec{n}) \\ \qquad\qquad \langle\vec{i}\rangle & = & \text{toIdx}_{\langle\vec{n}\rangle}(i \bmod p) \end{array}$$

The expression $\text{exchange}_\delta \delta'$ exchanges the elements in the shape δ' according to δ:

$$\begin{array}{l} \text{exchange}_{\langle\vec{p}^{(n)}\rangle}\langle\vec{q}^{(n)}\rangle = \langle q_{p_0}, \cdots, q_{p_{(n-1)}}\rangle \\ \quad \text{where } \forall ij.i \neq j \Rightarrow p_i \neq p_j \end{array}$$

Notice the partiality of the exchange function; if $\delta' = \text{exchange}_{\langle\vec{i}^{(k)}\rangle}\delta$ then $\vec{i}^{(k)}$ is known to be a permutation of $0, \cdots, (k-1)$.

A majority of the dynamic semantics rules are given below. We have left out the rules for `rotate`, `cat`, `cons`, `snoc`, `drop`, `compress`, `replicate`, and `scan`. We have also left out the rules for the shape-versions of the operations (e.g., `takeV`), which are all easily defined in terms of the non-shape versions.

Small Step Reductions $\boxed{e \hookrightarrow e' \text{ or } \textbf{\textit{err}}}$

$$\frac{e \hookrightarrow e' \quad E \neq [\cdot]}{E[e] \hookrightarrow E[e']} \quad (3.27) \qquad \frac{e \hookrightarrow \textbf{\textit{err}} \quad E \neq [\cdot]}{E[e] \hookrightarrow \textbf{\textit{err}}} \quad (3.28)$$

$$\frac{}{\texttt{let } x = v \texttt{ in } e \hookrightarrow e[v/x]} \quad (3.29) \qquad \frac{}{(\lambda x.e)\ v \hookrightarrow e[v/x]} \quad (3.30)$$

$$\frac{}{[\vec{a}^{(n)}] \hookrightarrow [\vec{a}^{(n)}]^{\langle n\rangle}} \quad (3.31)$$

$$\frac{i = i_1 + i_2}{\texttt{addi}(i_1, i_2) \hookrightarrow i} \quad (3.32) \qquad \frac{d = d_1 + d_2}{\texttt{addd}(d_1, d_2) \hookrightarrow d} \quad (3.33)$$

$$\frac{n \geq 0}{\texttt{iota}(n) \hookrightarrow [1, \cdots, n]^{\langle n\rangle}} \quad (3.34) \qquad \frac{n < 0}{\texttt{iota}(n) \hookrightarrow \textbf{\textit{err}}} \quad (3.35)$$

$$\frac{e = [v_f\ a_0, \cdots, v_f\ a_{(n-1)}]}{\texttt{each}(v_f, [\vec{a}^{(n)}]^\delta) \hookrightarrow \texttt{reshape0}(\delta, e)} \quad (3.36)$$

$$\frac{\begin{array}{c}\delta = \langle \vec{n}, m\rangle \quad k = \text{product}(\vec{n}) \quad i = [0; m[\\ e_i = v_f\ a_{(i*k)}\ (\cdots (v_f\ a_{(i*k+m-1)}\ v)\cdots)\end{array}}{\texttt{reduce}(v_f, v, [\vec{a}^{(n)}]^\delta) \hookrightarrow \texttt{reshape0}(\langle \vec{n}\rangle, [\vec{e}^{(k)}])} \quad (3.37)$$

$$\frac{m = \text{product}(\delta') \quad f(i) = i \bmod n \quad n > 0}{\texttt{reshape}(\delta', a, [\vec{a}^{(n)}]^\delta) \hookrightarrow [a_{f(0)}, \cdots, a_{f(m-1)}]^{\delta'}} \quad (3.38)$$

$$\frac{m = \text{product}(\delta') \quad a_i = a \quad i = [0; m[}{\texttt{reshape}(\delta', a, []^\delta) \hookrightarrow [a_0, \cdots, a_{(m-1)}]^{\delta'}} \quad (3.39)$$

$$\frac{\delta' = \text{rev}(\delta) \quad f = \text{fromIdx}_{\delta'}\ o\ \text{rev}\ o\ \text{toIdx}_\delta}{\texttt{transp}([\vec{a}^{(n)}]^\delta) \hookrightarrow [a_{f(0)}, \cdots, a_{f(n-1)}]^{\delta'}} \quad (3.40)$$

$$\frac{\begin{array}{c}\delta' = \text{exchange}_{\delta_0}(\delta) \\ f = \text{fromIdx}_{\delta'}\ o\ \text{exchange}_{\delta_0}\ o\ \text{toIdx}_\delta\end{array}}{\texttt{transp2}(\delta_0, [\vec{a}^{(n)}]^\delta) \hookrightarrow [a_{f(0)}, \cdots, a_{f(n-1)}]^{\delta'}} \quad (3.41)$$

$$\frac{\neg\exists\delta'.\delta' = \text{exchange}_{\delta_0}(\delta)}{\texttt{transp2}(\delta_0, [\vec{a}]^\delta) \hookrightarrow \textbf{\textit{err}}} \quad (3.42)$$

$$\frac{\begin{array}{c}m \geq 0 \quad \delta' = \langle m, \vec{n}\rangle \quad j = \text{product}(\delta') \\ f(i) = \text{if } i < k \text{ then } a_i \text{ else } a\end{array}}{\texttt{take}(m, a, [\vec{a}^{(k)}]^{\langle n, \vec{n}\rangle}) \hookrightarrow [f(0), \cdots, f(j-1)]^{\delta'}} \quad (3.43)$$

$$\frac{\begin{array}{c}m < 0 \quad \delta' = \langle -m, \vec{n}\rangle \quad j = \text{product}(\delta') \\ f(i) = \text{if } i < k \text{ then } a_{(k-1-i)} \text{ else } a\end{array}}{\texttt{take}(m, a, [\vec{a}^{(k)}]^\delta) \hookrightarrow [f(0), \cdots, f(j-1)]^{\delta'}} \quad (3.44)$$

$$\frac{k > 0}{\texttt{first}(a, [\vec{a}^{(k)}]^\delta) \hookrightarrow a_0} \quad (3.45) \qquad \frac{}{\texttt{first}(a, []^\delta) \hookrightarrow a} \quad (3.46)$$

The transitive, reflexive closure of $\hookrightarrow$, written $\hookrightarrow^*$, is defined by the following two rules:

$$\frac{e \hookrightarrow e' \quad e' \hookrightarrow^* e''}{e \hookrightarrow^* e''} \quad (3.47) \qquad \frac{}{e \hookrightarrow^* e} \quad (3.48)$$

We further define $e\uparrow$ to mean that there exists an infinite sequence $e \hookrightarrow e_1 \hookrightarrow e_2 \hookrightarrow \cdots$. The presented language does not support general recursion or uncontrolled looping, thus all programs represented in the intermediate language are guaranteed to terminate. The semantic machinery does support the addition of recursion (e.g., for implementing APL's recursion operator ∇).

3.1.4 Properties of the Language

In the following, we give a few definitions before we present a unique decomposition proposition. This proposition is used for the proofs of type preservation

and progress, which allow us to establish a type soundness result for the language, following standard techniques [80].

A *redex*, ranged over by r, is an expression of the form

$$r \quad ::= \quad (\lambda x.e)\ v \ \mid\ \texttt{let}\ x = v\ \texttt{in}\ e \ \mid\ op(\vec{v}) \ \mid\ [\vec{v}]$$

The following unique decomposition proposition states that any well-typed term is either a value or the term can be decomposed into a unique context and a unique well-typed redex. Moreover, filling the context with an expression of the same type as the redex results in a well-typed term:

Proposition 1 (Unique Decomposition) *If* $\vdash e : \tau$ *then either* e *is a value* v *or there exists a unique* E*, a unique redex* e'*, and some* τ' *such that* $e = E[e']$ *and* $\vdash e' : \tau'$*. Furthermore, for all* e'' *such that* $\vdash e'' : \tau'$*, it follows that* $\vdash E[e''] : \tau$*.*

PROOF By induction over the derivation $\vdash e : \tau$. □

The proofs of the following type preservation and progress propositions are then straightforward and standard [80].

Proposition 2 (Type Preservation) *If* $\vdash e : \tau$ *and* $e \hookrightarrow e'$ *and* $e' \neq$ ***err*** *then* $\vdash e' : \tau$*.*

PROOF By induction over the structure of the typing derivation $\vdash e : \tau$, using Proposition 1. □

Proposition 3 (Progress) *If* $\Gamma \vdash e : \tau$ *then either*

1. e *is a value; or*
2. $e \hookrightarrow$ ***err****; or*
3. *there exists an expression* e' *such that* $e \hookrightarrow e'$*.*

PROOF By induction over the structure of the typing derivation. □

Proposition 4 (Type Soundness) *If* $\vdash e : \tau$ *then* $e\uparrow$*, or* $e \hookrightarrow^* v/$***err****, or there exists* v *such that* $e \hookrightarrow^* v$*.*

PROOF By induction on the number of machine steps using Proposition 2 and Proposition 3. □

APL	*op*(s)		TySc(*op*)
	`index`	:	$\texttt{int} \to \texttt{int} \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma}$
	`update`	:	$[\alpha]^{\gamma+1} \to \langle\texttt{int}\rangle^{\gamma+1} \to \alpha \to \texttt{bool}$
`*`	`power`	:	$([\alpha]^{\gamma} \to [\alpha]^{\gamma}) \to \texttt{int} \to [\alpha]^{\gamma} \to [\alpha]^{\gamma}$

Figure 3.4: Operator type schemes for store-based operations.

3.1.5 Mutable Arrays

For the formalization to support mutable arrays, we extend the evaluation semantics with a store and the typing rules with a notion of store typing. We assume a denumerable infinite set of *locations* (l). Primitive operations (*op*) and values (v) are redefined as follows:

op	::=	`...` | `index` | `update` | `power`	(operations)
v	::=	$l \mid \lambda x.e$	(values)

We now define a *store* (s) to be a finite map from locations to array values. When *arr* is an array value and s is a store with l in its domain, we write $s[l \mapsto \mathit{arr}]$ to mean the store s with the location l updated to contain the array value *arr*. Further, a store s may be juxtapositioned with another store s', written s, s', assuming that the domains of s and s' do not overlap.

Types and type schemes remain unchanged. Type schemes for the store-based operations are presented in Figure 3.4.

The `index` operation takes two integers, d and i, and an array a as arguments and returns a new array (of rank one less than the rank of a) taken from a with dimension d dissected at index i. It is an error if d or i does not match with the shape a. The motivation for this particular form of `index` function is the often-used APL index construct, which, for instance, supports that a plane is extracted from a cube, as follows:

```
cube ← 10 20 30⍴⍳6000  ⍝ Dim x=10,y=20,z=30
plane ← cube[;4;]      ⍝ Dissect cube at y=4
```

where the index into cube, would in TAIL-notation be represented as `index(2,4,cube)` (i.e. $d = 2$ and $i = 4$). The `index` function can easily be implemented using dyadic transpose (i.e., `transp2`), `take`, `drop`, and `assert`, thus, we will not give the dynamic semantics for the operation here. The `update` operation we consider supports only individual array element updates. The operation takes an array and an index vector as arguments together with a scalar value of the proper type. The operation has the side-effecting behavior of updating the array at the given index position with the scalar value. Again, it is an error if the index vector does not match with the shape of the array. If the function returns a proper value, it returns `tt` (to keep the presentation simple, we do not support unit types). The `power` operation results from compiling APLs power operator `*`. The operation takes as arguments an iterator function f, a

number n, indicating the number of times the function should be applied, and, finally, an initial array argument a. The `power` operation $(f\text{⍣}n)\ a$, resembles the mathematical notation $f^n(a)$, that is, compose the function f with itself n times and apply the resulting function to the argument a.

The careful reader may now be worried that unless care is taken, combining subtyping with mutability can lead to problems. Indeed, unless the subtyping relation is modified slightly, we will run into problems showing type preservation and progress. The problem is subtyping rule (3.4) and the support for singleton vector types, which allows for the TAIL type system to reason statically about the content of a vector returned by a call to the `shapeV` operation. Now, if this vector is mutated in memory, using the `update` operation, a central invariant is violated. Fortunately, it turns out that if the subtyping relation is strengthened to not include subtyping rule (3.4), we can establish a type soundness result for the language. In what follows, we use $\subseteq_{\mathrm{s}}$ to denote the strengthened subtyping relation, which consists of all the previous subtyping rules from Section 3.1.1, except (3.4). Leaving this rule out requires the APL frontend to infer instead that a copying coercion is inserted whenever a value of type $\mathrm{SV}_\kappa(\rho)$ needs to be treated as a value of type $\langle\kappa\rangle^1$. This restriction is only a small price to pay for type soundness!

A *store type* (Σ) is now defined as a finite map from locations to types. We say that a store type Σ' *extends* another store type Σ, written $\Sigma' \sqsupseteq \Sigma$, if $\mathrm{Dom}(\Sigma') \supseteq \mathrm{Dom}(\Sigma)$ and for all $l \in \Sigma$, we have that $\Sigma(l) \subseteq_{\mathrm{s}} \Sigma'(l)$. Notice again the use of the strengthened subtyping relation.

The extended type system allows inferences among sentences of the form $\Gamma; \Sigma \vdash e : \tau$, which are read: "under the variable assumptions Γ and the store typing Σ, the expression e has type τ." All expression typing rules (rules (3.19)–(3.26)) remain unchanged except that a Σ needs to be added to all expression typing judgments and except that rule (3.21) is modified to use the strengthened subtyping relation.

Expression typing $\boxed{\Sigma, \Gamma \vdash e : \tau}$

$$\frac{\tau \subseteq_{\mathrm{s}} \tau' \qquad \Gamma; \Sigma \vdash e : \tau}{\Gamma; \Sigma \vdash e : \tau'} \quad (3.49)$$

The typing rules for values and stores are now given as follows.

Value typing $\boxed{\Sigma, \Gamma \vdash v : \tau}$

$$\frac{\Sigma(l) = \tau}{\Sigma, \Gamma \vdash l : \tau} \quad (3.50)$$

Store typing $\boxed{\Sigma \vdash s}$

$$\frac{\Sigma \vdash s \quad \vdash \mathit{arr} : \tau}{\Sigma, \{l \mapsto \tau\} \vdash s, \{l \mapsto \mathit{arr}\}} \quad (3.51)$$

We now define the notion of a *configuration* (k) to be a pair, written (e/s), of an expression e and a store s. Intuitively, the small step semantics turn a

well-typed configuration k into another well-typed configuration k' or into the error token ***err***. The relation is formalized as a set of reduction rules of the form $k \hookrightarrow k'$ *or* ***err***. The reduction rules from Section 3.1.3 may all easily be adapted to the refined setting. We only show a few of the modified rules below, together with rules for the new operations.

Small Step Reductions $\boxed{k \hookrightarrow k' \text{ or } \textbf{\textit{err}}}$

$$\frac{e/s \hookrightarrow e'/s' \quad E \neq [\cdot]}{E[e]/s \hookrightarrow E[e']/s'} \quad (3.52) \qquad \frac{e/s \hookrightarrow \textbf{\textit{err}} \quad E \neq [\cdot]}{E[e]/s \hookrightarrow \textbf{\textit{err}}} \quad (3.53)$$

$$\frac{e' = e[v/x]}{\texttt{let } x = v \texttt{ in } e/s \hookrightarrow e'/s} \quad (3.54) \qquad \frac{e' = e[v/x]}{(\lambda x.e)\ v/s \hookrightarrow e'/s} \quad (3.55)$$

$$\frac{s' = s, \{l \mapsto [\vec{bv}^{(n)}]^{\langle n \rangle}\}}{[\vec{bv}^{(n)}]/s \hookrightarrow l/s'} \quad (3.56)$$

$$\frac{\begin{array}{c} s(l_{\texttt{arr}}) = [\vec{bv}^{(m)}]^{\langle \vec{s}^{(n)} \rangle} \quad s(l_{\texttt{idx}}) = [\vec{i}]^{\langle n \rangle} \\ \vec{k} = \text{map}(\lambda x.x+1, \vec{i}) \quad j = \text{fromIdx}_{\langle \vec{s} \rangle} \langle \vec{k} \rangle \quad j < m \\ s(l_{\texttt{val}}) = bv_{\text{v}} \quad s' = s[l_{\texttt{arr}} \mapsto [\vec{bv}']^{\langle \vec{s}^{(n)} \rangle}] \\ \vec{bv}' = (bv_0, \cdots, bv_{(j-1)}, bv_{\text{v}}, bv_{(j+1)}, \cdots, bv_{(m-1)}) \end{array}}{\texttt{update}(l_{\texttt{arr}}, l_{\texttt{idx}}, l_{\texttt{val}})/s \hookrightarrow l'/s', \{l' \mapsto \texttt{tt}\}} \quad (3.57)$$

$$\frac{\begin{array}{c} s(l_{\texttt{arr}}) = [\vec{bv}^{(m)}]^{\langle \vec{s}^{(n)} \rangle} \quad s(l_{\texttt{idx}}) = [\vec{i}]^{\langle n \rangle} \\ \vec{k} = \text{map}(\lambda x.x+1, \vec{i}) \quad j = \text{fromIdx}_{\langle \vec{s} \rangle} \langle \vec{k} \rangle \quad j \geq m \end{array}}{\texttt{update}(l_{\texttt{arr}}, l_{\texttt{idx}}, l_{\texttt{val}})/s \hookrightarrow \textbf{\textit{err}}} \quad (3.58)$$

$$\frac{\begin{array}{c} s(l_{\text{n}}) = n \quad n > 0 \\ e' = \texttt{power}(\lambda x.e, \texttt{subi}(l_{\text{n}}, 1), e[l_{\texttt{arr}}/x]) \end{array}}{\texttt{power}(\lambda x.e, l_{\text{n}}, l_{\texttt{arr}})/s \hookrightarrow e'/s} \quad (3.59)$$

$$\frac{s(l_{\text{n}}) \leq 0}{\texttt{power}(\lambda x.e, l_{\text{n}}, l_{\texttt{arr}})/s \hookrightarrow l_{\texttt{arr}}/s} \quad (3.60)$$

To extend the formal results of Section 3.1.4 to the store semantics, we need to be explicit about how the stores evolve through the reduction semantics. The notions of evaluation contexts and redexes remain unchanged.

Proposition 5 (Unique Decomposition) *If $\emptyset; \Sigma \vdash e : \tau$ then either e is a value v or there exists a unique E, a unique redex e', and some τ' such that $e = E[e']$ and $\emptyset; \Sigma \vdash e' : \tau'$. Furthermore, for all e'' such that $\emptyset; \Sigma \vdash e'' : \tau'$, it follows that $\emptyset; \Sigma \vdash E[e''] : \tau$.*

PROOF By induction over the derivation $\emptyset; \Sigma \vdash e : \tau$. □

The proofs of the following type preservation and progress propositions are then straightforward and standard (see e.g., the Chapter on references in [86]).

Proposition 6 (Type Preservation) *If* $\emptyset; \Sigma \vdash e : \tau$ *and* $\Sigma \vdash s$ *and* $e/s \hookrightarrow e'/s'$ *then there exists a store type* $\Sigma' \sqsupseteq \Sigma$ *such that* $\emptyset; \Sigma' \vdash e' : \tau$ *and* $\Sigma' \vdash s'$.

PROOF By induction over the structure of the typing derivation $\emptyset; \Sigma \vdash e : \tau$, using Proposition 1. □

Proposition 7 (Progress) *If* $\Gamma; \Sigma \vdash e : \tau$ *and* $\Sigma \vdash s$ *then either*

1. e *is a value; or*
2. $e/s \hookrightarrow$ ***err****; or*
3. *there exists an expression* e'*, a store* s'*, and a store type* $\Sigma' \sqsupseteq \Sigma$ *such that* $e/s \hookrightarrow e'/s'$ *and* $\Sigma' \vdash s'$.

PROOF By induction over the structure of the typing derivation. □

Proposition 8 (Type Soundness) *If* $\emptyset; \Sigma \vdash e : \tau$ *and* $\Sigma \vdash s$ *then either* $(e/s)\uparrow$ *or there exist a value* v*, a store* s'*, and a store type* $\Sigma' \sqsupseteq \Sigma$ *such that* $e/s \hookrightarrow^* (v/s')$ *or* ***err***.

PROOF By induction on the number of machine steps using Proposition 6 and Proposition 7. □

3.2 Compiling the Inner and Outer Products

One of the main reasons for the complexity of the type systems presented in the previous sections is for the intermediate language to be a suitable target for an APL compiler. In particular, the intermediate language must be rich enough that complex array operations can be mapped to constructs in the intermediate language.

We now demonstrate how the technique used by Guibas and Wyatt [50] for compiling away the "dot" operator in APL, by representing it by a sequence of simpler APL-operations. The APL source code is given in Figure 3.5, which consist of the definition and use of an APL dyadic operator `dot`.

We shall not go into discussing and explaining the details of the APL code for the `dot`-operator, except from mentioning that the left and right function argument to the operator is referenced within the definition of the `dot`-operator using `⍺⍺` and `⍵⍵`, respectively. The intermediate language code resulting from compiling the `dot`-operator example is shown in Figure 3.6. The intermediate language code generated for the inner product of two matrices may seem a bit extensive. However, once traditional optimizations are applied, the code is simplified drastically, as can be seen in Figure 3.7, which contains the result after an extensive set of type- and semantics-preserving optimizations have been applied to the TAIL code.

After TAIL program optimization, the example program can be compiled into C code, as shown in Figure 3.8. We shall return to the topic of compiling TAIL in Section 3.4.

```
dot ← { ⍝: (κ→κ→κ)→(κ→κ→κ) →[κ](ρ1+1)
           →[κ](ρ2+1)→[κ](ρ1+ρ2)
  WA ← (1↓⍴⍵),⍴⍺
  KA ← (⊃⍴⍴⍺)-1
  VA ← ⍳ ⊃ ⍴WA
  ZA ← (KA⌽¯1↓VA),¯1↑VA
  TA ← ZA⍉WA⍴⍺         ⍝ Replicate, transpose
  WB ← (¯1↓⍴⍺),⍴⍵
  KB ← ⊃ ⍴⍴⍺
  VB ← ⍳ ⊃ ⍴WB
  ZB0 ← (-KB) ↓ KB ⌽ ⍳(⊃⍴VB)
  ZB ← (¯1↓(⍳ KB)),ZB0,KB
  TB ← ZB⍉WB⍴⍵         ⍝ Replicate, transpose
  ⍺⍺ / TA ⍵⍵ TB        ⍝ Compute final array
}

A ← 3 2 ⍴ ⍳ 5          ⍝ Example input A
B ← ⍉ A                ⍝ Example input B
R ← A + dot × B
R2 ← ×/ +/ R           ⍝ Reduce on the result

⍝         1  3  5
⍝         2  4  1
⍝
⍝   1 2   5 11  7    -+->    23  |
⍝   3 4  11 25 19    -+->    55  ×
⍝   5 1   7 19 26    -+->    52  |
⍝                         65780  v
```

Figure 3.5: The definition of a general dot-operator in APL together with an application of the operator to functions + and × and two matrices `A` and `B`.

3.3 APL Explicit Type Annotations

Consider again the `signal` function from Example 3.1 in the beginning of the chapter. Often a programmer is interested in specifying static properties of a program and to indicate the intention of some functionality. For instance, the `diff` function takes a vector as an argument and returns a vector of length one less than the length of the argument (unless the argument is the empty vector, in which case the length of the result vector will also be zero.) Using a typed intermediate language as a target for APL compilation, we can allow programmers to specify types for functions and operators and have the specifications checked, statically, by the compiler. Here is a new definition of the `diff` function, which can be applied only to vectors of non-zero length and which returns a vector of length one less than the length of the argument.

```
let v1:[int]2 = reshape([3,2],iotaV(5)) in
let v2:[int]2 = transp(v1) in
let v3:[int]2 = v1 in
let v4:[int]2 = v2 in
let v5:<int>3 = catV(dropV(b2iV(tt),shape(v4)),
                 shape(v3)) in
let v6:[int]0 = subi(firstV(shapeV(shape(v3))),
                 b2iV(tt)) in
let v7:<int>3 = iotaV(firstV(shapeV(v5))) in
let v8:<int>3 = catV(transp(vrotateV(v6,transp(
                 dropV(~1,v7)))),takeV(~1,v7)) in
let v9:[int]3 = transp2(v8,reshape(v5,v3)) in
let v10:<int>3 = catV(dropV(~1,shape(v3)),
                 shape(v4)) in
let v11:S(int,2) = firstV(shapeV(shape(v3))) in
let v12:<int>3 = iotaV(firstV(shapeV(v10))) in
let v13:<int>1 = dropV(negi(v11),transp(vrotateV(
                  v11,transp(iotaV(firstV(
                  shapeV(v12)))))))) in
let v14:<int>3 = catV(dropV(~1,iotaV(v11)),
                  snocV(v13,v11)) in
let v15:[int]3 = transp2(v14,reshape(v10,v4)) in
let v20:[int]2 = reduce(addi,0,zipWith(muli,v9,v15)) in
let v25:[int]0 = reduce(muli,1,reduce(addi,0,v20)) in
i2d(v25)
```

Figure 3.6: Intermediate language code (before optimization) for inner product of a 3×2 matrix and a 2×3 matrix with operations + and ×.

```
let v1:[int]2 = reshape([3,2],iotaV(5)) in
let v2:[int]2 = transp(v1) in
let v9:[int]3 = transp2([2,1,3],reshape([3,3,2],v1)) in
let v15:[int]3 = transp2([1,3,2],reshape([3,2,3],v2)) in
let v20:[int]2 = reduce(addi,0,zipWith(muli,v9,v15)) in
let v25:[int]0 = reduce(muli,1,reduce(addi,0,v20)) in
i2d(v25)
```

Figure 3.7: Intermediate language code for the inner product example after TAIL optimizations.

```
#include <apl.h>
double kernel(int n33) {
  int n9 = 1;
  for (int n10 = 0; n10 < 3; n10++) {
    int n12 = 0;
    for (int n13 = 0; n13 < 3; n13++) {
      int n15 = 0;
      for (int n16 = 0; n16 < 2; n16++) {
        int n23 = 1+((((n13*6)+(n10*2)+n16)%6)%5);
        int n27 = ((n10*6)+(n16*3)+n13)%6;
        int n32 = 1+(((n27%3)*2)+(n27/3)%5);
        n15 = n15+(n23*n32);
      }
      n12 = n12+n15;
    }
    n9 = n9*n12;
  }
  return i2d(n9);
}
```

Figure 3.8: Target C-like code for computing the inner product of a 3×2 matrix and a 2×3 matrix with operations + and ×.

```
diff ← {      ⍝: <κ>(ρ+1) → <κ>ρ
  1↓⍵-¯1⌽⍵
}
```

Similarly, a type specification for the `signal` function can specify that the function takes a vector of doubles as argument and returns a vector of the same length as the argument.

```
signal ← {    ⍝: <double>ρ → <double>ρ
  ¯50⌈50⌊50×(diff 0,⍵)÷0.01+⍵
}
```

Another example of using APL type annotations is for the `dot`-operator defined in Figure 3.5. This function takes two dyadic scalar functions as arguments together with two arrays, of ranks $\rho_1 + 1$ and $\rho_2 + 1$, respectively, and returns a new array of rank $\rho_1 + \rho_2$.

We retain compatibility with Dyalog interpreter and compiler, by placing annotations inside comments, which are written using the symbol ⍝.

3.4 TAIL Compilation

In this section, we demonstrate the feasibility of compiling TAIL programs into a low-level intermediate language through a low-level array intermediate language, or API, called Laila. The low-level intermediate language is separated into an expression language part and a statement part and resembles

in this sense a standard low-level imperative language, such as C. Indeed, it is a straightforward task to generate C code from the low-level intermediate code. On top of the low-level intermediate language, we construct a number of array abstractions, which forms the language Laila. As we shall see, the Laila language is a suitable setup for hosting a notion of hybrid pull-vectors, which support both multidimensional arrays with shapes and ordinary pull vectors, which form the basis for our compilation to low-level code. This representation is inspired by the work on optimizing Accelerate programs [78].

We use t to range over types at the low-level intermediate language level:

$$ty \quad ::= \quad \texttt{int} \mid \texttt{double} \mid \texttt{bool} \mid [ty]$$

We further assume a denumerable infinite set of program variables x^{ty}, annotated with a type *ty*. We sometimes leave out types from program variables when they are of no importance. Expression terms t and statements s for the low-level intermediate language are defined as follows:

$$\begin{aligned}
bop &::= \texttt{addi} \mid \texttt{addd} \mid \texttt{andb} \mid \texttt{ltei} \cdots && \text{(binary ops)}\\
uop &::= \texttt{negi} \mid \texttt{expd} \cdots && \text{(unary ops)}\\
t &::= x^t \mid i \mid d \mid b \mid c \mid uop(t)\\
&\quad \mid bop(t,t) \mid \texttt{malloc}(ty,t) \mid \texttt{subs}(x,t)\\
s &::= \texttt{For}(t, \lambda x.s, s) \mid \texttt{If}(t,t,t,s) \mid \epsilon\\
&\quad \mid \texttt{Decl}(x,t,s) \mid \texttt{Assign}(x,e,s) \mid \texttt{Update}(x,i,e,s)
\end{aligned}$$

The language contains looping constructs and conditional constructs at the statement level as well as support for declarations of variables. At the expression term level, the `malloc` construct gives support for allocating memory for a number of elements of the specified type. The `subs` construct gives support for indexing into allocated memory. For any expression term t, we write $\texttt{typeof}(t)$ to determine the result type of the expression. This function is easily defined as all expressions have an immediate result type.

In the following, we define a number of data types for specifying hybrid pull-arrays and code generation. We use an ML syntax for the presentation, but the definitions should easily carry over to other settings. First we assume that the meta language types for expression terms t, statements s, and variables *var* are available. We use the syntax $\texttt{I}i$ and $\texttt{B}b$ for injecting integer values i and boolean values b into the term expression language.

At the low-level array intermediate language level, we want to hide the details of name bindings and, in general, hide the fact that we are working with a language that distinguishes between expressions and statement. To this end, we follow the standard technique [36] of using a monad for encapsulating program construction:

```
infix >>=
structure M : MONAD = struct
  type 'a M = 'a * (s -> s)
  fun (v,sT) >>= g = let val (v',sT') = g v
                     in (v',sT o sT')
                     end
  fun ret x = (x, fn s => s)
end
```

We can now define a set of helper functions for building intermediate language constructs:

```
fun alloc ty t = let val x = newVar ty
                 in (v,fn s => Decl(x,malloc(ty,t),s))
                 end
fun update a i v = ((), fn s => Update(a,i,v,s))
fun index a i = (Subs(v,i), fn s => s)
fun lprod nil = I 1
  | lprod (x::xs) = muli(x,lprod xs)
fun lett e =
   let val v = newVar (typeof e)
   in (v, fn s => Decl(n,e,s))
   end
fun for n f = ((), fn s => For(n, f, s))
```

The `lett` combinator constructs a binding of an expression. We now make use of the assumption that the ranks of all arrays in a program are statically determined at compile time. This static property makes it possible to define shapes as meta-level lists of residual expression terms:

```
type INT = t
type sh = INT list
```

Our notion of a *hybrid pull-array* is defined by an `idx` data type and an `arr` data type:

```
datatype idx = F of INT -> t M  (* flat representation   *)
             | N of sh -> t M   (* shaped representation *)
type arr = ty * sh * idx
```

The `arr` denotes a multi-dimensional pull-array with underlying base type `ty`, shape *sh*, and index function `idx`. The `idx` type supports two different encodings of indices. The first form specifies that the pull-array is encoded using a flat representation, requiring calculation of the proper index into a row-major representation, if encoding a multi-dimensional array. The second form specifies that the user needs only supply indices for each dimension to "pull out" an underlying value of the pull-array; we call the second form a *shaped pull-array*.

As we shall see below, some array operations are indifferent to the choice of the form of the index function, whereas other array operations work best with a particular form of index function. In particular, reshape operations are free to perform on flat pull-arrays, whereas transpositions are free to perform on shaped pull-arrays. We assume meta-level definitions of the toIdx and fromIdx functions (from Section 3.1.3) with the following type signatures:

```
val fromIdx : sh -> sh -> INT M
val toIdx   : sh -> INT -> sh M
```

We can now define a couple of helper functions for converting between shaped pull-arrays and flat pull-arrays:

```
fun toF(ty,sh,F f) = (ty,sh,f)
  | toF(ty,sh,N f) = (ty,sh,fn i => toSh sh i >>= f)
fun toN(ty,sh,N f) = (ty,sh,f)
  | toN(ty,sh,F f) = (ty,sh,fn is => fromSh sh is >>= f)
```

Figure 3.9 shows an implementation of a selection of array operations on hybrid pull-arrays. Notice in particular how the implementation of the `each` combinator is indifferent to the form of the pull-vector; the supplied function composes well with both forms of index functions. Notice also the implementation of the `materialize` combinator, which uses different materialization strategies dependent on the form of pull array it is given.

We have left out many of the important array operations necessary for a complete implementation of hybrid pull-arrays for the TAIL language. Missing operations include the `reduce` and `power` operations for which details of the semantics can be found in Section 3.1.3. The actual implementation of the `power` primitive may recognize cases where the size of the iterated array values is known not to change over the iterations. In such cases, a double-buffering approach can be used and memory can be allocated outside of the generated loop. Finally we mention that arrays that are mutated by the programmer always use a materialized representation.

3.5 Benchmarks

We have evaluated our compiler by comparison with a state of the art interpreter, namely Dyalog and handwritten C code. The chosen benchmarks exhibit different computation patterns, and range from a a few lines (e.g. game of life) to 200 lines of APL (a Monte-Carlo option pricer translated from code provided by LexiFi).

In addition to comparing the pure sequential performance, we investigated the possibility of future automatic parallelization, by adding a few OpenMP pragmas [85] by hand, and comparing parallel speedup with hand-optimized OpenMP code (Section 3.5.5).

We conclude the section by mentioning a few identified bottlenecks in our approach, and possible solutions (Section 3.5.7).

3.5.1 Benchmark setup

All benchmarks have been executed on an AMD Opteron system, using 32 cores, model 6274, and running at 2.2 GHz. All benchmarks have been executed 30 times each, and we report averages of wall-clock timings together with standard deviations. Time spent on file I/O while reading datasets to memory are not included in the measurements.

```
fun transp a = case toN a of
   (ty,sh,f) => (ty, rev sh, N o f o rev)

fun rotate (n,a) = case toN a of
   (ty,x::xs,f) => (ty,x::xs,N (fn i::is =>
                     f(modi(absi(addi(i,n),x))::ix)))
 | (ty,nil,f) => (ty,nil,N f)

fun drop (n,a) = case toN a of
   (ty,x::sh,f) => (ty, maxi(0,subi(x,absi(n)))::sh,
                    N(fn i::ix => f((i+n)::ix)))
 | (ty,nil,f) => (ty,nil,N f)

fun each f (ty,sh,idx) = (ty,sh,fn i => idx i >>= f)

fun extend sh f =
  lett (lprod sh) >>= (fn sz =>
  ret (fn i => f (modi(i,sz))))

fun reshape sh' a = case toF a of
  (ty,sh,f) => extend sh f >>= (fn g => (ty,sh',F g))

fun fornest nil k = k nil
  | fornest (s::sh) k =
     for s (fn i => fornest sh (fn ix => k(i::ix)))

fun materialize (ty,sh,idx) =
  lett (lprod sh) >>= (fn sz =>
  alloc ty sz >>= (fn a =>
  (case idx of
      F f => for sz (fn i => f i >>= update a i)
    | N f => lett (I 0) >>= (fn n =>
             fornest sh (fn ix => f ix >>= (fn v =>
                update a n v >>= (fn () =>
                assign n (addi(n,I 1))))))
  ) >>= (fn () => ret (ty,sh,N (ret o (index a))))
```

Figure 3.9: Array operations based on our hybrid pull-array representation.

For the micro-benchmarks we compare ourselves against the commercial 32-bit Dyalog APL interpreter version 14.0.22502, which have been made available to us by Dyalog Ltd. With this version, Dyalog provides its own experimental APL compiler, but it is still in an experimental stage and does not cover all of APL. For example, functions using indexed assignments can not be compiled, and you are not allowed to use global names (including calls to other user-defined functions). Only one of our benchmarks (Easter) could benefit from turning on automatic compilation, giving it a speed up of around 1.5 from approximately 8.8 sec to 6 sec. We have not looked further into the possibility of getting the remaining benchmarks to compile on Dyalog with Dyalog's builtin compilation support, especially because of the limitation with respect to user-defined functions.

3.5.2 Micro-benchmarks

These five benchmarks are used to compare ourselves against the Dyalog interpreter. The exact same source code have been used with a different prelude library for certain operations. In particular: File I/O and bitwise operations are implemented differently in Dyalog and TAIL.

In ordinary APL, such as in Dyalog APL, bitwise operations are written as manipulation of bit arrays, together with encode/decode operations between integer representation and bit array representation. This coding requires the language implementer to handle bit vectors carefully, to be able to make use of ordinary shift instructions, when a user issues a `take` or a `drop` on a bit vector. Instead of going into that, which is not part of our research agenda, we decided to provide direct access to the bitwise operations in TAIL, through special symbols `⎕AND`, `⎕XOR`, and so on. The benchmarks are introduced below, with problem-sizes given in Table 3.1, and the performance measurements displayed in Table 5.1.

Game of life Simulates N iterations of Conway's Game Of Life. This simulation is a standard APL-benchmark. However, for compilation with TAIL, it was necessary to rewrite it to avoid uses of nested arrays.

Easter This example is taken from a presentation by Dyalog, and computes the date of Easter Sunday in all years from year 1 to year 3000. We do this 300 times, just to be sure!

Primes This benchmark computes the first N primes, and is short enough to be able to present in entirety:

```
A←1↓⍳10000
primes ← (1=+⌿0=A∘.|A)/A
```

The main ingredient is a big outer-join using the remainder (modulus) operation. With a vertical reduction, the number of divisors for each input is counted and a compress-operation selects the primes.

Benchmark	Problem size	Source
Primes	N = 100.000	-
Easter	N = 300×3.000	Dyalog
Black-Scholes	N = 100.000	-
Sobol MC π	N = 10.000	Our own
Game of life	N = 20.000 (board: 40×40)	Our own
Option pricer	N = 1048576, medium dataset	FINPAR
HotSpot	N = 300 (grid: 512×512)	Rodinia

Table 3.1: Benchmarks used. The first five are relatively small micro benchmarks. The bottom two are larger real world applications.

Benchmark	Dyalog (ms)	TAIL (ms)	Speed-up
Primes	1423 ± 8.0	3190 ± 6.4	0.45
Easter	18259 ± 157	46 ± 5.2	401
Black-Scholes	3086 ± 20.2	96 ± 5.8	32
Sobol MC π	12692 ± 202	14.5 ± 1.1	871
Game of life	1490 ± 6.5	969.77 ± 6.8	1.5

Table 3.2: Timings of the five micro benchmarks on Dyalog and TAIL. Each benchmark was executed 30 times, the standard deviations are annotated.

Benchmark	Dyalog	TAIL (ms)	C (ms)
Option pricer	57 min.	4587 ± 70	2267 ± 142
HotSpot	14832 ms ± 353	3679 ± 5	2044 ± 69

Table 3.3: Application benchmarks in Dyalog, TAIL and handwritten C code respectively. We have executed all benchmarks 30 times, except the Option pricer running on Dyalog.

Black Scholes This is a standard APL benchmark, pricing European (call) options. We price the exact same option N times.

Monte-Carlo π with Sobol-sequences For Monte-Carlo problems where sample correlation is non problematic, Sobol-sequences provides a good source of random numbers. Instead of sampling pseudo-randomly, they fill out the sample space systematically. This is an advantage in many situations, giving faster convergence. In this benchmark we use Sobol-numbers to compute a Monte-Carlo approximation to π.

3.5.3 Application benchmarks

Option Pricing This benchmark is a Black-Scholes Monte-Carlo model for option pricing a wide class of contracts (the model does not handle American options). The benchmark follows a nested Map/Reduce pattern. Involved

components are: Sobol-sequence generation [20], conversion from uniform to normal distribution, and generation of price evolution paths using Brownian bridge discretization. To simulate a market with dependencies among the underlyings, a Black-Scholes model is used to correlate the generated paths. Finally, the payoff for each option are calculated and in the reduction phase all the obtained prices are averaged.

The original C code for this benchmark came from production code by LexiFi, which were extracted to a stand-alone presentation in the FINPAR benchmark suite [7]. Implementations details and mathematical foundations can be found in the FINPAR paper.

We compare ourselves with the medium-sized dataset, which requires 3 underlyings, 5 time steps and repeating the pricing for $N = 1048576$ independent iterations. The large dataset was not manageable for the Dyalog interpreter.

The Dyalog implementation was initially implemented by us, and optimized with suggestions from Dyalog software engineers.

HotSpot HotSpot is a widely used tool in the VLSI design process, and is used to estimate processor temperature using simulation. The inputs are two $M \times M$ matrices of respectively power and initial temperatures. The benchmark progress by iteratively solving a series of differential equations. The output is an $M \times M$ grid where each cell represents the average temperature value for the corresponding area of the die.

This benchmark is taken from the Rodinia benchmark suite for heterogeneous computing [33]. We have ported the implementation from code in the APL-like language ELI, original presented by WM Ching et al. in connection with work on ELI-to-C compilation and parallelization [34].

We have benchmarked using the same problem size as in the original Rodinia paper: 512x512 grid and 360 iterations, though they also provide a dataset for a 1024x1024 grid simulation.

3.5.4 Hybrid pull-array performance

The hybrid pull arrays introduced in Section 3.4, was motivated by a desire to eliminate index space calculations, for example when doing transpositions, as they are free as long as the array is on the form of a *shaped pull-array*. To demonstrate the gain of this hybrid approach, we have compared our TAIL performance using hybrid push arrays with ordinary flat pull arrays in the style of Guibas and Wyatt [50]. The timings are presented in Table 3.4.

3.5.5 Parallelization potential

To test whether there is a potential for automatic parallelization of TAIL programs, we have attempted adding OpenMP pragmas by hand to the generated code. We have only attempted it on the two application benchmarks, and we only added OpenMP-annotations to a single loop in each case. In the option pricing benchmark, where the outer loop is a large Map/Reduce, we added

Benchmark	Flat	Hybrid	Speed-up
Primes	4243 ms	3190 ms	1.01
Easter	129 ms	46 ms	2.84
Black-Scholes	97 ms	96 ms	1.22
Sobol MC π	21 ms	14.5 ms	1.33
Game of life	1183 ms	970 ms	1.43
Option pricer	4947 ms	4587 ms	1.08
HotSpot	5238 ms	3679 ms	1.42

Table 3.4: Performance comparison between flat pull arrays and our hybrid pull array on the seven benchmarks. Each benchmark was executed 30 times.

a single OpenMP reduction-pragma, hoisted out all the memory allocations, letting the threads share the same allocation and padded the memory allocations to avoid memory conflicts. With these few applications we got the speedup shown in Figure 3.10. In the case of HotSpot, the outermost loop has a dependence between loop iterations, making it inherently sequential, but in each iteration of that loop there is a large map, which can be parallelized by adding a single OpenMP-annotation. The speed-up graph for this attempt is also shown in Figure 3.10. In both cases we can observe linear speed-up until 8 threads where after the curve flattens a bit, at 32 threads we get around 21 times speedup compared to the sequential versions. These measurements demonstrate that even a simple parallelization strategy might work, by adding parallelization pragmas to the outer-most maps, reductions or scans, in a breadth first search through the AST. To obtain good performance, it will also be necessary to be able to hoist memory allocations out of loops, when there is no forward dependency to next iteration.

3.5.6 TAIL on GPUs

A backend based on pull-arrays is of course only one out of many possibilities for compiling the intermediate language. Together with students and colleagues we have also compiled TAIL to the existing data-parallel GPU languages Accelerate and Futhark.

Compiling TAIL to Accelerate

The language Accelerate is an array language for GPU programming embedded in Haskell, supporting a wide range of data-parallel primitives, multidimensional and rank-polymorphic programming. Accelerate is limited by not supporting nested data-parallelism. We have previously presented a Accelerate-based backend for TAIL [23]. The backend was implemented as part of a student project. To support the primitives of APL, a library of APL-style functions were implemented, by the use of core Accelerate constructs. Various difficulties arose, especially the encoding of shapes as so-called *snoc*-list, made implementing operations operating on the outer dimension (inner most in

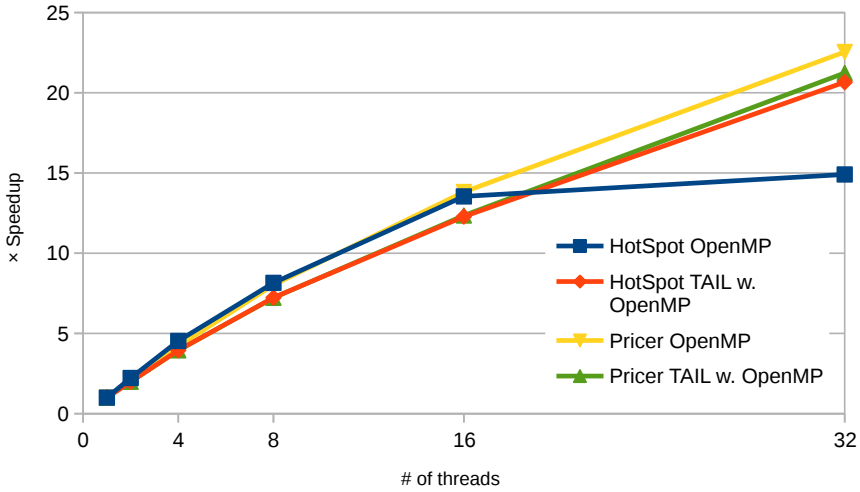

Figure 3.10: Speed-up compared with sequential version

Benchmark	**Problem size**	**TAIL C (ms)**	**TAIL Accelerate (ms)**
Signal	N = 50,000,000	209.03	16.1
Game-of-Life	40×40, N = 2,000	28.70	2.30
Easter	N = 3,000	33.96	-
Black-Scholes	N = 10,000	54.0	-
Sobol MC π	N = 10,000,000	4881.30	2430.30
HotSpot	1024×1024, N = 360	6072.93	2.03

Table 3.5: Benchmark timings in milliseconds. The timings are averages over 30 executions. TAIL C is using our sequential C-code backend, TAIL Accelerate is using the Accelerate backend.

the snoc-list), hard to write in a rank-polymorphic way without turning to potentially inefficient array transpositions.

Table 3.5 lists our timings on a few of the same benchmarks mentioned previously. All benchmarks were executed on a system with an Intel Xeon CPU E5-2650 and an NVIDIA GeFore GTX 780 Ti GPU.

Our TAIL-to-Accelerate compiler failed to generate code for the Easter and Black-Scholes benchmark, as the current formulation of those benchmarks are written using nested reduce operations, which is not supported by Accelerate. An alternative would be to attempt building a rewriting engine into the TAIL compiler, that could probably eliminate the need for nested parallelism in these smaller benchmarks.

Benchmark	Problem size	TAIL C (ms)	TAIL Futhark (ms)
Signal	N = 50,000,000	712.20	1.49
Game-of-Life	1200×1200, N = 100	1274.03	16.11
Easter	N = 10,000,000	321.80	1.80
Black-Scholes	N = 10,000,000	5396.10	9.13
Sobol MC π	N = 10,000,000	368.67	23.68
HotSpot	512×512, N = 360	1210.57	16.84

Table 3.6: Benchmark timings in milliseconds. The timings are averages over 30 executions. TAIL C is using our sequential C-code backend, TAIL Futhark is compiled using the Futhark backend.

Compiling TAIL to Futhark

As part of another student project, TAIL was also compiled into the language Futhark [55].

Futhark is a purely functional programming language targeting GPUs developed by colleagues at the HIPERFIT research center. Futhark is based on a calculi of array operators inspired by the Theory of Lists by Bird [16], allowing fusion and several other optimizations necessary to generate efficient GPU code.

As Futhark is not a polymorphic language, TAIL programs involving polymorphism was monomorphed before translation to Futhark. Monomorphization includes some of the built-in operations.

The Easter and Black-Scholes benchmarks that proved to be troubling for Accelerate, was manageable for Futhark to compile to efficient GPU code. Translating TAIL to Futhark did however still not allow us to translate the larger Option pricing benchmark from the FINPAR benchmark suite presented previously, because of the use of in-place updates which are necessary to write the so-called Brownian bridge path generation procedure.

Table 3.6 lists timings as previously reported [55]. The benchmarks were executed on the same system as the benchmarks of our TAIL-to-Accelerate compiler above, and are thus comparable, although (for some reason unknown to this Ph.D. student) the problems size of some of the benchmarks have been changed.

A comparison between Futhark and Accelerate is now also possible. In most cases Futhark drastically outperforms Accelerate, but in the case of HotSpot, Futhark is 8 times slower on a much smaller dataset (512×512, compared with 1024×1024 in the case of Accelerate).

The results presented here, for compilation of TAIL to both Accelerate and Futhark, are magnitudes better than what is attainable from the industry standard interpreter Dyalog. These results give us further confidence that TAIL is suitable as an intermediate language for compilation of APL programs to GPUs.

The benchmark suite is however limited to only a few small programs,

and we will have to obtain more realistic benchmarking programs from our industry partners to prove the practicality of the approach in larger settings.

3.5.7 Identified bottlenecks

While benchmarking we have discovered several shortcomings, that need to be addressed to obtain better performance, and which are especially important if we want to move into parallel computation.

Hoisting memory allocations outside loops when possible, has unsurprisingly shown to give some additional performance, and are especially important to get parallelization speedup. We should therefore also work at general expression hoisting and strength reduction, as memory allocations might depend on other expressions.

We have also observed a few loops, where a loop interchange could improve performance by making memory access patterns sequential. Memory access patterns are even more important for architectures such as GPUs where coalesced memory accesses is of key importance for performance.

3.6 Related Work

This chapter extends on our previous work on compiling APL [43] with a much more complete treatment of the source language, including support for boolean operations (e.g, compress), new data-parallel operations (e.g., scan), iterative computations (i.e., the power operator ⍣), and mutable array updates. The enriched treatment of the language also includes a refined type system for TAIL and a more complete coverage of primitives, both with respect to translation and target language operational semantics. Further, this work extends the previous work by using the idea of hybrid array representation from McDonell et al. [78], to reduce the amount of index-computations. Also, the previous work did not report any performance numbers of the compilation approach and it did not consider the possibility of allowing end-user APL programmers to use type annotations to assert static properties of defined functions and operators.

Other related work can be divided into a number of areas. Although, traditionally, APL is an interpreted language, there have been many attempts at compiling the language. For instance, Guibas and Wyatt have demonstrated how a subset of APL can be compiled using a delayed representation of arrays [50]. Other attempts include Timothy Budd's APL compiler [22] and the ELI-compiler [34]. One of the most serious attempts at compiling APL is the work on APEX [13], which also contains a backend for targeting SAC [47]. APEX has been reported both to provide good sequential and parallel performance on multi-processor machines. Our main contribution compared to these APL compilers, is that we have identified a typed array intermediate language, which can be understood in isolation from the gory details of APL, including APL operators and function overloading, identity item resolution,

scalar extension resolution, and more. Moreover, we have demonstrated that compiling APL through such a typed intermediate language can lead to a competitive performance compared to hand-written C code. We also pave the road for generating parallel code automatically by demonstrating the possibility of achieving good parallel performance by adding, although manually, simple OpenMP annotations to the generated C code.

A different area of related work relates to type systems for, and formal semantics of, array-oriented languages. This area of work includes the work by Slepak et al. [92] on giving a type system with static rank polymorphism for expressing APL-like operations. A related pool of work include the work on SAC [48], and in particular QUBE [101], on type systems for array languages. The SAC project seeks to provide a common ground between functional and imperative domains for targeting parallel architectures, including both multi-core architectures [46] and massively data-parallel architectures [51]. SAC uses `with` and `for` loops to express map-reduce style parallelism and sequential computation, respectively. More complex array constructs can be compiled into `with` and `for` loops, as demonstrated, for instance, by the compilation of APL into SAC [47]. The work on QUBE supports dependent types for verifying array related invariants [101].

Another pool of related work on array languages is the work on Futhark [58, 56, 57], which, as TAIL, and in opposition to SAC, holds on to the concept of second order array combinators (named SOACs) in the intermediate representations. The benefit of this approach is to perform critical optimizations, such as fusion, even in cases involving filtering and scans. Such operations are not straightforward constructs for SAC to cope with. Similar to Futhark, we seek to allow programmers to express parallel patterns in programs as high-level functional constructs, with the aim of systematically (and automatically) generating efficient (and even data-dependent) parallel code. Also related to the present work is the work on capturing the essential mathematical algebraic aspects of array programming [52] and list programming [16] for functional parallelization. Another piece of related work is the work on the Fish programming language [65], which uses partial evaluation and program specialization for resolving shape information at compile time.

A scalable technique for targeting parallel architectures in the presence of nested parallelism is to apply Blelloch's flattening transformation [17]. Blelloch's technique has also been applied in the context of compiling NESL [11] to GPU code, but is sometimes incurring a drastic memory overhead. In an attempt at coping with this issue and for processing large data streams, while still making use of all available parallelism, a streaming version of NESL, called SNESL has been developed [75] , which supports a stream datatype for which data can be processed in chunks and for which the cost-model is explicit. For utilizing parallel architectures efficiently, future compilation approaches for TAIL may also involve a variant of the flattening technique.

Finally, a large body of related work includes the work on embedded domain specific languages (DSLs) for programming massively parallel architectures, such as GPUs. This body of work includes the work on Obsidian [36],

from which the idea of push-arrays originates, and the Accelerate library [28], both of which targets GPUs.

3.7 Conclusion and Future Work

We have presented a statically typed intermediate language, used as a target for an APL compiler and demonstrated the ability to compile our selected subset of APL into efficient low-level code.

Our goal with this work has not been a desire to construct a minimal and pure language. We have instead been thriving to find a practical and pragmatic middle way between systems supporting dependent types such as QUBE, and languages without shape information in types. In contrast to systems based on dependent types, TAIL allows full type inference without user involvement.

There are several directions for future work. In particular, our intermediate language does not support nested arrays, although it does have support for expressing nested parallelism. Adding segmented operations and flattening transformations would be a possible future path.

Obtaining larger benchmark programs written by APL experts have been harder than anticipated, and to demonstrate our goals more clearly it would be interesting to obtain such realistic benchmark programs written by financial industry specialists.

Finally, it would be interesting to investigate the possibility for compiling the intermediate language directly into efficient code for multi-core CPUs and many-core GPUs. We have initiated efforts in this direction, by attempting to construct a language for building GPU kernels, that allows fusion. This work is presented in Chapter 5.

We ended the TAIL project in 2015, to pursue alternative strategy for generating GPU code. The HIPERFIT research center already had another ongoing project on GPU compilation, the before mentioned Futhark project, which is in many aspects similar to the TAIL language. To avoid too much internal competition at the research center. We therefore changed course to investigating GPU programming approach with more control of GPU resources and memory hierarchy, than what can be achieved by automatic optimizers. The work on such a resource aware language is presented in Chapter 5, but before we get to that, Chapter 4 will introduce the necessary background on GPU architectures and programming.

Chapter 4

GPU architecture and programming

Historically, hardware manufacturers has been able to increase clock rates and instruction-level parallelism on each new generation of processors, allowing single-threaded code to run faster without modifications. Heat dissipation, power consumption and other limiting factors have put an end to that trend. Clock rates have stalled around 4 Ghz, despite that the number of transistors manufacturers can put on a single chip is increasing at the same rate as previously, following Moore's law. Instead of increasing clock rates, hardware vendors are now applying the growing amount of transistors to increase the number of cores on a single chip.

Hardware vendors are thus exploring other options of increasing performance. At the extreme, thousands of individual processing units are placed on the same die, on devices such as graphics processing units (GPUs). The design of a GPU is vastly different from that of traditional CPUs. Caches and control logic such as branch predictors and data forwarding paths are exchanged for additional compute units (see Figure 4.1). This massively parallel design has become a cost-effective alternative to CPUs in areas such as image processing, machine learning, simulations in natural sciences, bioinformatics and computational finance. Many tasks in these domains can be formulated as data-parallel programs, which can take advantage of the massive degree of parallelism, and are not bound by requirements of low-latency context switching that control logic and caches provides for traditional CPUs.

Taking advantage of the highly parallel GPU hardware requires careful attention to a major underlying issue, namely the performance gap between the computational power of GPUs and the limited bandwidths of their memory systems. The GPU we have used in our experiments, an NVIDIA GTX 780 Ti, has the ability to perform 5,046 GFlop/s (single precision). However, its internal memory system is limited by a bandwidth of 336 GB/s. Data transfer rates with memory external to the GPU is further limited by the PCI Express bus. As GPUs are still in their infancy, their memory systems are complex and intimate knowledge of its different facets are required for writing efficient GPU algorithms. Programmers needs to limit the number of main memory

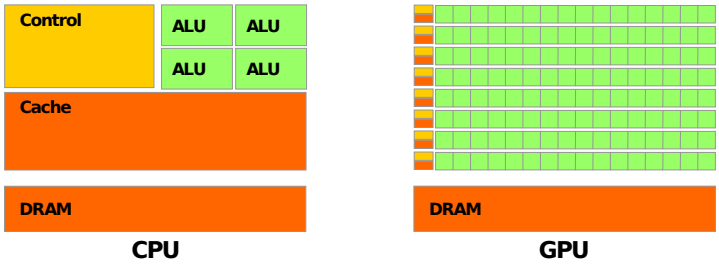

Figure 4.1: Transistor usage CPUs vs GPUs. Figure from NVIDIA CUDA C Programming Guide [82]

transactions through the use of memory efficient algorithms and the limited amount of manually controlled local memory and caches. In addition, one should make sure to schedule enough simultaneous computations to obtain latency hiding on otherwise I/O bound computations.

This chapter will introduce the central terminology on GPU programming, and give an overview of current GPU hardware, GPU programming and discuss optimisation strategies necessary for obtaining efficiency on GPU devices.

4.1 GPU architecture

A typical modern GPU device consists of a few hundred to a couple of thousand individual computational *cores*, all operating in parallel. These cores are clustered into larger units called *streaming multiprocessors* (SMs). As mentioned, the GPU we have used in our experiments is an NVIDIA GeForce GTX 780 Ti, which is based on the Kepler GK110 architecture. The layout of a streaming multiprocessor from the GK110 is shown in Figure 4.2. This particular streaming multiprocessor consists of 192 cores and the complete GK110 GPU consists of fifteen such streaming multiprocessors, with a total of 2880 cores.

GPUs are typically programmed by launching thousands of identical cooperating threads operating on different data elements of the input. The programmer partitions the threads into *cooperative thread arrays* (CTAs), also known as *thread blocks*. Each CTA is assigned to a single SM, and the threads in a CTA can communicate and synchronise through the use the SMs shared memory. Several CTAs can be executed concurrently on the same SM.

The fact that all threads of a CTA are executing the same program, allow the individual cores of an SM to share components such as instruction schedulers, caches and register files. The 192 cores of the GK110 streaming multiprocessor are sharing four instruction schedulers. The instruction scheduler groups adjacent threads of a CTA into *warps* of 32 threads, and dispatches instructions to groups of 32 cores at a time. All 32 cores of such a subgroup always execute exactly the same instructions in lock-step on different threads, *single instruction, multiple thread* (SIMT). Each thread have its own register set and can branch

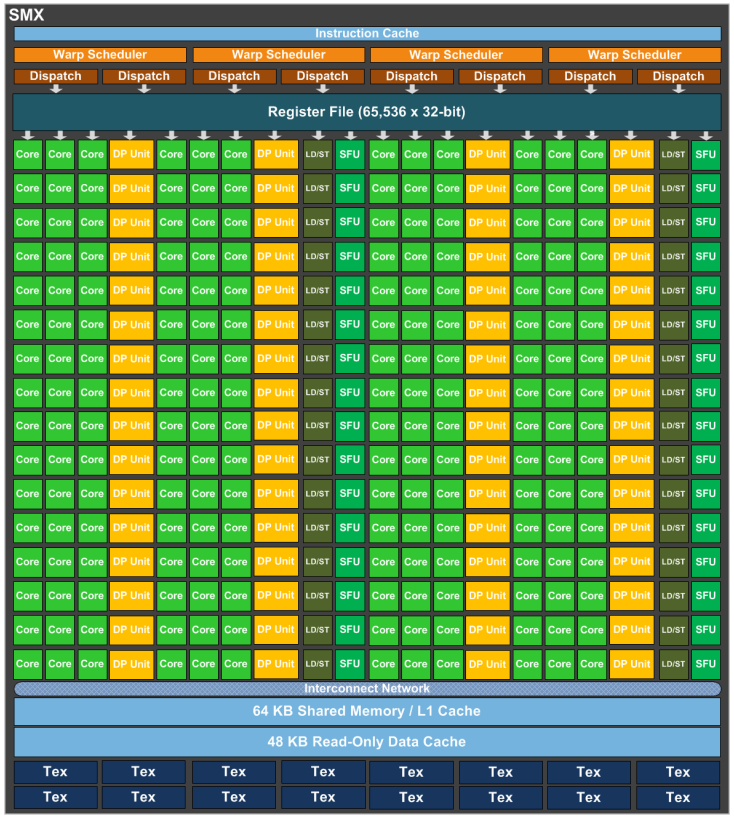

Figure 4.2: Streaming Multiprocessor of the NVIDIA Kepler GK110 GPU. Illustration from NVIDIA Kepler White paper.

independently of the other threads. However, as all threads of a warp execute in lock-step, *control divergence* between threads will be realised by disabling threads that are not taking a given branch. This means that full efficiency is only realised when all threads of a warp follow the same execution path.

Accessing memory on a GPU is typically slow and to obtain latency hiding it is necessary to be aware of the ratio of arithmetic operations to number of memory operations. Increasing arithmetic intensity can either come directly from compute-intensive problems with high-degree of arithmetic operations per thread or be achieved by launching orders of magnitudes more threads than the total number of cores, such that some threads can perform useful work while others wait for memory transactions to complete. A useful measure is the *occupancy* of a streaming multiprocessor. Occupancy is defined as the ratio of active warps to the maximum number of active warps that the SM supports. The maximum number of supported warps is limited by the number of registers and shared memory the program requires per group of threads, and thus depends on the individual program as well as hardware parameters.

4.1.1 Memory hierarchy

The memory system of a GPU is hierarchically structured, with a large main memory (global memory) of several gigabytes, a relatively small L2 cache shared between all streaming multiprocessors (512 KB to 2 MB), as well as various even smaller memory modules in each streaming multiprocessor. A streaming multiprocessor provides its cores with an L1 cache, a read-only texture buffer, a register file and an area of shared memory. The shared memory module makes it possible for the cores inside an SM to communicate and is also useful as a manually controlled read/write cache.

The register file and shared memory bank is the working memory of the threads. Each SM of the GTX 780 Ti provides 256 KB of registers and 64 KB of shared memory. However, but the register file and shared memory have to be shared by all active threads. The number of registers and amount of shared memory used per thread thus limit the number of active threads on the SM, thus reducing occupancy. To obtain good occupancy it is thus beneficial to limit register and shared memory usage.

Reading and writing to global memory is performed in transactions involving entire blocks of successive memory locations. In GPU terms, this is called memory coalescing, as many memory accesses are coalesced into a single transaction. Accessing memory through coalesced transactions is essential for performance, and we will discuss it further in Section 4.3 on optimisation of GPU programs.

The bandwidth of global memory accesses are bound by the width of the memory bus and the memory clock rate. For our NVIDIA Geforce GTX 780 Ti, the bus width is 384 bit, the memory is clocked at 1750 Mhz, and with GDDR5 memory it has double data rate and double data lines. This gives a theoretical bandwidth of:

$$1750 \text{ Mhz} \times 384 \text{ bit} \times 2 \times 2 = 336 \text{ Gbps}$$

The effective bandwidth is however much lower, and must be measured empirically. With NVIDIA's bandwidth tester (`bandwidthTest --cputiming`), we measure an effective bandwidth of 254.9 Gbps.

An even greater bottleneck is present in the connection between GPU and CPU memories. Currently the CPU and the GPU are most often connected through PCI Express bus, which in current versions are limited to a bandwidth around 8-16 GB/s. Reducing data movement over the PCI Express bus is thus even more crucial for performance. Alternative designs that allows CPUs and GPUs to share the same memory system have been developed to mitigate this.

4.2 GPU programming

General purpose computing on graphics processing units (GPGPU) is the technique of using GPUs (graphical processing units) for applications not necessarily

involving computer graphics[1]. There are two widely used GPU programming platforms, namely OpenCL and CUDA. Each platform uses its own terminology. We will not try to cover the differences, but stick mostly to the terminology of OpenCL.

The execution of a GPU program has to be conducted by an accompanied program executing on a CPU. The CPU is referred to as the *host* and the GPU is called the *device*. Other concepts such as pointers or arrays are often prefixed with the location (GPU or CPU) to specify where they reside. A *host pointer* is thus a pointer pointing to *host memory* and a *device-side array* is an array located in *device memory*. Observe, that OpenCL is not limited to interfacing with GPUs, so the device and the host might be the same hardware unit.

The smallest unit of work on a GPU is a single *thread* of execution on a core. However, to limit the number of scheduling units on the SMs, these cores are grouped into *warps* (or *wavefronts*) of 16 or 32 cores executing the same instructions in lock-step. All the threads are further arranged into equal-sized groups called *work-groups* (or *thread blocks*), which in turn are arranged into a *grid* of work-groups. Each *work-group* is executed on a single SM, which provides shared memory accessible from all cores of that SM.

Programming a GPU is done by writing *kernel programs* or simply *kernels*. A kernel specifies the work done by a single thread of the complete problem. Synchronisation across threads is not necessary within warps, as they always execute in lockstep. Synchronisation across threads in individual work-groups is accomplished by *barrier*-functions. These calls must not occur inside conditionals and shall thus always synchronise all threads inside the work-group. Synchronisation between work-groups is possible only by splitting work into several kernels.

Because synchronisation is not possible across work-groups, the common structure of all GPU algorithms is to *a)* partition the input data between work-groups, *b)* let each work-group process its designated part of the input, *c)* let each work-group distribute its output to distinct sections of global memory. In addition to writing such kernel programs, care has to be taken when scheduling kernel calls and data-movement between host and device.

The host program, conducting which GPU kernels to launch, is also responsible for managing data movement between host and device memory in between kernel invocations. However, newer generations of GPUs (which we have not had available for this project) allow a unified virtual address space, such that data movement can be performed on as-need basis, when a kernel requests specific portions of an array, instead of being explicitly managed by the host program.

[1]Some GPUs are built only for computation, and confusingly, sometimes the term "GPGPU" also refer to these headless GPUs.

4.3 Optimisation of GPU programs

Efficient use of a GPU first of all requires attention to the memory bottlenecks described earlier, and various optimisations can be used to address this bottleneck. Optimizing memory usage either reduce the amount of transfers necessary, or perform latency hiding that allow useful computations to run while the waiting for memory transactions to complete.

In addition, another class of optimization addresses the SIMT nature of GPUs, which introduces the problem of *control divergence*, where multiple threads sharing the same instruction schedulers are allowed to diverge in their control flow.

Finally, standard optimisation such as strength reduction, loop unrolling, constant folding, and so on also have an impact when trying to obtain optimal performance of GPU programs.

4.3.1 Memory throughput

Most significant are performance issues related to global-memory accesses. Accessing global-memory on current generations of NVIDIA GPUs incurs a 200-400 clock cycle stall [82], and can thus quickly overshadow any other algorithmic improvement.

The number of global-memory operations can be decreased by various means. We have already discussed how fusion techniques can lead to fewer memory transactions (See Section 2.3). Other possibilities include the use of shared-memory for intermediate storage; when the same data elements needs processing more than once, or optimizing data access patterns that allows coalescing of many memory accesses into a single transaction.

Blocking

Blocking [72], or tiling, is a technique for improving locality of reference, through subdividing a problem in memory blocks, or tiles, that fit into the cache. On GPUs shared memory can be used as a cache, and blocking can be achieved by loading data into shared-memory, before performing the actual computation.

Memory coalescing

Several memory transactions can be coalesced into a single transaction, if threads in the same warp access the same segment simultaneously. On NVIDIA GPUs, if 16 threads (a half-warp) access 16 words in the same aligned segment of 64 or 128 bytes, this access will result in only a single memory-transaction, regardless of the access pattern followed by the threads within that memory segment [82]. It is thus important to organise computations such that threads sharing the same warp will take the same branches and issue memory operations inside the same memory segments.

If input data is needed in a different order, than what can be achieved by doing a coalesced traversal of memory, it can often be wise to load the data into shared-memory in a coalesced fashion, before traversing it in the needed order. This pattern is illustrated in Section 5.2, in an implementation of a transpose operation.

Shared memory bank conflicts

Although shared-memory can be used to optimise global memory accesses, shared-memory itself presents us with additional problems. The shared memory system used in NVIDIA GPUs is partitioned into memory-banks, each accessible with a bandwidth of 32 bits per clock cycle [82]. Two simultaneous loads to the same bank results in a conflict, stalling the computation. Special care is thus also necessary when using shared-memory.

4.3.2 Utilisation

Utilising a GPU efficiently does not only require optimising memory transactions, it also requires optimised utilisation of the GPU's compute units (streaming multiprocessors).

The various reasons why a GPU might not be fully utilised ranges from: optimising algorithms for less inter-thread synchronisation, reducing resource requirements (such as number of registers used), and increasing the parallelism degree to be able to perform latency hiding.

Avoiding synchronisation

Threads are able to communicate at all levels of the hierarchy of the GPU, but with vastly different cost. Synchronisation between work groups are only possible through separating programs into individual kernels. This can in effect also require costly memory transactions to global memory, and should thus be avoided if possible. Mapping parallel algorithms such that synchronisation is mostly required between threads in the same work group is thus important to maximise utilisation.

Similarly, limiting inter-work group syncronisation can also be beneficial. This is for instance possible if threads that need to communicate are allocated to the same warp.

Latency hiding

Latency hiding is achieved when enough arithmetical work is available to keep the processor active while waiting for memory transactions to complete. Increasing the amount of arithmetic work per thread is not always possible, and is very dependent on the algorithm being implemented. When threads in a warp are waiting for I/O operations to complete, streaming multiprocessor can execute warps from the same or other work-groups.

The number of work-groups and warps residing on a streaming multiprocessor depends on the resource requirements of the individual kernel, for instance the amount of registers and shared memory required by the kernel. To increase the number of resident work-groups and warps it is thus necessary to decrease pressure on registers and shared memory. Choosing a smaller work-group size might, for instance, reduce the amount of shared memory required. However, smaller work-group sizes will often also increase the shared memory requirements.

4.3.3 Thread divergence

Another challenge posed by the SIMT architecture of GPU streaming multiprocessors, is the use of a shared instruction scheduler for warps of threads. When threads inside a warp deviate on their control flow paths, each separate path must be executed in sequence, with the threads not following a given path disabled. If data-dependent control flow is necessary in a given application, it is thus necessary to optimise the grouping of threads into warps, such that thread divergence inside warps is minimised. This can for instance be achieved by bucketing similar values in a separate pass.

Chapter 5

FCL: Hierarchical data-parallel GPU programming

Extended version of "Low-Level Functional GPU Programming" presented at FHPC'16

In recent years, several languages for general purpose, data-parallel computation on GPUs have been suggested [29, 59, 96, 27, 62, 11]. Most of these language developments have focused on providing users with high-level specifications of programs and performing a range of automatic optimizations. Often no cost-model is specified, and the language is thus a black box for users who want to reason about the performance of their programs. Parallel algorithms researchers are sidelined, as it is hard to reason about the actual efficiency and performance characteristics of algorithms. The user is decoupled from the hardware model, and cannot be sure whether an operation will result in a memory transaction or not, making unexpected performance hits hard to debug. Also, some algorithms require memory access patterns not supported by the prevalent set of primitives, or depend critically on hardware parameters that these languages do not expose [25].

A body of work exists on the topic of modelling I/O usage in sequential [66, 4] and parallel algorithms [3, 5, 103, 104], including modelling of hierarhical memory systems, such as GPUs. However, the work in this area is mostly concerned with machine models useful for analyzing algorithms, and does not specify a programming model for those machine models.

In the GPU niche of data-parallel languages, Obsidian is an exception [96], allowing for playfulness and invention on the low-level giving you (almost) complete control over the GPU, and still allowing computations to be composed efficiently using so called *pull arrays* and *push arrays*. These arrays are not directly stored in a region of memory, but are rather representations of *array-computations*. This means that most array operations are cheap: they do not incur the overhead of writing a modified array to memory. Instead, they

modify the underlying symbolic array-computation directly. Furthermore, the memory hierarchy of the GPU is exposed in the type system, allowing users to reason about how programs are mapped to hardware.

Obsidian uses a multi-staged compilation approach, that allows users to use Haskell as a meta-language generating Obsidian expressions, which can for instance be used to generate all the statements of an unrolled loop, or to precompute certain values already at code-generation time. However, this approach also has a few drawbacks. Obsidian does not provide any general loop constructs, kernels are compiled as individial programs, and no cross-kernel optimizations can thus be performed.

We present FCL, a reimplementation of Obsidian with an external syntax implemented in Haskell2010 as a self-contained compiler[1]. With FCL, we extend on the work on Obsidian; eliminating the need of using meta-programming techniques in program development, and introducing new operators and language constructs to maintain the same expressive power. The embedded nature of Obsidian also had its drawbacks, especially if used as an intermediate language, which is another reason this project came to be.

In both Obsidian and FCL, computations are polymorphic in their mapping to executions on the GPU hardware, by the use of *level*-annotations in array types. We have developed a dynamic operational semantics for FCL that details the computational model and makes it clear how the different *levels* map to various iteration schemes on the GPU.

The rest of the chapter is structured as follows. Section 5.1 explains *pull* and *push* arrays. In Section 5.2, we introduce FCL through three example programs: array reversal, matrix transpose, and parallel reduction. Section 5.5, we demonstrate that FCL is able to generate efficient OpenCL-code. In Section 5.3, we do a rigorous introduction to FCL, defining its type system and dynamic semantics. With language fully presented we present some further example programs in Section 5.4 and benchmark results in 5.5. Finally, we conclude with a discussion of future work in Section 5.7 and 5.8.

5.1 Obsidian

FCL inherits pull and push arrays from Obsidian [35]. As mentioned previously, these are not actual arrays manifested in memory, but are instead delayed array computations that describe how to produce an array. When the result of a pull or push array computation is written to memory, we say that the array has been *materialized*.

The two types of arrays complement each other; pull arrays allow array indexing, but array concatenation is inefficient. Push arrays on the other hand allow for efficient concatenation, but disallow array indexing.

In Obsidian, iteration schemes on push arrays are annotated in the array types, by a *level*-parameter. The level-parameter can be either `Grid`, `Block`, or `Thread`, corresponding to the hierarchy of organization for GPU threads, and

[1] FCL is available at `http://github.com/dybber/fcl`

annotates the sequential/parallel structure of the underlying iteration scheme. How levels are used will be explained in the context of FCL in the next section.

The main advantage of push arrays compared with pull arrays, is that they allow for efficient implementation of functions that combine arrays, for example array append and various interleavings. Combining pull arrays typically lead to conditionals evaluated at each index of the array. The performance hit of these conditionals can be severe, in particular in the cases when these conditionals lead to threads diverging within a warp. Interleaving two pull arrays is particularly bad as it means that each pair of consecutive threads take different paths through the conditional, wasting half of the resources within each warp in use by this interleaving.

Append and interleave of two push arrays can be achieved by generating two separate loop structures and offsetting the writer function.

In Obsidian, the programmer can access the underlying array representation based on index-functions and writer-functions of both pull and push arrays. In FCL, we keep the concepts of pull and push arrays, but abstract away from their actual representation, as will be illustrated in the rest of the chapter.

5.2 Case Studies in FCL

In this section we will demonstrate the use of FCL by implementing three different GPU algorithms: array reversal, array transposition using shared memory, and parallel reduction.

5.2.1 Array Reversal

Consider a program that reverses an array:

```
sig reverse : forall 'a. ['a] -> ['a]
fun reverse arr =
  let n = length arr
  in generate n (fn i => index arr (n - i - 1))
```

This program is implemented using the function `generate`, a language primitive that creates a new array by mapping the given function over the index-space $[0; n-1]$. The program here *cannot* be compiled directly to GPU code, as it does not mention how it should be mapped to sequential or parallel loops. The arrays in this example are *pull* arrays, which are identified by types of the form `['a]`, where $'a$ is a type variable, representing an arbitrary non-function type. To compile an FCL program into a kernel, we require the user to add an iteration scheme, detailing how this kernel should be mapped to the threads of the GPU. Such iteration schemes are annotated by a *level*, which can be either *thread* (sequential execution), *block*, or *grid*. The iteration scheme is added using a function called `push`. Let us demonstrate, the basic functionality by creating a block-level version of `reverse`, that is executed in parallel by a work-group of threads.

```
sig revBlock : forall 'a. ['a] -> Program<block> ['a]<block>
fun revBlock arr = return<block> (push<block> (reverse arr))
```

Notice how the iteration scheme is reflected in the array type, `['a]<block>`. This is a *push* array (using Obsidian terms), and is contructed using the `push` function, which, given a *level parameter*, associates an iteration scheme with the given pull array:

```
push : forall <level> 'a. ['a] -> ['a]<level>
```

The *push* array is further encapsulated in a `Program`-monad, using the `return` function. The Program monad allow us to restrict where values can be used, such that values computed in one kernel can not slip out, and be reused in another, where the values are no longer available. We will return to this aspect later.

To distribute the computation across several blocks, the input-arrays have to be partitioned and the resulting reversed array-chunks need to be concatenated back together again in the right order. In this case, the order of the chunks also needs to be reversed before concatenation.

```
sig revDistribute : forall 'a. int -> ['a]
                  -> Program <grid> ['a]<grid>
fun revDistribute chunkSize arr =
  splitUp chunkSize arr
   |> map revBlock
   |> reverse
   |> concat<block> chunkSize
```

The operator `|>` is reversed function application (the syntax is taken from F# and Elm), also known as forward-pipe. Notice that the same `reverse` function can be used both to reverse the order of elements and the order of the blocks. The operation `concat` distributes the computation across a grid of blocks, thereby raising the *level* from `block` to `grid`. This is also evident from the type of `concat`, where `1+`*level*, unifies with levels of one level higher in the hierarchy (details are given in Section 5.3).

```
concat : int -> [Program<level> ['a]<level>]
               -> Program<1+level> ['a]<1+level>
```

This means that each subarray is executed in a separate block, and `concat` makes sure that each block writes its result to adjacent subsections of the array it returns. Alternatively we could have applied `push <grid>` directly to the primitive reverse function, to add a grid-level iteration scheme to the array, but that is only possible in simple cases, where there is no dependencies between threads and where we do not need to manipulate the amount of data processed by each block or how results are combined. Neither `splitUp` nor `concat` is a primitive of FCL, and more complicated tiling and interleaving can thus be implemented, as we will see in the following example. `splitUp` can be implemented using two applications of the previously introduced `generate`:

```
sig splitUp : forall 'a. int -> ['a] -> [['a]]
fun splitUp n arr =
  generate ((length arr) / n)
    (fn i => generate n
       (fn j => index arr ((i * n) + j)))
```

5.2.2 Transpose in Shared Memory

The simple `revDistribute` function can also be implemented in Obsidian. Now consider the problem of matrix transposition. In FCL we only have one-dimensional arrays, which means that a two-dimensional matrix must be represented as its flat representation together with information about the number of columns and rows.

If we follow a naive approach, we can transpose a two-dimensional matrix, using the following `transpose` function:

```
sig transpose : forall 'a. int -> int -> ['a] -> ['a]
fun transpose rows cols arr =
  generate (rows * cols)
           (fn n =>
              let i = n / rows
                  j = n % rows
              in index arr (j * rows + i))
```

If this version of `transpose` were to be executed in parallel on the GPU, using the same type of partitioning and concatenation as in the reverse example, it would lead to uncoalesced writes. When a group of GPU threads collectively read or write a section of memory, the memory transactions can be *coalesced* if they all fall into the same segment of memory. In this case, when adding an iteration scheme to the final array, the final writes will always be coalesced in a single transaction, but the indexing into the input array will not, and the reads from the input-array will thus not be able to coalesce and we will incur a huge performance penalty.

A more efficient approach is to chunk up the matrix in smaller two-dimensional tiles, transpose each tile in shared memory, and finally stitch the tiles back together again (in transposed order). By using the shared-memory as intermediate storage, we make sure that both reads and writes to global memory are coalesced, as the threads can first collaborate on moving data to shared memory, and afterwards collaborate on copying data from shared memory to the output-array.

The transpose algorithm is presented in Figure 5.2. Notice the use of "`do`"-notation, for operating in `Program`-monad. The `transpose` algorithm follows roughly the same structure as the `reverse` example. However, instead of splitting the linear input-array into chunks (one following the other), we split and concatenate two-dimensional tiles with the functions `splitGrid` and `concatGrid`. The important thing to notice is that this reading/writing

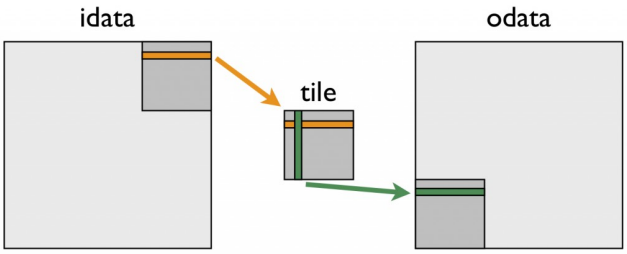

Figure 5.1: Transpose in Shared memory. Figure by NVIDIA.

```
sig transposeBlock : forall 'a. int -> int
                   -> Program<block> ['a]
                   -> Program<block> ['a]<block>
fun transposeBlock rows cols arr =
  do<block>
    { arr' <- arr
    ; return<block> (push<block> (transpose rows cols arr'))
    }

sig transposeTiled : forall 'a. int -> int -> int -> ['a]
                   -> Program <grid> ['a]<grid>
fun transposeTiled tileDim rows cols elems =
    splitGrid tileDim cols rows elems
     |> map push<block>
     |> map force<block>
     |> map (transposeBlock tileDim tileDim)
     |> transpose (rows / tileDim) (cols / tileDim)
     |> concatGrid<block> tileDim cols
```

Figure 5.2: Matrix transposition in FCL using shared-memory to obtain memory coalescing.

order is encapsulated in `splitgrid` and `concatgrid`, and a library of such operations can be provided to users.

Also, we apply the function `force` which *executes* an iteration scheme, writing the array to shared memory, which converts the push array into a pull array, allowing the array to be indexed arbitrarily:

```
force : [a]<lvl> -> Program<lvl> [a]
```

The result is a single kernel performing the transposition with all steps fused, performing just as well as the standard OpenCL implementation. For the sake of simplicity, the kernel in the form presented here, works only for matrices that can be evenly divided by `tileDim`. To use this method for other matrices, a reshape operation increasing its size can be performed and, the

surplus columns and rows, can afterwards be removed using `drop`. We expect that these operations will be able to fuse, such that no additional reads/writes are necessary.

5.2.3 Parallel Reduction

To implement a parallel reduction in FCL, we will perform a tree-reduction inside each work-group; this subcomputation is implemented by splitting the subarray in two, and performing an element-wise sum of the two halves. The implementation is similar to what has previously been shown in Obsidian [96].

The FCL prelude provides the following functions for splitting arrays in two parts and joining arrays element-wise.

```
sig zipWith : forall 'a 'b 'c. ('a -> 'b -> 'c)
            -> ['a] -> ['b] -> ['c]
fun zipWith f a1 a2 =
  generate (min (length a1) (length a2))
    (fn ix => f (index a1 ix) (index a2 ix))

sig halve : forall 'a. ['a] -> (['a], ['a])
fun halve arr =
  let half = (length arr) / 2
  in splitAt half arr
```

The tuple returned by `halve` is merely a syntactic construction, only the tuple element will be present in the OpenCL kernel code. Using these we can now write a function for taking one reduction-step:

```
sig step : forall <lvl> 'a. ('a -> 'a -> 'a) -> ['a]
         -> Program <lvl> ['a]<lvl>
fun step<lvl> arr =
  let x = halve arr
  in return<lvl> (push<lvl> (zipWith f (fst x) (snd x)))
```

Notice that the function is polymorphic in the level-variable lvl. This strategy makes it possible to postpone the decision of whether `step` will run sequentially or at one of the parallel levels of the hierarchy.

In Obsidian, we would have implemented this functionality as a recursive function on the meta-level. Recursion on the meta-level is possible in Obsidian, as the function is working on just a chunk of the array and we would statically know the chunk size. The meta-level recursion in Obsidian would generate an unrolled loop.

In FCL we instead provide a built-in looping-construct, `while`, which accepts a *stop-condition* and a *stepping* function as arguments as well as the initial array.

```
sig reduce : forall <lvl> 'a. ('a -> 'a -> 'a) -> ['a]
               -> Program<lvl> ['a]<lvl>
fun reduce<lvl> f arr =
  do <lvl>
   { a <- while<lvl> (fn a => 1 != length a)
                     (step<lvl> f)
                     (step<lvl> f arr)
   ; return<lvl> (push<lvl> a)
   }
```

The above program will generate a while-loop in the OpenCL kernel, and automatically force values to shared memory between operations as well as performing synchronization between threads in the work-group. In cases where the chunk size is known at compile time, we can use loop unrolling techniques to achieve the same code as if we had used Obsidian.

The `while`-construct assumes that arrays never need to grow during evaluation and thus reuses the same area of shared memory on each iteration. Also, `while` will always materialize the input array to shared memory before starting the iteration. To avoid doing a direct copy from global memory to shared memory, in the reduction kernel, we take one initial step before starting the while-loop, an optimization called "First add during load" by Mark Harris [53].

To perform the reduction using multiple work-groups, we need to split a larger array and concatenate the intermediate sums:

```
sig reducePart : forall 'a. ('a -> 'a -> 'a) -> ['a]
               -> Program<grid> ['a]<grid>
fun reducePart f arr =
 let chunkSize = 2 * #BlockSize
 in splitUp chunkSize  arr
     |> map (reduce <block> f)
     |> concat 1
```

Here `#BlockSize` will refer to either OpenCL's `get_local_size(0)` or a constant specified by the user as a configuration option at compilation time.

Another difference from Obsidian also comes to light here; as we no longer distinguish between statically known values and dynamically known values, we are not be able to infer that `reduce <block> f` always returns a single scalar. We solve this by requiring an extra argument to `concat`, an expression computing the size of each chunk to concatenate. The size is necessary as a parameter, as memory allocations must be performed before executing a GPU kernel.

In addition, static sizes are required by Obsidian to determine the sizes of shared memory allocations, and allow memory reuse, when two arrays are not live simultaneously. In FCL, two uses of shared memory will never overlap, even though they are not live at the same time. An alternative allocation algorithm is left as future work. In many cases we should be able to determine

these sizes, as long as block-sizes are declared, not computed, and in these cases we should be able to improve memory footprint.

The above `reduceGrid` function will result in a kernel that reduces chunks of the array in each of their work-group, to a single value. These values are concatenated, but to do the full reduction the reduce kernel has to be invoked again. We perform a similar while loop, as the one in the original `reduce` function, here we are instead iterating the `grid`-level function `reducePart`:

```
sig reduceFull : forall 'a. ('a -> 'a -> 'a) -> ['a]
                -> Program <grid> 'a
fun reduceFull f arr =
  do<grid>
   { a <- while<grid>
             (fn arr => 1 != length arr)
             (reducePart f)
             (reducePart f arr)
   ; return<grid> (index a 0)
   }
```

In the next section, we will introduce FCL more formally, before we present some larger examples in Section 5.4.

5.3 Formalisation

To better understand the limitations and performance of programs written in FCL, and to validate correctness, we will now turn to a more formal treatment of the language.

We use i, d, b, to range over *integers*, *doubles*, and *booleans*, respectively. Let α range over an infinite set of type variables, let δ range over an infinite set of *level variables*, and let x range over program variables.

Whenever z is some object, we write $\vec{z}$ to range over sequences of similar objects. When we want to be explicit about the size of a sequence $\vec{z} = z_0, \cdots, z_{(n-1)}$, we often write it on the form $\vec{z}^{(n)}$.

The core syntax of FCL is defined as follows:

$$
\begin{aligned}
bv &::= i \mid d \mid b && \text{(scalars)}\\
l &::= \delta \mid Z \mid 1 + l && \text{(levels)}\\
e &::= bv \mid x \texttt{<}\vec{l}\texttt{>} \mid (e_1, e_2) \mid () && \text{(expressions)}\\
&\quad \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e_1\ e_2\\
&\quad \mid \texttt{let}\ x\ \texttt{<}\vec{\delta}\texttt{>}\ =\ e_1\ \texttt{in}\ e_2\\
&\quad \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3
\end{aligned}
$$

The notation `()` is a unit value.

The set of built-in operations is not visible from this syntax presentation, and will be presented in the next sections. It is through the built-in operations that the specifics of the language really becomes clear.

Notice that a variable x is parameterised over a number of *level parameters*, $\vec{\delta}$, with the notation x <$\vec{\delta}$>, where δ is a *level variable*. Each use of a variable require specifying concrete values for all level arguments, for instance x `thread`. For variables that does not require any level arguments we will use the shorthand x for x <>.

We often use the following short-hands for the first three levels:

$$\begin{aligned} \texttt{thread} &= Z \\ \texttt{block} &= 1 + Z \\ \texttt{grid} &= 1 + (1 + Z) \end{aligned}$$

Unlike Obsidian, we do not support `warp`-level computations, which reduces the complexity of the implementation. It is possible to reintroduce `warp`-level into the model, and doing so will likely be beneficial, performance wise, for several applications.

5.3.1 Type System

The syntax of FCL types, kinds and type-schemes is defined as follows:

$$\begin{aligned} bt &::= \alpha \mid \texttt{int} \mid \texttt{double} \mid \texttt{bool} && \text{(base types)} \\ \tau &::= \alpha \mid bt \mid (\tau_1, \tau_2) \mid \tau_1 \to \tau_2 \mid \texttt{unit} && \text{(types)} \\ &\mid [\tau] && \text{(pull arrays)} \\ &\mid [bt]\texttt{<}l\texttt{>} && \text{(push arrays)} \\ &\mid \texttt{Program<}l\texttt{>}\ \tau && \text{(program monad)} \\ \kappa &::= \texttt{BT} \mid \texttt{GT} \mid \texttt{TYP} && \text{(kinds)} \\ \sigma &::= \forall\texttt{<}\vec{\delta}\texttt{>}\ \overrightarrow{\alpha : \kappa}.\ \tau && \text{(type-schemes)} \end{aligned}$$

As previously mentioned, the type of pull arrays with elements of type τ are written $[\tau]$. Push arrays can only contain values of base types (bt). The type of push arrays containing values of type bt is written $[bt]$<l>, where l denotes the level in the GPU hierarchy where it can be written if materialized. To encapsulate materialized arrays, the type system includes a hierarchy of monads `Program`<l> τ, with l denoting where on the GPU hierarchy the monadic actions are executed.

To define the set of valid types (under assumptions for free variables), we define a relation $\Delta \vdash \tau$ below, where Δ are kind environments, mapping type variables to kinds:

$$\Delta ::= \ \alpha : \kappa, \Delta \mid \epsilon$$

The kind-system divides types into three categories. Base types (`BT`), ground types (`GT`), and general types (`TYP`). Base types are types of scalar values, which are the only types of values allowed in push arrays. Ground types are all types except function-types, and are the types allowed in pull arrays.

In practice, we often leave out kinds, when they are obvious from the context.

Kind system $\boxed{\Delta \vdash \tau : \kappa}$

$$\frac{}{\Delta \vdash \texttt{int} : \texttt{BT}}\ (5.1) \qquad \frac{}{\Delta \vdash \texttt{double} : \texttt{BT}}\ (5.2) \qquad \frac{}{\Delta \vdash \texttt{bool} : \texttt{BT}}\ (5.3)$$

$$\frac{}{\Delta \vdash \texttt{unit} : \texttt{GT}}\ (5.4) \qquad \frac{\Delta(\alpha) = \kappa}{\Delta \vdash \alpha : \kappa}\ (5.5)$$

$$\frac{\Delta \vdash \tau_i : \kappa}{\Delta \vdash (\tau_1, \tau_2) : \kappa}\ (5.6) \qquad \frac{\Delta \vdash \tau : \texttt{BT}}{\Delta \vdash \tau : \texttt{GT}}\ (5.7) \qquad \frac{\Delta \vdash \tau : \texttt{GT}}{\Delta \vdash \tau : \texttt{TYP}}\ (5.8)$$

$$\frac{\Delta \vdash \tau : \texttt{TYP} \quad \Delta \vdash \tau' : \texttt{TYP}}{\Delta \vdash \tau \to \tau' : \texttt{TYP}}\ (5.9)$$

$$\frac{\Delta \vdash \tau : \texttt{GT}}{\Delta \vdash [\tau] : \texttt{GT}}\ (5.10) \qquad \frac{\Delta \vdash \tau : \texttt{BT}}{\Delta \vdash [\tau]\texttt{<}l\texttt{>} : \texttt{GT}}\ (5.11) \qquad \frac{\Delta \vdash \tau : \texttt{BT}}{\Delta \vdash \texttt{Program<}l\texttt{>}\ \tau : \texttt{GT}}\ (5.12)$$

A type environment Γ is a set of type assumptions of the form $x : \sigma$, mapping program variables to type-schemes:

$$\Gamma ::= x : \sigma, \Gamma \mid \epsilon$$

The types of built-in array combinators are shown in Figure 5.4. In addition a set of built-in scalar operations are defined in Figure 5.5. An initial typing environment Γ_{init} is formed by adding a binding $op : \text{TySc}(op)$ for each op.

The operation `interleave` is used to combine array operations at one level into programs at the level above. It is through `interleave` that operations such as `concat` and `concatGrid` used in the previous examples, can be implemented. The details of `interleave` will be explained when the semantics is described in Section 5.3.3. The operations `power`, `poweri`, and `while` allow iteration of array programs, in the same style as the power operator in APL.

To support sequential loops on thread-level, the operation `seqfor` is provided. The function `seqfor` constructs an array through repeatedly executing a function that returns index-value pairs, corresponding to the assignment that should be done in that iteration. In Figure 5.3 we shown how `seqfor` might be used to implement standard sequential algorithms such as folds and scans on arrays.

A substitution S maps type variables to types and level variables to levels. The result of applying a substitution S to an object X, written $S(X)$, is first to extend the substitution to be the identity outside its domain and then simultaneously substitute each type variable α in X with $S(\alpha)$ and each level variable l in $S(X)$ with $S(l)$, after appropriately renaming bound variables in X. The *support* of a substitution S, written Supp S, is the set of elements v for which $S(v) \neq v$.

```
sig foldl : forall 'a 'b. ('a -> 'b -> 'a) -> 'a -> ['b]
          -> ['a]<thread>
fun foldl f b array =
  seqfor 1 (1 + (length array))
    (fn read => fn i =>
       (0, if i == 0
           then b
           else f (read 0) (index array (i - 1))))

sig scanl : forall 'a 'b. ('a -> 'b -> 'a) -> 'a -> ['b]
          -> ['a]<thread>
fun scanl op b array =
 let n = length array
 in seqfor (1 + n) (1 + n)
      (fn read => fn i =>
          (i, if i == 0
              then b
              else op (read (i - 1)) (index array (i - 1))))
```

Figure 5.3: Sequential left fold and left scan, using `seqfor`.

A type scheme $\sigma = \forall\vec{l}.\,\vec{\alpha}.\tau$ *generalises* a type τ' via a substitution S, written "$\sigma \succ \tau'$ via S", iff Supp $S \subseteq \{\vec{\alpha}, \vec{l}\}$ and $S(\tau) = \tau'$.

The type system allows inferences among sentences of the form $\Delta, \Gamma \vdash e : \tau$, which are read: "under the assumptions Δ, Γ the expression e has type τ". The typing rules are shown in Figure 5.6.

The typing rules are mostly standard. The only interesting rules are the rules for let-bindings and the rule for variable instantiation, which are extended to support level parameters.

op		TySc(*op*)
`fst`	:	$\forall\alpha,\beta.\ (\alpha,\beta) \to \alpha$
`snd`	:	$\forall\alpha,\beta.\ (\alpha,\beta) \to \beta$
`length`	:	$\forall\alpha.[\alpha] \to$ `int`
`lengthPush`	:	$\forall$<δ> $\alpha.\ [\alpha]$<δ> $\to$ `int`
`map`	:	$\forall\alpha,\beta.\ (\alpha \to \beta) \to [\alpha] \to [\beta]$
`mapPush`	:	$\forall$<δ> $\alpha,\beta.\ ($`int` $\to \alpha \to \beta) \to [\alpha]$<$\delta$> $\to [\beta]$<δ>
`generate`	:	$\forall\alpha.$ `int` $\to ($`int` $\to \alpha) \to [\alpha]$
`index`	:	$\forall\alpha.\ [\alpha] \to$ `int` $\to \alpha$
`push`	:	$\forall$<δ> $\alpha.\ [\alpha] \to [\alpha]$<$\delta$>
`force`	:	$\forall$<δ> $\alpha.\ [\alpha]$<δ> $\to$ `Program`<δ> $[\alpha]$
`return`	:	$\forall$<δ> $\alpha.\ \alpha \to$ `Program`<δ> α
`bind`	:	$\forall$<δ> $\alpha,\beta.$ `Program`<δ> $\alpha \to (\alpha \to$ `Program`<δ> $\beta) \to$ `Program`<δ> β
`interleave`	:	$\forall$<δ> $\alpha.$ `int` $\to (($`int`$,$`int`$) \to$ `int`$) \to [$`Program`<δ> $[\alpha]$<δ>$] \to$ `Program`<$1+l$> $[\alpha]$<$1+\delta$>
`interleaveSeq`	:	$\forall$<δ> $\alpha.$ `int` $\to (($`int`$,$`int`$) \to$ `int`$) \to [$`Program`<δ> $[\alpha]$<δ>$] \to$ `Program`<δ> $[\alpha]$<δ>

op		TySc(*op*)
`power`	:	$\forall$<δ> $\alpha.$ `int` $\to ([\alpha] \to$ `Program`<δ> $[\alpha]$<δ>$) \to$ `Program`<δ> $[\alpha]$<δ> $\to$ `Program`<δ> $[\alpha]$
`poweri`	:	$\forall$<δ> $\alpha.$ `int` $\to ($`int` $\to [\alpha] \to$ `Program`<δ> $[\alpha]$<δ>$) \to$ `Program`<δ> $[\alpha]$<δ> $\to$ `Program`<δ> $[\alpha]$
`while`	:	$\forall$<δ> $\alpha.\ ([\alpha] \to$ `bool`$) \to ([\alpha] \to$ `Program`<δ> $[\alpha]$<δ>$) \to$ `Program`<δ> $[\alpha]$<δ> $\to$ `Program`<δ> $[\alpha]$
`seqfor`	:	$\forall\alpha.\ [\alpha]$<`thread`> $\to$ `int` $\to (($`int` $\to \alpha) \to$ `int` $\to ($`int`$,\alpha)) \to [\alpha]$<`thread`>

Figure 5.4: Built-in operators with type schemes.

`+` : `int` $\to$ `int` $\to$ `int`
`-` : `int` $\to$ `int` $\to$ `int`
`*` : `int` $\to$ `int` $\to$ `int`
`/` : `int` $\to$ `int` $\to$ `int`
`%` : `int` $\to$ `int` $\to$ `int`
...

Figure 5.5: Built-in scalar operators.

Expression typing $\boxed{\Delta, \Gamma \vdash e : \tau}$

$$\frac{}{\Delta, \Gamma \vdash i : \texttt{int}} \ (5.13) \qquad \frac{}{\Delta, \Gamma \vdash d : \texttt{double}} \ (5.14) \qquad \frac{}{\Delta, \Gamma \vdash b : \texttt{bool}} \ (5.15)$$

$$\frac{}{\Delta, \Gamma \vdash () : \texttt{unit}} \ (5.16) \qquad \frac{\Delta, \Gamma \vdash e_1 : \tau_1 \quad \Delta, \Gamma \vdash e_1 : \tau_2}{\Delta, \Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)} \ (5.17)$$

$$\frac{\Delta, (\Gamma, x : \tau') \vdash e : \tau}{\Delta, \Gamma \vdash \texttt{fn}\ x \texttt{=>}\ e : \tau' \to \tau} \ (5.18) \qquad \frac{\Delta, \Gamma \vdash e_1 : \tau' \to \tau \quad \Delta, \Gamma \vdash e_2 : \tau'}{\Delta, \Gamma \vdash e_1\ e_2 : \tau} \ (5.19)$$

$$\frac{\Delta, \Gamma \vdash e_1 : \texttt{bool} \quad \Delta, \Gamma \vdash e_2 : \tau \quad \Delta, \Gamma \vdash e_3 : \tau \quad \Delta \vdash \tau : \texttt{BT}}{\Delta, \Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau} \ (5.20)$$

$$\frac{\Gamma(x) \succ \tau \text{ via } S \quad S(\vec{\delta}^{(n)}) = \vec{l}^{(n)} \quad \Delta \vdash \tau : \texttt{TYP}}{\Delta, \Gamma \vdash x <\vec{l}^{(n)}> : \tau} \ (5.21)$$

$$\frac{\begin{array}{c}(\Delta, \vec{\alpha}^{(k)} : \vec{\kappa}), \Gamma \vdash e_1 : \tau' \quad \text{ftv}(\Gamma) \cap \{\vec{\alpha}, \vec{\delta}\} = \emptyset \\ \sigma = \forall \vec{\delta}^{(n)}\ \vec{\alpha}^{(k)}.\tau' \quad \Delta, (\Gamma, x : \sigma) \vdash e_2 : \tau\end{array}}{\Delta, \Gamma \vdash \texttt{let} <\vec{\delta}^{(n)}> x = e_1\ \texttt{in}\ e_2 : \tau} \ (5.22)$$

Figure 5.6: Static semantics for FCL

5.3.2 Monomorphization

Before compilation we perform *monomorphization*, which is performed following the rules in Figure 5.7. The only thing different from traditional monomorphization is the use of level variables which are explicitly given by the user at variable usage.

A *monomorphization environment* $\mathcal{E}$ maps program variables to objects of the form (σ, e):

$$\mathcal{E} ::= \ x \mapsto (\sigma, e), \mathcal{E} \mid \epsilon$$

For each predefined operator *op* with type scheme $\sigma = \Gamma_{init}(op)$, we add a binding $op \mapsto (\sigma, op)$ to create the initial monomorphization environment $\mathcal{E}_{init}$.

Following these rules, all `let`-defined variables are inlined. However, our implementation maintains these `let`-bindings, creating a new binding for each different usage of the variable and avoids introducing more copies than there are type instances.

5.3.3 Dynamic Semantics

We now present the semantics of the language, which will aid understand how FCL terms can be compiled and, in particular, how level-types guide the compilation.

The evaluation relation, we define below, is annotated with a location. Locations emulate the hierarchical tree structure of a parallel machine, and are

Monomorphization $\boxed{\mathcal{E} \vdash e \Rightarrow e'}$

$$\frac{}{\mathcal{E} \vdash i \Rightarrow i} \ (5.23) \qquad \frac{}{\mathcal{E} \vdash d \Rightarrow d} \ (5.24) \qquad \frac{}{\mathcal{E} \vdash b \Rightarrow b} \ (5.25)$$

$$\frac{\mathcal{E} \vdash e_1 \Rightarrow e_1' \quad \mathcal{E} \vdash e_2 \Rightarrow e_2'}{\mathcal{E} \vdash (e_1,\ e_2) \Rightarrow (e_1',\ e_2')} \ (5.26)$$

$$\frac{\mathcal{E}, x \mapsto (\tau, x) \vdash e \Rightarrow e'}{\mathcal{E} \vdash \texttt{fn}\ x : \tau . e \Rightarrow \texttt{fn}\ x : \tau . e'} \ (5.27) \qquad \frac{\mathcal{E} \vdash e_1 \Rightarrow e_1' \quad \mathcal{E} \vdash e_2 \Rightarrow e_2'}{\mathcal{E} \vdash e_1\ e_2 \Rightarrow e_1'\ e_2'} \ (5.28)$$

$$\frac{\mathcal{E} \vdash e_1 \Rightarrow e_1' \quad \mathcal{E} \vdash e_2 \Rightarrow e_2' \quad \mathcal{E} \vdash e_3 \Rightarrow e_3'}{\mathcal{E} \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3'} \ (5.29)$$

$$\frac{\begin{array}{c}\mathcal{E}(x) = (\sigma, e) \\ \sigma \geq \tau \text{ via } S \quad S(\vec{\delta}^{(n)}) = \vec{l}^{(n)}\end{array}}{\mathcal{E} \vdash x <\vec{l}^{(n)}> \Rightarrow S(e)} \ (5.30) \qquad \frac{\begin{array}{c}\mathcal{E} \vdash e_1 \Rightarrow e_1' \quad \text{fv}(\mathcal{E}) \cap \{\vec{\alpha}, \vec{\delta}\} = \emptyset \\ \sigma = \forall \vec{\delta}^{(n)} \vec{\alpha}^{(k)} . \tau \\ \mathcal{E}, x \mapsto (\sigma, e_1') \vdash e_2 \Rightarrow e_2'\end{array}}{\mathcal{E} \vdash \texttt{let}\ x : \sigma <\vec{\delta}^{(n)}> = e_1\ \texttt{in}\ e_2 \Rightarrow e_2'} \ (5.31)$$

Figure 5.7: Monomorphization of FCL expressions.

of the form:

$$\begin{array}{rll} loc ::= & \mathsf{Thread}(thread\_id) & thread\_id \in \mathbb{N} \\ | & \mathsf{Group}(\overrightarrow{loc}^{(n)}) & \end{array}$$

Locations relates to levels and we introduce similar shorthands for blocks and grids.

$$\begin{aligned} \mathsf{Block}(\overrightarrow{loc}^{(n)}) &= \mathsf{Group}(loc_0, \ldots, loc_{n-1}) \\ &\qquad \text{where } loc_i = \mathsf{Thread}(thread\_id_i), \text{for some } thread\_id_i \\ \mathsf{Grid}(\overrightarrow{loc}^{(n)}) &= \mathsf{Group}(\mathsf{Block}(loc_0), \ldots, \mathsf{Block}(loc_{n-1})) \end{aligned}$$

We also introduce a relation, $loc \rhd l$, which defines whether the location loc is *respecting* the level l:

$\boxed{loc \rhd l}$

$$\frac{}{\mathsf{Thread}(thread\_id) \rhd Z} \ (5.32) \qquad \frac{loc_i \rhd l \text{ for all } i}{\mathsf{Group}(\overrightarrow{loc}) \rhd 1 + l} \ (5.33)$$

To allow fusion, certain rules of the semantics are allowed to fire during compile time, such as the rules for creating and indexing into pull arrays. To distinguish between these two sets of rules, we annotate the judgment with a location and a symbol 0 or 1 depending on whether the rule may fire at compile time. In the semantics we allow all non-array accessing operations at arbitrary

$\boxed{loc \blacktriangleright \tau}$

$$\frac{}{loc \blacktriangleright bt}\ (5.34) \qquad \frac{}{loc \blacktriangleright ()}\ (5.35)$$

$$\frac{loc \blacktriangleright \tau_1 \quad loc \blacktriangleright \tau_1}{loc \blacktriangleright (\tau_1, \tau_2)}\ (5.36) \qquad \frac{loc \blacktriangleright \tau_1 \quad loc \blacktriangleright \tau_1}{loc \blacktriangleright \tau_1 \to \tau_2}\ (5.37)$$

$$\frac{loc \blacktriangleright \tau}{loc \blacktriangleright [\tau]}\ (5.38) \qquad \frac{}{loc \blacktriangleright [bt]\texttt{<}l\texttt{>}}\ (5.39)$$

$$\frac{loc \triangleright l \quad loc \blacktriangleright \tau}{loc \blacktriangleright \texttt{Program<}l\texttt{>}\ \tau}\ (5.40)$$

Figure 5.8: Relation $loc \blacktriangleright \tau$

levels, which, in particular, is used to model that such operations may occur at the host level, when calculating array sizes.

We define the relation $loc \blacktriangleright \tau$ in Figure 5.8 to denote types τ which can be evaluated at loc.

Values in FCL are either base values (bv), lambda abstractions, pull arrays, push arrays, delayed interleavings of push arrays, delayed sequential loops or a value in a `Program` context.

$$\begin{aligned}
v ::= {} & bv && \text{(base values)}\\
\mid {} & \texttt{fn}\ x\ \texttt{=>}\ e\\
\mid {} & [e_0, \ldots, e_{n-1}] && \text{(pull array)}\\
\mid {} & \texttt{Push <}\delta\texttt{>}\ v && \text{(push array)}\\
\mid {} & \texttt{Interleave}\ v_n\ v_f\ v && \text{(delayed interleave)}\\
\mid {} & \texttt{InterleaveSeq}\ v_n\ v_f\ v && \text{(delayed sequential interleave)}\\
\mid {} & \texttt{SeqFor}\ [e_0, \ldots, e_{n-1}]\ n\ i\ f\ g && \text{(sequential for)}\\
\mid {} & \texttt{Program <}l\texttt{>}\ v
\end{aligned}$$

While pull arrays are traditionally presented as function values (see Chapter 2), we use a representation of arrays of expressions, to be able to use them as a vehicle for modelling various evaluation orders in the small-step semantics.

An array value of type `Program <`l`>` $[bt]$`<`l`>` is essentially modelled as a tree structure, with "pushed" pull arrays at the leaves (`Push` $[\vec{e}]$), interleavings forming the body of the tree, and a `Program <`l`>` v at the root.

The `SeqFor` models a sequential loop constructing an array. The user provides an initial array, which is materialized before repeatedly executing f to update the array. In this way in-place updates are allowed. In practice, we also provide an unsafe version of `seqfor`, where the initial array is not initialized, and it is the user's responsibility to make sure all values are written and not to index into locations that do not yet have a value.

Value typing $\boxed{\Delta, \Gamma \vdash v : \tau}$

$$\frac{\Delta, (\Gamma, x : \tau') \vdash e : \tau}{\Delta, \Gamma \vdash \lambda x.e : \tau' \to \tau} \quad (5.41) \qquad \frac{\Delta, \Gamma \vdash e_i : \tau \quad \Delta \vdash \tau : \texttt{GT}}{\Delta, \Gamma \vdash [e_1, \ldots, e_n] : [\tau]} \quad (5.42)$$

$$\frac{\Delta, \Gamma \vdash v : [\tau] \quad \Delta \vdash \tau : \texttt{BT}}{\Delta, \Gamma \vdash \texttt{Push <}l\texttt{>}\ v : [\tau]\texttt{<}l\texttt{>}} \quad (5.43) \qquad \frac{\Delta, \Gamma \vdash v : \tau}{\Delta, \Gamma \vdash \texttt{Program <}l\texttt{>}\ v : \texttt{Program<}l\texttt{>}\ \tau} \quad (5.44)$$

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash v_n : \texttt{int} \quad \Delta, \Gamma \vdash v_f : \texttt{int} \to \texttt{int} \to \texttt{int} \\ \Delta, \Gamma \vdash v : [\texttt{Program<}l\texttt{>}\ [\tau]\texttt{<}l\texttt{>}]\end{array}}{\Delta, \Gamma \vdash \texttt{Interleave}\ v_n\ v_f\ v : [\tau]\texttt{<}1 + l\texttt{>}} \quad (5.45)$$

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash v_n : \texttt{int} \quad \Delta, \Gamma \vdash v_f : \texttt{int} \to \texttt{int} \to \texttt{int} \\ \Delta, \Gamma \vdash v : [\texttt{Program<}1 + l\texttt{>}\ [\tau]\texttt{<}l\texttt{>}]\end{array}}{\Delta, \Gamma \vdash \texttt{InterleaveSeq}\ v_n\ v_f\ v : [\tau]\texttt{<}l\texttt{>}} \quad (5.46)$$

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash v_i : \tau \quad \Delta, \Gamma \vdash n : \texttt{int} \quad \Delta, \Gamma \vdash i : \texttt{int} \quad \Delta, \Gamma \vdash g : (\tau \to \tau') \\ \Delta, \Gamma \vdash f : (\texttt{int} \to \tau) \to \texttt{int} \to (\texttt{int}, \tau)\end{array}}{\Delta, \Gamma \vdash \texttt{SeqFor}\ [\vec{v}^{(n)}]\ n\ i\ f\ g : [\tau']\texttt{<thread>}} \quad (5.47)$$

Figure 5.9: Typing of values

To support map operations on an array generated with `seqfor`, an extra function (g) is added, which initially is just an identity-function.

We extend the typing relation to include typing of values in Figure 5.9, and extend our notion of expressions (e) to include values (v):

$$e ::= \ldots \mid v$$

We further add two functions `ifP` and `permute` to the language, which are necessary for formalising the language as a small-step semantics; these constructs, however, are not part of the implemented language. Their types are given below. The `ifP` function allows conditionals on programs, which is a necessary construct for unrolling a single step of the iteration constructs. The `permute` function performs a scatter operation similar to the one described in Chapter 2 and which is also found in NESL and Accelerate. We use the `permute` function to simplify our formalisation of `interleave` below; in practice, however, these two operations are implemented as a single step, which avoids the overhead of storing the permutation vector to memory. We introduce `permute` mainly to simplify the formalisation.

op		TySc(*op*)
`ifP`	:	$\forall\texttt{<}\delta\texttt{>}\ \alpha.\ \texttt{bool} \to \texttt{Program<}\delta\texttt{>}\ \alpha \to \texttt{Program<}\delta\texttt{>}\ \alpha \to \texttt{Program<}\delta\texttt{>}\ \alpha$
`permute`	:	$\forall\texttt{<}\delta\texttt{>}\ \alpha.\ [\texttt{int}] \to [[\alpha]\texttt{<}\delta\texttt{>}] \to \texttt{Program<}\delta\texttt{>}\ [\alpha]$

Evaluation contexts

$$
\begin{aligned}
E ::= {} & [\cdot] \mid E\ e \mid v\ E \\
& \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
& \mid (E, e) \mid (v, E) \mid \texttt{fst}\ E \mid \texttt{snd}\ E
\end{aligned}
$$

Figure 5.10: Evaluation contexts for FCL

Evaluation contexts, ranged over by E are defined i Figure 5.10. When E is an evaluation context and e is an expression, we write $E[e]$ to denote the expression resulting from filling the hole in E with e.

The dynamic semantics is defined as a small step reduction semantics, with the judgment form $e \hookrightarrow^{c}_{loc} e'/\mathbf{err}$, where c ranges over costs 0 and 1. Intuitively, a large number of the rules for the operational semantics, namely those that are annotated with a cost of 0, denote "administrative reductions", which appear at compile time and therefore are not directly associated with a runtime cost.

We shall later present a property stating that a well-typed expression e is either a value or it evaluates in one step, with cost 0 or 1 to another expression e' or the special token $\mathbf{err}$, signaling an error. Such errors cover index-out-of-bounds errors when reading from and writting into arrays. Other errors, not considering non-termination, are ruled out by the static type system.

We shall not formally demonstrate a phase-distinction property for the language specifying that the 0-cost rules can be implemented through a compilation into a lower-level imperative language suitable for execution on parallel architectures. However, the implementation for FCL that targets OpenCL, for which benchmarks are given in Section 5.5, provides evidence for such a phase distinction.

The evaluation rules are separated into a number of groups and appear in Figures 5.11–5.14. The first group of rules appear in Figure 5.11. These rules, include contextual rules and rules that implements compile-time function application, compile-time elimination of tuples, and the introduction of pull arrays and associated operations. Only the operation of indexing into a pull-array is considered to have a cost different from 0. The reason is that the operation is considered to perform bounds-checking, which in the worst case will happen at runtime.

Small-step semantics $\boxed{e \hookrightarrow^{c}_{loc} e' / \mathbf{err}}$

$$\frac{e \hookrightarrow^{c}_{loc} e' \quad E \neq [\cdot]}{E[e] \hookrightarrow^{c}_{loc} E[e']} \ (5.48) \qquad \frac{e \hookrightarrow^{c}_{loc} \mathbf{err} \quad E \neq [\cdot]}{E[e] \hookrightarrow^{c}_{loc} \mathbf{err}} \ (5.49)$$

$$\frac{}{(\texttt{fn}\ x \texttt{=>}\ e_1)\ v_2 \hookrightarrow^{0}_{loc} e_1[v_2/x]} \ (5.50)$$

$$\frac{}{\texttt{fst}\ (v_1, v_2) \hookrightarrow^{0}_{loc} v_1} \ (5.51) \qquad \frac{}{\texttt{snd}\ (v_1, v_2) \hookrightarrow^{0}_{loc} v_2} \ (5.52)$$

$$\frac{}{\texttt{map}\ f\ [e_0, e_1, \ldots, e_{n-1}] \hookrightarrow^{0}_{loc} [f\ e_0, f\ e_1, \ldots, f\ e_{n-1}]} \ (5.53)$$

$$\frac{}{\texttt{interleave}\ v_1\ v_2\ v_3 \hookrightarrow^{0}_{loc} \texttt{Interleave}\ v_1\ v_2\ v_3} \ (5.54)$$

$$\frac{}{\texttt{interleaveSeq}\ v_1\ v_2\ v_3 \hookrightarrow^{0}_{loc} \texttt{InterleaveSeq}\ v_1\ v_2\ v_3} \ (5.55)$$

$$\frac{}{\texttt{seqfor}\ (\texttt{Push <thread>}\ [\vec{e}^{(n)}]\ m\ f \hookrightarrow^{0}_{loc} \texttt{SeqFor}\ [\vec{e}^{(n)}]\ m\ 0\ f\ (\texttt{fn}\ x \texttt{=>}\ x)} \ (5.56)$$

$$\frac{}{\texttt{push <}l\texttt{>}\ v \hookrightarrow^{0}_{loc} \texttt{Push <}l\texttt{>}\ v} \ (5.57)$$

$$\frac{}{\texttt{generate}\ n\ f \hookrightarrow^{0}_{loc} [f\ 0, \ldots, f\ (n-1)]} \ (5.58) \qquad \frac{}{\texttt{length}\ [e_0, \ldots, e_{n-1}] \hookrightarrow^{0}_{loc} n} \ (5.59)$$

$$\frac{i \in [0, n)}{\texttt{index}\ i\ [e_0, \ldots, e_{n-1}] \hookrightarrow^{1}_{loc} e_i} \ (5.60) \qquad \frac{i \notin [0, n)}{\texttt{index}\ i\ [e_0, \ldots, e_{n-1}] \hookrightarrow^{1}_{loc} \mathbf{err}} \ (5.61)$$

Figure 5.11: Small-step semantics

The second group of rules are presented in Figure 5.12, and include rules for sequencing monadic operations, rules for determining the length of a push-array, and rules for map-map fusion on push arrays. Each of these rules contribute with 0 runtime cost; in particular, map-map fusion is guaranteed to be performed at compile time.

Small-step semantics (continued) $\boxed{e \hookrightarrow^0_{loc} e' / \mathbf{err}}$

$$\frac{loc \rhd l}{\texttt{return} \texttt{<}l\texttt{>}\ v \hookrightarrow^0_{loc} \texttt{Program} \texttt{<}l\texttt{>}\ v} \quad (5.62)$$

$$\frac{loc \rhd l}{\texttt{bind} \texttt{<}l\texttt{>}\ (\texttt{Program} \texttt{<}l\texttt{>}\ v)\ f \hookrightarrow^0_{loc} f\ v} \quad (5.63)$$

$$\frac{}{\texttt{lengthPush}\ (\texttt{Push} \texttt{<}l\texttt{>}\ [e_0, \ldots, e_{n-1}]) \hookrightarrow^0_{loc} n} \quad (5.64)$$

$$\frac{}{\texttt{lengthPush}\ (\texttt{Interleave}\ m\ f \texttt{<}l\texttt{>}\ [p_0, \ldots, p_{n-1}]) \hookrightarrow^0_{loc} \texttt{muli}\ m\ n} \quad (5.65)$$

$$\frac{}{\texttt{lengthPush}\ (\texttt{InterleaveSeq}\ m\ f \texttt{<}l\texttt{>}\ [p_0, \ldots, p_{n-1}]) \hookrightarrow^0_{loc} \texttt{muli}\ m\ n} \quad (5.66)$$

$$\frac{}{\texttt{lengthPush}\ (\texttt{SeqFor}\ [\vec{v}^{(n)}]\ m\ i\ f\ g) \hookrightarrow^0_{loc} n} \quad (5.67)$$

$$\frac{}{\texttt{mapPush}\ g\ (\texttt{Push} \texttt{<}l\texttt{>}\ v) \hookrightarrow^0_{loc} \texttt{Push} \texttt{<}l\texttt{>}\ (\texttt{map}\ g\ v)} \quad (5.68)$$

$$\frac{\begin{array}{c}\forall i \in [0; n(\quad p_i = \texttt{Program} \texttt{<}l\texttt{>}\ [e_0, \ldots, e_{m-1}] \\ p'_i = \texttt{Program} \texttt{<}l\texttt{>}\ [g\ e_0, \ldots, g\ e_{m-1}]\end{array}}{\begin{array}{c}\texttt{mapPush}\ g\ (\texttt{Interleave}\ m\ f \texttt{<}l\texttt{>}\ [p_0, \ldots, p_{n-1}]) \\ \hookrightarrow^0_{loc} \texttt{Interleave}\ m\ f \texttt{<}l\texttt{>}\ [p'_0, \ldots, p'_{n-1}]\end{array}} \quad (5.69)$$

$$\frac{\begin{array}{c}\forall i \in [0; n(\quad p_i = \texttt{Program} \texttt{<}l\texttt{>}\ [e_0, \ldots, e_{m-1}] \\ p'_i = \texttt{Program} \texttt{<}l\texttt{>}\ [g\ e_0, \ldots, g\ e_{m-1}]\end{array}}{\begin{array}{c}\texttt{mapPush}\ g\ (\texttt{InterleaveSeq}\ m\ f \texttt{<}l\texttt{>}\ [p_0, \ldots, p_{n-1}]) \\ \hookrightarrow^0_{loc} \texttt{InterleaveSeq}\ m\ f \texttt{<}l\texttt{>}\ [p'_0, \ldots, p'_{n-1}]\end{array}} \quad (5.70)$$

$$\frac{}{\texttt{mapPush}\ h\ (\texttt{SeqFor}\ [\vec{e}^{(n)}]\ m\ i\ f\ g) \hookrightarrow^0_{loc} (\texttt{SeqFor}\ [\vec{e}^{(n)}]\ m\ i\ f\ (h \circ g))} \quad (5.71)$$

Figure 5.12: Small-step semantics (continued)

Figure 5.13 covers the third group of rules and detail the evaluation of sequential computations in a single GPU thread t. Rules (5.73)–(5.75) perform the actual thread-level computations, including scalar operations and conditionals.

Rules (5.76)–(5.80) specify how `force` expressions are evaluated, when invoked on sequential computations (thread level). The first two rules specify an array is computed sequentially by a single thread with thread ID t. The step cost is not limited, as it might both take 0 cost and 1 steps.

The rules for `seqfor` are described using monadic expressions. Rule (5.78) computes the input array through invoking the just explained thread-level `force`. Rule (5.79) computes a single iteration of the loop, through invoking the f function and writing the value u to index p in the array. Rule (5.80) is invoked after the final iteration and applies the function g to all elements.

Small-step semantics (continued) $\boxed{e \hookrightarrow^{c}_{\mathsf{Thread}(t)} e'/\mathbf{err}}$

$$\frac{}{\texttt{addi}\ v_1\ v_2 \hookrightarrow^{1}_{\mathsf{Thread}(t)} v_1 + v_2}\ (5.72) \qquad \frac{}{\texttt{muli}\ v_1\ v_2 \hookrightarrow^{1}_{\mathsf{Thread}(t)} v_1 \times v_2}\ (5.73) \qquad \ldots$$

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow^{1}_{\mathsf{Thread}(t)} e_1}\ (5.74)$$

$$\frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow^{1}_{\mathsf{Thread}(t)} e_2}\ (5.75)$$

$$\frac{e_i \hookrightarrow^{c}_{\mathsf{Thread}(t)} e_i'}{\begin{array}{l}\texttt{force}\ (\texttt{Push <thread>}\ [bv_0, \ldots, bv_{i-1}, e_i, \ldots, e_{n-1}]) \\ \quad \hookrightarrow^{c}_{\mathsf{Thread}(t)}\ \texttt{force}\ (\texttt{Push <thread>}\ [bv_0, \ldots, bv_{i-1}, e_i', \ldots, e_{n-1}])\end{array}}\ (5.76)$$

$$\frac{e_i \hookrightarrow^{c}_{\mathsf{Thread}(t)} \mathbf{err}}{\texttt{force}\ (\texttt{Push <thread>}\ [bv_0, \ldots, bv_{i-1}, e_i, \ldots, e_{n-1}]) \hookrightarrow^{c}_{\mathsf{Thread}(t)} \mathbf{err}}\ (5.77)$$

$$\frac{}{\begin{array}{l}\texttt{force}\ (\texttt{SeqFor}\ [\vec{e}^{(n)}]\ m\ i\ f\ g) \\ \quad \hookrightarrow^{1}_{\mathsf{Thread}(t)}\ \texttt{bind}\ (\texttt{force}\ (\texttt{Push <thread>}\ [\vec{e}^{(n)}]))\ (\texttt{fn}\ v \texttt{=>}\ \texttt{force}\ (\texttt{SeqFor}\ v\ m\ i\ f\ g))\end{array}}\ (5.78)$$

$$\frac{i < m}{\begin{array}{l}\texttt{force}\ (\texttt{SeqFor}\ [\vec{v}^{(n)}]\ m\ i\ f\ g) \\ \quad \hookrightarrow^{1}_{\mathsf{Thread}(t)}\ \begin{array}{l}\texttt{bind}\ (f\ (\texttt{fn}\ k \texttt{=>}\ v_k)\ i) \\ \quad (\texttt{fn}\ (p, u) \texttt{=>}\ \texttt{force}\ (\texttt{SeqFor}\ [v_0, \ldots, v_{p-1}, u, v_{p+1}, \ldots, v_{n-1}]\ m\ (i+1)\ f\ g))\end{array}\end{array}}\ (5.79)$$

$$\frac{i = m}{\texttt{force}\ (\texttt{SeqFor}\ [\vec{v}^{(n)}]\ m\ i\ f\ g) \hookrightarrow^{1}_{\mathsf{Thread}(t)} \texttt{force}\ (\texttt{Push}\ [g\ v_0, \ldots, g\ v_{n-1}])}\ (5.80)$$

Figure 5.13: Small-step semantics (continued)

The rules in Figure 5.14 specify how parallel expressions in the `Program`-monad are evaluated (above the thread level). Through the $loc \rhd l$ relation, it is specified how a computation is mapped to the hierarchy of the GPU. It should be noted that the `Program`-monad, is a means to restrict how values flow through the program and that a top-level program must be of type `Program<grid>` τ to be evaluated.

Rule (5.81) describes how expressions such as `force (push <`l`>` e`)` are evaluated, where a pull array e is converted to a parallel expression on the given level l. The input array is e is partitioned in n subarrays $[\overrightarrow{e_0}]$ through $[\overrightarrow{e_{n-1}}]$, however the rule does not mention exactly how the array is partitioned. Through the use of `force` at the level below, each subarray are allowed take a 0 or 1 step. We can understand this as follows: at the outermost level, a single step corresponds to a single step being executed on all locations at lower levels.

Small-step semantics with locations $\boxed{e \hookrightarrow^{c}_{loc} e' / \mathbf{err}}$

$$\frac{\begin{array}{c}\forall j \in J, \texttt{force}\ (\texttt{Push}\ \texttt{<}l\texttt{>}\ [\overrightarrow{e_j}] \hookrightarrow^{c_j}_{loc_j} \texttt{Program}\ \texttt{<}l\texttt{>}\ [e'_j] \qquad loc_j \rhd l \\ J \subseteq \{0,\ldots,n-1\} \quad |J| \leq m \quad c = \max_{j\in J}(c_j) \\ e''_i = \begin{cases} e_i & \text{if } i \notin J \\ e'_i & \text{if } i \in J \end{cases}\end{array}}{\begin{array}{l}\texttt{force}\ (\texttt{Push}\ \texttt{<}1+l\texttt{>}\ [\overrightarrow{e_0}, \ldots, \overrightarrow{e_{n-1}}]) \\ \quad \hookrightarrow^{c}_{\mathsf{Group}(\overrightarrow{loc^{(m)}})} \texttt{force}\ (\texttt{Push}\ \texttt{<}1+l\texttt{>}\ [\overrightarrow{e''_0}, \ldots, \overrightarrow{e''_{n-1}}])\end{array}} \quad (5.81)$$

$$\frac{loc \rhd l}{\texttt{force}\ (\texttt{Push}\ \texttt{<}l\texttt{>}\ [bv_0, \ldots, bv_{n-1}]) \hookrightarrow^{0}_{loc} \texttt{Program}\ \texttt{<}l\texttt{>}\ [bv_0, \ldots, bv_{n-1}]} \quad (5.82)$$

$$\frac{\begin{array}{c}\forall j \in J, p_j \hookrightarrow^{c_j}_{loc_j} p'_j \qquad loc_j \rhd l \\ J \subseteq \{0,\ldots,n-1\} \quad |J| \leq m \quad c = \max_{j\in J}(c_j) \\ p''_i = \begin{cases} p_i & \text{if } i \notin J \\ p'_i & \text{if } i \in J \end{cases}\end{array}}{\begin{array}{l}\texttt{force}\ (\texttt{Interleave}\ \texttt{<}l\texttt{>}\ k\ f\ [p_0, \ldots, p_{n-1}]) \\ \quad \hookrightarrow^{c}_{\mathsf{Group}(\overrightarrow{loc^{(m)}})} \texttt{force}\ (\texttt{Interleave}\ \texttt{<}l\texttt{>}\ k\ f\ [p''_0, \ldots, p''_{n-1}])\end{array}} \quad (5.83)$$

$$\frac{loc_j \rhd l \qquad p_i = \texttt{Program}\ \texttt{<}l\texttt{>}\ (\texttt{Push}\ \texttt{<}l\texttt{>}\ [bv^i_0, \ldots, bv^i_{k-1}])}{\begin{array}{l}\texttt{force}\ (\texttt{Interleave}\ \texttt{<}l\texttt{>}\ k\ f\ [p_0, \ldots, p_{n-1}]) \\ \quad \hookrightarrow^{1}_{\mathsf{Group}(\overrightarrow{loc^{(m)}})} \\ \qquad \texttt{bind}\ (\texttt{force}\ (\texttt{Push}\ \texttt{<}l\texttt{>}\ \ [f(0,0), f(0,1), \ldots, \\ \qquad\qquad\qquad\qquad\qquad\quad f(1,0), f(1,1), \ldots, \\ \qquad\qquad\qquad\qquad\qquad\quad \ldots, \\ \qquad\qquad\qquad\qquad\qquad\quad f(n-1,0), \ldots, f(n-1,k-1)])) \\ \qquad\qquad (\texttt{fn}\ v \texttt{=>}\ \texttt{permute}\ \texttt{<}1+l\texttt{>}\ v\ (\texttt{Push}\ \texttt{<}1+l\texttt{>}\ [\overrightarrow{bv^0}, \overrightarrow{bv^1}, \ldots, \overrightarrow{bv^{n-1}}]))\end{array}} \quad (5.84)$$

$$\frac{\exists k \notin [0; n(}{\texttt{permute}\ [\overrightarrow{k}^{(n)}]\ (\texttt{Push}\ \texttt{<}l\texttt{>}\ [\overrightarrow{bv}^{(n)}]) \hookrightarrow^{1}_{loc} \mathbf{err}} \quad (5.85)$$

$$\frac{\forall k \in [0; n(}{\texttt{permute}\ [\overrightarrow{k}^{(n)}]\ (\texttt{Push}\ \texttt{<}l\texttt{>}\ [\overrightarrow{bv}^{(n)}]) \hookrightarrow^{1}_{loc} \texttt{Program}\ \texttt{<}l\texttt{>}\ [bv_{k_0}, bv_{k_1}, \ldots, bv_{k_{n-1}}]} \quad (5.86)$$

Figure 5.14: Small step semantics continued. Rules for `force`.

Rule (5.82) corresponds to the final synchronisation after all subarrays have been fully evaluated to base values.

Enclosing values in the `Program`-monad allows us to limit at which points values can flow upwards in the hierarchy of computation. From the types, we know that escaping up to the above level is possible only through the `interleave`-operation. The `interleave` operation is thus central to the language, as it specifies how values are passed up through the hierarchy, as well as how these values are written in the combined array. Recall the type of `interleave`:

$$\begin{aligned}\texttt{interleave} : \forall\texttt{<}\delta\texttt{>}\ \alpha.\ \texttt{int} \\ &\to ((\texttt{int}, \texttt{int}) \to \texttt{int}) \\ &\to [\texttt{Program<}\delta\texttt{>}\ [\alpha]\texttt{<}\delta\texttt{>}] \\ &\to \texttt{Program<}1 + l\texttt{>}\ [\alpha]\texttt{<}1 + \delta\texttt{>}\end{aligned}$$

The third argument is a pull array of `Program` arrays that needs to be combined; each `Program` specifies a push array computation executed on a lower level processor (a streaming multiprocessor or a single compute unit). The first argument specifies the size of subarrays of the third argument, which allows us to precompute the size of the combined array during allocation. Where Obsidian only allows concatenation of array results, `interleave` allows specification of a function that determines where in the final array the individual values should be stored. For each computed value, the function is invoked with the corresponding index in the outer pull array and the inner push-array. The returned index determines where it should be placed in the final array.

Concatenating the subarrays in row-major order can thus be performed as follows:

```
sig concat : forall <l> 'a. int
                            -> [Program<l> ['a]<l>]
                            -> Program<1+l> ['a]<1+l>
fun concat<l> n arr =
  interleave<l> n (fn sh => ((fst sh) * n) + (snd sh)) arr
```

The `interleave` operation is not intended for direct usage by FCL users, but can be used to write operations such as `concat` and `concatGrid`, which can be provided in a standard library for users.

In the first rule for `interleave`, rule (5.83), it is specified that at most m sub-programs in the given pull array can take a 1 step, where m is the number of locations in the group. The set J determines which of the subprograms in the array $[\overrightarrow{p}^{(n)}]$ are taking a step. All subprograms p_j where $j \notin J$ are left unchanged.

The second rule for `interleave`, Rule (5.84), fires when all sub-programs are fully evaluated to vectors of base values, and constructs a permutation vector from the permutation function f, specifying the order in which the computed values should be written to memory. In this formalisation, the actual

$$\frac{}{\texttt{power}\ 0\ f\ v \hookrightarrow^1_{loc} v} \quad (5.87)$$

$$\frac{n > 0}{\texttt{power}\ n\ f\ (\texttt{Program <}l\texttt{>}\ v) \hookrightarrow^1_{loc} \texttt{bind}\ (\texttt{force}\ v)\ (\texttt{fn}\ v' \texttt{=>}\ \texttt{power}\ (n-1)\ f\ (f\ v'))} \quad (5.88)$$

$$\frac{}{\begin{array}{l}\texttt{while}\ f\ g\ (\texttt{Program <}l\texttt{>}\ v) \hookrightarrow^1_{loc} \\ \quad \texttt{bind}\ (\texttt{force}\ v)\ (\texttt{fn}\ v' \texttt{=>}\ \texttt{bind}\ (f\ v')\ (\texttt{fn}\ b \texttt{=>}\ \texttt{ifP}\ b\ (\texttt{while}\ f\ g\ (g\ v'))\ (\texttt{return <}l\texttt{>}\ v'))\end{array}} \quad (5.89)$$

$$\frac{}{\texttt{ifP true}\ x1\ x2 \hookrightarrow^1_{loc} x1} \quad (5.90) \qquad \frac{}{\texttt{ifP false}\ x1\ x2 \hookrightarrow^1_{loc} x2} \quad (5.91)$$

Figure 5.15: Small step semantics continued. Rules for `seqfor`.

permutation is performed by the previously mentioned `permute` function, which we introduced to simplify the presentation of `interleave`.

We have left out the rules for `interleaveSeq`. These rules are similar to the rules for `interleave`, however without moving data up the hierarchy. That is, the $<1 + l>$ level argument to `permute` and `Push`, should be changed to $<l>$.

It should be noted that these rules for `interleave` does not go into detail with how values flow up and down through the memory system. In fact, following this formalisation subprograms might be reassigned to various locations of the machine on every step. In practice, however, our implementation only pass values back up through the memory hierarchy when a subprogram is completely evaluated, not in every step. There is an implicit synchronization after all `interleave` operations. While the outset of the project aimed at providing a memory cost-model, the developed semantics only covers the use of computing resources, and the GPU memory systems effects on performance such as memory coalescing are not possible to cover.

Rules (5.87)–(5.91) in Figure 5.15 evaluates the loop constructs `power` and `while`. The operations `while` and `power` are similar to the power operation in APL, modified to work on push and pull arrays. The initial value is given as a push array, and before the loop begins, this array is forced. The resulting pull array will be given to the iteration function, which again returns a push array to be consumed in the next iteration. Note the use of `ifP` in rule 5.89, this is the only place it is used. In practice, the `power` and `while` constructs reuse the same memory area for each `force` operation. This scheme limits these iteration constructs to non-enlarging functions, as the result after an iteration step must always be smaller than the array given as initial value.

5.3.4 Properties of the language

In the following we present propositions stating type soundness of FCL, however only when excluding the `while` and `seqfor`.

We use the notation $e \hookrightarrow\!\!\!\!\rightarrow^{c}_{loc} e'$ for the transitive and reflexive closure of $\hookrightarrow^{c}_{loc}$, as defined below. In such judgments, c ranges over natural numbers $\mathbb{N}_0$.

$$\boxed{e \hookrightarrow\!\!\!\!\rightarrow^{c}_{loc} e'/\mathbf{err}}$$

$$\frac{}{e \hookrightarrow\!\!\!\!\rightarrow^{0}_{loc}} \ (5.92) \qquad \frac{e \hookrightarrow^{c_1}_{loc} e' \quad e' \hookrightarrow\!\!\!\!\rightarrow^{c_2}_{loc} e'' \quad c = c_1 + c_2}{e \hookrightarrow\!\!\!\!\rightarrow^{c}_{loc} e''} \ (5.93)$$

Proposition 9 (Type Preservation) *If $\Delta, \Gamma \vdash e : \tau$, then either*

(1) e is a value; or

(2) there exists c such that $e \hookrightarrow^{c}_{loc} \mathbf{err}$ and $loc \blacktriangleright \tau$; or

(3) there exists c such that $e \hookrightarrow^{c}_{loc} e'$ and $loc \blacktriangleright \tau$ and $\Delta, \Gamma \vdash e' : \tau$.

Proposition 10 (Progress) *Given a location $loc \blacktriangleright \tau$, and if $\Delta, \Gamma \vdash e : \tau$, then either*

(1) e is a value; or

(2) there exists c such that $e \hookrightarrow^{c}_{loc} \mathbf{err}$ is a value; or

(3) there exists e' and c such that $e \hookrightarrow^{c}_{loc} e'$

The following soundness proposition states that given a machine specification (location) loc and an expression with a type respecting that location, then an execution of said expression will either error, fail to terminate, or compute a value of type τ.

Proposition 11 (Type Soundness) *Given a location $loc \blacktriangleright \tau$, if $\Gamma_{init} \vdash e : \tau$ then either*

(1) there exists c such that $e \hookrightarrow\!\!\!\!\rightarrow^{c}_{loc} \mathbf{err}$; or

(2) there exists an infinite sequence $e \hookrightarrow^{c}_{loc} e' \hookrightarrow^{c'}_{loc} e'' \hookrightarrow^{c''}_{loc} e''' \ldots$; or

(3) there exists c and a value v such that $e \hookrightarrow\!\!\!\!\rightarrow^{c}_{loc} v$ and $\vdash v : \tau$

In practice, top level expressions always have the type `Program <grid> int`, we state this special case of the soundness proposition in the following corollary.

Corollary 1 *Given a location $loc \triangleright$ `grid`, if $\Gamma_{init} \vdash e :$ `Program <grid> int` then either*

*(1) there exists c such that $e \hookrightarrow\!\!\!\!\rightarrow^{c}_{loc}$ **err**; or*

(2) there exists an infinite sequence $e \hookrightarrow^{c}_{loc} e' \hookrightarrow^{c'}_{loc} e'' \hookrightarrow^{c''}_{loc} e''' \ldots$; or

(3) there exists c and a value v such that $e \hookrightarrow\!\!\!\!\rightarrow^{c}_{loc}$ `Program <grid>` v and $\vdash v$: `int`

5.4 Larger examples

We have now introduced FCL in its entirety, and we will now demonstrate its capabilities more fully, through some larger example programs. First, let us return to the `transpose` example presented in Section 5.2.

5.4.1 Transpose revisited: `splitGrid` and `concatGrid`

The `transpose` operation presented in Figure 5.2 used operations `splitGrid` and `concatGrid` to partition an array in two-dimensional tiles. The `splitGrid` operation can be implemented as two nested calls to `generate`, and boils down to a number of index calculations.

```
sig splitGrid : forall 'a. int -> int -> int -> ['a] -> [['a]]
fun splitGrid splitSize width height elems =
  let tileSize = splitSize * splitSize in
  let groupsWidth = width / splitSize in
  let groupsHeight = height / splitSize
  in
   generate (groupsWidth * groupsHeight)
    (fn gid =>
      let groupIDy = gid / groupsWidth in
      let groupIDx = gid % groupsWidth
      in generate tileSize
          (fn tid =>
            let localIDx = tid % splitSize in
            let localIDy = tid / splitSize in
            let xIndex = groupIDx * splitSize + localIDx in
            let yIndex = groupIDy * splitSize + localIDy in
            let ix = yIndex * width + xIndex
            in index elems ix))
```

The `concatGrid` operation is written using the `interleave` operation, again requiring a number of index calculations:

```
sig concatGrid : forall <l> 'a. int
                                -> int
                                -> [Program <l> ['a]<l>]
                                -> Program <1+l> ['a]<1+l>
fun concatGrid <l> splitSize width arr =
    let tileSize = splitSize * splitSize in
    let groupsWidth = width / splitSize in
     interleave <l> tileSize
          (fn sh =>
             let gid = fst sh in
             let tid = snd sh in
             let groupIDx = gid % groupsWidth in
             let groupIDy = gid / groupsWidth in
             let localIDx = tid % splitSize in
             let localIDy = tid / splitSize in
             let xIndex = groupIDx * splitSize + localIDx in
             let yIndex = groupIDy * splitSize + localIDy in
             let ix = yIndex * width + xIndex
             in ix)
          arr
```

It is the responsibility of the author of functions such as `concatGrid` to guarantee memory coalescing. The `interleave` n f operation allows an arbitrary permutation to performed before the array is written, using the function f. The programmer should make sure that invocations $f(i,j), f(i,j+1), f(i,j+2), \ldots$ return values in the same neighbourhood (corresponding to writes into the same memory segment).

The author of functions such as `splitGrid` can however not provide any guarantees, as he is merely constructing a pull array. The reading order of the actual array will be determined by how this array ends up being read. That such guarantees can not be made is a major drawback of pull arrays, and often makes it necessary to reason backwards, if you want to understand whether a given program accesses an array coalesced.

It should be noted that it is not the intention that users need to be aware of the details of these lengthy implementations; they can be provided in a standard library.

5.4.2 Tiled matrix multiplication

Another operation that can make use of `splitGrid` and `concatGrid` is matrix multiplication. Through tiling, we can limit the number of global memory accesses necessary, by moving complete tiles of the matrix into local memory, before computing. Such an optimization is possible for matrix multiplication, as the problem can be subdivided into a large number of smaller matrix multiplications.

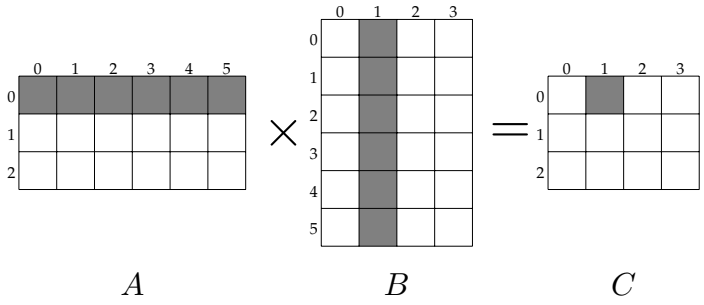

Figure 5.16: Tiled matrix multiplication. The matrix product $AB = C$ can be computed by subdivison into smaller matrices. Computing the greyed out tile of matrix C, corresponds to multiplying the greyed out rectangles of matrix A and matrix B.

When computing the matrix product $A \times B = C$, the two matrices A and B can be subdivided into horizontal and vertical rectangles, respectively, as shown in the Figure 5.16. Each rectangle consists of a number of smaller square matrices. Computation of the sub-matrix $C_{i,j}$ of matrix C, can be implemented as the sum of matrix products:

$$C_{i,j} = \sum_k A_{j,k} B_{k,i}$$

In FCL, the above algorithm can be implemented as follows. First, we will need a program for computing a simple matrix multiplication in a single work group. The standard matrix multiplication algorithm can be implemented by performing $m \times n$ dot products. We therefore need a sequential dot product operation, which we can implement using `foldl`:

```
sig dotp : forall 'a. ('a -> 'a -> 'a)
                   -> ('a -> 'a -> 'a)
                   -> 'a -> ['d] -> ['c] -> ['a]<thread>
fun dotp f g neutral vec1 vec2 =
  foldl f neutral (zipWith g vec1 vec2)
```

The block-wide matrix multiplication can then be implemented through concatenating $m \times n$ such dot products:

```
-- Block-wide matrix multiplication
-- multiply m*p matrix by p*n matrix
sig matmulBlock :
  forall 'a. ('a -> 'a -> 'a)
           -> ('a -> 'a -> 'a)
           -> 'a -> int -> int -> int -> ['a] -> ['a]
           -> Program <block> ['a]<block>
fun matmulBlock f g neutral m p n A B =
  generate (m * n)
    (fn i =>
       let col = i % n in
       let row = i / n in
       let rowVec = generate p
                      (fn j => index A (row * p + j)) in
       let colVec = generate p
                      (fn j => index B (j * p + col))
       in return<thread> (dotp f g neutral rowVec colVec))
   |> concat<thread> 1
```

To be able to implement matrix multiplication, we will also need a variant of `zipWith`, that zips a pull array and a push array into a push array. Such a function can be implemented as follows, using a variant of `mapPush` that passes index information to the mapping function:

```
sig zipWithPush : forall <lvl> 'a 'b 'c.
                    ('a -> 'b -> 'c)
                   -> ['a] -> ['b]<lvl> -> ['c]<lvl>
fun zipWithPush <lvl> f a1 a2 =
  mapPushIx<lvl> (fn i => fn x => f (index a1 i) x) a2
```

With these functions we can now perform a multiplication of two rectangular arrays, which have already been tiled. We first create an array containing the neutral element in all positions. We then use the `power` operation to iterate over each tile of the two rectangles, initially forcing each tile to shared memory. Finally, we compute the matrix-multiplication using `matmulBlock` and adding the result to the accumulated array.

```
sig matmulRect : forall 'a. ('a -> 'a -> 'a)
                         -> ('a -> 'a -> 'a)
                         -> 'a -> [['a]] -> [['a]]
                         -> int -> int
                         -> Program <block> ['a]
fun matmulRect f g neutral rectA rectB s q =
  let init = generate (s*s) (fn y => neutral) in
  let body =
    (fn i => fn accumulator =>
       do<block>
        { let subA = index rectA i
        ; let subB = index rectB i
        ; subA' <- force<block> (push<block> subA)
        ; subB' <- force<block> (push<block> subB)
        ; res <- matmulBlock f g neutral s s s subA' subB'
        ; return<block> (zipWithPush<block> f accumulator res)
        })
  in power<block> q body
                  (return<block> (push<block> init))
```

The final step is to partition the original matrices A and B using `splitGrid`, apply the `matmulRect` for all combination of rectangles from the two arrays, and concatenate the results using `concatGrid`.

```
sig matmulTiled : forall 'a. ('a -> 'a -> 'a)
                          -> ('a -> 'a -> 'a)
                          -> 'a -> int
                          -> int -> int -> int
                          -> ['a] -> ['a]
                          -> Program <grid> ['a]<grid>
fun matmulTiled f g neutral split hA wA wB A B =
 let tileSize = split * split in
 let p = wA / split in
 let q = wB / split in
 let tiledA = splitGrid split wA hA A in
 let tiledB = splitGrid split wB wA B
 in
  iota (hA*wB / tileSize)
   |> map
       (fn tile =>
         let row = tile / q in
         let col = tile % q in
         let rectA = generate p (fn i => index tiledA (p * row + i)) in
         let rectB = generate p (fn i => index tiledB (i * q + col))
         in matmulRect f g neutral rectA rectB split p)
   |> concatGrid<block> split wB
```

5.4.3 Parallel prefix sum

Like Obsidian, we can implement a scan operation in FCL. In the `scanChunked` function below `sklansky` is block-level scan, as previously presented in the context of Obsidian. We leave out the definition as it is almost identical to their presentation.

The main function `scan` performs three `grid`-level computations, the first performs a block level scan, the second compute the cumulative sums after each block, through a second scan operation, and finally we perform map that adjust the results, by adding the cumulative sum of all previous blocks to each value of the current block.

The final operation adjusting each value returns a `grid`-level push array, which is the result of the entire computation. When using `scan` in a larger program, it would thus be possible to fuse the final `grid`-level computation with any `map` operations immediately following it.

```
sig scanChunked : forall 'a. int -> ('a -> 'a -> 'a) -> ['a]
                            -> Program <grid> ['a]<grid>
fun scanChunked lgChunkSize op arr =
  let chunkSize = 1 << lgChunkSize
  in concatMap<block> chunkSize
                      (sklansky lgChunkSize op)
                      (splitUp chunkSize arr)

sig intermediateSums : forall 'a. int -> ['a] -> ['a]
fun intermediateSums chunkSize arr =
  map last (splitUp chunkSize arr)

fun adjust n f neutral scans sums =
  (generate (length scans / n)
     (fn i => generate n
       (fn j => f (index scans (i*n + j))
                  (if i == 0
                   then neutral
                   else index sums (i - 1)))))
   |> map (fn a => return <block> (push<block> a))
   |> concat <block> n

sig scan : forall 'a. ('a -> 'a -> 'a) -> 'a -> ['a]
                    -> Program <grid> ['a]<grid>
fun scan f neutral input =
 let lgChunkSize = 8 in
 let chunkSize = 1 << lgChunkSize
 in do<grid>
     { scansPush <- scanChunked lgChunkSize f input
     ; scans <- force<grid> scansPush
     ; let sums = intermediateSums chunkSize scans
     ; scanSumsPush <- scanChunked lgChunkSize f sums
     ; scanSums <- force<grid> scanSumsPush
     ; adjust chunkSize f neutral scans scanSums
     }
```

5.5 Performance

FCL is work in progress; thus certain optimizations are still not implemented. However, the performance on the previously shown examples is promising and we have identified the bottlenecks that are currently limiting performance. In this section, we compare the performance of each benchmark with hand-written OpenCL kernels from NVIDIA's OpenCL SDK.

When an FCL program is compiled, the result is a file containing one or more OpenCL kernels and an executable that instruments the execution of the OpenCL kernels.

To benchmark the generated code, we have used an NVIDIA GeForce GTX 780 Ti, which is built on the Kepler architecture. It has 2880 cores (875 Mhz), and 3GB GDDR5 ram (7 Ghz, bus-width: 384 bit). Calculating the theoretical peak bandwidth on global memory accesses, we get 7Ghz $\times$ 384bit $=$ 336GB/s. In practice we can expect a 254.90GB/s maximum bandwidth, when accessing global memory, which we have measured using NVIDIA's benchmarking tool (`bandwidthTest`).

Each benchmark has been executed on an array of 2^{24} 32-bit integers (64 MiB). Timing was measured using OpenCL profiling operations, on 1000 executions of the same kernel, preceded by a single warm-up run.

The achieved global-memory bandwidths are shown in Figure 5.17, and are measurements of a single kernel call (e.g. not a full reduction). The measured maximum bandwidth, measured with NVIDIA's `bandwidthTest` program is plotted as a dashed horizontal line. The actual achieved performance for each kernel is presented in Table 5.1.

In the reverse and transpose examples, we hit the measured maximum bandwidth as we hoped. The generated code is similar to the hand-written code from NVIDIA, except for block-virtualization, which is not used in NVIDIA's version.

The reduction example is interesting; here we generate a completely unrolled loop, which performs reasonably well, but does not quite hit the performance target set by NVIDIA's heavily tuned kernel. To identify how we can improve our solution, we have inspected the difference between the two kernels. To get on par with NVIDIA's kernel, we will need to make each thread do an initial sequential reduction on a few elements, before the parallel tree-reduction we already have implemented.

Furthermore, we synchronize across the complete work group between each reduction step, even when the amount of elements is below warp size. Adding a warp-level between thread-level and block-level, would make it possible to avoid such unnecessary synchronizations. We had such a warp-level in our earlier implementation, but we have removed it for simplicity; it should not be a problem to reintroduce it again.

For the matrix multiplication example, we are not quite on par with the handwritten code either. Investigation lead us to believe that the extensive usage of divisions and modulo operations are to blame. See Section 5.7 below

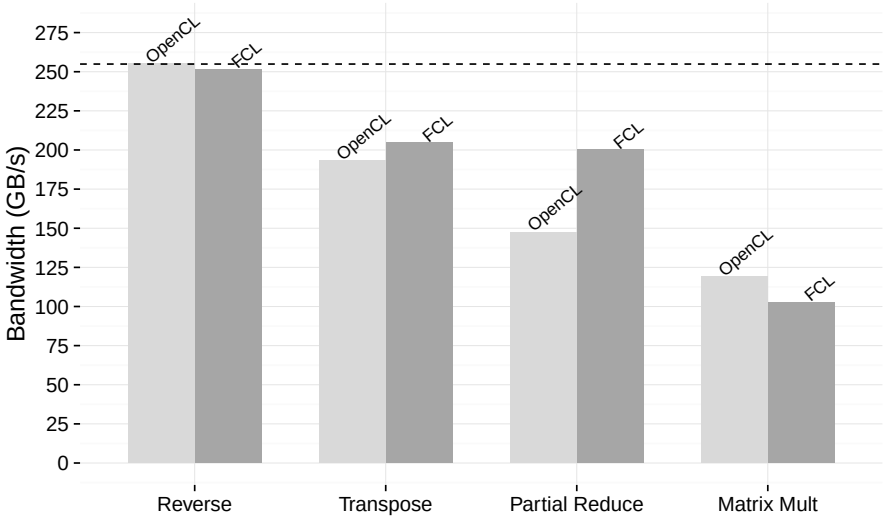

Figure 5.17: Measured bandwidths on our example programs. OpenCL bars are code from NVIDIA's OpenCL SDK, and we compare it to OpenCL kernels generated by FCL. The dashed line indicates the maximum achievable bandwidth as measured by NVIDIA's benchmarking tool.

Benchmark	**Input size**	**OpenCL**	**FCL**
Reverse	2^{24} 32-bit integers	0.526 ms	0.533 ms
Transpose	2^{24} 32-bit integers	0.695 ms	0.648 ms
Partial reduction	2^{24} 32-bit integers	0.331 ms	0.454 ms
Matrix multiplication	2^{24} doubles	8.692 ms	10.076 ms

Table 5.1: Timings of single-kernel executions of each micro benchmark. Average over 1000 invocations.

for suggestions for how we might avoid these expensive operations through introduction of multidimensional arrays.

Table illustrate that FCL also allow slightly bigger programs consisting of several kernels to compiled. We compare the performance of our `reduce` and `scan` programs with the kernels provided with the CUDA-based framework Thrust [83]. Timings are measured as wall-clock time.

For the reduction example we are on par with Thrust, which is curious as the optimised NVIDIA kernel where beating us in our previous single-kernel benchmark. For `scan` we are 50% slower than Thrust, which might be due to a different algorithm used.

Benchmark	Input size	Thrust	FCL
Full reduction	2^{24} 32-bit integers	0.5503 ms	0.533 ms
Full scan	2^{24} 32-bit integers	1.4543 ms	2.84 ms

Table 5.2: Wall-clock timings of full reduction (2 kernels) and scan (3 kernels).

5.6 Related work

Obsidian and FCL are not the first languages for hierarchical parallel machines. Sequoia is an imperative hierarchical language [44], inspired by previous work on Parallel Memory Hierarchies (PMH) [5], supporting both cluster computing through MPI and programming of multiple GPUs. Both Sequoia and PMH models a parallel machine as a tree of distinct memory modules. Programs are written to be machine independent, where function calls correspond to either executing a subtask on a child in the hierarchy (copying data to this memory module) or staying in the same memory module. Thus, the call/return of a subtask implies that data movement through the machine hierarchy *might* occur. The stopping criteria for recursive functions are left out, and instead specified separately in a *mapping specification*, which details how an algorithm maps to a concrete machine. Programs can also involve *tunable* parameters and various variants of the same algorithm; the mapping specification also controls these choices. Mapping specifications can potentially be automatically generated. The ideas from Sequoia are further generalized in the work on Hierarchical Place Trees [109].

Another hierarchical data-parallel language is HiDP by Zhang and Mueller [81] for hierarchical GPU-programming. In addition to the hierarchies of Obsidian and FCL, they add two sub-warp levels of size 4 and 8, respectively. Parallelism is embodied as nested parallel-for loops (which are called map-blocks) together with a set of built-in parallel array-operators (partition, reduce, scan, sort, and reverse). Arrays are multi-dimensional and nested irregular segmented arrays are built-in. For optimization purposes, it is however also possible to use regular arrays. Fusion decisions and use of shared-memory are completely controlled by the compiler.

The language discussed by Dubach et al. [94] is also related to FCL, operating at a similarly low-level. The main idea is to build a language that can be automatically tuned to hardware, by applying search strategies on the provided set of rewrite rules. It might be interesting to build a similar search-based rewrite-engine on top of FCL and allow the user to express rewrite-rules. Another interesting aspect of this work is its support for programming with vector-instructions (such as adding two `int4` values in OpenCL), which would correspond to a layer between `warp`-level and `thread`-level in Obsidian and FCL. Previous work have also been attempted in the context of Obsidian [105] and Qilin [74].

The hierarchy in FCL and Obsidian might also be compared to the concept of locales and sublocales in the Chapel language [31]. However, Chapel does

not currently target GPUs.

Functional approaches to GPU computing have typically concentrated on optimizing compilers that are intended to shield the user from the need to understand (or control) details of the GPU. Examples include Futhark [59], Accelerate [29], Delite [27], Harlan [62], and Nessie [11]. These projects might perhaps be considered to be at roughly the same level as NVIDIA's Thrust library [83]. FCL and Obsidian are rather at the level of NVIDIA's CUB library, which provides reusable software components for every level of the GPU hierarchy [84].

5.7 Discussion and future work

The FCL language is still at a very early prototype stage. We have already mentioned that reintroduction of `warp`-level would be beneficial for algorithms where extraneous work-group wide synchronisation are slowing the computation down. Doing so would not be difficult, but we chose to remove that feature for ease of implementation of our prototype compiler.

Another interesting and similar effort could be support for an extra level above `grid`-level. Currently, all arrays always reside on the device. However, this restriction is not feasible for large problems, because of the limited memory of GPU devices. Having a level above the `grid`-level would allow programmers to reason about data-movement between the host and the device. However, in our efforts towards implementing such a level we found that the push/pull array representation was not appropriate, as handles to the underlying arrays are not accessible across GPUs. Moving data to and from GPUs must be performed in large transactions, not elementwise, and a different array representation is thus necessary. On newer NVIDIA devices supporting unified memory, it may be less of a problem to use push/pull arrays, as the data movement will then be handled automatically by the OpenCL/CUDA runtime.

Adding extra levels to the language, such as a warp or a device level, however, would also require users to write even lengthier programs to specify how arrays are chunked and subproblems combined. Often the same split/interleave operations would be necessary for each level of the hierarchy until we arrive at the thread-level. To avoid such cumbersome programming, it would be relevant to investigate how recursion over the machine hierarchy could be achieved in a way similar to the programming language Sequoia [44], albeit in functional style. Adding recursion and a ad-hoc level-polymorphism in the form of `case`-statement working on levels should indeed be possible, as it would be possible to unfold and remove both after monomorphization.

Programs might then be written in a style similar to the following pseudocode, and users of such functions would obtain different versions depending on what level of the hierarchy that they require.

```
sig f : forall 'a 'b <lvl>. ['a] -> Program<lvl> ['b]<lvl>
fun f <lvl> xs =
  case <lvl> of
    | <thread> => ... sequential solution...
    | <1+l> => partition into subproblems
                |> map (f<l>)
                |> combine subproblems
```

Furthermore, as can be seen in the `transpose` program as well as the `splitGrid` and `concatGrid` functions, a large amount of index calculations are sometimes necessary in the current version of FCL. Supporting multidimensional arrays in the style of Repa [68] or Accelerate [28] would allow us to reduce the amount of index calculations. We could for instance implement `splitGrid` in a more straightforward maner, avoiding costly divisions and give it a type that more precisely describes its functionality:

```
sig splitGrid : forall 'a. int -> ['a]2 -> [['a]2]2
fun splitGrid chunkSize arr =
  generate2D
    (width arr / chunkSize, height arr / chunkSize)
    (fn (outerX, outerY) =>
       generate2D (chunkSize, chunkSize)
           (fn (innerX, innerY) =>
             let x = outerX * chunkSize + innerX in
             let y = outerY * chunkSize + innerY in
             let index = y * (width arr) + x
             in index elems index))
```

Here we use the notation $[\alpha]^n$ for the type of pull arrays of rank n, and after the `splitGrid` operation we would obtain a 2D array of 2D arrays. Making such a change would of course also require most other operations to be changed to work with higher-dimensional arrays. For example, a modified `force` function operating on multidimensional arrays, might be implemented with the following type:

$$\texttt{force} : \forall\alpha\texttt{<}l\texttt{>}.\ [\alpha]^n\texttt{<}l\texttt{>} \rightarrow \texttt{Program<}l\texttt{>}\ [\alpha]^n$$

Another area of future work relates to fusion. FCL currently allows producer/producer fusion and consumer/producer fusion. However, horizontal fusion is not possible in general, which can be a major drawback for many applications. Allowing push arrays to be fused horizontally requires that two push arrays can share the same loop structure. Currently we only allow push arrays to contain base types. However, if we also allowed tuples, an array of tuples, to tuples of arrays conversion would allow us to write two or more arrays simultaneously. Such an approach would however still require a manual composition of programs from the user, which is not desirable. Alternatively, it might be interesting to look at how the `force<grid>` operation could be

modified to accept several computations, and execute them in tandem, if they follow the same structure.

As stated previously, the use of pull and push arrays have several drawbacks. Improving on these representations to support larger memory transactions and better support for horizontal fusion is one path to explore. Alternatively, it would be interesting to avoid the distinction altogether, and instead provide a limited set of rewriting rules on array combinators that are simple enough that users can reason about when they will be applied.

Furthermore, with the current version of pull and push arrays, we found it difficult to give any precise cost-model, as the cost of accessing a pull array both depends on the order it will be accessed. It would be interesting to remodel the language such that data movement from global memory to local-memory is always required to be in blocks corresponding to the transaction size of the GPU, which would expose the cost of uncoalesced accesses directly to the user. In this way it should only be possible to access individual array element from within sequential code, when data is already moved to the streaming multiprocessor. In practice, the compiler might remove unecessary uses of shared memory, but forcing the user to always move data between all layers of the GPU would probably make the language more uniform, and perhaps easier to provide a meaningful cost-model.

Currently memory is allocated implicitly during `force`, and it is thus not possible to reuse the same memory, except using the `power/while` operations. We would want the possibility of writing an in-place version of for example `reverse`, writing the reserved array back to the same global-array. Similar problems occur in the sequential loop, as we always have to initialise the array, even though these initial values will never be used. Futhark [59] solves these issues using uniqueness types, allowing them to provide safe in-place updates to arrays, by guaranteeing that the reference to the given array is unique, and no execution path might use it.

5.8 Conclusion

We have presented FCL, a functional language for GPU algorithms. FCL is work in progress, but the prototype that we have presented shows promising results.

FCL builds on previous work on Obsidian [96], from which both the concepts of push arrays and level-variables originate. FCL distinguishes itself from Obsidian by adding support for more involved interleaving patterns when push arrays are forced, being a self-contained language, supporting wider range of sequential loops, and by using a similar programming style for all layers of the hierarchy.

In the previous section, we have identified a number of paths for future work. Other possibilities for futre work would be to implement some larger example programs in FCL, and attempt to use FCL as an intermediate language for our APL-compiler [43].

Chapter 6

Conclusion

We have presented two functional data-parallel programming languages for programming massively parallel architectures.

We first presented a statically typed array intermediate language, TAIL [43], for compiling the language APL, which has seen wide use in the financial industry. APL is originally a dynamically typed and interpreted language providing support for multidimensional array and a wide range of built-in shape-polymorphic primitives.

Our TAIL compiler supports a large subset of APL. The TAIL type system support several kinds of types for arrays, with gradual degree of refinement using subtyping. The type system keeps track of array ranks, and allows one-dimensional to be given a more refined type, which keeps track of vector lengths. Keeping track of vector lengths are necessary in situations where vectors are used as array shapes (e.g. in the `reshape` operation). We further demonstrate that TAIL is suitable as the target for an inference algorithm for an APL compiler.

After an APL program have been translated into TAIL, we have shown that the resulting TAIL program can be further translated into fused high-performance sequential code, or through various existing data-parallel functional programming languages into efficient GPU code [23, 55].

Following our TAIL project, we determined that it would be interesting to pursue a compilation strategy that allows users to reason about data movement and data access patterns, while still supporting efficient composition through fusion. It has been mentioned that a reason why functional data-parallel languages lack industrial adoption, is the limited ability for users to optimise their algorithms and ensure performance. Previous work on compilers for functional data-parallel languages often rely on heavy program restructuring to achieve fusion, without providing control of when fusion will or will not happen, or whether memory accesses are coalesced.

The language resulting from our work in this direction, FCL [40], is a purely functional data-parallel language based on previous work on Obsidian [96]. FCL allows expressing data-parallel algorithms in a style following the

hierarchical structure of the GPU, and allows reasoning about the location and movement of data through memory system.

The implementation of FCL is based delayed array representations and user annotations of when data should be written. We demonstrate through a number of micro benchmarks that FCL compiles to efficient GPU code. However, the delayed array representations makes horizontal fusion hard and does not allow bulk memory operations, which are necessary for efficient data movement between the GPU and the host. It is future work to determine if another compilation scheme can be used to solve some of these issues.

While FCL is still at a prototype stage, it might be useful as an explorative tool to try various algorithmic approaches when implementing parallel algorithms for the GPU.

Pointing forward, future work might include strengthening the FCL language with a cost-model, providing support for ad-hoc level-polymorphism, and work on compiling TAIL to FCL. A target worth pursuing might be to allow APL users the ability to annotate their APL programs with level annotations, instructing the underlying compiler framework how APL programs are mapped to hierarchical machines.

Bibliography

[1] Philip Samuel Abrams. *An APL machine*. PhD thesis, Stanford University, 1970.

[2] Advanced Micro Devices, Inc. *AMD Accelerated Parallel Processing, OpenCL Programming Guide*, 2013.

[3] Alok Aggarwal, Bowen Alpern, Ashok Chandra, and Marc Snir. A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 305–314. ACM, 1987.

[4] Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] Bowen Alpern, Larry Carter, and Jeanne Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers, 1993. Proceedings*, pages 116–123. IEEE, 1993.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[7] Christian Andreetta, Vivien Begot, Jost Berthold, Martin Elsman, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. A financial benchmark for gpgpu compilation. In *18th International Workshop on Compilers for Parallel Computing (CPC'15)*, CPC '15, 2015.

[8] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

[9] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[10] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. Data-only flattening for nested data parallelism. *SIGPLAN Not.*, 48(8):81–92, February 2013.

[11] Lars Bergstrom and John Reppy. Nested Data-parallelism on the GPU. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12. ACM, 2012.

[12] Robert Bernecky. The role of apl and j in high-performance computation. *SIGAPL APL Quote Quad*, 24(1):17–32, September 1993.

[13] Robert Bernecky. APEX: The APL parallel executor. Master's thesis, Department of Computer Science University of Toronto, 1997.

[14] Robert Bernecky and Stephen B. Jaffe. ACORN: APL to C on real numbers. In *ACM SIGAPL Quote Quad*, pages 40–49, 1990.

[15] Jost Berthold, Andrzej Filinski, Fritz Henglein, Ken Friis Larsen, Mogens Steffensen, and Brian Vinter. Functional high performance financial it. In *International Symposium on Trends in Functional Programming*, pages 98–113. Springer, 2011.

[16] R. S. Bird. An Introduction to the Theory of Lists. In *NATO Inst. on Logic of Progr. and Calculi of Discrete Design*, pages 5–42, 1987.

[17] Guy Blelloch. Programming Parallel Algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, 1996.

[18] Guy Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[19] Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[20] Paul Bratley and Bennett L Fox. Algorithm 659: Implementing sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):88–100, 1988.

[21] Richard P. Brent and H-T_ Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, 31(3):260–264, 1982.

[22] Timothy Budd. *An APL Compiler*. Springer-Verlag, 1988.

[23] Michael Budde. Compiling APL to Accelerate through a typed IL. MSc project, Department of Computer Science, University of Copenhagen (DIKU), November 2014.

[24] Chris Burke and Roger Hui. J for the APL programmer. *SIGAPL APL Quote Quad*, 27(1):11–17, September 1996.

[25] Philip Carlsen and Martin Dybdal. Option pricing using data-parallel languages. Master's thesis, DIKU, University of Copenhagen, Department of Computer Science, 2013.

[26] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. *ACM SIGPLAN Notices*, 46(8):47–56, 2011.

[27] Hassan Chafi, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Anand R Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *ACM SIGPLAN Notices*, volume 46. ACM, 2011.

[28] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP'11, pages 3–14. ACM, 2011.

[29] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[30] Bradford L Chamberlain. Chapel: Parallel programmability from desktops to supercomputers, 2016. URL: `http://chapel.cray.com/presentations/ChapelForCopenhagen-presented.pdf`.

[31] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[32] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[33] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, 2009.

[34] Hanfeng Chen and Wai-Mee Ching. An eli-to-c compiler: Production and performance, 2013.

[35] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, pages 21–30. ACM, 2012.

[36] Koen Claessen, Mary Sheeran, and Joel Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *7th workshop on Declarative aspects and applications of multicore programming*, DAMP'12. ACM, 2012.

[37] Robert Clifton-Everest, Trevor L McDonell, Manuel MT Chakravarty, and Gabriele Keller. Embedding foreign code. In *Practical Aspects of Declarative Languages*, pages 136–151. Springer, 2014.

[38] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07. ACM, 2007.

[39] Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. Gpu programming in a high level language: compiling x10 to cuda. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, page 8. ACM, 2011.

[40] Martin Dybdal, Martin Elsman, Bo Joel Svensson, and Mary Sheeran. Low-level functional gpu programming for parallel algorithms. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, pages 31–37. ACM, 2016.

[41] Conal Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, 2003.

[42] Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

[43] Martin Elsman and Martin Dybdal. Compiling a subset of APL into a typed intermediate language. In *1st ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY'14. ACM, 2014.

[44] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[45] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232. ACM, 1993.

[46] Clemens Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming (JFP)*, 15(3):353–401, 2005.

[47] Clemens Grelck and Sven-Bodo Scholz. Accelerating APL programs with SAC. In *Conference on APL '99: On Track to the 21st Century*, APL'99, pages 50–57. ACM, 1999.

[48] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *Int. Journal of Parallel Programming*, 34(4):383–427, 2006.

[49] Clemens Grelck and Fangyong Tang. Towards Hybrid Array Types in SAC. In *7th Workshop on Prg. Lang., (Soft. Eng. Conf.)*, pages 129–145, 2014.

[50] Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. In *5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'78, pages 1–8. ACM, 1978.

[51] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Procs. Workshop Decl. Aspects of Multicore Prog. (DAMP)*, pages 15–24. ACM, 2011.

[52] G. Hains and L. M. R. Mullin. Parallel functional programming with arrays. *The Computer Journal*, 36(3):238–245, 1993.

[53] Mark Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), 2007.

[54] Troels Henriksen. Exploiting functional invariants to optimise parallelism: a dataflow approach. Master's thesis, DIKU, Denmark, 2014.

[55] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. Apl on gpus: A tail from the past, scribbled in futhark. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, pages 38–43. ACM, 2016.

[56] Troels Henriksen, Martin Elsman, and Cosmin Eugen Oancea. Size slicing - a hybrid approach to size inference in futhark. In *Procs. Funct. High-Perf. Comp. (FHPC)*, FHPC '14. ACM, 2014.

[57] Troels Henriksen and Cosmin E. Oancea. A T2 graph-reduction approach to fusion. In *2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, September 2013.

[58] Troels Henriksen and Cosmin E. Oancea. Bounds checking: An instance of hybrid analysis. In *1st ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY'14. ACM, 2014.

[59] Troels Henriksen and Cosmin Eugen Oancea. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 47–58. ACM, 2013.

[60] Stephan Herhut, Sven-Bodo Scholz, Robert Bernecky, Clemens Grelck, and Kai Trojahner. From contracts towards dependent types: Proofs by partial evaluation. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 254–273. Springer Berlin Heidelberg, 2008.

[61] K. Hillesland and A. Lastra. GPU floating-point paranoia. *Proceedings of GP2*, 2004.

[62] Eric Holk, William E Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. Declarative Parallel Programming for GPUs. In *International Conference on Parallel Computing (ParCo 2011)*, 2011.

[63] Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, Inc, May 1962.

[64] Kenneth E. Iverson. Notation as a tool of thought. *Commun. ACM*, 23(8):444–465, August 1980.

[65] C. Barry Jay. Programming in fish. *International Journal on Software Tools for Technology Transfer*, 2(3):307–315, 1999.

[66] Hong Jia-Wei and Hsiang-Tsung Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333. ACM, 1981.

[67] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell workshop*, volume 1, pages 203–233, 2001.

[68] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, and Simon Peyton Jones. Regular, shape-polymorphic, parallel arrays in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*, ICFP'2010, September 2010.

[69] Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. Vectorisation avoidance. In *ACM SIGPLAN Notices*, pages 37–48. ACM, 2012.

[70] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, 100(8):786–793, 1973.

[71] Mads RB Kristensen, Simon AF Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: a virtual machine approach to portable parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[72] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 63–74. ACM, 1991.

[73] Bernard Legrand. *Mastering Dyalog APL*. Dyalog Limited, November 2009.

[74] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55. ACM, 2009.

[75] Frederik M. Madsen and Andrzej Filinski. Towards a streaming model for nested data parallelism. In *2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, September 2013.

[76] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. Exploiting vector instructions with generalized stream fusion. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13. ACM, 2013.

[77] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *Third ACM Haskell Symposium on Haskell*, Haskell'10, pages 67–78. ACM, 2010.

[78] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. *ACM SIGPLAN Notices*, 48(9):49–60, 2013.

[79] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 282–291. ACM, 1997.

[80] Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, USA, December 1995.

[81] Frank Mueller and Yongpeng Zhang. Hidp: A hierarchical data parallel language. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[82] NVIDIA. CUDA C Programming Guide, 2015.

[83] NVIDIA. NVIDIA Thrust Library, 2015. URL: `https://developer.nvidia.com/thrust`.

[84] NVIDIA Research. NVIDIA CUB Library, 2015. URL: `https://nvlabs.github.io/cub/`.

[85] OpenMP Architecture Review Board. OpenMP application program interface version 3.1, 2011. URL: `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

[86] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic book, January 2015. Version 3.2.

[87] David MacQueen Robert Harper and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University Of Edinburgh, March 1986.

[88] Amos Robinson, Ben Lippmeier, and Gabriele Keller. Fusing filters with integer linear programming. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 53–62. ACM, 2014.

[89] John Scholes. D: A functional subset of Dyalog APL. *The Journal of the British APL Association*, 17(4):93–100, April 2001.

[90] Mary Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *Journal of functional programming*, 21(01):59–114, 2011.

[91] Jack Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic computers*, pages 226–231, 1960.

[92] Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 27–46. Springer-Verlag, 2014.

[93] Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 27–46. Springer Berlin Heidelberg, 2014.

[94] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, 2015.

[95] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 65–76, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2594291.2594342`, `doi:10.1145/2594291.2594342`.

[96] Bo Joel Svensson, Ryan R. Newton, and Mary Sheeran. A language for hierarchical data parallel design-space exploration on GPUs. *Journal of Functional Programming*, 26, 2016.

[97] Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. *Implementation and Application of Functional Languages*, pages 156–173, 2011.

[98] Peter Thiemann and Manuel M. T. Chakravarty. Agda meets accelerate. In *24th Symposium on Implementation and Application of Functional Languages*, IFL'2012, 2013. Revised Papers, Springer-Verlag, LNCS 8241.

[99] Kai Trojahner and Clemens Grelck. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming*, 78(7):643–664, 2009. The 19th Nordic Workshop on Programming Theory (NWPT'2007).

[100] Kai Trojahner and Clemens Grelck. Descriptor-free representation of arrays with dependent types. In *Procs. Int. Conf. on Implem. and Appl. of Funct. Lang. (IFL)*, pages 100–117, 2011.

[101] Kai Trojahner and Clemens Grelck. Descriptor-free representation of arrays with dependent types. In *20th International Conference on Implementation and Application of Functional Languages*, IFL'08, pages 100–117. Springer-Verlag, 2011.

[102] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[103] Jeffrey Scott Vitter and E AM Shriver. Algorithms for parallel memory, i: Two-level memories. *Algorithmica*, 12(2):110–147, 1994.

[104] Jeffrey Scott Vitter and E AM Shriver. Algorithms for parallel memory, ii: Hierarchical multilevel memories. *Algorithmica*, 12(2):148–169, 1994.

[105] Michael Vollmer, Bo Joel Svensson, Eric Holk, and Ryan R Newton. Meta-programming and auto-tuning in the search for high performance gpu code. In *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*, pages 1–11. ACM, 2015.

[106] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical computer science*, 73(2):231–248, 1990.

[107] Arthur Whitney. K. *The Journal of the British APL Association*, 10(1):74–79, July 1993.

[108] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.

[109] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 172–187. Springer, 2009.

[110] A. P. Yershov. Alpha - an automatic programming system of high efficiency. *Journal of the ACM*, 13(1):17–24, January 1966.