Implementing an OpenMP backend for the Lift compiler

Federico Pizzuti

Master of Science
School of Informatics
University of Edinburgh
2016

Abstract

The widespread use of parallel hardware has exposed the limitation of traditional low-level parallel primitives, which force the programmer with the task of managing a great deal of incidental, machine-specific details.

This document presents the implementation of a code generator for the functional programming language Lift, capable of producing code targeting OpenMP, a standard parallel platform, derived from a high-language and platform independent specification.

Then, a performance analysis of the generated OpenMP is presented: it is shown how this approach yields fast execution times by comparing it with the already existing OpenCL Lift backend, and how it expresses good scalability by running the same benchmark with a varying number of execution threads.

Table of Contents

1 Introduction **1**

1.1 Overview 1

1.2 Contributions 2

1.3 Thesis outline 2

2 Background **4**

2.1 Overview 4

2.2 Parallel Patterns 4

2.3 Functional Programming 6

2.4 Lift 8

2.4.1 Basics 9

2.4.2 Type System 10

2.4.3 Maps and Reductions 10

2.4.4 Data Reordering Views 12

2.4.5 Operators 13

2.5 toGlobal 13

2.6 OpenMP 14

2.6.1 Basic worksharing constructs 14

2.6.2 The reduction directive 16

2.6.3 The private clause 17

2.7 Summary 18

3 Related Work **19**

3.1 Libraries and Framework 19

3.2 Domain specific languages 20

3.3 Lift specific work 22

3.4 Summary 22

4 Implementation **23**
4.1 Overview . . . 23
4.2 Language extensions . . . 24
4.2.1 Lift's parallel primitives . . . 24
4.2.2 OpenMP as a parallel platform . . . 24
4.2.3 Parallel Map . . . 25
4.2.4 Parallel Reduce . . . 26
4.3 Code generator . . . 27
4.3.1 OpenCL generator code refactoring . . . 27
4.3.2 Basic C generator . . . 28
4.3.3 OpenMP Generator . . . 29
4.4 Runtime . . . 32
4.4.1 Design decisions . . . 32
4.4.2 Parameter passing and result presentation . . . 33
4.4.3 Memory Allocation . . . 34
4.4.4 Executor . . . 35
4.5 Summary . . . 36

5 Evaluation **37**
5.1 Methodology . . . 37
5.1.1 Experimental setup . . . 37
5.1.2 Timing . . . 38
5.1.3 Inputs . . . 38
5.2 Benchmarks . . . 38
5.2.1 Overview . . . 38
5.2.2 Black-Scholes . . . 39
5.2.3 Matrix Multiplication . . . 40
5.2.4 N-Body problem . . . 41
5.2.5 Testing OpenMP Reduction . . . 42
5.3 Results . . . 42
5.3.1 Black-Scholes . . . 42
5.3.2 Matrix multiplication . . . 46
5.3.3 N-Body solver . . . 48
5.3.4 OpenMP parallel reduction . . . 50
5.4 Summary . . . 51

6 Conclusion **52**

6.1 Summary 52

6.2 Critical analysis & limitations 53

6.3 Future work 54

Bibliography **55**

List of Figures

4.1 Overall structure of the Lift compiler pipeline using the OpenMP backend 24
4.2 Schematic overview of the runtime generation and execution steps . . 32
4.3 Schematic overview of execution of compiled Lift code from Scala . . 35

5.1 Comparison of running times for Black-Scholes benchmark generated from OpenCL, OpenMP and Sequential backends 44
5.2 Strong-scaling profile of generated OpenMP implementation of Black-Scholes when varying number of execution threads 45
5.3 Comparison of running times for Matrix Multiplication benchmark generated from OpenCL, OpenMP and Sequential backends 46
5.4 Strong-scaling profile of generated OpenMP implementation of Matrix Multiplication when varying number of execution threads 47
5.5 Comparison of running times for N-Body solver benchmark generated from OpenCL, OpenMP and Sequential backends 48
5.6 Strong-scaling profile of generated OpenMP implementation of N-Body solver when varying number of execution threads 49
5.7 Comparison of running times for the square-sum benchmark, implemented with a single native ReduceOMP vs the composition of ReduceSeq o MapOMP 50

Chapter 1

Introduction

1.1 Overview

Current hardware trends are moving the computing industry more and more towards the use of heterogeneous and parallel architectures, motivated by their greater efficiency and higher theoretical performance. Software however still operates largely along the same principles of the last decade: as a result, programs that wish to fully exploit the capabilities of this new generation of parallel hardware are needlessly complicated by platform specific implementation details. Moreover, simply coding in more expressive languages does not necessarily help: compilers can rarely automatically parallelize an arbitrary block of code, and when it is done, the results tend to be far from being optimal (Armstrong and Eigenmann, 2001).

As always in Computer Science, dealing with this complex challenge requires the introduction of a suitable abstraction. In this case, parallel patterns can be a useful tool to structure and simplify parallel programs. The fundamental idea is to identify a set of fixed templates for algorithms, each specifying the parallel features but not the domain specific details, which are supplied by the user. Several such patterns have been identified(González-Vélez and Leyton, 2010).

The issue then becomes how to best utilize the parallel patterns abstraction. While a variety of different methods have been tried, this document focuses on Lift, a functional programming language modeling parallel patterns as higher order functions. The use of functional programming allows to support a high-level declarative style, so abstract that any program written on it is independent of the parallel system used to run

it, yet structured enough to enable automatic parallelization by the compiler.

1.2 Contributions

The project detailed in this document is composed of the following contributions:

- A Lift code generator producing standard sequential C code, to be derived from the previously existing OpenCL backend, to act as a fundation for more complex backends and as a baseline for testing purposes.
- A Lift code generator producing C code for the OpenMP platform, to be built on top of the C generator.
- A compile time "Harness generator" producing a C wrapper for the output of the above mentioned backends, using compile-time information to correctly setup the parallel computation, including performing memory allocation and data marshalling.
- Scala code to automate the above steps, allowing to seamlessly compile and call Lift code at runtime.

1.3 Thesis outline

The rest of the document is divided in the following chapters:

- Chapter 2 - Background: presents an overview of the concepts touched by the project. It focuses mainly on fours areas: Parallel Patterns and their role in the world of parallel programming, Functional Programming and how it can be use to best express parallel patterns, a quick tour of the syntax and salient aspects of the Lift language and finally an explanation of the parts of OpenMP which are used by the generated code.
- Chapter 3 - Related Work: a review of the available literature, cataloging works related to the project here described. This includes a range of different ways of implementing parallel patterns, including: library based approaches, distributed computation frameworks, avant-garde attempts such as mondaic computation and more Lift-like attempts, such as parallel DSLs and DSL generators. The chapter then lists the relevant literature about the Lift language in particular.

- Chapter 4 - Implementation: explains in greater detail the design of the code implementing the various contributions proposed by this project. It starts by explaining the methods used to translate Lift parallel primitives into OpenMP code. Then, the individual software components are examined bottom-up: starting from the C generator base, then moving to the OpenMP generator specifics, the Runtime system and finally Scala-to-Lift interoperation.

- Chapter 5 - Evaluation: attempts to establish the quality of the OpenMP code generator by presenting a series of experiments comparing the running times of its compiled Lift programs with those generated by the sequential and OpenCL backends, and examining the scalability of the produced software with increasing numbers of parallel processing units.

- Chapter 6 - Conclusion: the closing chapter includes an overall summary of the work done, a critical analysis and limitations sections highlighting the shortcomings of the adopted approaches, and finally a future work section proposing additional improvements, some of which addressing those problems.

Chapter 2

Background

2.1 Overview

This chapter presents and overview of the concepts touched within the project. It starts by looking at parallel patterns as a way to structure parallel programs. It then covers functional programming in general, and the connection it holds with the world of parallelism. Then, it gives a closer look at the details Lift programming language, and how it brings the two worlds together. An introduction to OpenMP, the parallelism platform target by this computer, closes the chapter.

2.2 Parallel Patterns

Writing parallel programs is a complex task. Much of traditional software programming is based around a single-threaded sequential environment, which can be quite comfortable to use due to its deterministic nature. This situation breaks down when one starts to operate with multiple execution threads: the non-deterministic order of operation leads to unpredictable results, and many of the traditional algorithms and abstractions might not be compatible with this new paradigm. To complicate matter further, in most programming languages using parallel features means dealing with a plethora of low-level details concerning the implementing machinery: locks, tasks, messages and threads are some of the most famous examples.

Measures can be taken to improve accessibility (and quality) of parallel programming, and as always in computer science, a new system of abstractions is needed to help manage complexity: Parallel Patterns(McCool et al., 2012). The fundamental idea is that

many parallel programs can be broken down to element which come from a limited set of templates, just with some small application specific change. Each of these templates - a parallel pattern - therefore can be thought of as an algorithm parametrized by one or more domain specific functions, which fill in the details. And, by imposing certain constraints on these parameter functions, it is possible to guarantee some invariants, the most important of which is to guarantee that the result can be parallelized without further work.

One of the most famous parallel patterns can act as a perfect example of the above: the transformation of a collection of elements into a new one of the same size and shape wherein each element is a function of another. Moreover, this transformation is executed in parallel, according to the nature of the data structure one maps over - in case of an array-like structure, this means that each element can be computed in parallel. This pattern is usually called Map, although it can also appear under the name of "Transform". It accepts one specialization parameter, a function that takes an object of the element type within the original collection, and returning a new object of the target's element type. Note that this parameter function needs not be parallel, or know anything about parallelism: it is the Map implementation which deals deal with the details.

One might then wonder how many parallel patterns are needed to express the majority of parallel programs, for if it is a great number, we have merely replaced a complicated abstraction with another. It turns out however that only a handful of parallel patterns are needed: the above mentioned Map for collection-to-collection transformations, and the Reduce pattern (collapsing a collection into a single value by repeated application of a pairwise operation) are enough to form a very expressive language, and can therefore be a good choice of primitives in any parallelism system which wants to relies on the pattern approach (the renowned distributed computing framework MapReduce adopts precisely this approach).

While Map and Reduce offer a very good theoretical basis, sometimes real world software needs require other kinds of parallel operations, such as Scattering and Gathering, which do not deal with data transformations but with re-organizing the work within the parallel structure. For example, one might want to explicitly scatter a large array before mapping over it to control the amount of work each thread has to perform, perhaps im-

proving memory access behavior. Systems relying on parallel patterns might support or not these operations depending on the level of abstraction they desire to obtain.

Finally, there are some very sound software engineering reasons to rely on a pattern based approach: Having a small handful of patterns to implement, system designers can dedicate a lot of time and attention to each, producing some highly optimized and tested implementations.

2.3 Functional Programming

The previous section has explored the merits of relying on a pattern-based approach to simplify and improve quality of programs using parallel features. The question then becomes: what are the features that a programming language should have to fully take advantage of this method?

The first requirement is that a language should be able to comfortably pass around behaviors, as procedures implementing parallel patterns needs to be parametrized with the program-specific code. Secondly, the language should have the ability to enforce properties of the behaviors passed around, in order to ensure the correct working of patterns. Finally, the language should easily allow to compose patterns together, in order to easily construct more complex parallel algorithms form the very small set of selected primitives.

All these features can be found in functional programming languages. These languages are structured quite differently from the more common imperative style. They are expression oriented languages, where values are usually immutable (and in some cases, entirely so). In these languages, programs are decomposed in pure functions: procedures which are only allowed to express their outputs in terms of their inputs (just like mathematical functions). Moreover, functions are treated as first class values: they can be computed at runtime (generally in a concise style similar to the abstractions of lambda calculus) and passed/returned from functions (functions that operates on other functions, known as higher-order, are in fact a common sight in functional programs). Finally, while not strictly being required, most functional programming languages feature rich static type systems.

Adopting a functional paradigm therefore has several important implications:

- The ease of use of first-class functions means that it is natural to implement parallel patterns as higher-order functions which, when applied to specific-behavior functions, yield the concrete parallel algorithm to be applied directly on the data. Moreover the strong type system ensures that, if a functional program compiles, then the parallel patterns formed are valid.

- Functional purity and immutability of values allows to rule out common bugs associated by parallelism: race conditions are not a problem if one acts on immutable data, and the use of pure functions ensures that the order of computation is somewhat more flexible - making the program safe in the presence of non-deterministic execution order.

- Reliance over composition over sequencing encourages the structuring of the program as a pipeline of parallel algorithms operating over the data. This approach is particularly appropriate for many computations which can make use of parallelism as a way of achieving a speedup. Moreover, unlike sequencing, composition is type-checked, allowing to catch some more errors at compile time.

To examplify the benefit of using parallel patterns with a functional style over the more common imperative paradigm, compare these equivalent dot product implementations, one a Lift-like snippet, the other a handwritten OpenMP C program.

Listing 2.1: Lift implementation

```
(input1, input2) ->
 Reduce((x,y) -> x + y) o
   Map(x -> x._1 * x._2) $ Zip(input1,input2)
```

Listing 2.2: OpenMP implementation

```
float input1[N] = ....
float input2[N] = ....
float intermediate[N];
float result;
//The map
#pragma omp parellel for
for(int i = 0; i < N; i++)
{
   intermediate[i] = input1[i] * input2[i];
```

```
}
//The reduce
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < N; i++) {
        result += intermediate[i];
}
return result;
```

Consider now the general shape of both programs, without being concerned too much with details. While the size difference is clearly the most striking feature, it is not the most significant part. The attentive reader might point out that the C program does not have Map and Reduce functions, but must instead spell out the for loop, and it is this deficiency that causes the code to be longer. However providing a Map and Reduce for C would not help much: these functions would expect function pointer as parameters, and the addition and multiplication operation would therefore need to be defined somewhere else, yielding to similar program length with added complexity. Therefore, it is simply not viable to adopt a pattern based approach in a language without lambdas.

Moreover, there are other advantages in adopting the functional way: since all iteration is handled implicitly by the pattern functions, the programmer must never specify array lengths or iteration variables, which are a common source of bugs. Nor is there any need to explicitly declare the variables holding the intermediate steps and final result: they are implicit in the expression based structure. And finally, if we were to swap the order of the mapping operation with the reduction one, the functional version would stop working due to typing problems.

2.4 Lift

The source language of the compiler this project is part of is Lift, a functional programming language for expressing data parallel programs. It attempts to leverage the benefit of functional programming expressed in the previous section to enable users to write high-performance programs at a high level of abstraction, across a variety of platforms, including CPUs and GPUs.

2.4.1 Basics

Before looking at the details of the Lift language, it is beneficial to have a high level look at the structure of the Lift language and compiler.

A Lift program is fundamentally composed of three different kinds of elements. First, there are the pattern primitives, expressing parallel algorithms and data reordering as functions. Secondly, there are "user functions", snippets of C code expressing specific functions on data, such as mathematical operators. Finally, there are language level operators such as lambda abstractions, function composition and application, used to glue together the program in a purely functional way.

In order to both guarantee a high level of abstraction when desired and control over implementation when needed, Lift's primitives are structured in a layered way, so that for some generic Lift primitive there may exist many low-level versions, exposing platform specific details. The compiler then translates the high level primitives into the appropriate low level ones.

While in theory Lift could be agnostic in terms of the language used as a compilation target, in practice and through this document this is assumed to be C by simplicity. This happens to be the target language both of the pre-existing OpenCL generator and of this project's addition, the OpenMP generator.

About the compiler's structure and implementation, Lift is currently embedded within the larger language Scala. Lift code is currently implemented in terms of Scala objects, and Lift expressions are therefore representation of the matching abstract syntax tree in Scala. In this regard, Lift is not dissimilar from a very heavyweight DSL. This eliminates the need of writing a front-end proper. As a side effect, programs with invalid structures are rejected by the Scala type system when they are first written.

Following this first step, a system of optimization based on rewrite rules is applied on the program to generate the low-level primitives from the generic ones. After that, the abstract syntax trees are given to a type checker, which annotates it with the inferred types. Finally, the annotated expression is passed to a code generator, which produces code for a given target platform.

2.4.2 Type System

Lift is a statically typed language, and its type system revolves around a few main elements:

- Scalar types: these act as the bottom entities from which more complex types follow. They mirror C primitive types.
- Vector types: represent fixed sized vectors or scalar types.
- Tuple types: fixed arity heterogenous tuples, built from any other type. The language tracks the length and the type of each element. For each tuple type, a C structure is generated as an underlying representation
- Array types: fixed size homogenous collection. The type system tracks both element type and length, which can ultimately be a fixed constant or a compile-time variable resolved at runtime. Arrays can contain anything, including other other arrays. Within this document, array types are noted by square brackets, wrapping around the element type and size.

The notable feature here is the tracking of array size in the type. This has implications in various areas of the compiler: it allows for the indexes resolution detailed in 2.4.4, and for the memory allocation techniques covered in 4.4.3.

2.4.3 Maps and Reductions

The two core parallel patterns previously identified appear in Lift under their canonical names of Map and Reduce, and are provided both high level and low level. Moreover, their low level versions include both parallel and non-parallel forms. While this might be surprised at first, one can see that each parallel pattern has a matching sequential implementation with the same interface and yielding the same results. Reasons for providing such versions include having a simple baseline to compare parallel algorithms with, offering a "last line" implementation which is sure to work correctly even if the target infrastructure is, for some reason, not able to parallelize some code, and finally as an explicit low-level choice for applying operations when the data size is so small that the parallel versions would be overall slower due to coordination overhead. This document only refers to two pairs of low-level patterns: MapSeq, MapOMP. ReduceSeq and ReduceOMP, respectively the sequential and parallel version (through OpenMP).

Generic Patterns		
Map	$(\alpha \to \beta) \to [\alpha, N] \to [\beta, N]$	Generic mapping
Reduce	$((\beta, \alpha) \to \beta) \to \beta \to ([\alpha, N] \to [\beta, 1])$	Tree-parallel reduction
Tuple	$\alpha_1 -> ... -> \alpha_N -> (\alpha_1, ..., \alpha_N)$	Tuple constructor
Get	$(\alpha_1, ..., \alpha_N) -> \text{int} -> \alpha_K$	where K is the integer value passed
Transpose	$[[\alpha, N], M] -> [[\alpha, M], N]$	Inverts dimensions
Sequential Patterns		
MapSeq	$(\alpha \to \beta) \to [\alpha, N] \to [\beta, N]$	Sequential mapping
ReduceSeq	$((\beta, \alpha) \to \beta) \to \beta \to ([\alpha, N] \to [\beta, 1])$	Sequential reduction
OpenCL Patterns		
MapGlb	$(\alpha \to \beta) \to [\alpha, N] \to [\beta, N]$	Parallel mapping through global items
MapWrg	$(\alpha \to \beta) \to [\alpha, N] \to [\beta, N]$	Parallel mapping with workgroups
toGlobal	$\alpha -> \alpha$	Used for explicit memory control

Table 2.1: A selection of Lift primitives

2.4.3.1 Map

As stated before, Lift implements the Map pattern as an higher order function with type:
$(\alpha \to \beta) \to ([\alpha, N] \to [\beta, N])$
That is, given a function from some type α to β, it returns a new function from an Array containing elements of type α of size N, constructing a new Array of the same size N, containing elements of type β. For example, if we have a function *floor* of type float $\to$ int, and an array of ten floating point numbers *nums* of type $[\text{float}, 10]$, it is possible to compute the array of floored numbers (of type $[\text{int}, 10]$ with the Lift snippet:

Listing 2.3: Simple map example

```
Map(floor) $ nums
```

(The difference of treatment for the floor parameter, passed within parenthesis, and the numns parameter, passed by application with $, is due to the current implementation of Lift: conceptually both parameters can be treated equally).

An implication which might not be immediately apparent is that Map, being itself a function, can be passed to another instance of itself. This is very useful for iterating over nested arrays: this lift program for example can be used to floor every entry of a

matrix

Listing 2.4: Nested maps

```
Map(Map(floor)) $ matrix`
```

2.4.3.2 Reduce

Reductions are also implemented in a similar way, with a function of type:
$((\beta,\alpha) \to \beta) \to \beta \to ([\alpha,N] \to [\beta,1])$
The first argument is a binary function combining two items into one, the second parameter is a starting value, which is combined with all the elements in sequence, until there are no more elements left. The binary function need not combine two elements of the same type: merely one of the accumulator type and one of the element array type.

Logically, a reduction operation results in a single element type. In Lift however the preferred approach is to have reductions return a single-element array. This is in tune with Lift's data-parallel processing nature: arrays are very natural elements to the lift ecosystem, while scalar values are not necessarily so (additionally, doing so simplifies the implementation: operating exclusively on arrays allows to operate in a uniform way).
A standard example of Lift reduction is to sum up all elements in an array. The following snippets demonstrates

Listing 2.5: Simple reduction example

```
Reduce(add, 0.0f) $ nums
```

The reduction is initialized with the constant 0.0f, the neutral element of addition. This is a common practice when using reductions.

2.4.4 Data Reordering Views

The remaining lift primitive patterns deal with transformations which do not modify the contents of arrays or tuples, but either project out data from composite structures, or that modify the shape of such structures. Here is a quick overview:

- Get_N, Head and Tail: Used to extract values from composite structures: Get_N projects out the Nth entry of a tuple. Head and Tail are used to deconstruct an array in a first element/remainder way.

- Zip and Unzip: Zip takes two arrays of identical length and produces an array of pairs (Tuple_2), matching element one-by-one. Unzip reverses the operation
- Split_N and Join:Split_N takes an array and groups elements in sub-arrays of size N. Join reverses the operation, concatenating every 1st level element in a nested array, and eliminates the redundant array dimension.
- Transpose: considering a two dimensional array as a row-column structure, this operation regroups the elements such that rows and columns are swapped.

2.4.5 Operators

Finally, these are the hard-coded operators of the Lift language, used to structure the computation.

- $ operator: an infix operator performing functional application, of type
  $(\alpha \to \beta) \to \alpha \to \beta$
  to be used every time one wishes to pass concrete values into Lift functions.
- $\circ$ operator: an infix operator implementing function composition. Of type
  $(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$ to be used as the main tool for constructing programs.
- fun operator: a lambda abstraction operator, used for creating anonymous functions. Every Lift program is ultimately wrapped into a fun structure.

2.5 toGlobal

A closing mention should go to the toGlobal function, which appears quite a lot in the benchmarks code shown later in the evaluation function. This function performs no actual computation and should almost considered as a compiler directive, used to explicitly influence memory allocation. The Lift compiler infers which variables are to be stored in global memory, accessible to all execution threads but slow to access due to the synchronization required, and private memory, which is replicated for each thread and is much faster to access.

Clearly, the final results of each program should be stored in global memory (this

is in fact enforced by Lift at compile time). To achieve this, many of the benchmarks have, as a last and outermost operation, some function of the form

Listing 2.6: Enforcing global memory

```
        toGlobal(MapSeq(id)) o ...
```

The MapSeq(id) operation is semantically a no-operation: id is the function that returns and element unchanged, and mapping it over an array has a similar effect at the collection level. However, since it is embedded in a toGlobal call, this has the actual effect of ensuring that the final results are going to be placed, unmodified, in global memory.

2.6 OpenMP

OpenMP is the parallel platform used as a compilation target in the project described by this document. It employs a shared-memory parallel model, and it supports the C, C++ and Fortran programming languages. Whatever the language, most OpenMP features are not accessed directly using language level features such as function calls, but are instead encoded in suitable placed comment, known as pragmas. These are then processed by the compiler, which generates the parallel code accordingly. By utilizing this system, the OpenMP authors are free to structure the API as they wish. Moreover, since most pragmas act as modifiers of already valid sequential code (as shown by the examples below), some OpenMP programs have the pleasant side effect of also being valid if the library is not supported by the system: the pragmas are simply be ignored, yielding a correct traditional program.

OpenMP is quite a large library, supporting various forms of parallelism at different levels of abstraction. This document is only concerned by the features employed in the project: the more commonly used higher-level abstractions.

2.6.1 Basic worksharing constructs

The most basic way of expressing parallelism in OpenMP is through the use of the fork-join "worksharing" constructs. These pragmas are used to define regions of code which the runtime is free to parallelize. The simplest form is accessed by using the **parallel** pragma. Consider the following C snippet

Listing 2.7: Parallel pragma

```
int x = 0;
#pragma omp parallel
{
  x++;
}
printf("%d",x);
```

In the above programs, the lexical scope limited by the braces is parallelized. In particular, the OpenMP runtime allocates a certain number of threads, each incrementing the variable x. No explicit synchronization code is necessary: access to the shared variables is controlled by the runtime, avoiding therefore unpredictable results - the variable x is incremented exactly once per thread. Then, upon exiting the lexical scope, all threads but the main one are be killed, and the program returns to normal sequential execution. This model of computation is known as fork-join.

Notice however that the above pragma allows minimal control over the parallelism: the user has no control over the number of threads employed, and each thread executes the entire block of code. As such, this approach is only of limited use. One can however make additions to this basic parallel pragmas, called directives, in order to specify certain properties of the parallel computation.

One of the most common directives, used extensively within this project, is the **for** directive. It informs the OpenMP runtime that the parallel section following it is not just a generic block, but instead a C for loop, and that the loop needs to be parallelized in a way which preserves the total number of iterations.

Listing 2.8: Parallel for

```
int input[N] =...;
int output[N] = ...;
#pragma omp parallel for
for(int i = 0; i < N; i++)
{
  output[i] = i*i;
}
```

The behavior is simply the expected one: the output array of squares is computed from input in a totally parallel way. Just like the simple **parallel** pragma, the runtime retains

control over the number of threads and the precise work subdivision.

2.6.2 The reduction directive

A very common operation in parallel programs working with numbers is to take an array, apply transformations to each element, and in the end combine all the elements in one result value. As such, the OpenMP library offers a directive aimed at simplifying the implementation of such algorithms. One can add the **reduction** clause to any parallel region to specify that, before exiting the scope, a certain variable is to be used as the target of a reduction operation. An example can be used to better clarify:

Listing 2.9: Reduction

```
float data[N] = 0;
float sum;
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < N; i++)
{
  sum += sqrt(input[i]);
}
```

The above program sums up all the square roots of the numbers contained within the input array. The computation of the square root happens in parallel as specified by the **parallel for** clause. As for the summing up, the reduction clause specifies that the variable **sum** is to be the accumulator.For each thread executing in the parallel region, a private copy of sum is initialized by the runtime. It is to this copy that the thread sums up a certain number of input entries, depending on how the library schedules the parallel for loop. Then, when exiting the parallel section, the various private copies of sum are combined using the operator specified in the reduction clause - in this case, addition - restoring the original semantics of the program, with only one sum variable existing.

While this is definitely the most common usage of the reduction clause, one could use it in more creative ways. For example, there is no particular limit to what happens within the body of the loop: instead of incrementing sum, one could multiply it by the square root of the input values. This would yield a program whose final value of sum depends on how the for loop has been scheduled. As another example, one could employ a reduction without the presence of the for clause, but just with basic parallel regions. Experimenting with this forms, while of limited practical use, can be very

helpful in understanding how OpenMP reductions work.

2.6.3 The private clause

By default, the OpenMP runtime determines which variables are to be shared between threads, and which variables are to be duplicated for each thread, by looking at their scoping: every variable declared inside a parallel region is assumed to be private to the threads living within that region: copies are made, and no synchronization code is produced. Variables that are declared outside of the lexical scope are assumed to be shared, and the runtime synchronizes access to them.

This behavior is quite natural and often the desired one. There are however cases where these rules do not necessarily hold. In fact, one of such case is covered in the previous section: the target variable of a reduction is declared outside the parallel section, yet it is treated as a private variable.

Sometimes however the user might wish to obtain control over this process. An example of this is shown later in this document: many threads need to obtain read-only access to a variable holding a runtime constant. This variable is however loaded and initialized before the parallel computation takes place, and therefore declared in the outside scope. Without further corrections, OpenMP considers the variable shared, and therefore serializes accesses to it, which in the case documented here actually turned a theoretically parallel loop into a de facto sequential one. Another use case is found when employing older versions of C, which force all variables to be declared before any other statement in a function.

To solve this false sharing problem one can use the **private**, passing in the list of variables which must be made private in the following parallel section, as in the following snippet, mimicking the one counting the number of threads used in a parallel section, but using a constant to specify the amount to count by:

Listing 2.10: Private variables

```
int one = 1;
int x;
```

```
#pragma omp parallel private(someConstant)
{
  x += one;
}
printf(%d, x);
```

2.7 Summary

This chapter has presented an overview of the concepts which lay at the fundation of the project. It has first covered Parallel Patterns as a generic method to structure parallel computation and how the functional programming paradigm can offer a very convenient realization of them. A quick tour of the Lift programming language follows, and then the chapter concludes with an overview of the OpenMP constructs used to implement Lift's parallel primitives.

Chapter 3

Related Work

This chapter presents a review of the research in the field of parallel programming, showing the different approaches to parallelism adopted, and focusing in particular on those attempts which rely on designing appropriate language level abstractions.

3.1 Libraries and Framework

The first part of the review focuses on those attempts of tackling the problem of simplifying the writing of parallel computation without introducing a new programming language or a domain-specific language, but directly, structuring themselves as software libraries or frameworks, to be use idiomatically within the hosting language.

One of the older and most established example of exposing parallel patterns through library support is the Thread Building Blocks (TBB) library from Intel (Reinders, 2007). This C++ library provides function templates implementing parallel algorithms, such as “parallel_for” and “parallel_reduce”.
TBB is an excellent example for analyzing the virtues and pitfalls of using a language library approach: on one hand, there is easy integration within a larger C++ program: the template functions can just be invoked seamlessly, and working within the same language means there are no problem with data format. However, some inconvenience is added for the programmer: in this case, the parametric behavior expected by TBB requires extra boilerplate code to be specified. Moreover, the user must be sure that the arrangement of the parallel computation as a whole is correct, as the library cannot help in that regard.

Another approach that has become popular is to provide a fixed runtime structure, and have user code, written in a general purpose programming language, fit some specified interface. It is then this outer shell that takes care of structuring the entire computation. This approach has been made popular by distributed computing frameworks such as MapReduce(Dean and Ghemawat, 2008): this approach is particularly suited for their domain, as the infrastructural code is much more complex then in the case of local parallelism. The clear limitation of this approach is the inability to vary much the structure of the computation, sometimes forcing the user to express their program in an unidiomatic way.

Yet another method for simplifying parallel coding without necessarily relying on any explicit notion of parallelism is to provide the user with a set of high performance parallel primitive operations. For example, the Intel MKL math library includes a variety of hand-optimized routines calculating commonly used algorithms (such as Fast Fourier Transform), some of which have a parallel implementation. Clearly, this approach is only suitable for popular niches but cannot scale for general purpose computation (Wang et al., 2014).

Not all efforts are directed to achieve data-parallelism. Task-level parallelism, while operating at a coarser level, is useful for structuring a parallel computation differently. Message passing frameworks, such as OpenMPI(Graham et al., 2006), are particularly well suited for this kind of parallelism.
Finally, the increasing popularity of functional programming languages have increased the possibility space for developing new abstraction. An example of these possibilities is the parallelism library proposed in (Marlow et al., 2011), where instead of using traditional parallel patterns, monadic combinators are used to construct the parallel program.

3.2 Domain specific languages

The second part of this review focuses on techniques which go beyond simply being a software library or framework, and instead act on the more fundamental code generation level.

A first example is the one represented by the Copperhead language project(Catanzaro

et al., 2011). The idea behind this endeavour is to identify a subset of the Python language whose behavior can be easily predicted at compile time (restrictions include things like forcing each expression to return a given value). Then, this python subset is enriched with functions implementing parallel patterns: for the programmer, working with Copperhead is not too dissimilar then TBB, only using a more pleasant higher level language. Copperhead programs however are not executed like conventional Python scripts: they are instead fed to a compiler, which targets the CUDA platform.

Another interesting project is Delite (Sujeeth et al., 2014), a DSL generation framework for data parallel languages. The backing idea is that even traditional pattern-based parallelism might result too generic for total ease of use, at it is still require explicit mention of the parallel features. Instead, by using Delite it is possible to generate domain specific languages which takes care of almost all aspects of parallelism, for example an arithmetic-like language operating on vectors whose basic mathematical operators are automatically parallel.

In (Eijkhout, 2014), a peculiar parallel DSL is presented which is based on the theory known as "Integrative Model of Parallelism". This theoretical model allows the user to express parallelism in terms on how data is to be distributed and combined, in an highly general way. The DSL system can then infer dependencies from the supplied description, and generate correct parallel code from that. The claimed benefit of this approach is that it is very suitable for expressing programs where the parallel form is more complex then the traditional map or reduce, such as for example combining elements arranged in a "neighborhood" of the array, or when some more complex pairing is needed.

Finally, one can find that work has been performed in providing parallel DSLs which are dedicated to a particular application type (this is somewhat like the case of the Intel MKL math library covered before, only at the language level rather then at the library level). Examples of this can be found in the fields of Machine Learning(Sujeeth et al., 2011) and Computer Vision(Chiw et al., 2012).

3.3 Lift specific work

The last part of this review concentrates of notable work done regarding the Lift language and its OpenCL backend. An overview of the compilation process and fundamental ideas behind Lift can be found in (Dubach et al., 2012). The Lift rewrite rule system, used to derive lower-level constructs from high level functional primitives, is explained in detail here (Steuwer et al., 2015). Finally, an analisys of the performance of various matrix multiplication implementations generated by Lift on a variety of computing platforms, including CPUs and GPUs, can be found in (Dubach, 2016).

3.4 Summary

This chapter has offered a quick tour of the works concerning automatic parallelization through the use of parallel patterns. It has started from conventional library-based approaches and progressively moved towards more pervasive changes, such as devising parallel DSLs. It has finally concluded by mentioning literature concerning the core aspects of Lift, the parallel processing language extended in this project.

Chapter 4

Implementation

4.1 Overview

This chapter describes the design decisions and implementation details of the software proposed in this project. Before covering each component in depth however, it is useful to see the entire program at a higher level of abstraction. (TODO: here goes a picture of the various components interacting)

The first component of the back-end's pipeline is internally called the "code generator". It receives an object representing the abstract syntax tree of a Lift program , and it produces a C function implementing it.

This C implementation however does not take care of memory allocation, input parsing or output. Therefore, an "harness generator" is coded around it. This produces a stand alone C program, which takes care of input handling, memory allocation, timing (for evaluation purposes) and outputting results.

Finally, the project includes code for easily invoking the compiled program, and automatically convert native Scala values in the required input.

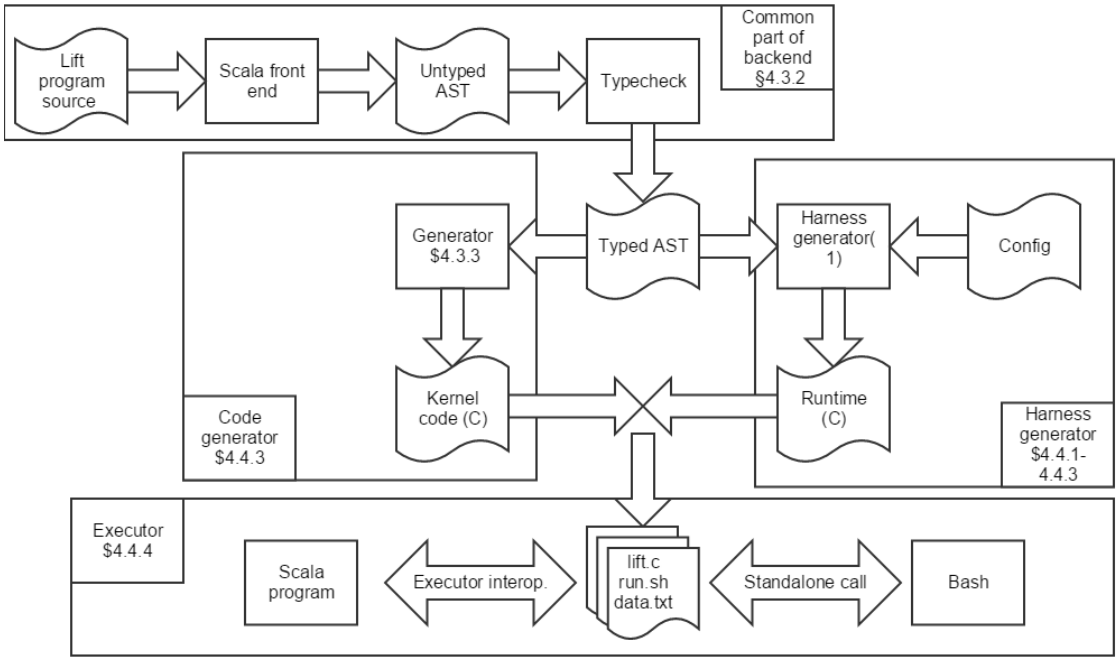

Figure 4.1: Overall structure of the Lift compiler pipeline using the OpenMP backend

4.2 Language extensions

4.2.1 Lift's parallel primitives

Lift is a functional language for expressing parallel algorithms. A typical Lift program is structured as the composition of a series of primitive functions, some implementing parallel algorithms(mapping, reducing, etc), and other implementing data-reordering transformations (such as zipping together two arrays to generate an array of pairs). Finally, algorithms that are specific to the program (such as particular operations on scalars) are expressed directly in C by the programmer, resulting in "User Functions". Since no code is actually generated for the data reordering (all arrays are of known size at runtime,therefore it is possible to track the transforms on the indexes and generate the correct array access expressions without the need of actually computing intermediate collections), one needs only to generate code for the parallel patterns to have a working implementation.

4.2.2 OpenMP as a parallel platform

It is therefore necessary to find a way to express Lift's parallel primitives in terms of OpenMP, the target platform of choice for this project. OpenMP is quite a large library,

and has both high level and low level constructs. The choice for this project has fallen on the higher level ones, as they offer a very natural and straightforward mapping for the functional patterns employed by lift.

In particular, since all the required primitives iterate over an array element by element, the choice has fallen on the **omp parallel for** family of OpenMP primitives, which parallelizes a C for loop statically dividing the work among available worker threads.

4.2.3 Parallel Map

The first parallel primitive of Lift is the ubiquitous Map function, which transforms an array by applying a given function to each element. The natural implementation for this operation in C is a for loop, setting each element of the target array to the value of the matching element in the source array, transformed by the supplied function. This for loop can then be parallelized in OpenMP through the use of the **omp parallel for** directive. Therefore, we can implement the following lift statement, mapping the function f over the array called xs, is translated as follows

Listing 4.1: Lift code

```
MapOMP(f) $ xs
```

Listing 4.2: OpenMP implementation

```
#pragma omp parallel for
for(int i = 0; i < N; i++)
{
        target[i] = xs[i];
}
```

The reader may wonder where do the target and N variables come from, as they do not appear in the original expression. These are resolved by the Lift compiler during code generation: every expression has a memory region where the result is stored, which accounts for target. As for N, the Lift type system tracks size as part of the array type, and therefore N is always known at compile time.

4.2.4 Parallel Reduce

The other parallel pattern implemented is the Reduce function, which provided with a binary operation an a starting value, it accumulates the entries of an array, yielding a

single result.

Unlike mapping, reducing is not strictly speaking a primitive operation: provided with an implementation for MapOMP, there is a general way to produce a parallel reduce implementation, based on the notion of tree parallelism. However, the OpenMP platform provides native constructs implementing such parallel reductions, and a design choice was made to also provide Lift with direct access to the native implementation.

Once again, the implementation of ReduceOMP here proposed is built over a C for loop by prepending the directive **omp parallel for reduce(*op*:*acc*)** to the generated sequential code. Therefore, we have the following translation

Listing 4.3: Lift code

```
ReduceOmp(+:(Float),0.0f) $ xs
```

Listing 4.4: OpenMP implementation

```
acc = 0;
#pragma omp parallel for reduction(+:acc)
for(int i = 0; i < N; i++)
{
        acc += xs[i];
}
```

As it can be seen in the above listings, the pragma specifies which operation is to be applied. This operation cannot however be any binary function, but can only be drawn from a fixed set (+,-,*,/,min,max and average). However, the Lift type system has no notion of this limitation, and therefore allows any operation with the correct signature to be supplied to a reduction operation, resulting in programs that typechecks correctly but that cannot be compiled.

The solution adopted in this project is to introduce a new class of entities, called reduction operators, each one representing one of the parallel reduction. These objects subtype Lift's functions, and are implemented under the hood as hard-coded blocks of C code, specifying the body of the for loop required for each operator, and works seamlessly within the rest of the language (it is possible, for example, to employ the parallel operator **+:** whenever one would use regular addition. In this case, the operator is parametrized by the Lift type it has to operate on. This eliminates the need for

having differently named versions for each element type).

Then, the reduce parallel constructor no longer accepts any function of the correct type, but a reduction operator. Therefore, the Scala typesystem ensures that only allowed combination can be produced. This approach however has the drawback of introducing a different kind of entity which is transparent to the Lift typesystem, and only works as a side effect of the implementing technology.

4.3 Code generator

Having seen how each parallel pattern is mapped onto it's own OpenMP implementation, it is now time to look at the details of the code generation module as a whole. This is the part of the compiler which is responsible for the generation of C code implementing a given Lift function. The code generator itself is split in two main sub-modules: the first producing a C-subset abstract syntax tree from the supplied program, and another one printing the C source code from it.

4.3.1 OpenCL generator code refactoring

At the time this project was started, the Lift compiler already had a functioning code generator, targeting OpenCL instead of OpenMP. Since OpenCL is also a parallelism library for C, the opportunity was present for significant code reuse. However, several modification were needed: OpenCL programs have an unconventional structure and use a large amounts of non standard C concepts and keywords, making it incompatible with the OpenMP generator.

In addition to this purely technical issue, there are software engineering concerns to be raised by the addition of a secondary backend. The OpenCL code generator is written as an independent unit without regard for the existence of any alternative generator, which means that the implementation is full of OpenCL specific details all over, including the reusable parts.

It was therefore decided, in order both to reuse as much code as possible and to better equip the compiler for the addition of further back-ends, to generate a basic C generator, acting as an abstraction layer, over which to build the OpenMP module and,

in the future, the OpenCL one as well. This C generator, detailed in the following subsection, has been obtained by "purging" away OpenCL specifics from the original implementation, while at the same time retaining the same structure and interface in order to maximize reuse and simplify any future merger. Therefore, in the following look at the details of this module, one must bear in mind that the bulk of the work was already present at the beginning of this work.

4.3.2 Basic C generator

The C generator, while implemented as a Scala object, can really be seen as one large function, accepting typecheked Lift abstract syntax trees and producing C code. Upon the parameter expression, the following steps take place:

- Adress space inference: computes whether the expression is acting on Global or Local memory. While the difference between these two types of memories is non-existent outside of OpenCL, this step was kept for the aforementioned uniformity reasons. Moreover, some of the work done is still very important to the standard C implementation, including such the decision to assign values to local variables (the OpenCL private memory) as opposed to shared arrays (the OpenCL global memory), which has have similar performance implications in OpenMP (access to shared data still requires synchronization, which should be avoided if possible). An interesting side effect of this arrangement is that the Lift programs targeting OpenMP can - and at time must - explicitly use of the **toGlobal** and **toLocal** function to explicitly control the process, even at times where the use could be inferred by the context.

- Ranges and Sizes: this second step takes care of computing the sizes of the C variables holding the various parameters and intermediate values. This information is vital for performing memory allocation, and happens to be mostly platform independent. It was therefore reused verbatim.

- Barrier elimination: this step was completely removed from the generator, as plain C does not have any notion of barriers, nor OpenMP.

- Types generation: Lift has a richer type system than C, and provides language level support for tuples. These can be implemented as C structures, and are generated following a mechanical process: the structure name is **Tuple_a_b_c**, where a b and c are the types of the individual elements in this example triple.

Then, the first component is called **_1**, the second **_2** and so on. Once again, this system is platform agnostic as far as C is concerned as was not modified.

- User function generation: similar to the above, only even simpler: user functions, already expressed as blocks of C code, are written in after the types.

- Kernel generation: the final step of the process, it takes care of producing the code implementing the Lift function proper - henceforth known as the Kernel (a name retained from the OpenCL implementation). This step is recursively applied over the supplied abstract syntax tree, generating a matching C syntax tree from it. When the process is completed, the C tree, which now represents a free floating block of code, is packaged up into a C function definition.

  This stage is the one that has undergone the greatest change. Firstly, all OpenCL parallel constructs have been removed, as the C generator can only support the most general patterns, such as reorderings and sequential mapping and reducing. Secondly, the C abstract syntax tree has been extended to represent pragma statements, which were absent in the original implementation (these being needed for implementing the OpenMP patterns). Finally, the production of the Kernel function signature is so rich of with OpenCL specifics that it has been rewritten from the ground up to be purely C. The algorithm is relatively simple: typing information is extracted from the Lift abstract syntax tree to determine the parameter and result types of the Kernel function.

4.3.3 OpenMP Generator

The OpenMP generator is an extension of the C generator, adding in support for parallel constructs. This is achieved by augmenting the latest step of the C generator (the recursive translation of Lift constructs into a C abstract syntax tree) with the two new parallel patterns MapOMP and ReduceOMP. In order to achieve the maximum degree of code reuse possible, these are declared as subtypes of MapSeq and ReduceSeq, the semantically-equivalent sequential versions present in the C generator. While this might appear baffling at first, there are solid reasons for this: by leveraging inheritance this way, there is no need to provide any code for all the complex internal steps such as address space inference and size computation.

Moreover, the correctness of this approach is guaranteed by the nature of the OpenMP library, which sets as one of its objectives to be backwards-compatible with C (that is, if one compiles an OpenMP program without the library, the code written is still valid C, and produces a semantically equivalent sequential program). Therefore, by generating the same code for MapSeq and MapOMP (module a slight different soon to be shown), and having ensured the correctness of MapSeq, one can rely entirely on OpenMP for the correctness of MapOMP.

The only addition that the OpenMP generator must therefore provide is in the final code generation phase, where the parallel implementations must decorate the C code produced by the matching implementation with the appropriate pragmas as discussed in section 4.2

The approach stated above is however not complete: in its naivity, it ignores dangerous interactions that might arise between the C generator optimizer and the blind addition of an OpenMP pragma to the its output. An example of hazard is encountered with loops that have a small, fixed number of iterations. Such loops might be unrolled, and when that that happens, a pragma cannot be added before the produced code block, as it would attempt to parallelize a non-existing loop. The OpenMP generator therefore is also able to detect wether the code was optimized in some form which can no longer be parallelized and does not add the pragmas in that occasion.

4.3.3.1 Private memory optimization

For the most part the OpenMP code generator implements MapOMP and ReduceOMP following the rules given in sections 4.2.3 and 4.2.4. This yields correct programs, but not necessarily high performing ones. There can be several causes for poor performance in the generated OpenMP code, the most common of which is repeated access to a shared variable, which requires large amount of synchronization overhead.

Consider, for example, this simple Lift fragment, which takes a two dimensional array, and produces a one-dimensional array by summing up each row.

Listing 4.5: A nested reduction

```
MapSeq(xs -> ReduceSeq(+,0.0f) $ xs) $ yss
```

Listing 4.6: The straightforward translation

```
acc = 0;
//The map
#pragma omp parallel for
for(int i = 0; i < N; i++) {
        //The reduce
        for(int j = 0; j < N; j++) {
                acc += yss[i * N + j];
        }
        result[i] = acc
}
```

The variable **acc**, like all C variables used in a Lift implementation, is declared before the main body of code. However, in this particular case, having a single instance of **acc** is problematic: each thread in the outer parallel loop tries to constantly access it, leading to a de-facto serial program. However, there is no semantic reason why **acc** should be in accessed by all threads. This condition, known as false sharing, can therefore be resolved by instructing the OpenMP framework to generate a private copy of the **acc** variable within in each. This is achieved by decorating the parallel for pragma with a **private** clause. The correct implementation therefore is:

Listing 4.7: Non-sharing implementation

```
acc = 0;
//The map
#pragma omp parallel for private(acc)
for(int i = 0; i < N; i++) {
        //The reduce
        for(int j = 0; j < N; j++) {
                acc += yss[i * N + j];
        }
        result[i] = acc
}
```

The OpenMP code generator is capable of performing this optimization: before generating the parallel pragma for MapOMP or ReduceOMP, it recursively explores children expression, resolve which variables are private in their scopes, and if any are present adjust the implementation accordingly. (The N-Body benchmark presented in the Eval-

uation chapter is a notable case of this optimization being needed)

4.4 Runtime

The previous section has detailed the workings of the code generator, which produces a C implementation of the Lift kernel function. However, the code produce is far from being directly usable: memory allocation and I/O are non existent at this stage. It is therefore necessary to provide some form of Runtime providing all these capabilities, and unlike the generator, no code of the OpenCL runtime can be reused.

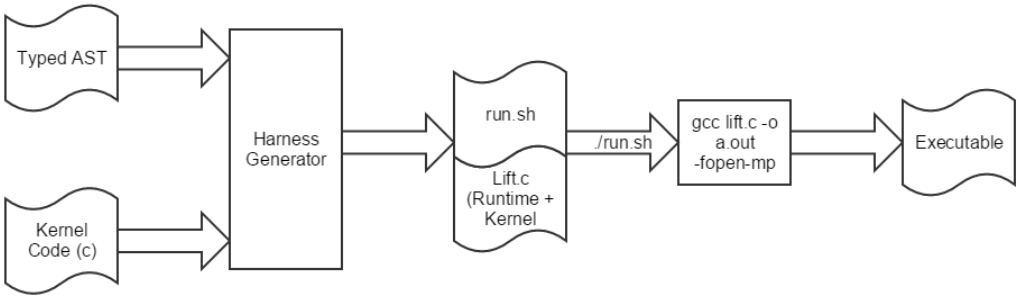

Figure 4.2: Schematic overview of the runtime generation and execution steps

4.4.1 Design decisions

Unlike the generator, whose structure was forced by having to work in concert with the rest of the compiler architecture, the Runtime is independent and therefore a variety of implementation strategies were possible.

- JVM to C interop: the most immediate idea is to directly call the compiled C code from the executing environment of a Lift program, the JVM. This has the benefit of being a very direct approach: the lift kernel is the only thing to be compiled, and no particular memory allocation or IO code is necessary: the kernel is invoked directly on the data, which is marshalled using the JNI technology. However, this option was discarded, as JNI is very unpleasant to work with: the protocol is quite complex and makes for some poorly readable code which would

ultimately result in a worse implementation, even if the process is theoretically easier.

- Fixed C runtime linked with kernel: The second option is to produce a static C runtime, capable of handling input and output, to be linked with the Lift kernel and provide the needed functionality. This has several merits, such as being a single, reusable compiled C file. Such file could be written by hand, and therefore be as complex as needed.

  The damning issue here however is that this Runtime does not have access to typing information, which makes memory allocation and invocation of the Lift kernel overly complicated.

- The third and final approach, which is the one fully implemented in this project, is to dynamically generate the Runtime every time a Lift program is compiled. By using typing information, is possible to generate code that is tailored to the exact Lift program being compiled. The downside here of course is the presence of an additional external layer for every lift program, and of one more compilation step necessary.

4.4.2 Parameter passing and result presentation

The first task the Runtime must take care of is interfacing the Lift kernel function with the outside world . This can be divided into to independent sections: receiving data and passing it in the kernel function, and then the presentation of the kernel output back to the invoking program. Two simple communication protocols have been devised, one for Input and the other for Output.

The input protocol is as follows: scalar values are symbolically encoded (that is, they are written out in a human-readable format, as opposed to be encoded byte-per-byte). The entire encoded string is a sequence of such scalar values, alternated by punctuation marks. The following entries make up the parameter string: first, the size variables, integers that determine the length of dynamic arrays. Then, the program parameters, in the same order as the Lift function parameters. Arrays are passed in as sub-sequences, without any clear start or end within the final encoding, and are automatically flattened.

The lack of any punctuation besides separation of individual scalar values is of no matter: since the Runtime is compiled with full typing info, the number of size variables and the number and order of parameters is statically known at compile time, providing all the information needed. In fact, we can leverage this to be able distinguish when the program fails due to having be supplied an incoherent argument string, as opposed to failures due to Lift programming errors.

The input protocol was designed to be so minimal for a variety of reasons. The first concern was simplicity of implementation in the Runtime, which can only rely on the minimal capabilities offered by the C standard library (using a custom parsing system would add an unnecessary dependency to such a core piece of code). The reliance on strings, while greatly inefficient, is there to make the Runtime code platform independent: being a parallel data processing language, is not unusual to run Lift programs on more exotic hardware, which could use an unconventional endianess, or machine word size, all details which can be safely ignored by using the ASCII representation.

The output protocol, on the other end, focuses around human-readability: scalars are once again given in their canonical ASCII representation, but arrays are not encoded using run-length, but instead nested brackets. This simplifies the testing of Lift programs, even as stand-alone entities. And while it makes the parsing slightly more complicate, this is not a problem if the invoking environment is coded in a suitably rich language, such as the Scala used within the Lift compiler.

4.4.3 Memory Allocation

The Lift kernels do not include memory allocation code: every array-typed variable needs to be allocated before the kernel function is invoked, and are then passed in as pointer parameters. There are several reasons for this: firstly, this arrangement is needed in an OpenCL setting, where memory is not an uniform resource, and secondly it makes the kernel more easily portable across different platforms.

There are two possible allocation methods;

- Arrays whose size is a compile time constant get allocated on the stack, for simplicity and speed: no calls to malloc-free needs to be made.
- Arrays whose size is dynamic: these must be allocated using malloc and then

release using free. The amount of bytes to allocate is computed by using typing information (for the element type) and from the size variables read in at the beginning of the input data.

Stack allocation was preferred over having the arrays as globals (which, being part of the program executable itself, require no allocation). The main reasons behind this choice is that there is not much performance difference between the two, but using globals for arrays greatly increases executable size, especially in the case of large volumes of data processing, which is not unexpected given the nature of Lift.

The memory allocation code also includes logic to handle several subtle complexities. The most prominent example is in the handling of nested arrays, which are supported in the Lift language but are then expected to be flattened when passed as parameters to the kernel. This is of particular importance in the output generation, which needs to be aware of the level of nesting in order to be human readable.

4.4.4 Executor

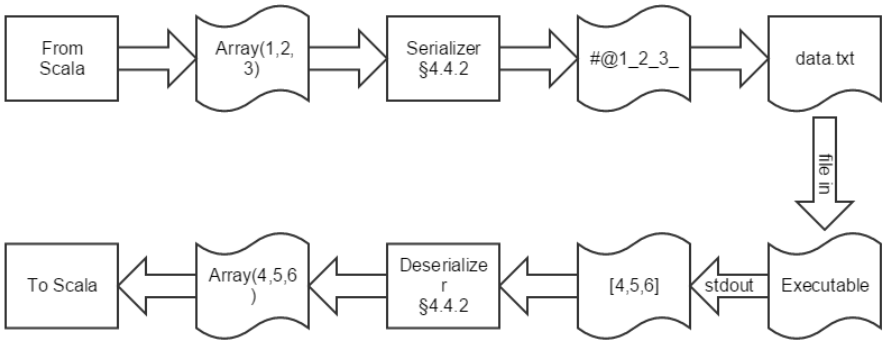

Figure 4.3: Schematic overview of execution of compiled Lift code from Scala

The last component in the Lift compiler pipline, known as the executor, allows both to seamlessly invoke Lift from Scala, and to generate a portable bundle which can then be run on a Unix-like system with gcc installed. The idea is to take a Lift expression and the input data to apply it over, and then generate a script which compiles the program and feed in the data.
If the invoking Scala program is executing in the right environment (and operative system supporting bash and gcc), then the shell script can be immediately run, and by

redirecting the standard output, the calling Scala program can read back the output, which is de-serialized and can the directly used within Scala. Otherwise, the script and source bundle can be distributed and then re-compiled and run on site.

Distributing the source code and expecting it to be recompiled on the target platform, while initially slow, is motivated by the intent of maximizing portability:Lift programs are expected to run on a variety of hardware, and so distributing the binary is not always a possibility.

4.5 Summary

This chapter has presented a step-by-step vi of the main software components of the Lift compilation and execution pipeline. It has by giving an high-level view of the translation method utilized to derive OpenMP code from Lift primitives. It has then covered the main stages of the basic C sequential generator. Then, the OpenMP generator built on it is analyzed, placing particular emphasis on the platform specific optimization performed. Finally, the chapter covers the design and implementation of a dynamically generated Runtime for Lift programs, and how it allows interaction with the invoking environment, particularly in the case of Scala.

Chapter 5

Evaluation

This chapter covers the methodology used to evaluate the success or failure of this project, the data gathered in the process and the conclusions that one can draw from this.

5.1 Methodology

5.1.1 Experimental setup

The machine used to run the tests is sporting a 32-core Intel Xeon L7555 CPU, therefore particularly suited for running parallel computation. At the moment of running the experiments, the machine had 10 GB of RAM available, which is sufficiently large to ensure no swapping has occurred.

The C files generated by the Lift compiler have been compiled with gcc version 4.7.2. The following compilation options have been used: -std=c99, -fopenmp (enabling OpenMP), -O3 (common optimization level and needed for next flag) -ftree-vectorize (enables gcc's embedded vectorizer, needed here because it also assists gcc's optimization of OpenMP code, eliminating some false-sharing issues) -lm (some benchmarks depend on the math library). Running time information is collected through the timing system covered in section 4.4.4. The OpenCL programs have been compiled with the already existing executor system present in the Lift compiler, which also provides running times.

5.1.2 Timing

In order to successfully evaluate the outcome of the project, running time information needs to be collected. If one however uses any "external" timing system, such as the bash **time** utility, the results obtained would include secondary computations, such as feeding the parameters to the kernel or memory allocation. To obtain just the running time of the Lift-generated code therefore, the harness generator can be configured to record a timestamp right before, and just after, calls to the kernel. Then, a time difference is computed and presented as the running time. In particular, the C function **gettimeofday** is used to extract the timestamps.

5.1.3 Inputs

The inputs for the benchmarks below are produced by a random number generator. All the experiments detailed here use floating point number as inputs, which are also the unit of measurement for the input size in all data presented.

The input size range have been determined by empirical exploration: the criteria used to arrive at the values used is that the input size shall be large enough to get reliably readings which are not perturbed by background noise (such as a well-timed OS context switch), but not so large to complicate data gathering operations by taking too long, or thrashing tche computer's performance by consuming excessive amounts of RAM.

5.2 Benchmarks

5.2.1 Overview

Four benchmarks have been selected to evaluate the project. Three of these are common computation tasks: computing the Black-Schoals model of a series of prices, multiplication of two square matrices, and solving one iteration of the n-body problem. The benchmarks are order by increasing computational load. For each, the different implementations of each benchmark are tested against each other: a sequential version acting as a baseline, the OpenMP version and the OpenCL version generated by the other Lift backend. Only the code for the OpenMP version is given: the sequential and OpenCL version can be obtained by replacing the MapOMP calls with the equivalent

form for each implementation. No further modification is applied to the generated C code. A fourth benchmark is presented at the end, exploring the usage of the native ReduceOMP against the traditional alternative of composing a parallel map with a sequential reduction.

No other variable, but input size, is modified in the runtime comparison benchmarks. This is quite important to take notice of, because two identical OpenMP programs can have wildly different performance, depending on how the runtime is configured. For example, altering the "OMP_NUM_THREADS" variable, which controls the amount of worker threads used in the OpenMP program, can have drastic effect on the running time. This is however an external element to the Lift compiler, which is ignorant even of the existence of this feature, and therefore the default value is used in the experiments.

In any case, each individual benchmark-size-backend triple is run ten times, and then the mean of running times is used.

Additionally, for the three canonical benchmarks, a second experiment has been performed, aiming to ascertain the strong-scaling properties of each implementation. This is achieved by controlling the amount of threads the OpenMP runtime can use and then executing the same program, on the same input size, and measuring the speed ratios obtained. Execution with 1 to 32 threads is tested.

5.2.2 Black-Scholes

The first benchmark presented is the Black-Scholes algorithm. Computationally speaking, it is structured as the application of a complex sequential function to a set of elements. Since this function only depends on the element it is applied, the entire process is easily parallelizable. Therefore, Black-Scholes can be used to test the effectiveness of parallel mapping under ideal conditions.

The Lift code implementing this benchmark is quite straightforward:

Listing 5.1: OpenMP Black-Scholes

```
fun(
```

```
    ArrayType(Float, N),
    inRand => MapOMP(blackScholesComp)  $ inRand
  )
```

As can it be seen, the program simply maps the blackScholesComp user function over the supplied array. The code of blackScholesComp has been taken from the already existing benchamarks in the Lift compiler, and it's implementation have been omitted here for space reasons.

5.2.3 Matrix Multiplication

The second benchmark presented is matrix multiplication. This classic computation can be written in a variety of different flavours and techniques, such as using tiling to increase memory locality. While such implementations were available in the Lift benchmark suite, this project uses a hand-written matrix multiplication version, focused on simplicity, clear ordering and separation of operations: the computation is executed "for each row, for each column", and for each row-column pair, the multiplication step is separated from the addition one (one could improve the performance by fusing the multiplication into the addition. However would require writing a custom "multiply and accumulate" user function in C. This sort of optimization would move away from the functional nature of the language, and are therefore avoided here).

Listing 5.2: OpenMP matrix multiplication

```
fun(
    ArrayType(ArrayType(Float, N), N),
    ArrayType(ArrayType(Float, N), N),
    (A, B) => {
      toGlobal(MapSeq(MapSeq(MapSeq(id)))) o
      MapOMP(fun(Arow =>
        MapOMP(fun(Bcol =>
          toGlobal(MapSeq(id)) o ReduceSeq(add, 0.0f) o
          MapSeq(mult)  $ Zip(Arow, Bcol)
        )) o Transpose() $ B
      )) $ A
    })
```

As it is possible to see, only the outer iterations have been parallelized. The rationale for keeping the inner computation sequential is that the amount of work performed for each row-column pair is not sufficient for justifying the overhead. Moreover, the

outer layers of parallelism ensure that there is more then enough work for all the cores anyways. Finally, since two layers of MapOMP calls are within each other, this benchmark is still effective at verifying that that parallel loops can be successfully nested.

5.2.4 N-Body problem

The third benchmark is the N-Body problem solver, whose code follows:

Listing 5.3: OpenMP n-body

```
fun(
    ArrayType(float4, N),
    ArrayType(float4, N),
    Float,
    Float,
    (pos, vel, espSqr, deltaT) =>
      toGlobal(MapOMP(fun(p1 =>

        (MapSeq(fun(acceleration =>
          update(Get(p1, 0),
          Get(p1, 1), deltaT, acceleration))))

          o ReduceSeq(fun((acc, p2) =>
          calcAcc(Get(p1,0), p2, deltaT, espSqr, acc)),
          float4zero) $ pos

      ))) $ Zip(pos, vel)
  )
```

This program is based on the similar one existing in the Lift compilerS codebase. The main modification here as redefining the OpenCL-specific float4 type to be an alias for the four-tuple. The omitted user functions update and calcAcc implement the specifics of the N-Body algorithm.

This benchmark shows the correct implementation of two fundamental features: first, the parallel version can easily operate on structured data (the float4 in this case) as well as the previously encountered scalars. Secondly, this program makes use of the “private memory optimization” mentioned in 4.3.3.1: the parameters espSqr and deltaT

are correctly distributed as private copies to each thread.

5.2.5 Testing OpenMP Reduction

The final benchmark here shown is not a well known or useful computation. Instead it is an ad-hoc program, attempting to compare the performance characteristics of using ReduceOMP against the equivalent composition of a parallel Map and a sequential reduce. This is achieved by having a standard summation type reduction, in conjunction with a custom function mapped over the input elements. This function is an arbitrary arithmetic function, which has no particular meaning: it is just complex enough to avoid unexpected C compiler optimizations which may sabotage the results and also not be so fast as to hide any performance gain within overhead times, but still small enough to model the common " arithmetical transformation and sum" pattern. The selected expression is:
$f(x) = x^2$
The two programs are:

Listing 5.4: Standard version

```
fun(
    ArrayType(Float,N),
    in =>  toGlobal(MapSeq(id)) o
        ReduceSeq(add,0.0f) o
         MapOMP(f) $ in)
```

Listing 5.5: Native ReduceOMP version

```
fun(
    ArrayType(Float,N),
    in =>  toGlobal(MapSeq(id)) o
        ReduceOMP(:+(Float),fAcc,0.0f) $ in)
```

5.3 Results

5.3.1 Black-Scholes

Regarding raw speed, the clear winner here is the OpenCL implementation, which results faster than all others in all cases. Comparing the OpenCL and OpenMP versions, we can notice what seems to be a large constant factor present in the latter which is

absent from the former. This seems to be the main drawback of relying on OpenMP: there is a significant overhead in setting up the parallel computation, which does not seem to be justified by the amount of work Black-Scholes requires.

However, comparing the slowdown as a function of input size, it is possible to see a much better picture for the OpenMP implementation: OpenCL-128000 is 5 times slower then OpenCL-1280, but for OpenMP the factor is just 1.8 times. One can therefore hypothesize that there exist a break-even point after which OpenMP would take over OpenCL, however this is a risky hypothesis, as the mass of data required for this to happen for Black-Scholes is quite massive, and other factors, such as RAM availability, might enter in play at that point.

As far as strong scaling is concerned, on the hosting machine the Black-Scholes computation keeps a linear speedup profile up until 16 threads. After that point, the data suggests that there is no more performance to be gained from further cores, and in fact the program results slower with 32 cores. This is probably due to the shrinking amount of work done per execution thread: the overhead costs of parallelism become greater than the time saved by having more threads.

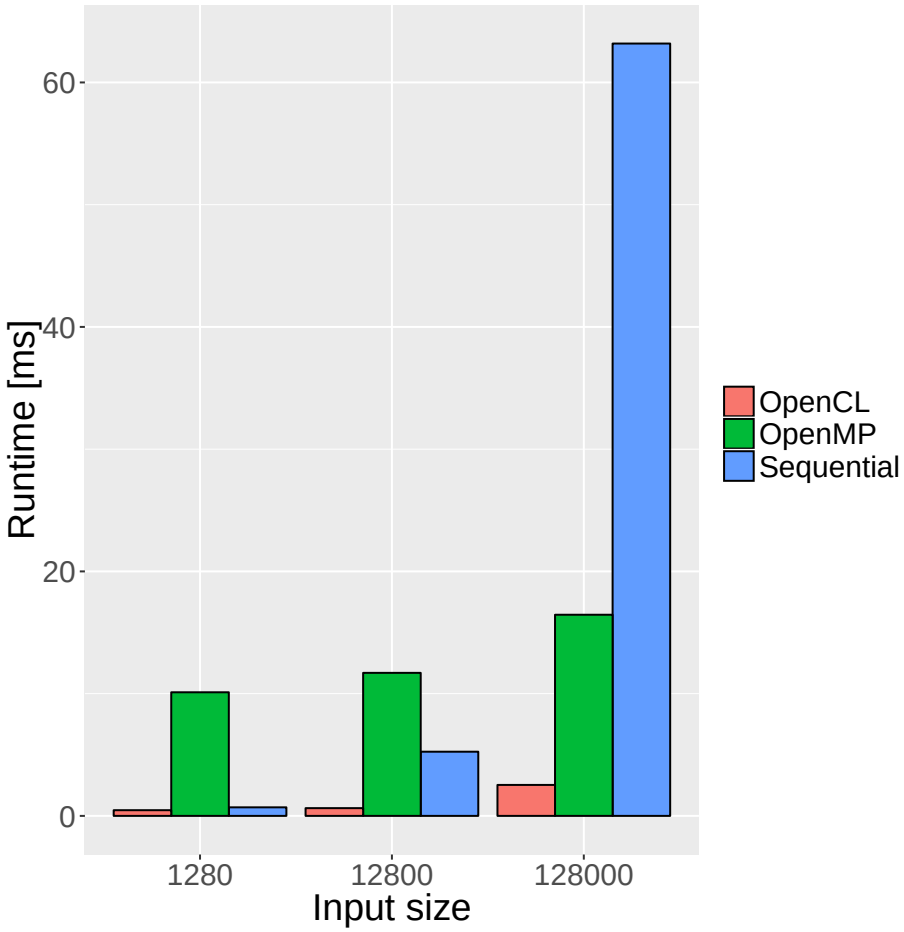

Figure 5.1: Comparison of running times for Black-Scholes benchmark generated from OpenCL, OpenMP and Sequential backends

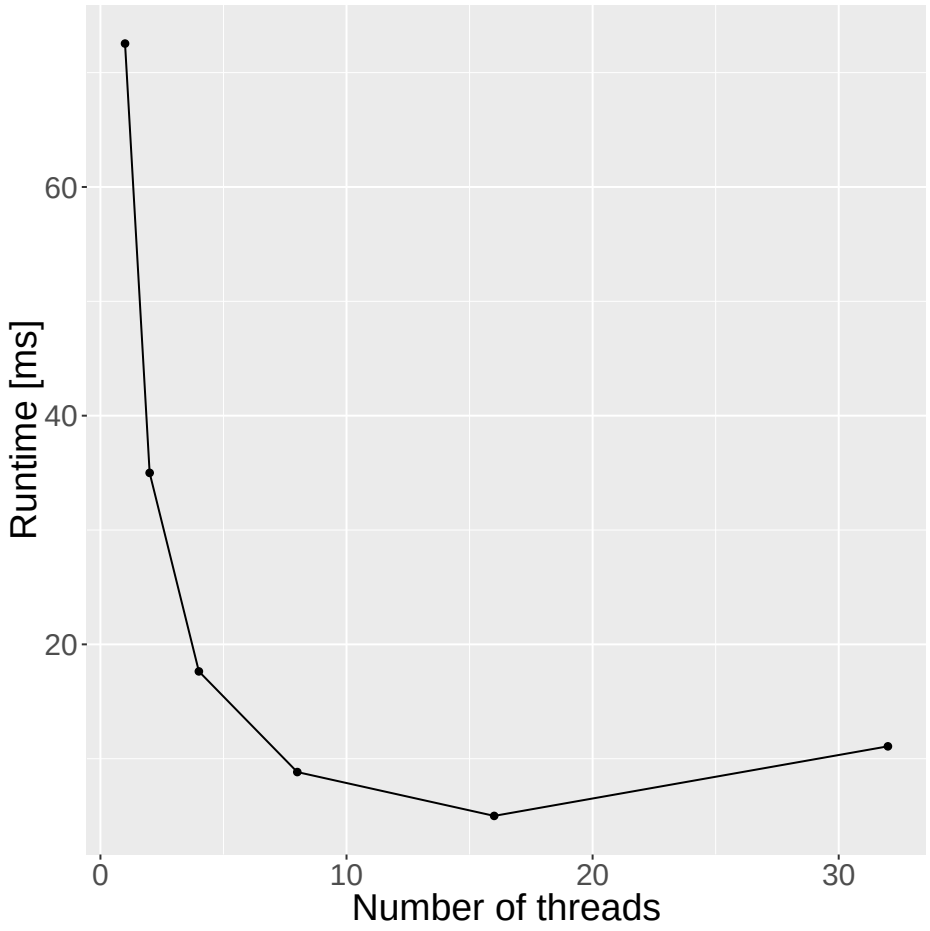

Figure 5.2: Strong-scaling profile of generated OpenMP implementation of Black-Scholes when varying number of execution threads

5.3.2 Matrix multiplication

The data comparison shows that this implementation tracks the OpenCL preformance much better: the amount of work is sufficiently large that the initial OpenMP overhead is an almost insignificant factor.

The runtime shows a linear-like speedup, potentially continuing after a thread size of 32

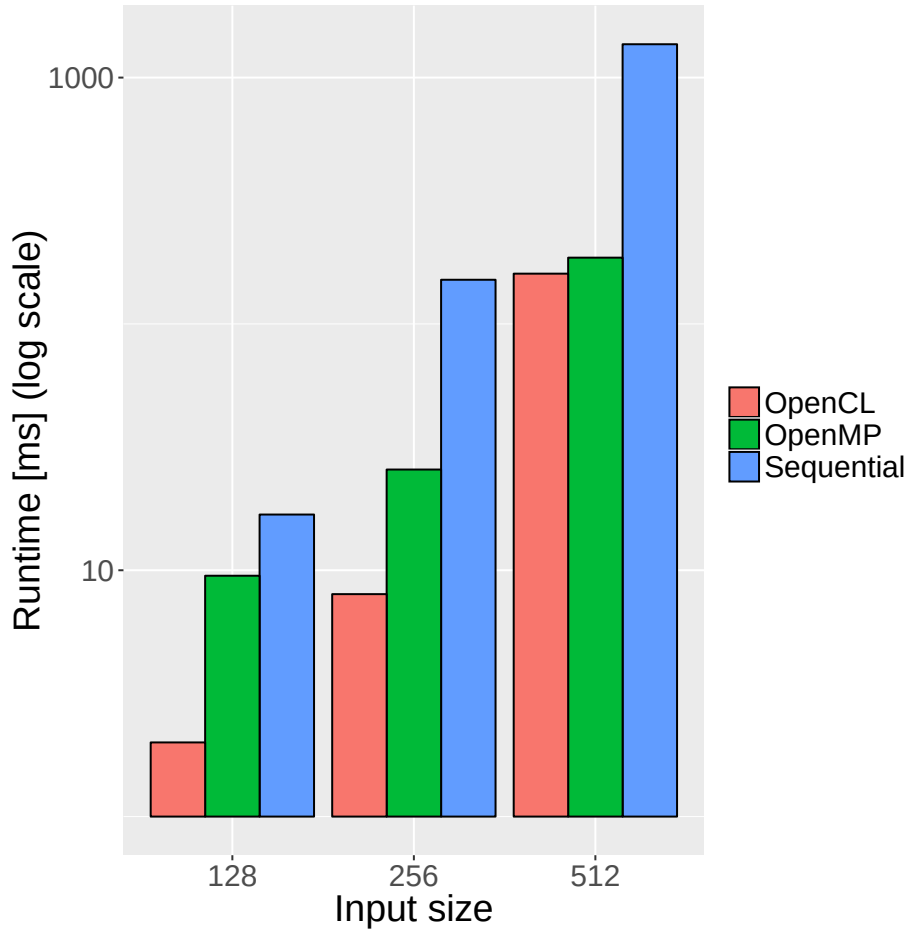

Figure 5.3: Comparison of running times for Matrix Multiplication benchmark generated from OpenCL, OpenMP and Sequential backends

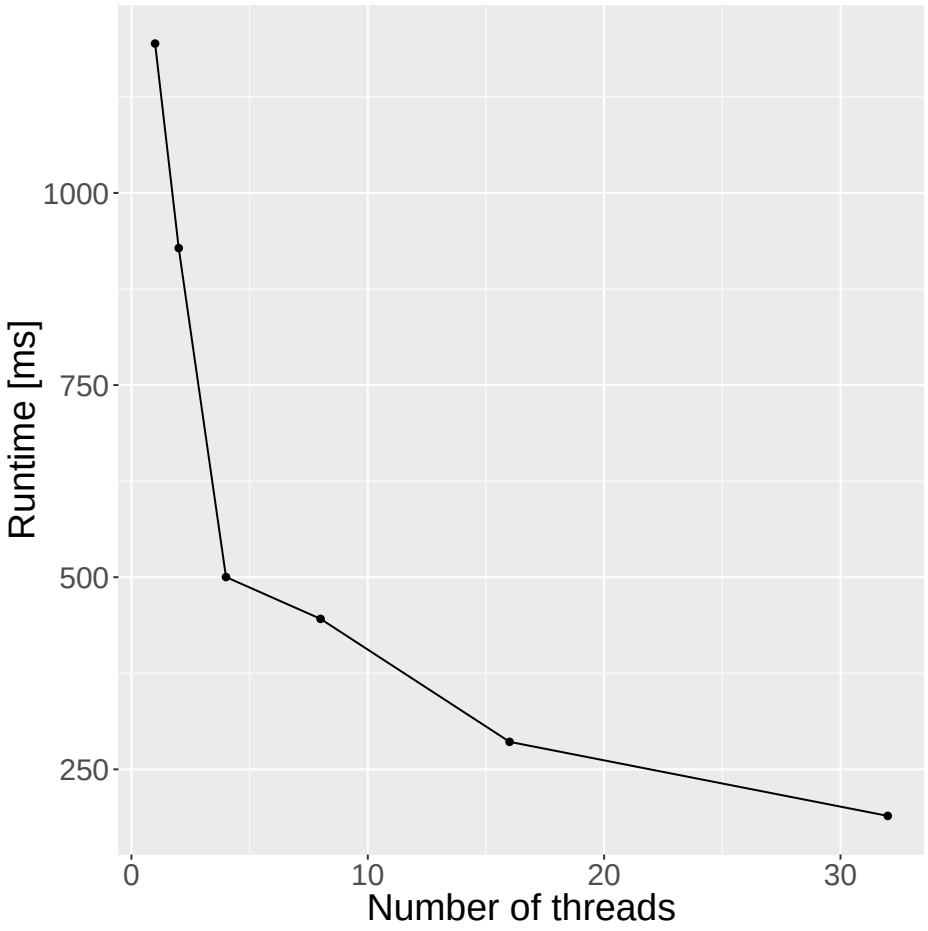

Figure 5.4: Strong-scaling profile of generated OpenMP implementation of Matrix Multiplication when varying number of execution threads

5.3.3 N-Body solver

Among all the benchmarks, this is the most computationally intensive: as such, it exasperates the trends seen in the previous two experiments: at minimal size, the sequential implementation dominates, but as computational load becomes more and more significant, the parallel implementations become better, especially the OpenMP version. The scaling profile also seems to confirm this hypothesis: the greater load means a greater fraction of computation time is dedicated to useful work (as opposed to coordination), which gives an even more linear speedup-like curve.

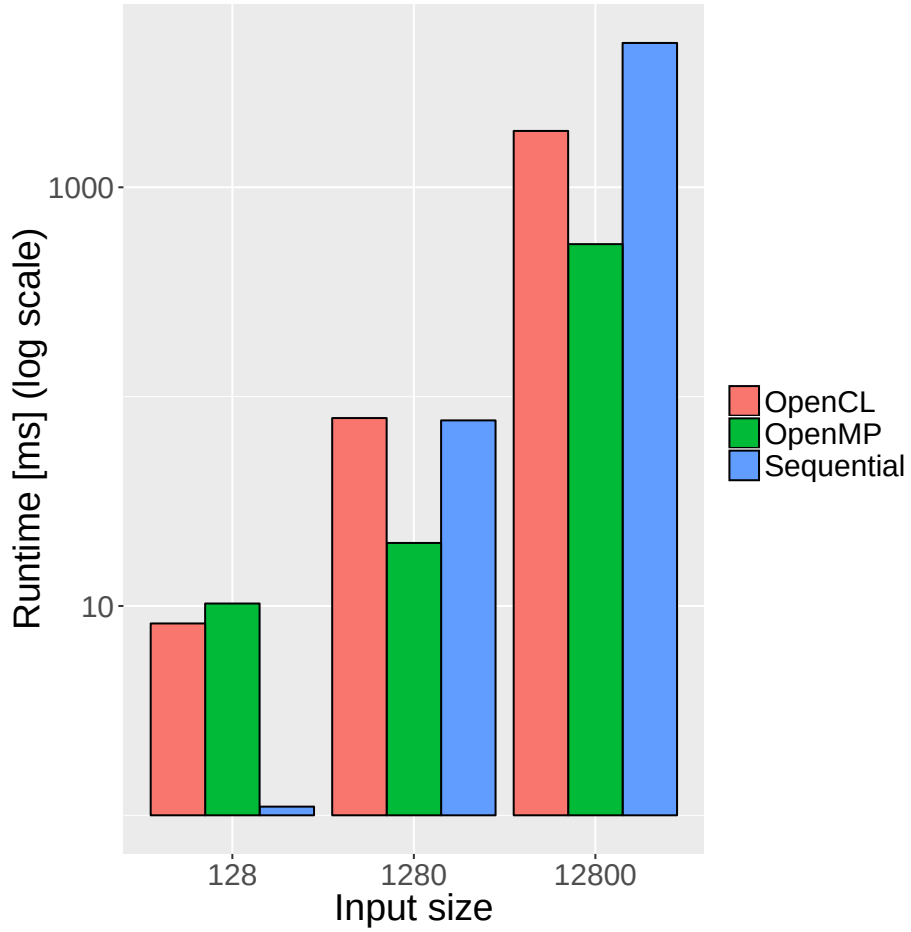

Figure 5.5: Comparison of running times for N-Body solver benchmark generated from OpenCL, OpenMP and Sequential backends

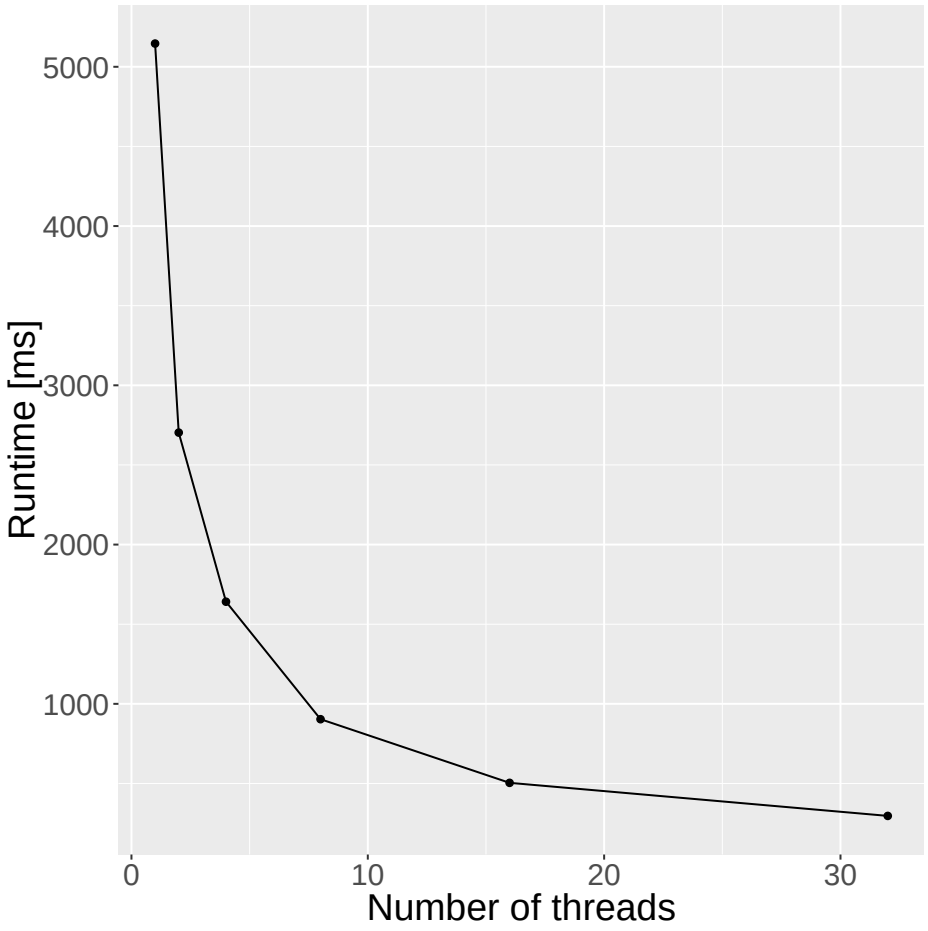

Figure 5.6: Strong-scaling profile of generated OpenMP implementation of N-Body solver when varying number of execution threads

5.3.4 OpenMP parallel reduction

The data seems to show that, at least in the small, employing the native reduction yields only a moderate performance gain. In fact, the runtimes are so close that it is tempting to hypothesize on the nature of gcc's OpenMP parallel reduction, and state that it is also implemented internally as a sequential reduction over a parallel mapping, and that the difference in performance reported is only due to elimination of overheads resulting from the fusion, such as having only one C function call (fAcc) as opposed to the two of the standard lift implementation (f and add).

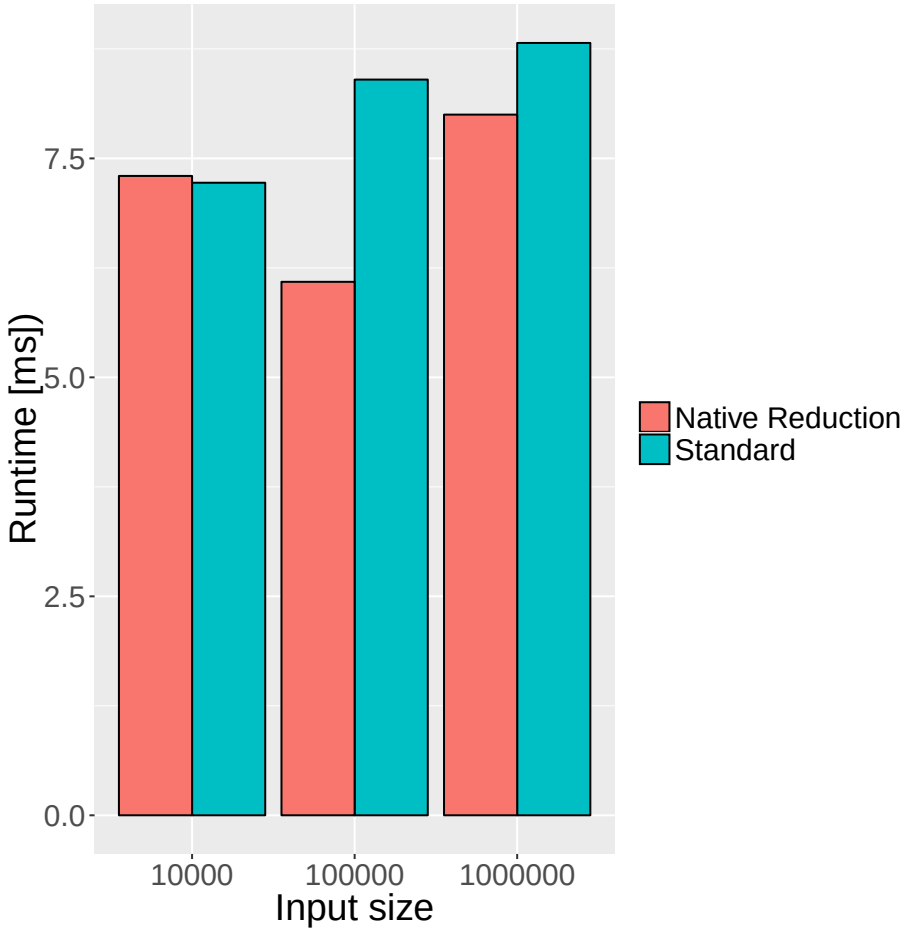

Figure 5.7: Comparison of running times for the square-sum benchmark, implemented with a single native ReduceOMP vs the composition of ReduceSeq o MapOMP

5.4 Summary

The following conclusions seem to follow from the experiments shown above:

- The OpenMP implementation successfully parallelizes the code, and if the amount of work is sufficiently large, a high number of threads can be used in a beneficial way.
- For the smallest computational loads, the sequential version can easily compete with the parallel alternatives, due to the lack of parallel overhead, but also because of the compilers ability to better optimize sequential code.
- For larger but still relatively small loads (BlackScholes, Matrix Multipication) the OpenCL implementation is superior to the OpenMP one: this is due to the apparent presence of a tens of millisecond startup cost in the latter.
- After a certain breaking point, OpenMP seems to outperform OpenCL. This is seen in NBody, and from the relative slowdown of OpenMP being smaller in the Black-Scholes and Matrix Multiplication benchmarks.
- Under normal circumstances, using ReduceOMP yields only small benefits compared to the native parallel version. Further work is needed here to determine whether this is a gcc related issue.

Chapter 6

Conclusion

This closing chapter starts by offering a summary of the work done in the documented project. It then proceeds to a critical review of the work done, and closes by proposing a few future avenues of development.

6.1 Summary

This document begins by providing the necessary background knowledge to understand the work done in the linked project: the idea of using parallel patterns and functional programming to express parallel programs, a quick glance at the relevant parts of the Lift language and at the worksharing constructs of OpenMP, the code generator's targeted platform. It than moves to cover related work, some referencing alternative approaches and other referring the other backend used in Lift.

The focus then shits to the workings of the code generators. It starts from a high-level view of how to translate Lift's parallel patterns into OpemMp, it moves towards the implementation details: from the initial refactoring of existing OpenCL code, it presents an overview of the generation pipeline, focusing on the parts where modifications have been made. Finally, the document covers the motivation and design behind providing an harness to supplement the generated C code, and the implementation challenges it posed.

The final part of the document is dedicated to evaluating the performance of the OpenMP code generator: it is compared with a baseline sequential implementation and the OpenCL version. Moreover, a small strong-scalability analisys is performed to as-

certain “how parallel” the implementation actually is. The experiments led to the conclusion that the OpenMP implementation is indeed working as intended, with good scalability and performance after a certain computational load, but that there are still some deficiencies, particularly when dealing with smaller tasks.

6.2 Critical analysis & limitations

There are several instances of difficulties in this project which, in hindsight, should have been tackled differently. The first notable problem has been going with an ASCII-based protocol for data encoding. While this might seem minor, it was one of the stifling factors imposing an upper bound on the experiment size: it caused data files containing the arguments to the application to become huge and therefore become to unwieldy: the memory consumption for loading them in memory alone caused serious performance issues, and even crashed some programs, even in presence of proper memory-management techniques.

The second main issue deals with “unknown unknowns” in the behavior of the gcc compiler, with respect to OpenMP: initially some OpenMP implementations had absymal performance (worse then the sequential version) due to redundant synchronization. The inclusion of the -ftree-vectorize flag, a fortuitous find, actually solves that problem as a side-effect of enabling native gcc vectorization. It is however impossible to tell whether similar problems are present in the current project implementation: their magnitude could be too small to be singled out clearly, and yet affect the overall performance.

A notable deficiency of this project is the absence of any effort of implementing explicit vectorization, which is poorly replaced with reliance of the native gcc’s vectorizer. Gcc’s implementation of OpenMP’s explicit vectorization has resulted quite unpredictable, and the exploration of alternative compilers, such as the Intel C Compiler, have been cut short due to time constraints.

The use of OpenMP’s higher level primitives has been motivated by the almost perfect correspondence between Lift’s parallel patterns Map/Reduce and OpenMP’s parallel for and reduction. However a case could be made that after implementing MapOMP with the work-sharing constructs, the work could have shifted to implementing alterna-

tive primitives with lower-level OpenMP primitives such as tasks and barriers, perhaps yielding better performance by eliminating the relatively large parallelism overhead which the works-sharing constructs seem to bring.

6.3 Future work

Several possible avenues of development for the Lift OpenMP backend are open: as simple improvement, one could re-write the serialization protocol to use a more information-efficient encoding, such as writing the values directly in C binary format. Perhaps implement a system of splitting up the data in smaller files, as to reduce the maximum memory footprint without making the data parsing overly complex.

One could also extend the backend by implementing parallel patterns in terms of lower level OpenMP features, such as tasks and barriers. Tasks in particular could be used for a Map implementation which is not statically-scheduled like the current MapOMP, and being better suited for unpredictably asymmetric computations.

Having a working OpenMP backend means that now Lift code can be run free of the shackles imposed by conforming to the fixed OpenCL kernel structure. This means that, on the OpenMP backend at least, one could enrich Lift with some language features such as input processing and better flow control, allowing one to write entire programs in Lift.

Bibliography

Armstrong, B. and Eigenmann, R. (2001). Challenges in the automatic parallelization of large-scale computational applications. In *ITCom 2001: International Symposium on the Convergence of IT and Communications*, pages 50–60. International Society for Optics and Photonics.

Catanzaro, B., Garland, M., and Keutzer, K. (2011). Copperhead: compiling an embedded data parallel language. *ACM SIGPLAN Notices*, 46(8):47–56.

Chiw, C., Kindlmann, G., Reppy, J., Samuels, L., and Seltzer, N. (2012). Diderot: a parallel dsl for image analysis and visualization. In *Acm sigplan notices*, volume 47, pages 111–120. ACM.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dubach, C., Cheng, P., Rabbah, R., Bacon, D. F., and Fink, S. J. (2012). Compiling a high-level language for gpus:(via language support for architectures and compilers). In *ACM SIGPLAN Notices*, volume 47, pages 1–12. ACM.

Dubach, M. S. T. R. C. (2016). Matrix multiplication beyond auto-tuning: Rewrite-based gpu code generation.

Eijkhout, V. (2014). A dsl for integrative parallel programming. In *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*, pages 27–34. IEEE.

González-Vélez, H. and Leyton, M. (2010). A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160.

Graham, R. L., Shipman, G. M., Barrett, B. W., Castain, R. H., Bosilca, G., and Lumsdaine, A. (2006). Open mpi: A high-performance, heterogeneous mpi. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9. IEEE.

Marlow, S., Newton, R., and Peyton Jones, S. (2011). A monad for deterministic parallelism. In *ACM SIGPLAN Notices*, volume 46, pages 71–82. ACM.

McCool, M. D., Robison, A. D., and Reinders, J. (2012). *Structured parallel programming: patterns for efficient computation*. Elsevier.

Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.".

Steuwer, M., Fensch, C., Lindley, S., and Dubach, C. (2015). Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. *SIGPLAN Not.*, 50(9):205–217.

Sujeeth, A., Lee, H., Brown, K., Rompf, T., Chafi, H., Wu, M., Atreya, A., Odersky, M., and Olukotun, K. (2011). Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616.

Sujeeth, A. K., Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. (2014). Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134.

Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., and Wang, Y. (2014). Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer.