Idiom Specific Code Transformations for Parallelising Compilers

Philip Ginsbach

Master of Science
School of Informatics
University of Edinburgh
2016

Abstract

Parallel computing has become pervasive in most computational domains and this has lead to profound changes to programming languages, compilers and software libraries. The advent of heterogeneous computing requires a reevaluation of the currently established approaches to parallel computing, as many of them do not generalise well to heterogeneous parallelism.

The concept of computational idioms is well suited to describe the specific computational capabilities of individual components in heterogeneous systems. Furthermore, computational idioms contain contextual information that could enable compilers to reason about parts of code that are unsuited for traditional static analysis due to irregularities such as indirect memory accesses. The lack of formal specifications for computational idioms however prevents their use in automatic compiler analysis.

We investigate an idiom based approach to automatic parallelisation using a formal specification language for the description of computational idioms. We develop algorithms to automatically detect computational idioms in single static assignment intermediate representation code and implement idiom specific code transformations. Our evaluation is based on homogeneous systems, which allows a meaningful comparison to established methods for automatic parallelisation. We intend to extend the approach to heterogeneous computing in future research.

In a case study, we analyse the performance bottlenecks of two established benchmark collections and classify the identified bottlenecks according to the computational idioms that they represent. We establish the three most important computational idioms in these benchmarks and study the relevant parallelisation and optimisation techniques that can be applied to them.

We use the results from this case study to implement an optimisation pass in the LLVM infrastructure that is able to automatically detect computational idioms and apply idiom specific parallelisation transformations. We prove the general feasibility of the approach for a limited set of computational idioms. Further work is necessary to show that the approach can be extended to more complex programs and additional computational idioms.

Acknowledgements

I would like to thank my supervisor Michael O'Boyle for his scientific advice, his helpful feedback and the many interesting discussions about our research. Furthermore I want to thank my supervisor Björn Franke for new perspectives and interesting conversations about reduction operations and other topics.

I want to thank ARM and in particular my industrial supervisor Chris Ryder for the support and the opportunity to work together with one of the most exciting technology companies in Europe.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Philip Ginsbach*)

Table of Contents

Chapter 1

Introduction

Writing parallel computer programs is difficult and no single satisfactory solution to this problem has been found. Multi-core processors are now pervasive in most computational domains, resulting in profound changes to programming languages, compilers and software libraries. The advent of heterogeneous computing requires a reevaluation of currently established approaches to parallel computing, as many of them do not generalise well to heterogeneous parallelism.

In this thesis we introduce a new approach to automatic parallelisation based on the concept of computational idioms. It involves the formal specification of computational idioms and the integration of idiom specific transformations into compilers. With an idiom based approach, we can circumvent shortcomings of conventional static analysis by relying on idiom specific contextual information.

1.1 Motivation

Uncovering, expressing and exploiting parallelism in computer programs remains a big challenge. Ever since multi-core processors became mainstream in desktop computing, finding efficient and usable approaches to developing multithreaded applications has been a major research topic (Asanovic et al. (2006)).

Early hopes of avoiding profound changes to the established programming stack by enabling compilers to parallelise sequential source code automatically were largely disappointed. Instead of a single-size-fits-all solution, almost every aspect of application development was touched by efforts to mainstream parallel programming. This includes the development of parallel programming languages, supporting software libraries and modifications to the compiler infrastructures.

Compiler based parallelisation approaches were for a long time confined to very limited forms of parallelism, such as instruction level parallelism in the form of automatic vectorisation (Larsen and Amarasinghe (2000)). Exploiting thread level parallelism of larger scopes on the other hand turned out to be a very difficult problem. This is mostly due to the conservative nature of static program analysis. In many cases, it can not provide conclusive evidence for the validity of parallelising program transformations (Niall Murphy and Mullins (2015)). Speculative parallelisation with dynamic validity verification has helped to overcome the limitations of traditional static program analysis and revitalised interest in compiler based parallelisation (Tournavitis et al. (2009); Wang et al. (2014)).

The emerging awareness that future hardware platforms will be increasingly heterogeneous (Chung et al. (2010)) demands the reevaluation of the established parallelisation techniques and the development of new approaches. Heterogeneous computing in this context means the use of structurally different processing cores together in a single computing system. This can mean central processor cores working more closely together with accelerator hardware such as graphics processors. Other examples are platforms like ARM big.LITTLE, which pair very energy efficient processor cores with high performance cores to be able to provide the potential for high performance with very low energy consumption during idle times.

Heterogeneous multi-core systems can feature multiple different instruction sets, distinct memory spaces and complicated performance characteristics. This makes reasoning about efficient parallel implementations much more difficult. Heterogeneous computing is mostly a generalisation of parallel computing, but not all of the established approaches to parallel computing can be extended easily to support heterogeneity. This is because the main challenge of parallel computing on homogeneous systems is the exposure of parallelism, where in heterogeneous computing the scheduling of parallelism and the distribution of memory become an additional difficulty (Cong and Yuan (2012); Emani and O'Boyle (2015)).

In this dissertation we present a new approach to compiler based automatic program parallelisation that is designed around the concept of computational idioms. This concept allows us to integrate domain specific parallelisation and optimisation approaches into general purpose compilers.

1.2 Idiom Specific Parallelisation

Heterogeneous computing hardware often consists of general purpose processor cores together with other, more specialised processing hardware. An example of this concept is a desktop computer with a multi-core main processor and a general purpose programmable graphics processing unit (GPU). In mobile platforms there are often additional hardware accelerators, such as hardware media codecs and digital signal processors.

The individual processing devices in heterogeneous systems are optimised for specific computational workloads that occur frequently and are critical to the performance of the device. In software, such reoccurring computational structures can be classified as computational idioms. We argue that the capabilities of heterogeneous are therefore naturally described by computational idioms. This makes an idiom based approach to parallelisation very natural in the context of heterogeneous computing.

The fundamental approach is to detect idioms in program code and to then use domain specific knowledge of these idioms to achieve optimal parallelisation. As opposed to standard approaches, the awareness of higher level structures gives our system additional contextual information to resolve irregularities in program code that normally inhibit static analysis. This includes in particular pointer arithmetic and indirect memory access.

1.3 Work Undertaken

The work undertaken for this dissertation can be split into several distinct parts. The individual sections cumulate in the implementation of an optimisation pass in the LLVM infrastructure that uses the concept of computational idioms to apply code transformations that result in better hardware utilisation and runtime speedup by better exploiting available parallelism.

In the first stage of our research, we performed an explorative study of some established benchmark suites to assemble the computational idioms that are most pervasive in common computational workloads. To achieve this, we used profiling techniques to establish the bottleneck computations of the individual benchmark programs. We then classified the bottlenecks in terms of computational idioms and established which of them we need to support for significant coverage.

For each of the identified computational idioms, we investigated in what way we can best achieve runtime performance. This involved comparing alternative implementations of the functionality both using optimised libraries and manual code transformations. Most important was the efficient parallelisation of the performance critical program parts. We performed extensive measurements of the resulting runtime behaviour and compared the results to optimised parallel versions of the benchmarks that were created by the original implementers.

Using the results from the previous steps, we devised a formal language to specify computational idioms in a constraint based fashion that can be evaluated mechanically and we developed algorithms to efficiently process the specifications.

We then implemented a constraint based idiom detection system in the LLVM infrastructure based on this formal specification language to enable compilers based on LLVM to automatically spot computational idioms. We evaluated this system on the selected benchmark suites and compared the findings to the manual classification that we performed earlier.

Finally, we added routines to the LLVM optimiser to recreate our manual idiom specific code transformations based on this constraint based automatic idiom detection functionality. The results of this were compared to the runtime speedups that were previously attained by manual optimisations.

1.4 Contributions in this Dissertation

The following contributions are part of this MSc dissertation.

- An analysis of relevant computational idioms in two benchmark suites.
- A survey and evaluation of idiom-based optimisation and parallelisation strategies.
- The specification of a formal language for the description of computational idioms in LLVM IR.
- The derivation of an algorithm for the automatic detection of computational idioms in LLVM IR.
- An LLVM optimisation pass for idiom specific parallelisation.

Chapter 2

Background

The concept of computational idioms is central to our research. We use it as an umbrella term for several related concepts that have been studied by distinct research communities. We introduce the particular challenges of some individual computational idioms and describe the most commonly used parallelisation and optimisation strategies.

The LLVM compiler infrastructure will be used for implementing a prototype in a later chapter, we give an overview of the basic principles of the LLVM code base and intermediate representation language. We give some background on the benchmark software that we used.

2.1 Computational Idioms

Computational idioms are program structures that occur frequently and can be exploited via specific optimisation techniques to achieve performance increases. Examples of computational idioms are linear algebra, stencil computations and reduction operations. We will discuss each of these examples in more detail in later sections.

Several groupings and classifications of program structures were introduced by different scientific communities. They differ in their formality and motivation. We understand the term computational idiom as an umbrella term to encompass these different conceptualisations. We will use this concept of computational idioms as a vehicle to transfer parallelisation and optimisation techniques from well studied groups of programs into general purpose compilers.

2.1.1 Higher Order Functions

In functional programming languages, such as Haskell and OCaml, some idioms can be expressed very elegantly as higher order functions. Higher order functions take other functions as parameters to implement the actual function behaviour. Examples for this are most notably `map` and `fold`, which are used to implement two fundamental groups of computations, both with an inherent potential for parallelism. There are many more examples of higher order functions, among them `filter` and `scan`.

Functions that can be implemented via `map` take as their input an array and perform a per element operation to generate an output array of the same size. Crucially the same computation is performed for each value in the input array and each value in the output array is only dependent on the corresponding input value. In Haskell, `map` is defined as follows.

```
map           :: (a -> b) -> [a] -> [b]
map f []      =  []
map f (x:xs) =  f x : map f xs
```

In contrast, the `fold` idiom takes an array as input but computes only a single value. The following Haskell function definition fully specifies its functionality. The most common example of fold operations are sums over arrays.

```
foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      =  z
foldl f z (x:xs) =  foldl f (f z x) xs
```

Higher order functions can be used to describe many common computational workloads. The popularity of the MapReduce framework (Dean and Ghemawat (2008)) for example stems from the observation that many big data workloads exhibit characteristics that can be expressed efficiently using a combination of the higher order functions `map` and `reduce` (another name for `fold`). It provides an idiom based approach to the development of big data applications, enabling shorter development times and more predictable performance. The inspiration for this framework came from the observation that much of the existing, individually programmed software algorithms at Google implicitly used similar programming models already.

The implementation of higher level functions requires a powerful type system and can be only partially realised in most mainstream programming languages, such as C and Fortran. Template meta-programming in C++ is fundamentally a functional programming language and therefore allows for many of these concepts to be transferred to C++, but this is a painful endeavour.

The awareness of higher order functions can however still help to reason about potential parallelism and the best approach to uncover it. This was the main inspiration for the study of algorithmic skeletons as described in section 2.1.3. For example, the nature of the `map` function implies data-parallel behaviour. With a bit more sophistication, the `fold` function captures inherent parallelism as well, as will be discussed in detail in section 2.2.3.

2.1.2 Parallel Dwarfs

The so called Berkeley Dwarfs are a collection of 13 computational methods that together comprise a large portion of the common parallel computing workloads (Asanovic et al. (2006)). Each Dwarf is a computational pattern that is common in important applications and has persisted more or less unchanged for many years. The Dwarfs are inspired by numerical computations that arise in the scientific computing community, although the authors claim that the knowledge from this domain may prove useful in other areas as well.

As opposed to higher order functions, the Berkeley Dwarfs are a much more informal concept that was not developed for the use in programming languages and compilers. They were instead intended as a guideline for the evaluation of new hardware architectures and as a basis for new benchmarking tools. The Berkeley Dwarfs group applications mostly by the underlying mathematical methods that are used instead of specifying the algorithmic behaviour of the idioms.

The complete list of Berkeley Dwarfs as outlined in the technical report are **Dense Linear Algebra**, **Sparse Linear Algebra**, **Spectral Methods**, **N-Body Methods**, **Structured Grids**, **Unstructured Grids**, **Monte Carlo**, **Combinatorial Logic**, **Graph Traversal**, **Graphical Models**, **Finite State Machines**, **Dynamic Programming**, **Backtrack and Branch-and-Bound**. The first seven of these are taken directly from a previous work by Colella (2004).

Due to the lack of formality, the definition of the Berkeley Dwarfs is not directly suitable for compiler analysis. In later sections, we will show how parts of the Berkeley Dwarfs can be described in a more formal way however and some of them will prove useful for our approach.

2.1.3 Algorithmic Skeletons

Another concept that can be considered a subset of computational idioms is the notion of "Algorithmic Skeletons" (Cole (1991)). It was introduced as a way to classify the behaviour of parallel programs according to their organisation of workload distribution. This is mostly done with the motivation to introduce new, higher level programming models and tools for parallel programming. One major inspiration to the concept was the lack of functionality in mainstream programming languages to equal the well established higher order functions of functional programming languages, such as the **map** function to express element wise application of a single function to an array.

Among the "Algorithmic Skeletons" are **Fixed Degree Divide & Conquer** and the **Task Queue**. The concept of "Algorithmic Skeletons" has been used to implement many programming frameworks and libraries, including Skandium (Leyton and Piquer (2010)), Eden (Loogen et al. (2005)), eSkel (Cole (2004)) and SkelCL (Steuwer et al. (2011)). Furthermore, the Intel Thread Building Blocks library (Reinders (2007)) is based on this concept.

As opposed to the higher order functions used in functional programming and similarly to the Berkeley Dwarfs, the definitions for algorithmic skeletons are informal and not intended for automated reasoning. They are however heavily inspired by higher order functions and similarly describe the algorithmic structure of computations. This distinguishes them from the Berkeley Dwarfs, which are more focused on mathematical domains.

2.2 Idiom Specific Parallelisation Techniques

In this section we will discuss three specific computational idioms in more detail: stencil computations, linear algebra and reduction operations.

Stencil kernels and linear algebra are particularly important in the domain of scientific computing. Parallelism is generally easy to expose in many stencil computations. Achieving good performance however requires sophisticated code transformations to guarantee good cache locality. Numeric linear algebra is one of the best understood computational patterns and fast implementations of the relevant linear algebra operations have existed for many years. Different reduction operations are widespread in all of computing. We introduce basic parallelisation techniques and discuss the generalisation of reduction operations beyond their usual scope using insights from functional programming.

2.2.1 Stencil Computations

Stencil computations are a class of iterative computational kernels that operate on multidimensional arrays of floating point values. In each iteration, the values of each cell in the array are updated with a value computed as some function of a neighbourhood of that cell. The crucial feature of the stencil idiom is that the precise shape of the neighbourhood as well as the function applied to it are the same over the entire domain of the array.

A typical example of a stencil computation is the Jacobi kernel, where the new value of each cell is the average of the previous values of its four direct neighbours.

```
for(i = 1; i + 1 < n; i++)
  for(j = 1; j + 1 < n; j++)
    new[i][j] = 0.25 * (old[i][j+1] + old[i+1][j]
                        + old[i-1][j] + old[i][j-1]);
```

Stencil kernels are crucial in many computational domains, including most notably image processing and many areas of scientific computing, particularly computational fluid dynamics. While stencil computations are very regular and exposing parallelism is often easy, naive implementations can be very inefficient in their cache utilisation. The result of this is that stencil specific optimisation techniques can result in speedup of orders of magnitudes due to increased cache locality. Particularly important optimisation techniques in this context are the different forms of multidimensional iteration space tiling.

Traditional optimising compilers are generally unable to automatically apply these optimisation transformations, as they require higher-level reasoning about the code. This makes stencil kernels the perfect candidate for idiom specific compiler optimisations.

These characteristics also make stencil computations well suited for domain specific languages that can use domain specific knowledge to reason automatically about best cache utilisation and often create faster code than programmers using traditional languages. We want single out Halide (Ragan-Kelley et al. (2013)) here, that we intend to use in future iterations of our research to generate even faster stencil codes.

2.2.2 Linear Algebra

Efficient implementations of numeric linear algebra have been studied for many decades. A major advantage of linear algebra over the previously mentioned two computational idioms is the more restricted nature of linear algebra operations. While reduction operations and stencil kernels contain arbitrary computations as part of the computational idioms, linear algebra functionality can be implemented as a set of cleanly encapsulated subroutines.

The Basic Linear Algebra Subprograms (Anderson et al. (1999)) define a set of standard function interfaces that many library implementations of linear algebra functionality use. There are many competing library implementations of linear algebra functionality, in our research we used the Intel Math Kernel Libraries and OpenBLAS.

The most computationally intense linear algebra function is usually the multiplication of two large matrices. Similarly to stencil kernels, the tiling of the data into smaller chunks for increased cache locality is the most important optimisation strategy for optimising matrix multiplications and is used by most professional implementations. Furthermore the use of vector intrinsics for instruction level parallelism can lead to additional speedups.

Aside from the traditional dense linear algebra routines that are represented in the BLAS specification, there is sparse linear algebra. It deals with matrices that have almost all entries set to zero, making the standard approaches to linear algebra inefficient. The Intel Math Kernel Libraries implement sparse linear algebra routines that more or less mirror the BLAS interfaces.

2.2.3 Reduction Operations

2.2.3.1 Simple Reductions

The term **reduction operation** is generally used for simple computations that reduce the elements of an array of scalars onto a single output value. Other names for the same class of operations are **fold**, **accumulate** and **aggregate**. The most prototypical reduction operation is the sum over an array of integers.

```
for(i = 0; i < n; i++)
  output += input[i];
```

Reduction operations are of particular interest in the context of optimising and parallelising compilers as they exhibit inter-iteration dependencies but can still be mechanically parallelised. This works fundamentally by utilising the associativity of the underlying operation (in our example the addition). The simplest way to achieve this is as follows.

```
int temp = 0;

for(i = 0; i < n/2; i++)
  output += input[i];

for(i = n/2; i < n; i++)
  temp += input[i];

output += temp;
```

The two loops in the above code snippets have no interdependencies and can be executed concurrently. In the same way, the workload can be distributed over more than two processors by privatising the reduction variable to each thread as shown above (**temp** is a private copy of the reduction variable **output**).

The above example holds more generally for all binary operators that are associative, for example multiplication, minimum or bitwise union. It also extends to numeric data types besides integers, although there are subtle problems when working with floating point variables. This is because the approximate nature of floating point numbers does not preserve the law of associativity in all cases and the results of the above parallelisation approach are thus not guaranteed to be bitwise equal to the sequential implementation. The detailed investigation of these problems is beyond the scope of this work.

2.2.3.2 Insights from the Functional World

The concept of reduction operations can be extended far beyond the scope of sums and associative operations on scalars. To understand better the full generality of reduction operations, we can study the definition of reduction operations in the context of functional languages, which provide much more expressiveness for higher order functions than traditional programming languages.

In Haskell, one of the functions to implement reduction operations is foldl (fold left, corresponding to the order in which the loop is traversed).

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     =  z
foldl f z (x:xs) = foldl f (f z x) xs
```

This function can be used to generate code that is equivalent to the C snippets in the previous subsection.

```
let output = foldl (+) 0 input
```

The function interface allows for much more however. The elements of the array being reduced can be of any type, in particular it allows arrays and complex composite types. The same is true for the resulting value, which is also not restricted to be of the same type as the array elements.

For example, a simple integer histogram computation with four bins can be implemented as a reduction operation in the following way using foldl.

```
incBin::[Int]->Int->[Int]
incBin (x:xs) 0 = x+1 : xs
incBin []     n = []
incBin (x:xs) n = x : increment_bin xs (n-1)

histogram::[Int]->[Int]
histogram input = foldl increment_bin [0,0,0,0] input
```

Indeed histogram operations as above can be parallelised in much the same way as the simple sum of integers that was used as an example before. There are however a some difficulties with this.

The operator **incBin** in this example is an external binary operator meaning that it operates on two different types (**Int** and [**Int**]) and returns a value of one of those types ([**Int**]). It can therefore not be associative in the classical sense, as reordering computations naively would result in a type error.

```
incBin (incBin (incBin [1,2,3] 1) 2) 3 -- [2,3,4]
incBin (incBin [1,2,3] 1) (incBin 2 3) -- type error
```

There is however a way to make this work using an additional binary operator, in this example the element wise list addition.

```
addBins::[Int]->[Int]->[Int]
addBins  []     []    =  []
addBins (x:xs) (y:ys) = (x+y) : add_lists xs ys
```

We can now meaningfully change the parentheses by inserting the neutral element of the binary operator **addBins**, which is a list of zeros. The resulting values coincide and we get a relationship that resembles the law of associativity.

```
addBins (incBin (incBin [1,2,3] 1) 2) 3 -- [2,3,4]
addBins (incBin [1,2,3] 1)
        (incBin (incBin [0,0,0] 2) 3)   -- [2,3,4]
```

Using a more more abstract and lucid mathematical notation, writing **incBin** as '+' and **addBins** as '*', we get the following relationship for all $n, m \in \mathbb{N}$, integers x_i and starting histogram H (parentheses are omitted where enforced by type constraints to increase the readability).

$$(H + x_1 + \cdots + x_{n+m}) = (H + x_1 + \cdots + x_n) * ([0,0,0] + x_{n+1} + \cdots + x_{n+m}) \quad (2.1)$$

It is now clear that the terms on both sides of the '*' operator can be computed in parallel and thus the same methodology that was used for sums in our original example applies equally to histograms. This works whenever a complementing binary operator '*' exists to achieve the generalised definition of associativity that is implicit in the equation above. Simple reductions fit this scheme as well, the complementing operator in those cases simply happens to be the same as the original operator.

We have collected some examples of computations that can be written as reduction operations and parallelised using this approach.

computation	binary operator	complementing operator
sum	addition	addition
product	multiplication	multiplication
minimum	binary minimum	binary minimum
histogram	incBin	addBins
insertion sort	order preserving insertion	order preserving merge

2.3 The LLVM compiler infrastructure

LLVM is a compiler infrastructure and serves as the technical underpinning of many newer compilers, among them in particular the clang C/C++ compiler. The LLVM project encompasses many different components that are required for the development of high quality optimising compilers, all built around an intermediate program representation called LLVM IR.

One exceptional feature of LLVM is that it fully specifies its internal intermediate representation and freely exposes it to external programs. The LLVM intermediate representation (LLVM IR) is a fully typed, static single assignment assembly style language. Most LLVM optimising passes work on the level of LLVM IR and there are many tools in the LLVM infrastructure to handle LLVM IR, including an assembler, an interpreter and a linker to merge multiple LLVM IR modules. The significance of this intermediate representation in the design philosophy of LLVM is reflected in the former name of the project: "Low Level Virtual Machine".

The LLVM IR is designed to be agnostic to both programming languages and target hardware architecture. Most of LLVM is built around the intermediate representation and therefore the code base is very portable, supporting a plethora of different source languages and backends.

The presence of a well specified and accessible intermediate representation and the modular structure of the project make the LLVM infrastructure very extensible and allow its functionality to be easily utilised in external programs. This is reinforced by its permissive open source license, the LLVM Release License, which is based on similar terms as the BSD and MIT software licenses.

LLVM is a mature software project with backing from some of the most important technology companies of today, most notably Apple Inc. The optimiser is state of the art and the clang compiler is one of the most advanced and standard compliant C/C++ compilers and the default choice in several commercially used programming environments such as the XCode IDE.

Many device drivers use parts of the LLVM infrastructure to compile OpenCL source code and LLVM IR serves as the basis for the upcoming heterogeneous computing standard Standard Portable Intermediate Representation (SPIR). All these factors make LLVM the obvious choice as a code basis to prototype the ideas that we develop in this work.

2.4 Benchmark Software

The NAS Parallel Benchmarks (Bailey et al. (1991)) were developed by the NASA Advanced Supercomputing Division as a software tool to measure the capabilities of parallel supercomputers. The individual programs are specified algorithmically without stipulating implementation characteristics, explicitly allowing competing implementations. The individual benchmark programs perform computational tasks that are typical for computational fluid dynamics. In total, the benchmark suite consisted of five kernels and three more complex "simulated applications". It was further extended multiple times, adding four additional benchmark programs for unstructured computations, parallel I/O and data movement.

The Rodinia benchmarks (Che et al. (2009)) were developed specifically for heterogeneous computing environments. The collection includes OpenMP as well as CUDA versions to measure the impact of heterogeneous hardware accelerators on program performance. The individual benchmark programs are taken from scientific computing, engineering and data mining applications.

The PARSEC benchmark suite (Bienia (2011)) bundles thirteen benchmark applications from a number of different computational domains, including data mining, computer vision and financial analysis. Particular focus was put into selecting a representative range of programs that are not biased toward particular domains and to capture diverse program characteristics.

The Parboil Benchmarks (Stratton et al. (2012)) are a collection of benchmark programs that are each provided in different versions, including multiple levels of optimisation and heterogeneity. The individual programs are collected from different scientific and commercial fields including image processing, biomolecular simulation, fluid dynamics, and astronomy.

The PolyBench collection was conceived to assess the impact of compiler optimisations based on the Polyhedral Framework. It is a collection of simple computation kernels that are made up of nested loops that can be modelled using this mathematical framework. In total there are 30 individual benchmarks (version 3.2) implemented in plain C.

The San Diego Vision Benchmark Suite (Venkata et al. (2009)) is a collection of applications from the computer vision domain. It contains nine individual programs, each in a MATLAB and a C version. The reasoning for this is that MATLAB is the preferred language of researchers in computer vision, whereas C is the traditional systems programming language. As the intention of the benchmark collection is to enable platform developers to reason better about the performance of computer vision applications, the inclusion of the established languages of both communities is helpful. The applications are typically adopted from previously established MATLAB programs and have been ported by the authors to a subset of C that more or less directly adopts the original MATLAB program structure.

Chapter 3

Related Work

Related work to this dissertation comprises research from two different areas. First there is research on idiom specific compiler optimisation and parallelisation. Much of this work uses different terminologies and perspectives than we do, but they try to achieve similar goals.

Other relevant work is on automatic compiler based parallelisation in the context of heterogeneous hardware. This is not limited to idiom based approaches but comprises many different methodologies.

3.1 Idiom Specific Optimisations

The polyhedral framework (Kelly and Pugh (1995)) is a mathematical framework for the optimisation of well behaved nested loops. As the polyhedral framework requires particular algorithmic structures in the source code (well behaved nested loops), we consider it an idiom specific optimisation approach. It essentially defines an internal representation for instructions in nested loops that is well understood mathematically and can be manipulated with well defined mathematical transformations to expose parallelism and to improve runtime behaviour (e.g. cache locality).

Many domain specific languages have been build to perform optimisations based on the polyhedral framework, for example the PolyMage (Mullapudi et al. (2015)) system. Optimising functionality based on this system has also been integrated as a compiler pass into LLVM by the Polly project (Grosser et al. (2012)).

Extensions to the Polyhedral Framework have been proposed to allow it to capture more computational idioms as well, in particular regarding reductions. Such efforts are for example described in Doerfert et al. (2015). The authors discuss and implement a reduction-enabled scheduling approach as part of Polly and use the Polybench benchmark suite to evaluate it, achieving speedups of up to 2.21x.

Another approach to incorporate the reduction idiom into the Polyhedral Framework is described in Chandan Reddy and Cohen (2016), based on the Platform-Neutral Compute Intermediate Language (Baghdadi et al. (2015)). The approach in this publication is based on the Polyhedral Parallel Code Generator (Verdoolaege et al. (2013)) to generate CUDA and OpenCL code for multiple compute platforms.

The Halide domain specific programming language (Ragan-Kelley et al. (2013)) has been developed specifically for stencil kernels in the context of image processing. It allows programmers to specify optimisation techniques such as loop tiling and parallelisation strategies independent from the implementation of the functional logic. Halide can be used in conjunction with OpenTune, a program autotuner (Ansel et al. (2014)).

The authors of Mendis et al. (2015) use Halide to optimise stencil kernels that they reconstruct from compiles x86 binaries. Similar methods of code tuning using Halide are used by Kamil et al. (2016) but starting from Fortran code and using sophisticated automatic verification to guarantee correctness.

Andión (2015) describes a compiler based parallelisation approach for heterogeneous computing that is based on an idiomatic intermediate representation called KIR. This intermediate representation is based on the concept of diKiernels, which constitute algorithmic building blocks and are used to automatically generate OpenMP and OpenHMPP code. The authors propose a system that detects diKernels in conventional compiler IR and concatenates them to form contiguous sections of KIR. Individual examples of diKernels are scalar reductions and irregular assignments.

3.2 Compilation for Heterogeneous Computing

Much work has been done to improve the situation for heterogeneous systems consisting of CPUs and GPUs. The OpenCL standard Stone et al. (2010) provides a unified language and interface for compatible GPUs and CPUs, enabling a degree of portability between devices of different vendors. Several device dependent considerations remain important in practice, in particular the distribution of work among multiple available OpenCL devices, as OpenCL offers no scheduling capabilities. Several approaches have been proposed as abstraction layers on top of OpenCL to improve the ease of use and device utilisation by implementing rudimentary scheduling capabilities.

Sun et al. propose an interface that allows applications to enqueue OpenCL kernels for execution without explicitly assigning target devices Sun et al. (2012). The application may further stipulate interdependencies between these kernels to enforce sequential execution. The kernels are then scheduled to available OpenCL devices at runtime, using a scheduling policy that is provided by the user application. The runtime manages the individual OpenCL contexts and takes care of the necessary data transfers for devices with separate address spaces.

The PALMOS (Margiolas and O'Boyle (2015)) software is implemented as a separate process that runs in Linux user space and intercepts all communication between user programs and the actual OpenCL interface. PALMOS exposes a single, virtual OpenCL device to the user applications and manages the assignment of kernels to specific computing devices autonomously. This approach has advantages in that it requires no modification of existing software and enables the sharing of a pool of OpenCL devices amongst multiple applications.

Lee et al. take a different approach with the proposed Single Kernel Multiple Devices (SKMD) system Lee et al. (2013). It implements the OpenCL API and provides the user program with the illusion of a single OpenCL device. Instead of relying on the user program to provide multiple OpenCL kernels for effectively utilising multiple devices, SKMD distributes the threads of individual kernels across the available devices. In doing so, SKMD can simultaneously use several CPUs and GPUs.

These approaches still require the programmer to explicitly determine which parts of the program to run via OpenCL but remove the necessity to specify which of the available OpenCL devices to use. Other approaches that do not require the programmer to use OpenCL or similar APIs in the first place have been less successful and generally have to impose heavy restrictions on the initial program.

Notable in this context is the research by Barik et al. on the Concord system Barik et al. (2014). They present an approach for targeting heterogeneous architectures consisting of a CPU with an integrated GPU without the necessity for the programmer to use SPMD programming languages such as OpenCL or CUDA explicitly. The proposed system is build around a clang-based compiler for a subset of C++, lacking in particular the support for proper recursion, function pointers and exceptions and is limited to specific integrated GPUs by Intel that use the same physical memory as the CPU.

Approaches for automatically targeting heterogeneous MPSoCs include an article by Sheng et al., in which they investigate the difficulties of programming these platforms. They argue that the relatively short life cycle of embedded products call for a compiler framework that is adaptable and quickly retargetable for new platforms. Based on the model of Kahn process networks, the authors propose a C language extension called C for Process Networks (CPN) that requires the programmer to heavily annotate the code with data flow directives. The proposed compilation system is centred around a source-to-source compiler that generates C code for the individual components of the MPSoC.

Instead of relying on custom annotations, Chandramohan and O'Boyle implemented compiler based methods for targeting the processing cores of the Texas Instruments OMAP4430 MPSoC using the well established OpenMP programming model Chandramohan and O'Boyle (2014). The MPSoC consists of several processor cores using different ISAs that share the same physical main memory but provide no cache consistency and by default use distinct address spaces. Barrier placement and cache flush minimisation turned out to be crucial for performance and were optimised using cross-processor data dependence analysis. The system achieves a speedup of 38% and an improvement in energy efficiency of 40% on average over execution on the big cores alone.

3.3 Summary

There is a large body of work on compilation techniques for targeting heterogeneous architectures. Much of it is based on languages such as OpenCL and CUDA that are made specifically for heterogeneous computing but require experienced programmers. Other work that uses source code written in conventional languages such as C and Fortran generally relies on complex library interfaces and runtime systems. Some work based on the OpenMP programming model exists as well.

Idiom specific optimisation and parallelisation techniques have been studied extensively in the context of reduction operations and in the polyhedral framework. The notion of idiom specific approaches is not generally established in this research however, instead each of the approaches are confined to their own set of constraints. Some of the idiom based parallelisation techniques incorporate heterogeneous computing as well.

Chapter 4

Idiom Based Code Transformations - A Case Study

We study in detail the performance bottlenecks of two established benchmark collections. After extensive profiling work, we classify the identified bottlenecks according to the computational idioms that they represent, wherever we are able to identify such idioms. We establish the three most important computational idioms in the context of our benchmarks and study the relevant parallelisation and optimisation techniques that can be applied to them. Using only these idiom specific techniques, we develop parallel benchmark versions of the benchmarks and compare them to the optimised parallel versions of the original benchmark implementers.

4.1 Identification of Relevant Patterns

The underlying approach of this research project is to enable compilers to recognise specific computational idioms and to implement optimisation routines that exploit the knowledge of these idioms for parallelisation and runtime speedup. In this chapter, we seek to justify this approach by substantiating our expectation that a lot of bottleneck computations in common software can be described in terms of a limited set of specific computational idioms. We therefore performed an explorative study to identify the most important computational idioms and to find ways to exploit their characteristics to achieve runtime speedup.

Benchmark suites are generally designed to contain programs that accurately represent a cross section of specific computational domains in terms of the most

common computational workloads. It was therefore a natural choice to evaluate our hypothesis on several widely established benchmark collections.

Most benchmark programs are designated to evaluate the capabilities and performance characteristics of hardware platforms, but since they are usually distributed as source code, they are regularly used to compare the quality of the optimisers and code generators of different compilers as well. There exist some benchmarks that were developed specifically to test particular capabilities of optimising compilers, notably the PolyBench collection. These are however often less inspired by existing code bases and can misrepresent the effectiveness of optimisation techniques due to their benign and artificial nature. We therefore focused our efforts on more standard, widely accepted alternatives that originate from outside the compiler community.

We examined several established benchmark suites to get an understanding of the most important computational patterns. Particular benchmark suites that we considered for our work are the NAS Parallel Benchmarks, Rodinia, Parsec, Parboil and the San Diego Vision Benchmarks.

Different benchmarks of course focus on different computational domains and therefore the appropriate choice of benchmarks is of particular importance for our work. After some explorative analysis of the mentioned benchmark collections, we focused our efforts on two benchmarks suites, the Parboil benchmarks and the NAS Parallel Benchmarks. Both of these collections have their roots in scientific computing, where parallel computing and heterogeneous computing in the form of general purpose graphics processing units (GPGPUs) are firmly established and recognised as critical to performance.

4.1.1 NAS Parallel Benchmarks

The NAS Parallel Benchmaks (NPB) are a very mature and well studied set of programs from the domain of computational fluid dynamics. They were devised by the NASA Advanced Supercomputing Division to supersede previously used benchmarks such as the Livermore loops and the LINPACK benchmark, which suffered from many inadequacies, including their limited problem sizes. The original benchmark collection consisted of five computational kernels and three pseudo applications that attempt to cover the most important computational workloads that occur in computational fluid dynamics. Several smaller additions

have been made to the original set of programs but those will not be of major importance for this work.

In contrast to all the other benchmark suites that we investigated, the NAS Parallel Benchmarks are specified in a "pencil-and-paper" fashion. This means that the original developers specified what exactly an implementation of the NAS Parallel Benchmarks has to compute but implementation details such as the used programming languages are not stipulated. As a result, there are competing implementations written in different languages by multiple implementers. The NAS Parallel Benchmarks were originally conceived in the early 1990s and as such they are very well established and there are mature and extremely well optimised implementations available. This makes them very useful to explore the limits of our approach, as the parallel versions of the NPB are quite hard to optimise any further.

For our experiments, we used the implementation by Seoul National University (SNU NPB Suite). This is due to their choice of the programming language C and the very carefully crafted, high quality source code. Furthermore this implementation ships with multiple versions, including sequential, OpenCL and OpenMP based versions.

4.1.2 Parboil Benchmarks

The Parboil Benchmarks were developed at the University of Illinois and span a larger amount of computational domains than the NAS Parallel Benchmarks. It comprises eleven individual programs from the fields of image processing, biomolecular simulation, fluid dynamics and astronomy. Each of he individual benchmark programs comes in several varieties, including sequential, parallel and OpenCL versions.

The Parboil Benchmarks are less mature than the NAS Parallel Benchmarks and the implementation is not as carefully optimised. This makes the Parboil Benchmarks more representative of average real code bases and helps us highlight the strengths of our approach in a less unforgiving environment.

4.1.3 Bottleneck identification

Using profiling techniques we identified the bottleneck computations of each of the benchmark programs. We first used gprof to establish the bottleneck to

a function level granularity and then inserted manual timing routines into the original source code to narrow down the crucial computations. We then tried to express each bottleneck in terms of computational idioms and grouped the benchmark programs by our findings. In some cases, the bottlenecks did not correspond to any computational idiom that we are aware of.

For most of the computational kernels in the NAS benchmarks, the important performance bottlenecks were very clearly identifiable and corresponded perfectly to a narrow range of computational idioms.

The **Conjugate Gradient** kernel spends the majority of its runtime in a basic sparse matrix vector multiplication. The **Integer Sort** kernel has its only relevant performance bottleneck in a simple integer histogram computation that comprises only two source code lines in the sequential version of the benchmark. The **Multi-Grid** kernel spends most of its runtime in two routines that perform standard three dimensional first order stencil computations. Roughly half of the runtime of the **Embarrassingly Parallel** kernel is spent on a complex reduction computation to generate a histogram of previously generated (pseudo-)random variables. We were unable to classify the **Discrete 3D Fast Fourier Transform** kernel as well as the two additional kernels (**Unstructured Adaptive Mesh** and **Data Cube**) that were not part of the original version of the NAS Parallel Benchmarks into any computational idiom that would be useful for our further methodology.

All the three pseudo applications contained in the NAS Parallel Benchmarks (**Lower-Upper Gauss-Seidel solver**, **Block Tri-diagonal solver** and **Scalar Penta-diagonal solver**) spend significant portions of their runtime performing stencil computations.

The programs of the Parboil Benchmarks paint a similar picture. Two of them have clear-cut stencil computations as their bottleneck, the **Lattice-Boltzmann Method Fluid Dynamics** benchmark and the unambiguously named **stencil** program. The **Breadth First Search** kernel unsurprisingly performs breadth first graph traversal. The **sgemm** and **spmv** programs implement undisguised dense and sparse linear algebra respectively. The **histo** program implements a saturating histogram and the **Two Point Angular Correlation Function** program spends the vast majority of its runtime on a rather complex computation that fundamentally is a histogram.

4.1.4 Bottleneck classification

Using the previous findings, we came up with a set of three computational idioms that cover more than 60% of the identified bottlenecks of both benchmark suites. We decided to focus on these three idioms for our further investigations.

The first computational idiom is linear algebra. This finding should be entirely uncontroversial as linear algebra has for a long time been studied in the context of high performance computing and there is a broad consensus on its importance. In practice, the term linear algebra of course actually covers quite a lot of different computations (BLAS alone specifies several dozen function interfaces), but we found only a very limited set of them to be performance critical in the studied benchmarks, namely dense matrix-matrix multiplications sparse matrix-vector computations.

Stencil computations form the second relevant computational idiom. This again is entirely unsurprising, as stencil computations, like linear algebra, are a widely studied subject in high performance scientific computing. Sophisticated optimisation approaches such as the polyhedral model stem from the recognition of stencil computations as a crucial and well-understood computational pattern. All of the stencils that we found came in fact from a very restricted subset of general stencil computations, namely first order stencil computations.

The third computational idiom that we identified as pervasive are complex generalised reduction operations. While scalar reductions are a well studied subject, we feel that our identification of more complex reduction operations as an important computational idiom needs some further elaboration (cf. the corresponding section in the Background chapter). We will address one example here in detail to make clear what kind of reduction operations we mean in this context.

4.1.5 The Reduction Idiom in NPB EP

We use as an example one of the two computational bottlenecks of the **Embarrassingly Parallel** benchmark from the NAS Parallel Benchmarks. The benchmark performs statistical calculations using Monte-Carlo methods. It comprises the following short code snippet. We omit the variable declarations, as most of the types are implicit from the shown source code.

```
for (i = 0; i < NK; i++) {
  x1 = 2.0 * x[2*i] - 1.0;
  x2 = 2.0 * x[2*i+1] - 1.0;
  t1 = x1 * x1 + x2 * x2;
  if (t1 <= 1.0) {
    t2   = sqrt(-2.0 * log(t1) / t1);
    t3   = (x1 * t2);
    t4   = (x2 * t2);
    l    = MAX(fabs(t3), fabs(t4));
    q[l] = q[l] + 1.0;
    sx   = sx + t3;
    sy   = sy + t4;
  }
}
```

We claim that this is a complex reduction operation and can be parallelised in the mechanical way set out in the Background section. To prove this, we need to find the two binary operators as described before. First we need to indentify the types and the resulting function signatures for the operators that will be used in the reduction definition.

```
struct Type1
{
  double q[10];
  double sx;
  double sy;
};

struct Type2
{
  double x1;
  double x2;
};

Type1& operator+=(Type1& a, const Type2& b);

Type1& operator*=(Type1& a, const Type1& b);
```

Now we can implement the operators. The "+=" operator can be taken more or less directly from the original source code. The complementary "*=" operator on the other hand is implied by the definition of the "+=" operator.

```
Type1& operator+=(Type1& a, const Type2& b) {
  double x1 = 2.0 * b.x1 - 1.0;
  double x2 = 2.0 * b.x2 - 1.0;
  double t1 = x1 * x1 + x2 * x2;

  if (t1 <= 1.0) {
    double t2 = sqrt(-2.0 * log(t1) / t1);
    double t3 = (x1 * t2);
    double t4 = (x2 * t2);
    int    l  = MAX(fabs(t3), fabs(t4));
    a.q[l]    = q[l] + 1.0;
    a.sx      = a.sx + t3;
    a.sy      = a.sy + t4;
  }
  return a;
}

Type1& operator*=(Type1& a, const Type1& b) {
  for(int i = 0; i < 10; i++) a.q[i] += b.q[i];
  a.sx += b.sx; a.sy += b.sy;
  return a;
}
```

Using these definitions we can now rewrite the original code snippet in a way that makes the reduction character of the performed computations immediately obvious. The code below is still sequential but can be parallelised using the "*=" operator in the way described in the background section 2.2.3.

```
Type1 result;
for(i = 0; i < 10; i++) result.q[i] = q[i];
result.sx = sx; result.sy = sy;

for (i = 0; i < NK; i++) {
  Type2 value = {x[2*i], x[2*i+1]};
  result += value;
}

for(i = 0; i < 10; i++) q[i] = result.q[i];
sx = result.sx; sy = result.sy;
```

4.1.6 Summary

The tables 4.1 and 4.2 summarise how we classified the performance bottlenecks in the individual programs making up the two studies benchmark suites.

Table 4.1: Computational Patterns in the NAS Parallel Benchmarks

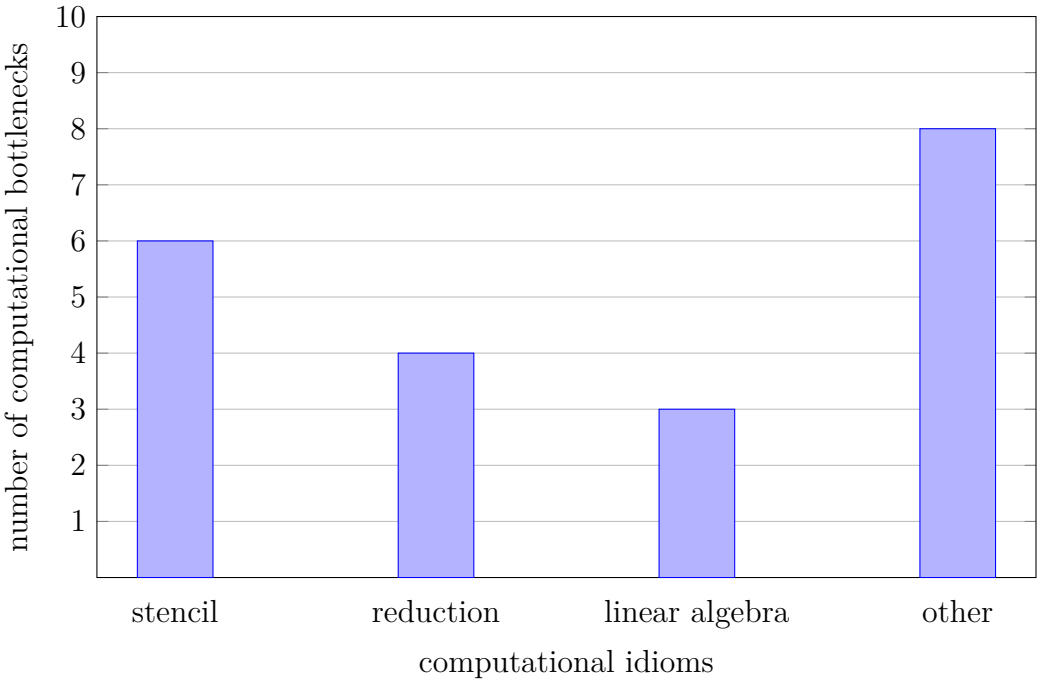
Name	Description	Bottleneck
BT	Block Tri-diagonal solver	stencil computation
CG	Conjugate Gradient	sparse linear algebra
DC	Data Cube	other
EP	Embarrasingly parallel	reduction operation
FT	Discrete 3D fast Fourier Transform	other
IS	Integer Sort	reduction operation
LU	Lower-Upper Gauss-Seidel solver	stencil computation
MG	Multi-Grid	stencil computation
SP	Scalar Penta-diagonal solver	stencil computations
UA	Unstructured Adaptive Mesh	other

Table 4.2: Computational Patterns in the Parboil Benchmarks

Name	Description	Bottleneck
bfs	Breadth-First Search	other
cutcp	Distance-Cutoff Coulombic Potential	other
histo	Saturating Histogram	reduction operation
lbm	Lattice-Boltzmann Fluid Dynamics	stencil computation
Mri-gridding	Magnteic Resonance Imaging Gridding	other
Mri-q	Magnetic Resonance Imaging Q	other
sad	Sum of Absolute Differences	other
sgemm	Dense Matrix-Matrix Multiply	dense linear algebra
spmv	Sparse-Matrix Dense-Vector Multiply	sparse linear algebra
stencil	3-D Stencil Operation	stencil computation
tpacf	Two Point Angular Correlation	reduction operation

We were particularly encouraged by the similarity of the results in both benchmark suites, hinting at a generalisability of our findings. The prevalence of the three identified important bottlenecks is shown in figure 4.1.

Figure 4.1: Prevalence of individual computational idioms



4.2 Fast Pattern Implementations

After having identified the most important computational idioms in both examined benchmark collections we then looked for their best performing implementations. We did this by experimenting on the original benchmark source codes while keeping track of our modifications and their impact, trying to single out the most effective code transformations.

4.2.1 Linear Algebra

Linear algebra is a very well studied field of numerical mathematics and there are many established library implementations of it. The specification of the Basic Linear Algebra Subprograms (BLAS) serves as an common established interface to many different library implementations of fundamental linear algebra functionality.

We experimented with two well known implementations of BLAS, the Intel Math Kernel Libraries (Intel MKL) and OpenBLAS. Both of them performed very similar in our experiments and we decided on OpenBLAS for dense linear algebra in the end, due to its open source nature and better performance portability. For

sparse linear algebra, we experimented with the corresponding implementations in Intel MKL but achieved no significant speedups over naive parallelisation for sparse matrix vector products.

The BLAS specification subdivides its functionality into three levels, level 1 for vector operations only involving vectors, level 2 for matrix-vector operations and finally level 3 for operations involvng multiple matrices. We found only the functionality of level 3 to be beneficial to performance, presumably by superior cache behaviour when compared to naive C code. Implementing the functionality of the first two levels using function calls to BLAS on the other hand frequently was detrimental to performance in the benchmarks. We assume that this is because such an approach impedes the compiler from optimising properly, for example by fusing the loops of consecutive vector additions.

4.2.2 Stencil Kernels

Stencil computations are also well studied but do not fit a library interface as nicely as linear algebra. This is because the concept of stencil kernels is parameterised by arbitrary neighbourhoods as well as update functions. Implementations using compile time concepts such as template meta-programming in C++ could be conceived but a concise specification like BLAS for linear algebra is not viable for stencil computations.

The optimisation of Stencil computations and in particular the efficient distribution of stencil workloads with associated synchronisation constructs is however a very well studied subject. Exposing parallelism in stencil computations is generally relatively easy. The main challenge is instead to achieve good cache locality, which often requires optimisations across several stencil codes. Techniques such as overlapped tiling and split tiling have been developed to this aim.

This knowledge used to be mostly preserved in terms of best practices in the high performance computing community. It would be very desirable to conserve this knowledge into reusable libraries, but as mentioned above this is not easily achievable. In the last years however, several domain specific languages have been developed to specifically generate fast stencil computations, most notably Halide. We experimented with it but found that for the relatively benign stencil computations in our benchmark programs we were able to match its performance by handwritten code. We found it sufficient to fuse together consecutive stencils

where possible and to use overlapped tiling. In later stages of our research, we plan to revisit Halide more in depth.

4.2.3 Reduction Computations

Reduction operations are well established as well, but rarely in the generality that we require. The way to implement fast parallel reduction operations however is well established and relies on privatising the reduction variable(s) and then merging the results at the end. This of course adds some overhead but is generally faster than using a mutex for every single read modify write access to the reduction variable, which would otherwise be necessary.

For histogram operations in particular (a subset of reduction operations) it can also be possible to guarantee the access keys of the different threads to be distinct. This however generally requires knowledge about the distribution of the key values. We demonstrate we will demonstrate this on a simple code example from the NAS Integer Sort benchmark.

```
for(i=0; i<NUM_KEYS; i++)
  key_buff_ptr[key_buff_ptr2[i]]++;
```

We evaluated three different possibilities to split this loop into parts that can be executed in parallel. Firstly, the read-modify-write access to the histogram array can be treated as a critical section. This results in significant performance degradation, as almost the entire loop content becomes serialised (with the execption of reading values from the second array).

Secondly, we can use the approach outlined in the corresponding Background chapter, namely privatising the reduction variables and merging them afterwards.

```
int local_key_buff[MAX_KEY];
for(i = 0; i < MAX_KEY; i++)
  local_key_buff[i] = 0;

for(i = 0; i < NUM_KEYS/2; i++)
  key_buff_ptr[key_buff_ptr2[i]]++;

for(i = NUM_KEYS/2; i < NUM_KEYS; i++)
  local_key_buff[key_buff_ptr2[i]]++;

for(i = 0; i < MAX_KEY; i++)
  key_buff_ptr[i] += local_key_buff[i];
```

As a third option, we can split the second array in two parts in such a way that we can guarantee their contents to be disjunct. This would typically be done by a threshold value, which should be as close as possible to the median value of the array contents to result in an even split. This method can therefore not be applied in general, only when the distribution of the values is known at least approximately.

```
int buff_part1[NUM_KEYS];
int buff_part2[NUM_KEYS];
int num_buff_part1 = 0;
int num_buff_part2 = 0;

for(i = 0; i < NUM_KEYS; i++)
  if(key_buff_ptr2[i] < MAX_KEY/2)
    buff_part1[num_buff_part1++] = key_buff_ptr2[i];
  else
    buff_part2[num_buff_part2++] = key_buff_ptr2[i];

  key_buff_ptr[key_buff_ptr2[i]]++;

for(i = 0; i < MAX_KEY; i++)
  local_key_buff[i] = 0;

for(i = 0; i < num_buff_part1; i++)
  key_buff_ptr[buff_part1[i]]++;

for(i = 0; i < num_buff_part2; i++)
  key_buff_ptr[buff_part2[i]]++;
```

The reference parallel implementation of NAS Integer Sort uses an approach along the lines of the third option. In our evaluations this turned out to be generally the fastest approach (20% to 50% faster than the second approach on an Intel processor with 8 threads depending on the data sizes). This is mostly due to the relatively large histogram size with associated privatisation costs as well as due to the completely uniform distribution and hence perfect division of the values in the second array.

4.3 Evaluation

In most cases idiom specific parallelisation approaches can match or even outperform parallel versions implemented by expert programmers. The one major exception to this is the Embarrassingly Parallel benchmark of the NAS parallel benchmark suite, where the parallelism is not contained in a computational idiom but instead encompasses the entire program.

4.3.1 Experimental Setup

We used our findings of the previous section to devise optimisation 'recipes' for the different computational idioms.

In **dense linear algebra** the performance critical sections were always matrix matrix multiplications, operations that involved only vectors never constituted performance bottlenecks. As an optimisation recipe we simply replaced all naive implementations of dense generalised matrix matrix multiplications with calls to the appropriate OpenBLAS `*gemm` routine. OpenBLAS then automatically parallelises the operation. For **sparse linear algebra** we parallelised instances of sparse matrix dense vector multiplications using simple OpenMP directives. As mentioned previously, more sophisticated approaches using the Intel MKL library did not provide additional speedup versus this naive parallelisation approach and performed worse on our AMD Opteron based test system.

Sencil computations required more sophisticated methods. In a fist step, we parallelised stencil computations whenever possible using OpenMP directives. Whenever possible we merged consecutive stencils. Finally we applied overlapped tiling strategies wherever possible and experimented manually with different tile sizes to get maximum performance.

Reduction computations required the most drastic changes to the original source code. For each reduction bottleneck we generated several functions and used the pthreads library to orchestrate them according to the approach that we outlined in section 2.2.3. First we defined structures to hold the reduction variables. We then created a function that contains a modified version of the original source code of the reduction operation that uses this structure to encapsulate the reduction variables and that can be restricted to a subsection of the iteration space. Another function was generated to coordinate the creation of threads, each of which call the aforementioned function. Finally a third function is used

to merge the reduction variables of the individual reduction threads together at the end. We then replaced the original reduction source code with a call the the thread coordinating function and determined the optimal number of threads to spawn.

We ended up with three versions for each of the benchmark programs. As baselines we used both the original sequential and parallel versions of the benchmarks that were provided by the implementers. As a third version we used the version that we obtained from applying our idiom based optimisation recipes to the sequential versions. In cases were the original benchmark shipped with multiple parallel versions, we always chose the fastest one to compare against.

Since our optimisations were restricted to idiom specific approaches, the original parallel versions had more optimisation potential and can therefore expected to outperform our idiom based parallel versions in many cases. As our idiom based optimisations relied on established programming techniques, the original parallel version and our idiom based parallel version were in same cases quite similar. This concerns mainly the stencil based benchmarks. For two of the benchmarks of the NPB collection the original parallel version was entirely parallelised using idiom based approaches and we did not develop a separate version (LU and MG). The same is true for the CG and spmv benchmarks.

We built the three versions of each benchmark with the highest compiler optimisation setting `-O3` using the `clang` and `clang++` compilers version 3.8. All performance measurements were done on a single computer featuring four AMD Opteron processors with 16 processor cores each. Each benchmark version was executed five times to avoid random fluctuation in the runtime measurements and the average runtime was recorded. When benchmark programs required additional data input (Parboil Benchmarks), we used the original input files that are provided with the benchmark source code. For the NAS Parallel Benchmarks we used the problem size class A.

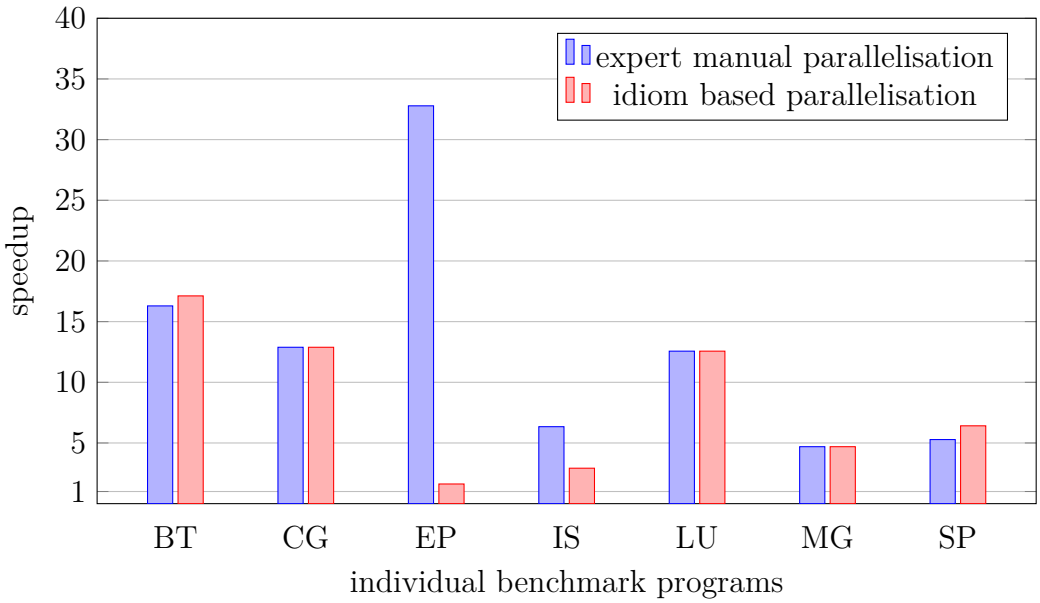
4.3.2 Results

In most of the NAS Parallel Benchmarks we are able to match the original parallel versions. unsurprisingly there is not a lot of optimisation potential over the original parallel versions as they have been hand tuned by expert programmers over several decades. The fusing of consecutive stencil computations in the **BT** **SP** benchmarks resulted in minor speedups but the difference is close to negligible.

We achieve good speedup against the sequential baseline in the **IS** benchmark but reach only about 50% of the performance of the original parallel version. This is because the original parallel version uses an approach that is based on the fact that the computation performed is a histogram with uniformly distributed input data. This allows the implementers to sort the keys into disjunct buckets and then to parallelise the histogram computation very efficiently.

The parallelism of the **EP** is not contained in a computational idiom and instead encompasses the entire program. Although our idiom based approach achieves more than an order of magnitude of speedup on the main bottleneck of the program (a reduction operations), the remaining sequential section of the program results in a mere 1.6x speedup against the sequential baseline.

Figure 4.2: NAS Parallel Benchmarks, sequential baseline



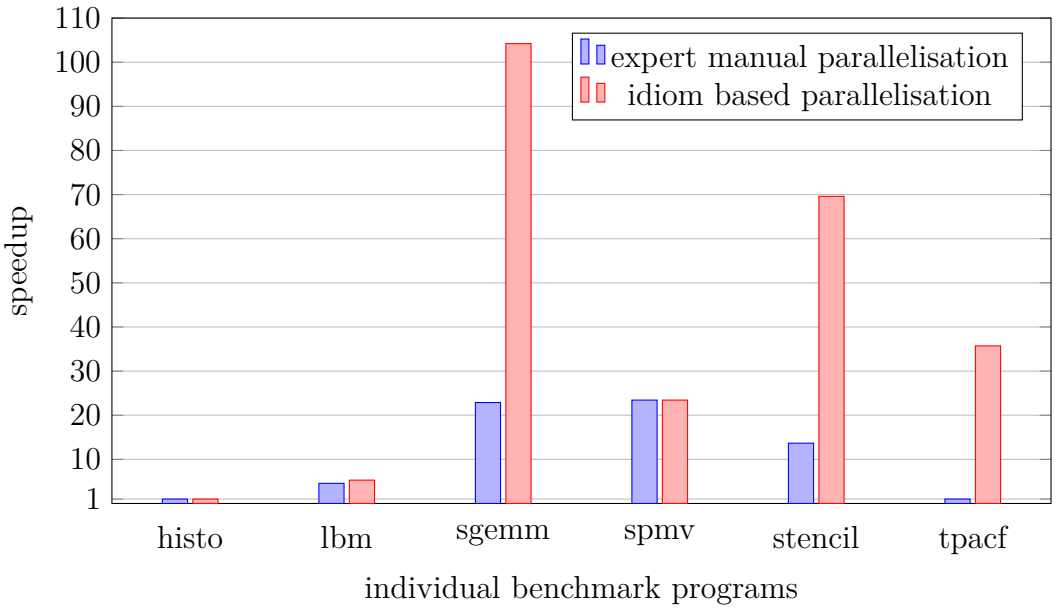
The Parboil Benchmarks offered much more unused parallelisation and optimisation opportunity over the original parallel versions than the NAS parallel Benchmarks. We were able to outperform all of the benchmarks aside from the

spmv program.

The original parallel reduction operations were implemented using critical sections, resulting in abysmal scalability. For the **tpacf** benchmark, we achieved very good scalability and a 35.7x speedup versus the sequential benchmark version. No parallel implementation of the **histo** program was able to beat the sequential version. This was due to the very unfavourable input data that was so small that the overhead of multithreading ruined all potential parallel speedup.

On the linear algebra benchmark **sgemm** we achieved 104x speedup against the sequential baseline using 64 processor cores, compared to 22.8x speedup of the original parallel version. This was mostly due to the superior cache locality of the OpenBLAS implementation. On the **stencil** benchmark the use of efficient tiling strategies resulted in 5.1x speedup when compared against the original parallel version.

Figure 4.3: Parboil Benchmarks, sequential baseline



4.4 Summary

The computational bottlenecks of many important programs can be meaningfully classified into different computational idioms. Important computational idioms include linear algebra, reduction operations and stencil kernels. Computational idioms contain contextual information that is often sufficient to parallelise sequential code efficiently.

Chapter 5

Constraint Based Idiom Detection - Theory and Practice

In this chapter we specify a formal language for the description of computational idioms. We develop algorithmic methods to identify the specified computational idioms in LLVM intermediate representation code. We implement thee algorithms in a modified version of the LLVM `opt` optimising tool and analyse to what extend we can recreate the manual results from the previous chapter.

5.1 Problems with Syntax

In the previous chapter we showed that computational idioms in many cases contain all the contextual and semantic information that is necessary to optimise and parallelise applications. The proof of concept study in that chapter relied on manually applying the idiom based code transformations to the program source code however. In this chapter, we will develop the methodology to automatise this process and draft a prototype implementation as a compiler optimisation pass in the LLVM compiler infrastructure.

For compilers to automatically recognise and replace specific computational idioms, we needed to specify them in a way that can be evaluated automatically inside the optimisation infrastructure. We considered several different approaches to this challenge. Maybe the most obvious approach, at least from a linguistic perspective, is to define the idioms as grammatical structures in the syntax of a specific programming language. This is a very natural approach to formalising computational idioms but it has some severe shortcomings, the limited portability

between programming languages being only the most superficial of them.

Programming languages generally contain multiple syntactic structures for similar underlying algorithmic concepts. A typical example of this is the presence of 'for' and 'while' loops in languages inspired by the programming language C, both of which can be used to implement the exact same functionality. While these semantically redundant syntactic constructs add to the complexity of the programming language, they can serve as syntactic sugar and make programmer intentions more explicit.

Any syntax based idiom specification would be massively complicated by these redundancies however, as its grammatical rules would have to consider all of them. This would likely make those rules too complicated to be practically viable. Instead we need a normalised form of the source code that removes as much as possible the programmer's freedom to express the same algorithmic concepts in multiple different ways.

Furthermore an approach based on syntax alone can not enforce semantic constraints that are outside the scope of the grammatical structure. This is the more fundamental of the two shortcomings because the syntax of complex programming languages is often next to meaningless without a significant amount of contextual information, in particular concerning types. The syntactic structure is quite hollow in C++ for example, because all values could be objects of classes with overloaded operators. As an example consider this C++ source code.

```
for(i = 0; i < n; i++)
   output += input[i];
```

This looks like a typical reduction operation when only the syntax is known. However without exact knowledge of the types of all the variables, this can be a very misleading assumption. For example, the variable `output` could be an object of the following class.

```
class Duplicator
{
public:
  Duplicator& operator += (int& operand) {
    operand *= 2;
    return this;
  }
};
```

This changes the meaning of the above syntax completely and the performed computation is actually a mapping operation without reduction characteristics. This problem can be partially solved by using grammatical patterns only as a preprocessing pass to find idiom candidates and then resorting to additional, custom rules that use contextual information and type constraints to weed out false positives. This however then simply moves the crucial bits of the idiom detection methodology into these additional rules and prevents any systematic and unified analysis.

We therefore concluded that an approach based on the syntactic structure of the source programming language is fundamentally flawed, as it struggles to deal with syntactic sugar and generally with grammatically complex languages and fails almost entirely in the presence of complex type systems. Instead we formalise the idioms not in the source programming language but on a less ambiguous level in modern compilers, more concretely on single static assignment intermediate representation.

Such an intermediate representation has multiple desirable properties. Firstly it is designed to be a suitable language for compiler optimisation transformations. This means that it is relatively easy to reason about, syntactically primitive and its semantics are well specified. Secondly the embedding into an existing compiler means that we can use optimisation passes to normalise the code. Furthermore an approach based on an intermediate representation is agnostic to the source programming language and the target hardware architecture.

The main disadvantage of this approach is however that we can not use the established techniques from domains such as formal languages, as all meaningful information of intermediate representation code is contained in the semantics and the syntactic structure is very primitive and useless for analysis. Instead of being able to rely on algorithms like AST matching or acceptors for formal languages we therefore had to come up with a novel way of detecting structure in single static assignment intermediate representation.

5.2 Arguments for LLVM

Most modern optimising compilers for traditional programming languages utilise an intermediate representation based on the Static Single Assignment form. This is mainly because it offers desired properties that make reasoning about it relatively easy in the context of valid optimisation transformations.

Instead of relying on an abstract definition of SSA intermediate representations, we decided to use LLVM IR as a concrete implementation of the concept. This gives us the advantage that we are able to quickly transition from our theoretical underpinnings to a functioning prototype implementation at the cost of some loss of abstraction. We think that this is a favourable tradeoff but are aware of a certain dichotomy when formalising abstract concepts such as computational idioms in something very concrete and somewhat arbitrary like LLVM IR.

The choice for LLVM IR in particular was made for the following reasons.

- The code base of LLVM is generally considered of high quality and more accessible than those of competing projects, gcc in particular. It is well documented, state of the art, under active development and designed to be easily extensible and reusable.
- LLVM supports a wide variety of popular programming languages as well as target architectures.
- We can use the already existing optimisation and code transformation passes to complement our own work.
- It is used by many device drivers for compiling OpenCL and upcoming standards such as SPIR will also be based on LLVM IR.

All the concepts that we introduce can easily be transferred to other SSA based intermediate representations. Since most modern compilers use such a form of intermediate representation internally, this makes our general methodology independent of the LLVM infrastructure. However we did not see any gains in understanding to be achieved from abstracting away the concrete LLVM based details such as the names of instruction opcodes.

5.3 Formal Problem Description

To understand our problem at a deeper level, we want to formalise the task that we try to achieve. It is our goal to specify computational idioms in such a way that for a given segment of LLVM IR code, a program can tell us whether this code contains the computational idiom or not. Futhermore the program should be able to tell us which exact part of LLVM IR corresponds to which component of the idiom.

If the computational idiom is for example a matrix vector multiplication, then the program should tell us which exact LLVM IR value constitutes the array that contains the vector elements. We can express this in the following entirely shallow definition.

Definition 1 *A computational idiom is defined as a pair $(\mathcal{I}, c)$, where $\mathcal{I}$ is an index set and c is a binary predicate on $(\mathcal{V} \cup \{*\})^{\mathcal{I}}$, where $\mathcal{V}$ is the set of all LLVM IR values, corresponding to the* ***llvm::Value*** *class.*

In prose this simply means that a computational idiom consists of a certain number of LLVM IR elements, some of which can be omitted using $*$, and each tuple of that many elements either is or is not an implementation of the idiom. To give a bit more meaning to this formalism, we give an example.

Example 1 *To specify the sum of two integers as a computational idiom, let*

$$\mathcal{I} = \{\textit{first summand}, \textit{second summand}, \textit{result}\}.$$

We can then specify the binary predicate c as follows

$$\begin{aligned} c(x) = \textit{true} \iff\ & x_{\textit{result}} \textit{ is an add instruction and} \\ & x_{\textit{result}} \textit{ is of integer type and} \\ & x_{\textit{first summand}} \textit{ is the first operand of } x_{\textit{result}} \textit{ and} \\ & x_{\textit{second summand}} \textit{ is the second operand of } x_{\textit{result}}. \end{aligned}$$

Notice how some properties in the example are left unspecified, for example the summands do not have to be instructions but can also be constants, function arguments etc. Furthermore the type of the summands is enforced by LLVM IR, so we do not have to explicitly mention it.

We need to find a good way to encode the binary predicates in some descriptive language that we can then realise in our prototype implementation. The above example gives a good indication as to how we can achieve that. It shows two of the main characteristics of the description language that we devised. Firstly, the predicate c is made up by logically concatenating several conditions that form the building blocks of our description language. Secondly, the individual conditions are based on the LLVM type system, instruction opcodes and on the structure of the data flow graph.

5.4 Constraint Based Idion Specification

Idiom specifications in our system are constructed by hierarchically combining simple conditions using logical operators. This means that fundamentally we are not defining 'bottom-up' what a computational idiom looks like but instead restrict the space of all programs down until only the constructs remain that we desire. Therefore we call this a constraint based approach.

In this section we will introduce a formal language for the specification of computational idioms. From what was already discussed, part of the language specification in Backus-Naur Form is already implicit.

```
constraint      ::= (<constraint>)  |   <or-constraint>
                 | <and-constraint> | <atomic-constraint>

or-constraint   ::= <constraint> or <constraint>

and-constraint  ::= <constraint> and <constraint>
```

The semantics of this part of the language should be immediately obvious, the crucial missing part is only the specification of the atomic constraints that are supported by the system. They can be grouped roughly into two different groups. Firstly there are graph based constraints that are based on the different dependency graphs underpinning the structure of the intermediate representation code. Secondly there are constraints that are restrict the properties of individual values by determining types, constant values etc.

5.4.1 Graph Based Constraints

The graphs that capture the dependencies between individual instructions are an important source of semantic information in SSA intermediate representation languages. These include the data flow graph (DFG), control flow graph (CFG), control dominance graph (CDG) and program dependence graph (PDG).

Enforcing certain graph edges, reachability and graph domination properties and excluding others therefore has to be the basis of any idiom specification system. This gives rise to the following specifications.

```
graph ::= DFG | reverse DFG | CFG | reverse CFG
        | CDG | reverse CDG | PDG | reverse PDG
two-element-graph-constraint ::=
    <element> has edge to <element> in <graph>        |
    <element> dominates <element> in <graph>          |
    <element> strictly dominates <element> in <graph> |
    <element> can reach <element> in <graph>
```

In these definitions, `<element>` is an element of the set $\mathcal{I}$ that was introduced in the formal definition of computation idioms that we introduced previously.

Aside from the standard graph constraints, we perceived the need for two generalisations of the concept of graph dominance. Graph dominance means roughly speaking that one node 'blocks' the reachability of the other when starting from the graph origin. This concept is very powerful but the use of the global origin of the graph can be limiting. For our methods we need a concept of local dominators, which we specify as follows.

```
three-element-graph-constraint ::=
    <element> can not reach <element>
    without passing <element> in <graph>
```

The second generalisation that we need is the concept of joint dominance. This concept simply requires that any path from the graph origin to a specific node has to pass through at least one of a specified set of nodes. In general, none of the nodes in that set have to dominate the destination node on their own.

```
n-element-graph-constraint ::=
    <element> can not reach <element>
    without passing any of <element-list> in <graph>
element-list ::= <element> | <element>,<element-list>
```

5.4.2 Miscellaneous Constraints

Aside from the graph constraints, we use constraints that restrict the properties of individual values. These include firstly constraints based on the type of an LLVM IR value, the value of LLVM IR constants and the opcodes of instructions. We added new constraints to this collection whenever the need arose and there is no overarching theory for them, they are simply an agglomeration of things that proved useful for describing computational idioms.

```
llvm-constraint ::= <element> is any instruction            |
                    <element> is an <opcode> instruction    |
                    <element> is a basic block label        |
                    <element> is a constant                 |
                    <element> is a global variable          |
                    <element> is a function argument        |
                    <element> is an integer value           |
                    <element> is a floating point value     |
                    <element> is a pointer value            |
                    <element> is integer constant <integer>
```

Finally we need a way to specify optional parts of computational idioms to make our specifications more flexible. We achieve this with an 'unused element' constraint, corresponding to the '*' in our original idiom definition.

```
unused-element-constraint ::= <element> is not used
```

We can now combine the previous derivation into a formal language specification.

5.4.3 Formal Language Specification

```
constraint         ::= (<constraint>)  |   <or-constraint>
                     | <and-constraint> | <atomic-constraint>
or-constraint      ::= <constraint> or <constraint>
and-constraint     ::= <constraint> and <constraint>
atomic-constraint  ::= two-element-graph-constraint   |
                       three-element-graph-constraint |
                       n-element-graph-constraint     |
                       llvm-constraints               |
                       <element> is not used

two-element-graph-constraint ::=
    <element> has edge to <element> in <graph>         |
    <element> dominates <element> in <graph>           |
    <element> strictly dominates <element> in <graph>  |
    <element> can reach <element> in <graph>

three-element-graph-constraint ::=
    <element> can not reach <element>
    without passing <element> in <graph>

n-element-graph-constraint ::=
    <element> can not reach <element>
    without passing any of <element-list> in <graph>

llvm-constraints ::= <element> is any instruction            |
                     <element> is an <opcode> instruction    |
                     <element> is a basic block label        |
                     <element> is a constant                 |
                     <element> is a global variable          |
                     <element> is a function argument        |
                     <element> is an integer value           |
                     <element> is a floating point value     |
                     <element> is a pointer value            |
                     <element> is integer constant <integer>

unused-element-constraint ::= <element> is not used

element-list ::= <element> | <element>,<element-list>
graph ::= DFG | reverse DFG | CFG | reverse CFG
        | CDG | reverse CDG | PDG | reverse PDG
```

5.5 Some Constraint Examples

In the previous section we drafted a description language for the specification of computational idioms. To prove the expressiveness of the language, we will give some examples.

Example 2 *The first example is a staple of any program analysis, the single entry single exit (SESE) region.*

$$\mathcal{I}_{SESE} = \{precursor, begin, end, successor\}$$
$$c(x) = true \iff$$
$$x_{begin} \text{ dominates } x_{end} \text{ in CFG and}$$
$$x_{end} \text{ dominates } x_{begin} \text{ in reverse CFG and}$$
$$x_{precursor} \text{ has edge to } x_{begin} \text{ in CFG and}$$
$$x_{precursor} \text{ dominates } x_{begin} \text{ in CFG and}$$
$$x_{end} \text{ has edge to } x_{successor} \text{ in CFG and}$$
$$x_{end} \text{ dominates } x_{successor} \text{ in CFG and}$$
$$x_{begin} \text{ can not reach } x_{precursor} \text{ without passing } x_{end} \text{ in CFG and}$$
$$x_{successor} \text{ can not reach } x_{end} \text{ without passing } x_{begin} \text{ in CFG}$$

Example 3 *The second simple example specifies a simple loop that has not been inverted. The above defined SESE region is used as a language construct for brevity. The atomic constraints "is the same as" as well as "is different from" should be self-explanatory.*

$$\mathcal{I}_{loop} = \{head, body\} \times \mathcal{I}_{SESE}$$
$$c(x) = true \iff$$
$$x_{body,successor} \text{ is the same as } x_{head,begin}$$
$$x_{head,end} \text{ is conditional branch instruction}$$
$$x_{body,precursor} \text{ is same as } x_{head,end} \text{ in CFG and}$$
$$x_{head,\text{-}} \text{ is SESE}$$
$$x_{body,begin} \text{ is different from } x_{head,successor} \text{ in CFG and}$$
$$x_{body,end} \text{ is branch instruction}$$
$$x_{body,\text{-}} \text{ is SESE}$$

5.6 Solving the Constraints

The idiom specification language that we use is designed such that it is easy to implement functionality that checks whether or not some chunk of intermediate representation corresponds to a specific computational idiom. All that is required is to check the atomic constraints, all of which can be evaluated using standard methods. What we actually need however is an algorithm that lists all occurrences of a computational idiom (if there are any). In other words, we want to search for subsets in intermediate representation code that confirm to a given idiom specification.

This is a very computationally expensive task when approached naively. We can formalise it as follows. Let $\mathcal{D}$ the set of instructions, constants, globals, basic block labels and function arguments of a given section of LLVM IR code und let $(\mathcal{I}, c)$ be the specification of a computational idiom. We are looking for an algorithm that enumerates the set $\mathcal{S}$ of all idiom occurrences.

$$\mathcal{S} = \{x \in \mathcal{D}^{\mathcal{I}} \mid c(x) = \text{true}\}. \tag{5.1}$$

For complex idioms in realistically sized functions we often have $\#\mathcal{I} > 100$ and $\#\mathcal{D} > 50$ and so the direct computation of this set by enumerating $\mathcal{D}^{\mathcal{I}}$ is clearly not viable as it contains more elements than the total number of atoms in the universe. Instead, we need a smarter approach that utilises the knowledge that we have about the composition of the binary predicate c to get a more efficient algorithm.

The main idea that we use is that idioms are made up in a modular fashion. Instead of testing every appropriately sized tuple of LLVM IR values for adherence to the idiom specification, we identify characteristic smaller parts of the idiom and look for those. We then extend the subset of the idiom that we are looking for until we capture it entirely. To formalise this approach we introduce some additional notation.

Definition 2 *For a given constraint $(\mathcal{I}, c)$ and a subset $\mathcal{J} \subset \mathcal{I}$ we define the restricted constraint $(\mathcal{J}, c_{\mathcal{J}})$ as follows. For any valid LLVM IR function $\mathcal{F}$ and $\mathcal{D}$ as defined above and any $x \in \mathcal{D}^{\mathcal{J}}$ the following equation holds.*

$$c_{\mathcal{J}}(x) = true \iff \exists y \in \mathcal{D}^{\mathcal{I}} \colon c(y) = true \text{ and } \forall j \in \mathcal{J} \colon y_j = x_j$$

The existence quantor makes such a construct very difficult to implement. In particular a restricted constraint of a constraint that can be formulated in our description language does not always inherit that property. We therefore introduce a second, weaker concept.

Definition 3 *For a given constraint $(\mathcal{I}, c)$ and a subset $\mathcal{J} \subset \mathcal{I}$ we call $(\mathcal{J}, c_\mathcal{J})$ an approximated restricted constraint if for any valid LLVM IR function $\mathcal{F}$ and $\mathcal{D}$ as defined above and any $x \in \mathcal{D}^\mathcal{J}$ the following holds.*

$$c_\mathcal{J}(x) = true \iff \exists y \in \mathcal{D}^\mathcal{I} \colon c(y) = true \text{ and } \forall j \in \mathcal{J} \colon y_j = x_j$$

This definition is of course very broad, as it allows the approximated restricted constraint to be true everywhere without any constraints being enforced. We will later see that sharpness of the chosen approximation will be crucial for the runtime behaviour of our system but irrelevant to its correctness. In the aforementioned case of a constantly true approximated restricted constraint, the algorithm will degenerate to the naive search through all tuples. To better understand the significance of these definitions, we give an example.

Example 4 *We define the computational idiom "addition involving a constant" as the pair $(\mathcal{I}, c)$ with the following properties.*

$$\mathcal{I} = \{\textit{constant summand}, \textit{other summand}, \textit{result}\}$$

$c(x) = true \iff$

x_{result} *is an add instruction and*

$x_{constant\ summand}$ *is a constant value and*

$(($ $x_{constant\ summand}$ *is the first operand of* x_{result} *and*

$x_{other\ summand}$ *is the second operand of* $x_{result})$ *or*

$($ $x_{other\ summand}$ *is the first operand of* x_{result} *and*

$x_{constant\ summand}$ *is the second operand of* $x_{result}))$

To find this idiom in a function naively, one has to check all these constraints for each triple of LLVM IR values. Intuitively it is a much better idea to instead look for all addition instructions and to then check whether one of their operands is a constant. This approach is expressed in the following approximated restricted constraint $(\mathcal{J}, c_\mathcal{J})$ where $\mathcal{J} = \{result\}$ and $c_\mathcal{J}$ is defined as follows.

$$c_\mathcal{J}(x) = true \iff x_{result} \text{ is an add instruction}$$

We can furthermore construct a restricted constraint $(\mathcal{J}', c_{\mathcal{J}'})$ *using the index set* $\mathcal{J}' = \{$*constant summand, result*$\}$ *and the following definition of* $c_{\mathcal{J}'}$*.*

$c_{\mathcal{J}'}(x) = true \iff$

x_{result} *is an add instruction and*

$x_{constant\ summand}$ *is a constant value and*

($x_{constant\ summand}$ *is the first operand of* x_{result} *or*

$x_{constant\ summand}$ *is the second operand of* x_{result})

With these definitions we can now construct an algorithm to find computational idioms in intermediate representation code in a very efficient way. It works fundamentally by constructing a chain of iteratively larger approximated restricted constraints and using them to sieve out more and more idiom candidates.

5.6.1 The Detection Algorithm

Let $(\mathcal{I}, c)$ be a computational idiom amd $\mathcal{F}$ a function with $\mathcal{D}$ defined as before. We choose an enumeration $i_1, \ldots, i_n$ of $\mathcal{I}$ and an enumeration $d_1, \ldots, d_K$ of $\mathcal{D}$ such that $\mathcal{I} = \{i_1, \ldots, i_N\}$ and $\mathcal{D} = \{d_1, \ldots, d_K\}$, where $N = \#\mathcal{I}$ and $K = \#\mathcal{D}$.

Furthermore, we choose a finite sequence of constraints $((\mathcal{I}_n, c_n))_{n=1,\ldots,N}$ such that $\mathcal{I}_n = \{i_1, \ldots, i_n\}$ and $(\mathcal{I}_n, c_n)$ is an approximated restricted constraint for $n < N$ and $c_N = c$. How such a sequence of approximations can be constructed will be discussed in a later section of this chapter. The correct choice of enumerations for $\mathcal{D}$ and $\mathcal{I}$ will also be discussed in that section.

We define a total order on $\mathcal{D}$ as induced by the enumeration $d_1, \ldots, d_K$. For convenience we introduce the symbols $-\infty$ and ∞ as values being smaller than or greater than any other value of $\mathcal{D}$ to avoid complex case distinctions.

We can now search for computational idioms using the following algorithm.

Algorithm 1 Constraint Solver Algorithm

1: $n \leftarrow 1$
2: $x = -\infty \in \mathcal{D}^{\mathcal{I}_1}$
3: **while** $n > 0$ and $n \leq N$ **do**
4: $\quad y \leftarrow \min\{x_n < x' \in \mathcal{D} \mid c_n(x_1, \ldots, x_{n-1}, x') = \text{true}\}$
5: $\quad$ **if** $y < \infty$ **then**
6: $\quad\quad n \leftarrow n+1$
7: $\quad\quad x \leftarrow (y_1, \ldots, y_n, -\infty) \in \mathcal{D}^{\mathcal{I}_{n+1}}$
8: $\quad$ **else**
9: $\quad\quad n \leftarrow n-1$
10: $\quad\quad x \leftarrow (x_1, \ldots, x_{n-1})$
11: **if** n=0 **then**
12: $\quad$ Function $\mathcal{F}$ does not contain idiom $(\mathcal{I}, c)$.
13: **else**
14: $\quad$ The tuple $x \in \mathcal{D}^{\mathcal{I}_n} = \mathcal{D}^{\mathcal{I}}$ matches to the idiom specification.

5.6.2 Interpretation as Graph Search

The algorithm is fundamentally a depth-first graph search algorithm in the graph (V, E) where the vertices and edges are defined as follows.

$$\begin{aligned} V =& \{*\} \cup \mathcal{D}^{\mathcal{I}_1} \cup \cdots \cup \mathcal{DI}_N \\ E =& \{(*, b) \in V \times V \mid b \in \mathcal{D}^{\mathcal{I}_1}, c_1(b) = \text{true}\} \cup \\ & \{(a, b) \in V \times V \mid \exists n \colon a \in \mathcal{D}^{\mathcal{I}_n}, b \in \mathcal{D}^{\mathcal{I}_{n+1}}, c_{n+1}(b) = \text{true}\} \end{aligned}$$

In this graph, all the elements of $\mathcal{D}^{\mathcal{I}_N}$ that are reachable from the origin $*$ fit the specification of the idiom. This property is independent of the choice of the approximated restricted constraints, which is clearly a requirement for correctness. The less precise the approximations are however, the more 'dead ends' there are in the graph. In the most extreme case of all the approximations being true constants, the algorithm degenerates to a lexicographic search through $\mathcal{D}^{\mathcal{I}}$ with very undesirable runtime properties as described before.

In the case of non approximated restricted constraints on the other hand, the time complexity of the algorithm is $O(NK)$ when assuming that the restricted constraints can be evaluated in constant time (this breaks down in practice).

5.6.3 Generating Restricted Constraints

We now have an algorithm to solve constraints but it requires not only the idiom specification itself but also total orders on $\mathcal{D}$ and $\mathcal{I}$ as well as a sequence of approximated restricted constraints. In the previous subsection we saw that the choice of approximated restricted constraints is very important to achieve good runtime behaviour. How can we however automatically generate an approximate constraint decomposition from a given idiom? The formal specification of computational idiom clearly can not help us a lot here but we can use knowledge about how constraints are constructed in our specification language. In particular, we can use the modularity of the construction process.

The order on $\mathcal{D}$ is mostly irrelevant, so we simply use the order of occurrence in the LLVM function for this. The order on $\mathcal{I}$ is extracted from the idiom specification by ordering the elements according to their first occurrence. This is by no means a substantiated approach but it turned out to work better than other methods. Among other things we also tried to use the Cuthill-McKee algorithm to order $\mathcal{I}$ by treating each atomic constraint as an edge between elements of $\mathcal{I}$ but the results were disappointing. Finally we can construct a specialisation of a constraint specification by simply discarding all atomic constraints which use elements of $\mathcal{I}$ not contained in the specified subset. The details of this approach will become clearer in a later section that describes our C++ implementation.

5.7 Prototype Implementation Architecture

We extended the LLVM infrastructure to test our methodology in a real world compiler infrastructure. Our modifications consist of two additional optimisation passes that can be enabled in a modified version of the LLVM “opt” program, a normalisation pass and a replacement pass. The replacement pass implements the functionality that we derived in the previous sections.

The normalisation pass applies several relatively simple code transformations to the LLVM IR that result in normalised structures to make the idiom detection easier. In particular, it normalises the way in which multidimensional arrays of dynamic size are accessed and it helps restructure polynomial computations to Horner’s scheme. We will not go into details of the implementation of the normalisation pass here.

The normalisation pass is intended to be used in conjunction with the default LLVM optimisation passes, which also result in normalised IR. In our experimental setup, we run all the optimisation passes from optimisation level '-O2' with the exception of loop unrolling and vectorisation passes. After this, we execute our custom normalisation pass, followed again by all '-O2' passes with the aforementioned exceptions. After these passes, the IR is normalised enough to run the replacement pass, followed by a final '-O3' to achieve maximum performance.

5.7.1 Implementation Overview

In our prototype implementation of this system, several class interface interact to provide the functionality described above. We will describe only the rough structure of the implementation due to space constraints. In total, our additions to the LLVM code base consist of 7017 lines of code.

The most important class of the system is the abstract class `Constraint` with its nested classes `Constraint::Specialized` and `Constraint::Solver`. This class provides only a single member function, `get_specialisations`, with return type `std::vector<std::pair<T*,Constraint::Specialized*>>`. This vector of pairs contains two structures at once. The first elements together form $\mathcal{I}$ and the second elements are corresponding to the previously defined approximate restricted constraints.

```
template<typename T>
class Constraint
{
public:
    virtual ~Constraint() { }

    enum class SkipResult;

    class Specialized;

    class Solver;

    virtual std::vector<std::pair<T*,Specialized*>>
                                        get_specializations() = 0;
};
```

The most important member function of the `Constraint::Specialized` class provides functionality to find the minimum as required by the detection algorithm that we introduced in section 5.6.1. This is encapsulated in the member function `skip_invalid`, which increments the provided value reference if the approximate restricted constraint is not already valid (thus halting at the minimal value that adheres to the constraints).

Depending on the outcome, the function returns `PASS` if the constraint was already fulfilled, `FAIL` if it can not be fulfilled by incrementing, `CHANGEPASS` if the constraint is fulfilled after incrementing and `CHANGE` if the constraint is not fulfilled after incrementing (but not obviously unfulfillable). This function behaviour might seem unnecessary complicated but the choice of possible return values allows for very performance efficient implementations of this interface.

The member function `begin`, `resume`, `fixate` and `cancel` are used to transition between different approximate restricted constraints. Whenever the variable n was incremented by the algorithm in section 5.6.1, this would be reflected by calls to `fixate` and `begin`. Similarly, calls to `cancel` and `resume` are necessary for decrements of n. This will become clearer later, when we show the implementation of this algorithm in C++.

```
enum class Constraint::SkipResult
{
    FAIL       = 0,
    PASS       = 1,
    CHANGE     = 2,
    CHANGEPASS = 3
};

class Constraint::Specialized
{
public:
    virtual ~Specialized() { }

    virtual SkipResult skip_invalid(T& c) = 0;

    virtual void begin () = 0;
    virtual void resume() = 0;
    virtual void fixate() = 0;
    virtual void cancel() = 0;
};
```

Finally the `Constraint::Solver` class implements the actual solving functionality, following the previously developed algorithm in spirit.

```
class Solver
{
public:
    Solver(Constraint& constr);

    ~Solver();

    bool next_solution(unsigned max_steps = UINT_MAX);

private:
    std::vector<std::pair<T*,Specialized*>> specializations;
    unsigned                                working_index;
};
```

The crucial part of the `next_solution` member function should explain the exact meaning of the `Constraint::Specialised` interface definitions.

```
while(n < specializations.size() && (max_steps--)) {
    SkipResult result = SkipResult::CHANGE;
    while(result == SkipResult::CHANGE) {
        result = specializations[n].second
                        ->skip_invalid(*specializations[n].first);
    }
    if(result != SkipResult::FAIL) {
        specializations[n].second->fixate();
        n ++;
        if(n < specializations.size()) {
            *specializations[n].first = T();
            specializations[n].second->begin();
        }
    }
    else {
        specializations[n].second->cancel();
        n --;
        if(n < specializations.size()) {
            specializations[n].second->resume();
            ++*specializations[n].first;
        }
    }
}
```

Most of the source code that we added to the LLVM infrastructure comprises implementations of classes that implement the the `Constraint` interface. Many of them required sophisticated methods for good performance, in particular the `ConstraintAnd` and `ConstraintOr` classes that implement the logical operators that are used to concatenate constraints. We will not go into any detail here but the following figure shows the inheritance tree of all classes in our prototype that implement the `Constraint` interface. All the constraints under `ConstraintAnd` are composed constraints, all others are atomic constraints.

```
Constraint
    ConstraintSingle
        ConstraintUnused
        ConstraintLLVMSingle
            ConstraintLLVMIntConstant
            ConstraintConstant
            ConstraintPreexecution
            ConstraintOpcode
            ConstraintIntegerType
            ConstraintPointerType
            ConstraintFloatType
    ConstraintOrdering
        ConstraintSame
        ConstraintDistinct
        ConstraintOrder
    ConstraintEdge
        ConstraintDFGEdge
        ConstraintDFGEdge0
        ConstraintDFGEdge1
        ConstraintDFGEdge2
        ConstraintDFGEdge3
        ConstraintCFGEdge
        ConstraintCDGEdge
    ConstraintReachable
        ConstraintDFGReachable
        ConstraintCFGReachable
        ConstraintCDGReachable
        ConstraintPDGReachable
```

```
ConstraintDominate
    ConstraintCFGBlocked
    ConstraintDFGBlocked
    ConstraintDFGDominate
    ConstraintDFGPostdom
    ConstraintCFGDominate
    ConstraintCFGPostdom
    ConstraintPDGDominate
    ConstraintPDGPostdom
    ConstraintDFGDominateStrict
    ConstraintDFGPostdomStrict
    ConstraintCFGDominateStrict
    ConstraintCFGPostdomStrict
    ConstraintPDGDominateStrict
    ConstraintPDGPostdomStrict
ConstraintOr
ConstraintAnd
    ConstraintProduct
    ConstraintDistributive
    ConstraintPrematureGEP
    ConstraintSESE
    ConstraintLoop
    ConstraintFor
    ConstraintNestedFor
    ConstraintExtendedInt
    ConstraintAffineAccess
    ConstraintAffineRead
    ConstraintScale
    ConstraintWeightedSum
    ConstraintSumReduction
    ConstraintPermute2
    ConstraintMatrixmatrix
    ConstraintSparseMV
    ConstraintDominatedExprInflow
    ConstraintDominatedExpr
    ConstraintHistogram
    ConstraintOffsetAffineAccess
    ConstraintStencilRead
    Constraint3DStencil
```

5.8 Automatic Pattern replacement

After we have spotted the computational patterns, our system needs to replace them by optimised implementations, as discussed in a previous section. We will not go into much detail here due to space limitations of this thesis.

For dense linear algebra, it is simply required to substitute the relevant LLVM IR instructions with library calls to BLAS routines. We do this by first generating an IR instruction that calls the appropriate BLAS `*gemm` function and insert it before the start of the loop that contains the naive implementation in the original source code. This step requires the analysis of the detected idiom to extract all the necessary parameters to the function call. We then erase only the one instruction in the original code that stores the new cell value into the resulting matrix, as all other instruction that constitute a matrix multiplication are free of side effects. The remaining code will be automatically removed by dead code elimination.

The situation for reduction operations was a bit more complex, as it required some sophisticated code transformations to achieve array privatisation and the creation and synchronisation of threads. Our replacement system currently only works for histogram reduction. All the reductions that we found in performance bottlenecks of the two benchmark collections were histogram reductions.

For each found histogram, our system generates four functions. One function contains a modified version of the original reduction IR code. The main modifications stem from the encapsulation into a function that can be passed to `pthread_create`, meaning that all parameters are contained in a single structure that is passed by reference as the only function argument. Furthermore the function can be restricted to perform the reduction on a subset of the original iteration space to represent the work of an individual thread.

As mentioned before, the operation is assumed to be a histogram reduction, meaning that the reduction variable is assumed to be an array or pointer. Since the size of the array can in general not be statically known at compile time in the second case, we insert boundary checks and generate a second function to dynamically resize the array. A third function is generated that spawns individual threads using the first function. This function is also responsible for merging together the resulting histograms from the individual worker threads.

Finally the original code is replaced by a call to this third function in a similar fashion as described before in the context of linear algebra.

5.9 Summary

We constructed a formal language for the specification of computational idioms and use it to formalise some of the previously studied idioms. After gaining some formal understanding of the problem at hand we devised algorithms to automatically detect computational idioms in single static assignment intermediate representation code. We implemented a prototype that implements these methods as an optimisation pass in the LLVM compiler infrastructure. This prototype automatically recognises computational idioms and applies idiom specific code transformations for parallelisation and optimisation.

Chapter 6

Evaluation

The idiom detection works well but some issues remain. The runtime of the idiom detection algorithm is generally negligible but for some complex programs the algorithm seems to almost diverge. The coexistence of many different formats for sparse matrices requires more complex idiom specifications and hinders the detection of some legitimate computational idioms as of now.

The replacement approaches for linear algebra using OpenBLAS work well. For reduction operations we achieve reliable detection and good speedup on a set of artificial test programs. On the original benchmark collections, the system works on the IS and tpacf benchmarks but fails on EP due to additional memory accesses.

6.1 Experimental Setup

For the evaluation of our detection algorithms we used artificial test programs as well as the original NAS and Parboil benchmark collections. We measured some preliminary performance results on the same system as in the previous evaluation, using a four processor machine with 16 core AMD Opteron processors. We did not yet perform a full evaluation across the entire set of benchmark programs but intend to due this once our system has been extended to cover stencil kernels as well. Instead we focused on the individual computational idioms and observed the behaviour of both the detection and the replacement routines.

The additional test programs consisted mostly of code snippets taken from the original benchmark programs with slight modifications that we used to investigate the limits of our detection algorithms.

We evaluated the prototype with respect to the following criteria:

- Does the prototype detect all the computational idioms that we classified manually in the computational bottlenecks?
- Does the prototype detect any additional computational idioms and can we manually veritfy them as correct?
- Do the automatic code transformations work reliably and produce valid programs?
- To what extend does the automatically generated parallel version differ from our manual implementations?
- How fast is the generated code?

6.2 Results

For dense linear algebra, the detection algorithm reliably recognised the matrix multiplication in the bottleneck of the **sgemm** benchmark. No other benchmark program in the two collections that we studied hat a bottleneck comprised of linear algebra. We ran some additional tests using slightly varying implementations of matrix multiplications, including the introduction and removal of temporary variables. Some examples are shown in figure 6.1. None of these superficial changes affected the reliability of the detection.

There were no additional false of correct examples of matrix multiplications in any of the other benchmark programs that the compiler pass identified. The automatic code transformations worked reliably and produced code that was functionally identical to the previously attained manual version. The performance was therefore identical. In conclusion, our methodology worked reliably and efficiently for dense linear algebra.

For the sparse linear algebra, the results were more inconsistent. This was due to the vastly differing implementations of sparse linear algebra in the different benchmark programs. In the CG benchmark of the NAS Parallel Benchmarks, the sparse matrix is stored in the Compressed Sparse Row format and the sparse matrix vector multiplication is implemented accordingly. The detection algorithm works well for this program but no code transformations are implemented yet

in our optimisation pass for this computational idiom. The other benchmark with sparse linear algebra in the computational bottleneck was spmv from the Parboil Benchmarks. This program however uses an entirely different storage format for the sparse matrix and is thus inaccessible for our system. This could be solved with additional constraint specifications, we might investigate this in further work.

Figure 6.1: Different matrix multiplication implementations

```
for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {
      float a = A[mm + i * lda];
      float b = B[nn + i * ldb];
      c += a * b;
    }
    C[mm+nn*ldc] = C[mm+nn*ldc] * beta + alpha * c;
  }
}
```

```
for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {
      c += A[mm + i * lda] * B[nn + i * ldb];
    }
    C[mm+nn*ldc] = C[mm+nn*ldc] * beta + alpha * c;
  }
}
```

```
for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {f
      float b = B[nn + i * ldb];
      c = c + A[mm + i * lda] * b;
    }
    C[mm+nn*ldc] = c;
  }
}
```

There were four reduction operations in the benchmark bottlenecks, all of which were histogram reduction. The simplest reduction was found in the the **Integer Sort** program of the NAS Parallel Benchmarks and the most complex reduction constituted the bottleneck of the **tpacf** benchmark from the Parboil collection. The detection algorithm detected the idiom in the **IS** and **histo** benchmarks reliably. The detection of the **EP** bottleneck worked reliably when the bottleneck was separated into its own function but we were unable to reproduce this in the context of the original source code for reasons unknown to us.

The bottleneck of **tpacf** was recognised by previous version of our compiler pass. When implementing our replacement routines we observed some regressions however and the benchmark is no longer recognised as a reduction computation. This is due to the use of an additional array structure for a binary seach computation that results in the final index that is used to access the histogram. This additional array is not accounted for by the computational idiom itself and confuses our system. The problem will be addressed in further research.

The code transformations for the parallelisation of reduction computations work reliably on all three detected benchmarks. The resulting code for **histo** needs some minor manual adjustment to generate valid results. This is because our system is unable to properly deduce the complementary operator as described in section 2.2.3. Instead it always assumes this complementary operator to be an element wise addition of the histogram bins, which is incorrect for the saturating histogram that is implemented in **histo**.

The resulting programs match the performance of our previously implemented versions. In particular, we achieve 14x speedup on the bottleneck of **EP** when compared against the sequential baseline, which is the input source code to our compiler.

6.3 Summary

The detection and replacement of dense linear algebra works reliably and produces code that is functionally identical to the parallel versions that we obtained in the preceding case study. The detection of sparse linear algebra generally works but is problematic due to the many different storage formats for sparse matrices. No code transformations for sparse linear algebra were implemented so far.

The detection of reduction operations is much more complex. The general methodology works but there are still some examples that do not fit our idiom specification. For the idioms that we detect successfully, the automatic code transformations are able to reproduce previous results.

Chapter 7

Conclusion

We were able to integrate idiom specific functionality for parallelisation and optimisation into an existing compiler infrastructure. Even though our prototype at this point only implements two computational idioms, we were able to show the general feasibility and usefulness of the system.

The main limitations of our work are the evaluation on only a small set of benchmark programs and the lack of actual hardware heterogeneity.

Further work is necessary to prove that the approach extends well to more complex programs and additional computational idioms. Furthermore the application to heterogeneous computation, which was beyond the scope of this project, will be a major part of subsequent research.

7.1 Summary

We used several established benchmark collections to get an understanding of typical computational workloads in the domain of scientific computing. Using profiling techniques, we established the performance critical sections of the source code of the individual benchmark programs. We collected these benchmark bottlenecks and compared them with each other to find similarities and to group them together according to computational idioms that they followed. Using a limited set of only three fundamental computational idioms (stencil kernels, reductions and linear algebra), we achieved good coverage of the performance bottlenecks in the studied benchmark collections.

We investigated the performance increases that can be achieved by idiom specific automatic code parallelisation. Using manual optimisation techniques

and domain specific knowledge, we found that many computational idioms can be parallelised and optimised in a mechanical way.

Using optimised library implementations for linear algebra, established optimisation techniques for stencil computations and generalisations of basic parallelisation approaches for reduction computations we modified the sequential benchmark version to get idiom based parallel versions. We compared the results that we achieved by idiom based code parallelisation with parallel versions of the benchmark programs that were written by experts. In most cases we were able to match or even surpass the optimised parallel versions, proving that computational idioms often contain all the contextual and semantic information that is required for efficient parallelisation.

We thought of ways to formalise this previously unstructured work and came up with a description language for computational idioms that turned out to be sufficiently expressive to formulate many important computational idioms. Using this language as the basis for our reasoning, we developed an efficient algorithm to automatically detect computational idioms in real source code. We implemented this algorithm in C++ and integrated it into a modified version of the LLVM compiler infrastructure.

We specified formally the computational idioms of dense matrix multiplication, sparse matrix vector multiplication and general histogram reduction operations. These patterns represent two of the three three computational idioms that we originally identified as important. We evaluated the accuracy of our automatic detection algorithm on them. The results were encouraging as we were able to recreate our manual classification quite accurately without any false positives.

Finally we tried to reproduce our manual optimisation results by implementing functionality in LLVM to imitate our parallelisation efforts. This was based on the previous work on the detection of idioms. The necessary effort for this differed widely between the individual idioms. For linear algebra, all the relevant optimisation work was encapsulated in BLAS implementations and thus the automatic parallelisation consisted only of inserting BLAS calls. The reduction idiom on the other hand required complex code generation. While this part of our research remains unfinished, automatic parallelisation of complex reduction computations appears to be possible on a level that far surpasses previous approaches in the literature.

7.2 Critical Discussion

Automatic parallelisation approaches have in the past often struggled to scale well beyond simple benchmark programs. We tried to avoid this problem by choosing well respected benchmarks from outside the compiler community and a flexible idiom detection method that is not easily confused by slight syntactic differences. Despite this, a much larger evaluation on more diverse and complex software will be necessary to prove the feasibility of our methods in real world code bases. This will in particular concern the generality of the formal specifications for idioms that we use. While our approach is independent from the original source code, some nonstandard implementations of computational idioms might still be obscured in intermediate representation code. Some of this might be mitigated by additional normalisation passes, but additional evaluation of the scale of these problems is necessary.

We chose LLVM IR as the base language for all our detection and replacement algorithms for computational idioms. We are very confident that this is a better choice than original source code but it is unclear how well our methods can scale on intermediate representation code. Fundamentally the choice for LLVM IR was mostly made due to the easy accessibility and tool support, the obvious shortcomings of source code analysis approaches and the lack of real alternatives. Several inconveniences of LLVM IR showed during the implementation of our prototype that required additional normalisation passes. In the future it might become useful to put restrictions or extensions onto the language specification for our purposes.

In this work we were able to express two computational patterns in our formal idiom specification language. It remains to be shown that the language is also expressive enough to allow for other computation idioms that arise in different computational domains. Related to this issue is the question whether any limited set of idioms can really cover the majority of computational bottlenecks. Maybe the abundance of discernible computational idioms is just an artefact of our focus on artificial benchmark programs and does not translate well to large scale software projects.

One major motivation for this research was the development of parallelisation techniques that can target heterogeneous hardware. So far we have not given any proof that idiom specific parallelisation is capable of this transition.

7.3 Future Work

In future work we intend to do a more thorough evaluation of our approach to automatic parallelisation. This will firstly involve the use of additional benchmark suites to investigate how well our methods generalise to other computational domains and alternative implementations. Furthermore, we intend to use more complex code bases that go beyond the scope of traditional benchmark programs to investigate the impact of our optimisations on more diverse and unstructured source code. We plan to use popular open source software from platforms such as github for this analysis. This work will require much more methodology to accurately measure the performance implications for interactive software and was beyond the scope of this project.

We have not implemented detection and parallelisation functionality for the stencil idiom yet. As this is one of the most common and important computational idioms in scientific computing and was one of the original three idioms that we assembled in our case study, it will be crucial to introduce it to our system. There is a profound body of work on optimising stencil computations and fierce competition in the field, which is why we decided to implement the other two idioms first. Aside from adding this new idiom to our system, many of the already specified idioms could often be implemented more elegantly and in a more generic way. We will use the experience that we gathered to revise these idioms.

Another direction of further research is the extension of our system to allow for arbitrarily sized index sets in our idiom specifications. Currently some constraint definitions require undesirable compile time limits, e.g. for stencil computations the maximum size of the neighbourhood must be fixed (although it can be set very large). This is generally not a huge problem but could be solved more elegantly. Extensions to the specification system and the corresponding idiom detection algorithms could alleviate this problem and allow for more general idioms to be implemented.

A more structured approach to idiom replacement will be a central part of follow up research. While currently we have developed a very structured and formally substantiated way of specifying and finding computational idioms, the automatic parallelisation is still quite ad-hoc. As we intend to move on to much more complicated, heterogeneous parallelism it will be necessary to develop a more sophisticated and scalable approach for this.

Another interesting research questing is to implement methods not for the increase of parallelism in programs but for the restructuring of parallelism. Such approaches would for example take already parallelised code and tune the parallelism for new parallel architectures, particularly of course for heterogeneous ones. This might also expose some problems that arise from our choice of LLVM IR as our implementation basis, because LLVM IR does not have a native concept of parallelism.

Bibliography

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition.

Andión, J. M. (2015). *Compilation Techniques for Automatic Extraction of Parallelism and Locality in Heterogeneous Architectures*. PhD thesis, University of A Coruña.

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. (2014). Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA. ACM.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY.

Baghdadi, R., Cohen, A., Grosser, T., Verdoolaege, S., Lokhmotov, A., Absar, J., Van Haastregt, S., Kravets, A., and Donaldson, A. (2015). PENCIL Language Specification. Research Report RR-8706, INRIA.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA. ACM.

Barik, R., Kaleem, R., Majeti, D., Lewis, B. T., Shpeisman, T., Hu, C., Ni, Y., and Adl-Tabatabai, A.-R. (2014). Efficient mapping of irregular c++ applications to integrated gpus. In *CGO '14*, pages 33:33–33:43, New York, NY, USA. ACM.

Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University.

Chandan Reddy, M. K. and Cohen, A. (2016). Reduction drawing: Language constructs and polyhedral compilation for reductions on gpus. In *Proceedings of the 25rd International Conference on Parallel Architectures and Compilation*, PACT '16.

Chandramohan, K. and O'Boyle, M. F. P. (2014). A compiler framework for automatically mapping data parallel programs to heterogeneous mpsocs. In *CASES '14*, pages 9:1–9:10, New York, NY, USA. ACM.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA. IEEE Computer Society.

Chung, E. S., Milder, P. A., Hoe, J. C., and Mai, K. (2010). Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA. IEEE Computer Society.

Cole, M. (1991). *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA.

Cole, M. (2004). Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406.

Colella, P. (2004). defining software requirements for scientific computing.

Cong, J. and Yuan, B. (2012). Energy-efficient scheduling on heterogeneous multicore architectures. In *Proceedings of the 2012 ACM/IEEE International Sym-*

posium on Low Power Electronics and Design, ISLPED '12, pages 345–350, New York, NY, USA. ACM.

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

Doerfert, J., Streit, K., Hack, S., and Benaissa, Z. (2015). Polly's polyhedral scheduling in the presence of reductions. *CoRR*, abs/1505.07716.

Emani, M. K. and O'Boyle, M. (2015). Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments. *SIGPLAN Not.*, 50(6):499–508.

Grosser, T., Größlinger, A., and Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4).

Kamil, S., Cheung, A., Itzhaky, S., and Solar-Lezama, A. (2016). Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 711–726, New York, NY, USA. ACM.

Kelly, W. and Pugh, W. (1995). A unifying framework for iteration reordering transformations. Technical report, College Park, MD, USA.

Larsen, S. and Amarasinghe, S. (2000). Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA. ACM.

Lee, J., Samadi, M., Park, Y., and Mahlke, S. (2013). Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *PACT '13*, pages 245–256, Piscataway, NJ, USA. IEEE Press.

Leyton, M. and Piquer, J. M. (2010). Skandium: Multi-core programming with algorithmic skeletons. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 289–296.

Loogen, R., Ortega-mallén, Y., and Peña marí, R. (2005). Parallel functional programming in eden. *J. Funct. Program.*, 15(3):431–475.

Margiolas, C. and O'Boyle, M. F. (2015). Palmos: A transparent, multi-tasking acceleration layer for parallel heterogeneous systems. In *ICS '15*, pages 307–318, New York, NY, USA. ACM.

Mendis, C., Bosboom, J., Wu, K., Kamil, S., Ragan-Kelley, J., Paris, S., Zhao, Q., and Amarasinghe, S. (2015). Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide dsl code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 391–402, New York, NY, USA. ACM.

Mullapudi, R. T., Vasista, V., and Bondhugula, U. (2015). Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443.

Niall Murphy, Timothy Jones, S. C. and Mullins, R. (2015). Limits of static dependence analysis for automatic parallelization.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA. ACM.

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition.

Sheng, W., Schürmans, S., Odendahl, M., Bertsch, M., Volevach, V., Leupers, R., and Ascheid, G. (2013). A compiler infrastructure for embedded heterogeneous mpsocs. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 1–10, New York, NY, USA. ACM.

Steuwer, M., Kegel, P., and Gorlatch, S. (2011). Skelcl - a portable skeleton library for high-level gpu programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182.

Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73.

Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D., and Hwu, W.-m. W. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127.

Sun, E., Schaa, D., Bagley, R., Rubin, N., and Kaeli, D. (2012). Enabling task-level scheduling on heterogeneous platforms. In *GPGPU-5*, pages 84–93, New York, NY, USA. ACM.

Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M. F. (2009). Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187, New York, NY, USA. ACM.

Venkata, S. K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., and Taylor, M. B. (2009). Sd-vbs: The san diego vision benchmark suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 55–64, Washington, DC, USA. IEEE Computer Society.

Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. (2013). Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23.

Wang, Z., Tournavitis, G., Franke, B., and O'boyle, M. F. P. (2014). Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Trans. Archit. Code Optim.*, 11(1):2:1–2:26.