Symbolic Generation of Parallel Solvers for Unconstrained Optimization

Symbolic Generation
of
Parallel Solvers
for
Unconstrained Optimization

By
Jessica L. M. Pavlin B.A.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University
© Copyright by Jessica L. M. Pavlin, May 29, 2012

MASTER OF SCIENCE(2012)	McMaster University
COMPUTER SCIENCE	Hamilton, Ontario

TITLE: Symbolic Generation of Parallel Solvers for Unconstrained Optimization

AUTHOR: Jessica L. M. Pavlin B.A. (McGill University)

SUPERVISOR: Dr. Christopher K. Anand & Dr. W. Spencer Smith

NUMBER OF PAGES: xv, 48

LEGAL DISCLAIMER: This is an academic research report. I, my supervisor, defence committee, and university, make no claim as to the fitness for any purpose, and accept no direct or indirect liability for the use of algorithms, findings, or recommendations in this thesis.

Abstract

In this thesis we consider the need to generate efficient solvers for inverse imaging problems in a way that supports both quality and performance in software, as well as flexibility in the underlying mathematical models. Many problem domains involve large data sizes and rates, and changes in mathematical modelling are limited only by researcher ingenuity and driven by the value of the application. We use a problem in Magnetic Resonance Imaging to illustrate this situation, motivate the need for better software tools and test the tools we develop. The problem is the determination of velocity profiles, think blood-flow patterns, using Phase Contrast Angiography. Despite the name, this method is completely noninvasive, not requiring the injection of contrast agents, but it is too time-consuming with present imaging and computing technology.

Our approach is to separate the specification, the mathematical model, from the implementation details required for performance, using a custom language. The Domain Specific Language (DSL) provided to scientists allows for a complete abstraction from the highly optimized generated code. The mathematical DSL is converted to an internal representation we refer to as the Coconut Expression Library. Our expression library uses the directed acyclic graphs as an underlying data structure, which lends itself nicely to our automatic simplifications, differentiation and subexpression elimination. We show how parallelization and other optimizations are encoded as rules which are applied automatically rather than schemes that need to be implemented by the programmer in the low-level implementation. Finally, we present results, both in terms of numerical results and computational performance.

Acknowledgments

First and foremost, I would like to thank Dr. Christopher Anand for the patience and kindness that he has shown me throughout my Masters, as well as the extensive knowledge that he shared with me. I could not have asked for a better supervisor. I would also like to thank Dr. Spencer Smith, for his guidance and support. Finally, I would like to thank my parents, Charles and Margaret, and my brother and sister-in-law, Michael and Joanne, for their generous encouragement, love and support.

Contents

Appendices i

Abstract iii

Acknowledgments v

List of Figures ix

List of Tables xi

List of Algorithms xii

List of Abbreviations xiv

1 Introduction 1
- 1.1 Motivation: Using Phase Contrast Angiography (PCA) . . . 1
  - 1.1.1 Physics and Mathematics of PCA . . . 1
  - 1.1.2 Barriers to PCA in Practice . . . 3
  - 1.1.3 Experimentation is Key . . . 4
- 1.2 Objective . . . 6

2 Domain Specific Language (DSL) 9
- 2.1 Background . . . 9
- 2.2 Implementation . . . 9
- 2.3 Graph Modifications . . . 10
- 2.4 Use Cases . . . 11

3 Common Subexpression Elimination (CSE) 13
- 3.1 Background . . . 13
- 3.2 Common Subexpression Elimination (CSE) for the Coconut Expression Library (CEL) 14

4 Differentiation 17
- 4.1 Background . . . 17
  - 4.1.1 Symbolic Differentiation (SD) . . . 17
  - 4.1.2 Automatic Differentiation (AD) . . . 18
- 4.2 Vector Differentiation in CEL . . . 18
- 4.3 Isolating Differential Variables . . . 19

5 **Simplification** 21
5.1 Background . . . 21
5.2 Simplification in CEL . . . 21
5.3 Well-definedness . . . 23
5.3.1 Confluence . . . 23

6 **Compilation, Runtime & Verification** 25
6.1 Background . . . 25
6.1.1 Single Instruction, Multiple Data streams (SIMD) . . . 25
6.1.2 Distributed Shared Memory . . . 26
6.1.3 Westmere-EP . . . 27
6.2 AVOps in CEL . . . 27
6.3 The Runtime System . . . 27
6.4 Verification . . . 29
6.5 Results . . . 30

7 **Graph Transformations for Parallelism** 33
7.1 Memory Locality and Image-Space Decomposition . . . 33
7.2 Operation Transformations . . . 34
7.2.1 Breaking apart Fourier transforms (FTs) that are followed by Projections: . 34
7.3 Column Block Ordering . . . 36

8 **Parallelizing L-BFGS** 37
8.1 Background . . . 37
8.1.1 Newton's Method . . . 37
8.1.2 Broyden-Fletcher-Goldfarb-Shanno (BFGS): . . . 37
8.1.3 Limited Memory BFGS (L-BFGS): . . . 38
8.2 Improvements . . . 38
8.2.1 Skipping Copies . . . 38
8.2.2 Fusing Dot Products . . . 40
8.3 Results . . . 40

9 **Conclusion** 43

A **Simplification Rewrite Rules** 45

B **Code Snippet: Fusing Dot Products for L-BFGS** 47

List of Figures

1.1 Conservation of Mass . . . 2
1.2 Numerical phantom without and with noise. . . . 4
1.3 Failed experiments caused by over-use of ℓ^2 and Conservation of Mass regularizers, and aliasing inherent to the use of non-convex objectives, like the velocity objective. 5
1.4 Results of preliminary experimentation with ℓ^2 and conservation of mass regularizers, as well as convexity factors. . . . 5

2.1 Data structure for expressions . . . 10
2.2 Data structures for operations . . . 10
2.3 Data structures for vector spaces . . . 11
2.4 Building operations in the CEL DSL . . . 12
2.5 Code to build a regularizer for the conservation of mass . . . 12

3.1 A simple example of how to remove repetitive calculations through common subexpression elimination . . . 13
3.2 Tree of: $\left\|\text{ft}(x+iy)\right\|_2$, where $(x+iy) : D \to \mathbb{C}$ is a discretization of a complex function. . . . 15
3.3 DAG of: $\left\|\text{ft}(x+iy)\right\|_2$, where $(x+iy) : D \to \mathbb{C}$ is a discretization of a complex function. . . . 15
3.4 Hashing DAG nodes to maximize common subexpression elimination. . . . 16

4.1 How Automatic and Symbolic Differentiation are related . . . 18
4.2 CEL's differentation of $\text{d}_x\left\|\text{ft}(x+iy)\right\|_2$ . . . 19
4.3 DAG of the differentation of $\text{d}_x\left\|\text{ft}(x+iy)\right\|_2$ . . . 19
4.4 Isolating differential variable dx in the expression $\Re(\text{ft}(\text{d}x+i0))) \cdot \Re(\text{ft}(x+iy))$. 20
4.5 Graph rebuilt to isolate differential variable . . . 20

5.1 Simplifying into a normal form to expose CSEs. . . . 22
5.2 Three simplification code examples written in Haskell: $scale(1, x_i) \to x_i$, $\Re(x_i + ix_j) \to x_i$, $scale(c_1, scale(c_2 * x_i)) \to scale(c_1 * c_2, x_i)$, and $\text{ft}^{-1}(\text{ft}(x_i)) \to x_i$ respectively. . . . 23
5.3 Visual demonstration of the diamond property for the rules $scale(s_0, scale(s_1, x_0)) \to scale(s_0 * s_1, x_0)$ and $sum(x_0) \to x_0$ . . . 24

6.1 Flynn's taxonomy . . . 25
6.2 Typical memory hierarchy with rough approximations for average access times and capacity [Tan08] . . . 26
6.3 AVOps distributed onto worker core ring buffers . . . 28

6.4 Verifying a safe AVOp stream 29
6.5 Verifying an unsafe AVOp stream 30
6.6 Scalability results 31
6.7 Normalized execution time results 32

7.1 First ordering of operations for core 0 of the expression 33
7.2 A second ordering of operations for core 0, optimized for memory locality 34
7.3 Data decomposed into strips, with the worker cores each running operations on their own strip. 34
7.4 DAG for 2D Projection $\circ$ ft 35
7.5 Transformation of DAG for 2D Projection $\circ$ ft for optimization. 35
7.6 A vector with 32 rows and 4 columns in column block order 36
7.7 Transformation of DAG for 2D Projection $\circ$ ft optimized for preference of column FTs over row FTs 36

8.1 First ten iterations of the real part of the SENSE problem. 40
8.2 First ten iterations of the real part of the SENSE problem using 54% of the data from three coils with approximately 50% noise added. 41

List of Tables

1 Functions in Coconut Expression Library. xv

1.1 Results from experimentation with regularizers pre-CEL 4

8.1 Times in seconds for the SENSE problem using 86% of the data from two coils. . . . 41

A.1 Simplification rewrite rules for scale 45
A.2 Simplification rewrite rules for sum 45
A.3 Simplification rules for product 45
A.4 Simplification rewrite rules for complex numbers 46
A.5 Simplification rules for Fourier transforms 46
A.6 Simplification rules for dot products 46
A.7 Simplification rewrite rules for sparse convolution zips 46
A.8 Simplification rewrite rules for inputs 46

List of Algorithms

1 The BFGS Algorithm 38
2 The L-BFGS Algorithm 39

List of Abbreviations

The following is the notation that will be used in this thesis.

Math	CEL DSL	CEL Internal	Description
$:=$			Assignment
$\sum x_0 ... x_{n-1}$	`+`	`Sum(x0,...,x{n-1})`	The sum of any number of inputs of the same dimensions
$\prod x_0 ... x_{n-1}$		`Prod(x0,...,x{n-1})`	The product of any number of inputs of the same dimensions
$scale(s, x)$	`scale`	`s :*: x`	The result of scaling x by the scalar s
$x_0 \cdot x_1$	`dot`	`x0 . x1`	The dot product of two vectors of equivalent length
$x_0 + ix_1$	`+:`	`ReIm(x0,x1)`	The complex number produced of the real vector inputs (with same dimensions) $x_0 + ix_1$
$\Re(x)$	`xRe`	`Re(x)`	The real part of the complex vector x
$\Im(x)$	`xIm`	`Im(x)`	The imaginary part of the complex vector x
$\text{ft}(x)$	`ft`	`FT(x)`	The fast fourier transform of the complex vector x
$\text{ft}^{-1}(x)$	`iFt`	`iFT(x)`	The inverse fast fourier transform of the complex vector x
$proj(ss, x)$	`proj`	`proj`	The projection of the elements to keep from the vector x
$inject(ss, x)$	`inject`		The injection of the zeros into the missing elements of x (those not in kept)
d	`diff`		Differentiation

Table 1: Functions in Coconut Expression Library.

Chapter 1

Introduction

As physicists continuously come up with new and ingenuous models for *Magnetic Resonance Imaging (MRI)*, reducing the gap between theory and practice is too frequently stymied by computational challenges. In this thesis we present a symbolic code generation toolbox that utilizes classic and novel approaches to maximize efficiency in an effort to ease these challenges, while presenting an abstracted layer for rapid development for physicists and mathematicians alike.

1.1 Motivation: Using Phase Contrast Angiography (PCA)

Researchers in MRI developed *Phase Contrast Angiography (PCA)* to quantify velocity, of blood and other tissues, in the body, but its use in modern healthcare has been limited by its long acquisition time. Utilizing this technique could provide physicians with a technique to safely diagnose and track congenital heart disorders, brain aneurisms, and circulatory diseases. For this reason, researchers are searching for more efficient PCA experiments, which will largely depend on more sophisticated models and better processing to solve them.

While an in-depth understanding of the physics of MRI and the inner-workings of PCA are outside the scope of this thesis, a high level overview of the origin of the signal-generation process, and the resulting inverse problem will illustrate both the common structure of inverse problems (especially inverse imaging problems) and the frequent variations that make the automatic generation of efficient solvers particularly helpful in this field.

In this section we will briefly review the physics behind this process and the challenges of using PCA in practice, and look at how a domain specific language closely matching the informal language used in modelling can help us meet these and many similar challenges.

1.1.1 Physics and Mathematics of PCA

The process of acquiring the velocity encoded signal begins with the subject in a large magnet which causes the net magnetization of its protons to align. The application of a radio frequency pulse tips the protons over, at which point they begin to precess at a uniform rate. A first linear field variation, known as a gradient, is applied causing the tipped protons to precess relative to the uniform rate at a rate proportional to the field strength. Finally, a second (and exact opposite) field variation is applied causing the protons to relatively precess in the opposite direction.

For protons at rest, such as in static tissue, the relative precession caused by the positive and negative gradients cancel out. For protons that moved during the experiment (for example, the

protons that make up the blood in an artery) the positions during the opposite gradients are different so the precessions do not cancel each other out exactly. How far the proton moved during the experiment is therefore revealed in the signal as a change in phase of rotation, which is captured in the digitized signal as a difference in complex phase, proportional to the (dot) product of the applied gradient and the tissue velocity. The signal is expressed by Equation (1.1), with the unknowns ρ_0 and V, proton density and velocity respectively, and the known G, the gradient.

$$\text{signal} = \rho \cdot e^{i \cdot \mathcal{G} \cdot V} \tag{1.1}$$

In some cases, the signal equation can be solved directly, but more commonly, we need to add a statistical model for the noise incorporated in the measurements and find the maximum likelihood estimate. For independent, identical, normally distributed error, this leads to a least-squares minimization problem. The resulting objective function, including the measurements, m, is

$$\sum_{G \in \mathcal{G}} \left\| m - \text{ft}(\rho \cdot e^{i \cdot \mathcal{G} \cdot V}) \right\|^2, \tag{1.2}$$

where the norm is the ℓ^2 vector norm summing the individual differences over the array of measurements, and $\mathcal{G}$ is a set of gradient sensitizations, which are parameterized by vectors in $\mathbb{R}^3$. A minimum of dimensions + 1 sensitizations are required, so for three-dimensional velocity profiles, four sensitizations are required, but more could be used for robustness or signal-to-noise reasons.

Regularization

Due to the high level of noise, ill-conditioning of the signal equation, incomplete data, or a combination of these issues, it is beneficial to introduce additional *a priori* information to the equation. This is added in the form of a penalty to the objective function coming from the maximum likelihood estimate. In our example, we expect the result to have smooth transitions between pixels, and we can introduce this information into the optimization problem by adding the squared distance, called the ℓ^2 *norm* of the differences between neighbouring pixels, to penalize any large jump in values between neighbouring pixels. This is a commonly used technique for making inverse problems easier to solve, especially in the presence of noise, and is known as *regularization*, introduced by Tikhonov [TA77].

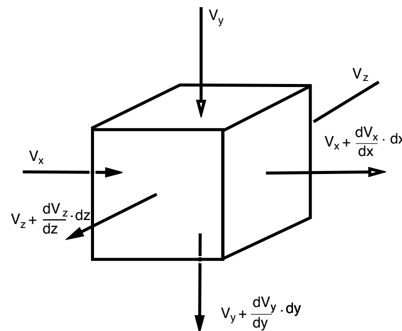


Figure 1.1: Conservation of Mass

We incorporate this information by adding a term to the objective function,

$$\min \left\{ \sum_{G \in \mathcal{G}} \left\| m - \mathrm{ft}(\rho \cdot e^{i \cdot G \cdot V}) \right\|^2 + \lambda \Delta(V)^2 \right\},$$

with λ acting as a tuning parameter to regulate how heavily the non-smoothness will be penalized, and Δ encoding the a priori information in the form of a regularizer.

Among the most common regularizers, ℓ^2-norms and ℓ^2-difference norms are good for getting rid of background noise and noise on top of smooth image segments. They lead to an easy-to-solve quadratic objective functions, but can result in over smoothing, or blurring. On the other hand, an ℓ^1-norm will penalize the total variation between tissues, rather than large transitions, and hence preserve sharp edges. In Equation 1.3 we see the ℓ^2-difference defined for a 3D discretization. The variable δ will be replaced by each of the velocity directions (v_x, v_y and v_z) as well the real and imaginary parts of ρ (ρ_r and ρ_i).

$$\begin{aligned} &\textstyle\sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} \sum_{k=0}^{n_z-1} \left\| \delta_{(i,j,k)} - \delta_{(i+1,j,k)} \right\|^2 + \\ &\textstyle\sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} \sum_{k=0}^{n_z-1} \left\| \delta_{(i,j,k)} - \delta_{(i,j+1,k)} \right\|^2 + \\ &\textstyle\sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} \sum_{k=0}^{n_z-1} \left\| \delta_{(i,j,k)} - \delta_{(i,j,k+1)} \right\|^2 \end{aligned} \tag{1.3}$$

In addition to these common regularizers, there are problem-specific regularizers. For example, in our problem we have experimented with regularizing using the physical property of conservation of mass. This property is illustrated in terms of velocity profiles and pixels in Figure 1.1. It simply says that if blood is flowing into a pixel then it has to be flowing out of the pixel. This property has been used as a regularizer in meteorological optimizaiton problems [HMH08, CMP02], where it has reduced solving time. To our knowledge, however, this is the first time it has been used in medical imaging. The equation for computing the conservation of mass regularizer for the velocity vectors is produced in Equation 1.4.

$$\textstyle\sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} \sum_{k=0}^{n_z-1} \left\| (v_{x(i+1,j,k)} - v_{x(i,j,k)}) + (v_{y(i,j+1,k)} - v_{y(i,j,k)}) + (v_{z(i,j,k+1)} - v_{z(i,j,k)}) \right\|^2 \tag{1.4}$$

1.1.2 Barriers to PCA in Practice

In practice, utilizing PCA as a means for diagnostics in hospitals using current methods is limited by long processing and acquisition times. Having the patient immobilized within the MRI machine for long periods is challenging for both the patient and for the budgetary concerns of the healthcare system.

Collecting the full data for this experiment requires the patient to be in the machine for upwards of 40 minutes, and, even then, recent work yields computational times well above acceptable levels. For example, an image with a resolution of 128×128 required 13 minutes for computation [IMB+10], and 3-dimensional images with resolutions of $128 \times 128 \times 128$ required between 180 minutes and 780 minutes for computation [Mar10]. Some experiments collecting less data and using better mathematical optimization techniques have demonstrated some promising results for low resolution images. For example, the problem has been solved using 28% of the data at a resolution of 128×128 (meaning the patient spends about 12 minutes in the MRI machine) in five minutes [HMB+10].

While these results are promising, much more work needs to be done before PCA can realistically be used as a diagnostic tool in our cash-strapped medical system.

1.1.3 Experimentation is Key

The most important things to gather from the previous sections is the following: with a better computational solution, PCA could become a practical, powerful, and non-invasive diagnostic tool. However, finding that solution will require a lot more experimentation. Currently, that leaves physicists hacking together code that is not reusable, and difficult to test and debug.

Before the creation of CEL, our group experimented with problems using numerical phantoms of resolution $28 \times 28 \times 28$. The model with and without noise is shown in Figure 1.2.

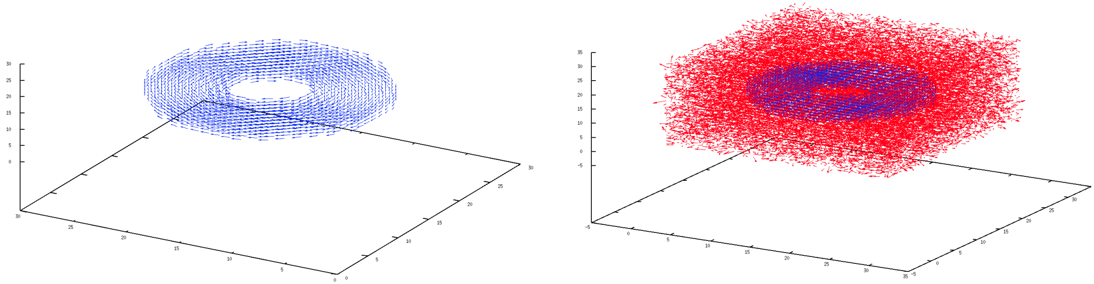


Figure 1.2: Numerical phantom without and with noise.

With experimentation we found failures caused by of over-smoothing or blurring with ℓ^2, the conservation of mass regularizer attempting to align all of the velocity vectors, and aliasing caused by gradients that were too small for the velocities in the in the phantom, resulting in convergence to the wrong minimum of our non-convex objective. Examples of these failures can be seen in Figure 1.3. However, with experimentation we were able to faithfully reproduce the phantom with greatly minimized noise, as can be seen in Figure 1.4.

These regularizers also effected numerical error and the time it took to run the experiments. Without these regularizers, we were able to solve for velocity in approximately 7 minutes, as seen in Table 1.1.3. However, experimentation with the conservation of mass regularizer demonstrated that we were able to cut that time almost in half. Meanwhile ℓ^2 elevated the time needed to solve the problem, but reduced the error by 90%. While these results were interesting, the project was stalled at that point due to the unusably long run times that came with the addition of just a few pixels once the data no longer fit in cache.

Regularizer	**Time (s)**	**ℓ^2 Error**
None	419	2.20
ℓ^2	702	0.26
CoM	248	2.10

Table 1.1: Results from experimentation with regularizers pre-CEL

MRI physicists would benefit greatly from having a code generation toolbox that allows them to enter equations and *a-priori*-information-driven regularizers, and produces efficient, parallel code suitable for real-time experiments, without requiring them to concern themselves with the

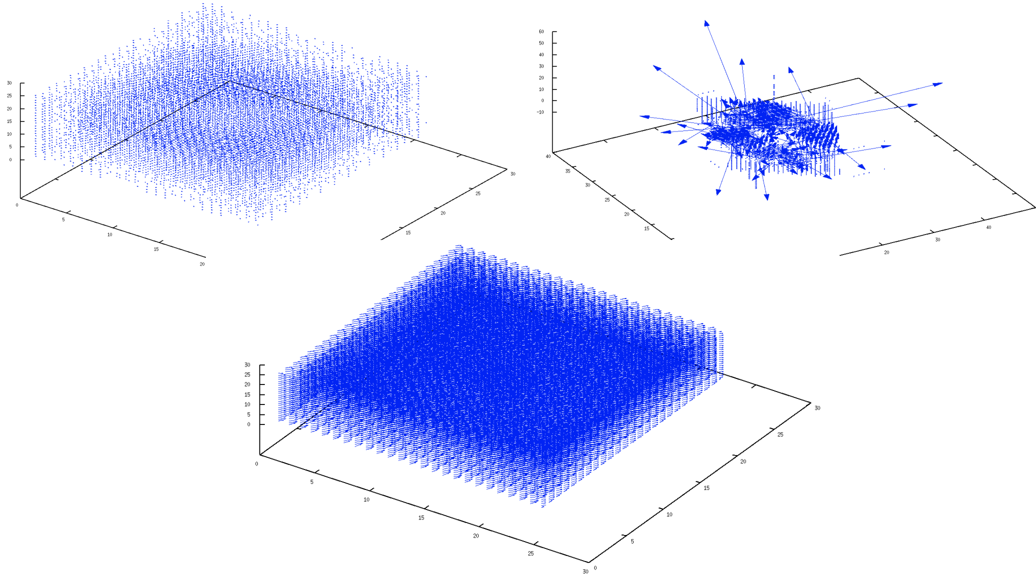


Figure 1.3: Failed experiments caused by over-use of ℓ^2 and Conservation of Mass regularizers, and aliasing inherent to the use of non-convex objectives, like the velocity objective.

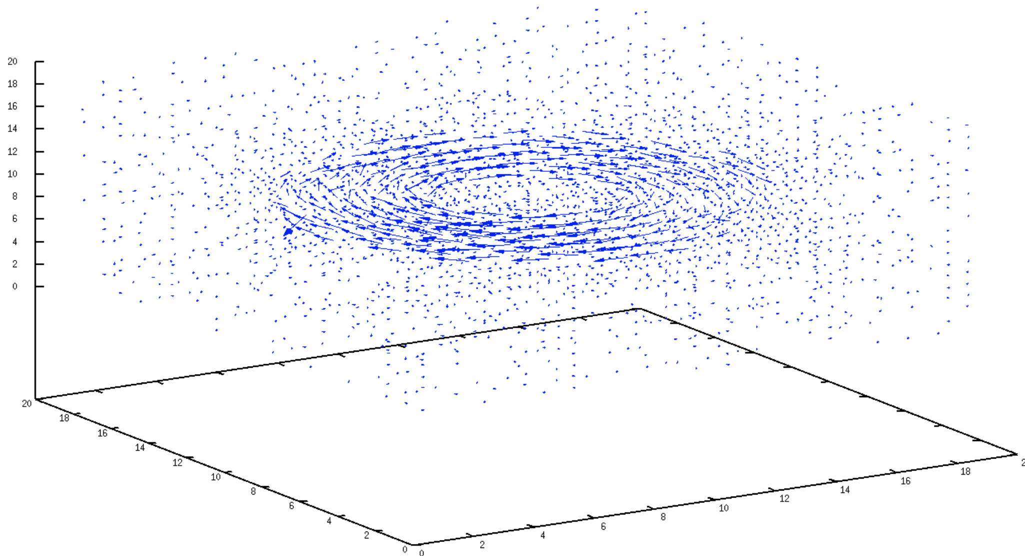


Figure 1.4: Results of preliminary experimentation with ℓ^2 and conservation of mass regularizers, as well as convexity factors.

computer science behind writing such code, let alone the software engineering factors necessary for the high reliability we expect from medical imaging software. In short, these experimenters would benefit greatly from a domain specific language suited to their field that generates code to do this work for them.

1.2 Objective

In the previous section, we have seen a part of the numerical experimentation possible with a simple symbolic code generation library implemented using algebraic data types in Haskell. While it was only capable of representing scalar expressions, through the use of a convention for encoding indices in scalar variable names, it was possible to generate the regularizers above, although the implementation was non-transparent, and fragile. It was, however, impossible to incorporate more general linear transformations, like the *Fourier transform (FT)*, which made it impossible to consider inverse problems involving partial-data collection. By definition, partial-data experiments are those in which the full set of data required by a *Fast Fourier Transform (FFT)* are not collected. In more pedestrian full-data experiments, we can assume that the Fourier transform has already occurred and model measurements in image space, which enabled the work above, but as we have suggested, the main barrier to PCA in practice is data-collection time, and reducing collection time requires the use of parallel data. Furthermore, most potential clinical applications will involve both novel experiment design and modelling and regularization. In these cases, functional evaluation must be performed in-vivo, and even willing volunteers can only lie still for so long while experimenters wait for data processing and iterate on experiment designs. So, as a practical requirement, we require short turn-arounds for optimized multi-core software, and high levels of software quality, even for experimental software.

Goals of Coconut Expression Library (CEL)

The Goal of the Coconut Expression Library is to develop a tool capable of handling the full range of mathematical models and regularizers, capture modelling errors as early and often as possible, include symbolic differentiation, and computation minimization (through common subexpression elimination (CSE) and algebraic simplifications), support correctness through automation and static checking wherever possible, and generate efficient parallel implementations to support rapid prototyping and live experimentation.

We feel these needs can be best met by

1. a domain specific language close to current use by applied mathematicians and scientists in in-formal model development; this must include
2. first-class vector variables,
3. derivation with respect to vector variables,
4. built-in linear operators with existing optimized implementations like the FFT,
5. user-defined zip-shift-map operations including the stencils used in the PDE community; and be interfaced with
6. an interface for computer scientists to specify simplification and parallelization operations algorithmically;
7. an efficient parallel run-time system to execute generated code;
8. an efficiently verifiable model of synchronization;
9. an efficient parallel unconstrained optimizer into which to plug generated function and gradient evaluators.

In this thesis we will explain how our DSL allowed us to reach goals 2 and 4 (see Chapter 2) and our differentiation techniques for reaching goal 3 (see Chapter 4). Additionally, we will show how rule based simplification and transformations for parallelism helped us to reach goal 6 (see Chapters 5 and 7). Finally, our parallel runtime and verification systems will lead us to reaching goals 7, 8 and 9 and are explained in Chapter 6.

Chapter 2

Domain Specific Language (DSL)

In this chapter we will briefly discuss DSLs in general and, specifically, the Coconut Expression Library and the advantages that it provides for users. The main novelty is the incorporation of *2. first-class vector variables*, as opposed to vectors of scalar expressions, which is required to support *4. built-in linear operators with existing optimized implementations like the FFT*. Doing so makes these higher-level abstractions available to later graph transformations performing simplification and parallelization.

2.1 Background

Domain Specific Languages (DSLs) are programming languages that provide a clear and concise set of features to express groups of tasks in a particular domain. They are used to abstract away the complex details of an application, providing the user with a simple interface with which to program.

Although requiring both domain expertise and knowledge of the language itself, a DSL provides an abstraction from the implementation details and facilitates faster development once these two are achieved.

Coconut (COde COnstructing User Tool) is a compiler research project at McMaster University [KAC06, ACK$^+$04, AK07b, AK07a], targeting medical imaging applications. Motivated by the desire to generate verifiable and highly optimized architecture-specific machine code based on mathematical specifications [KAC06, ACK$^+$04], Coconut provides low-level optimizations to account for higher-level model characteristics which are generally considered to be too risky from a software engineering stand point. The user is provided with a highly abstracted DSL written in Haskell.

2.2 Implementation

We provide a simple grammar to build expressions, which provides access to their data types, however, for safety, we want the users to use instances of classes which are designed around mathematical objects, such as real vector spaces and 1D, 2D, 3D and 4D discretizations (which are vector spaces with more structure, allowing for some compile-time type checking). In Figure 2.1 we demonstrate the underlying data type for expressions, and in Figure 2.2 the class for real vector spaces.

```
data ExpressionEdge = Op   Dims                           -- dims of output
                           OpId                           -- the operation
                           [Node]                         -- dims and nodes of inputs
                    | Var Dims ByteString                 -- variable X
                    | DVar Dims ByteString                -- differential variable dX
                    | RelElem  {reArray::Int              -- array number
                               ,reBoundary::Boundary      -- how to treat boundary
                                   elems
                               ,reIdx::[Int]              -- offsets in array indices
                               }                          -- length reIdx == length
                                   outDims
                    | Const {unConstDims::Dims, unConst ::Double}
    deriving (Eq, Show, Ord)
```

Figure 2.1: Data structure for expressions

```
data OpId  = FT Bool       -- FT:  True = forward , False = Inverse
           | PFT Bool Dir -- Partial FT with Dir of row, column or slice
           | Transpose Swap
           | Sum
           | SubMask
           | NegMask
           | Neg
           | Abs
           | Signum
           | Prod
           | Div
           | Sqrt
           | Sin
           | Cos
           ...
```

Figure 2.2: Data structures for operations

2.3 Graph Modifications

The coding for optimization of the internal data structures for increased parallelization and new simplifications is itself a DSL. Throughout the code examples in the following chapters (especially in Chapter 5), a consistent language will be made apparent that, when learned, will allow for developers to easily experiment with new optimizations and initiatives as they see fit.

```
class RealVectorSpace v s | v -> s where
  scale :: s -> v -> v
  dot :: v -> v -> s
  subMask :: v -> v -> v
  negMask :: v -> v -> v
  mapR :: (s -> s) -> v -> v
  norm2 :: v -> s
  norm :: (Floating s) => v -> s
  norm2 v = dot v v
  norm = sqrt . norm2
  projSS :: Subspace -> v -> v      -- project onto a subspace
  injectSS :: Subspace -> v -> v    -- inject (adjoint to project)
```

Figure 2.3: Data structures for vector spaces

2.4 Use Cases

The core expression language in the Coconut Expression Language (CEL) is meant to provide mathematicians and physicists with a programming language that allows them to easily express their models as new expressions.

All experimentation with Coconut DSLs sit on top of Haskell, and therefore require the user to have at least a base understanding of the language. That said, the advantages of the high-level abstraction from developing for multicore architectures outweigh the challenges.

Separation of model and optimizations

The DSL gives the user the ability to write their mathematical model without being concerned with optimizations. This abstraction allows the user to focus on the equations, rather than concerning themselves with code efficiencies.

Algebraic descriptions of expressions

With implementation details abstracted away, the user is able to build algebraic expressions that look like algebraic expressions. For example, in Figure 2.4 a user utilizes the CEL DSL to build a 3D Fourier transform, ft$(x + iy)$. The variable constructor `var3d` needs to know the dimensions ($16 \times 16 \times 16$ in this case) and the names (“x” and “y” respectively) of each variable. The symbols `+:` and `ft` are functions that build operations to build a complex number and the Fourier transform respectively.

Mix-and-match (objectives and regularizers)

As stated in the introduction, experimentation requires the ability to see how different regularizers effect the runtime and the quality of the output. The DSL for Coconut allows the user to write small expressions that represent these different models and then add them or break them apart as needed.

```
> let x = var3d (16,16,16) "x"
> let y = var3d (16,16,16) "y"
> ft (x +: y)
(FT((x(16,16,16)+:y(16,16,16))))
```

Figure 2.4: Building operations in the CEL DSL

```
massConservation :: (ThreeD,ThreeD,ThreeD) -> Scalar
massConservation (ThreeD vx,ThreeD vy,ThreeD vz)
    = Scalar $ Expression dotted exprs2
  where
    (Expression _ exprs0,nodesXYZ@[nX,_,_]) = mergeL [vx,vy,vz]

    -- velocity vectors inside convolution function
    vX = relElem 0 ZeroMargin
    vY = relElem 1 ZeroMargin
    vZ = relElem 2 ZeroMargin

    -- get dimensions
    dims = getDimE exprs0 nX

    -- create convolution function
    Scalar f =  (vX [1,0,0] - vX [-1,0,0])
              + (vY [0,1,0] - vY [0,-1,0])
              + (vZ [0,0,1] - vZ [0,0,-1])

    -- add the convolution function to the graph
    (exprs1,convN) = addEdge exprs0 $ Op dims (SparseConvZip dims f) nodesXYZ

    -- add the convolution function dotted with itself to the graph
    (exprs2,dotted) = addEdge exprs1 $ Op Dim0 Dot [convN,convN]
```

Figure 2.5: Code to build a regularizer for the conservation of mass

In Figure 2.5 we see a quick function to build the conservation of mass regularizer. This regularizer can be added to the fit-to-data term and sequenced with any number of other regularizers.

Rapid development

All in all, this provides the user with very fast development and very re-usable code.

Chapter 3

Common Subexpression Elimination (CSE)

The basic idea behind *Common Subexpression Elimination (CSE)* is to seek out instances of identical expressions (places that evaluate the same value multiple times) and replacing them with the already computed value.

The following code has a repetitive calculation:

```
d = a * b + c
f = a * b + e
```

Can be changed to:

```
temp = a * b
d = temp + c
f = temp + e
```

Figure 3.1: A simple example of how to remove repetitive calculations through common subexpression elimination

In this section we will briefly review the origins of this technique and then discuss how it is implemented to maximize efficiency in CEL. Although it does not directly contribute to any of the enumerated goals, CSE is required to produce efficient code with acceptable compile times.

3.1 Background

In 1970, IBM's John Cocke talk entitled *Global Common Subexpression Elimination* demonstrated how to remove repetitive calculations in code, using a directed graph with flow analysis. [Coc70] Cocke was looking at problems like the one expressed in Figure 3.1, where the compiler seeks out the culprits of repetitive calculations and, when certain that it will produce the same result (with

program flow understanding), makes the replacement using a temporary variable.

When dealing with large algebraic expressions, the problem is actually simpler. The classic understanding of an expression tree is remodeled into a *Directed Acyclic Graph (DAG)*. The head node of the expression is an ancestor to all other nodes in the graph. Variables and constants are either the entire graph (for example for the expressions x or 1) or children to the operations in the expression. They cannot be parents (and therefore cannot have any edges coming out of them). A subexpression of the expression consists of a node, all of its descendent nodes and the edges that connect them. Figures 3.2 and 3.3 we see the tree and DAG, respectively, of the expression $\left\|\mathrm{ft}(x+iy)\right\|_2$. Even in this simple example we can see a dramatic reduction in the expression size through the use of CSE.

Using DAGs as the underlying structure for physics expressions is popular methodology. Maple has been using them for well over a decade [Hec00], as have some institutions doing projects similar to CEL, including the University of Maryland's Quark-PLASMA project [YKD11] and the Barcelona Supercomputing Center's SMPS [SPBL06].

3.2 Common Subexpression Elimination (CSE) for the Coconut Expression Library (CEL)

In CEL, expressions are stored as DAGs with a hash function that stores each subexpression as an integer value. When an expression is created by the user, internal functions recursively build the DAG and populate the hash table. The recursion stops when a variable or constant is found, and a hash function adds them to the table. The hashed values of those leaf nodes are utilized in the hash for each of the nodes in the DAG building up from them (so $x + y$ does not hash to the same node as $x + z$ or even $y + x$ — more on that in Chapter 5). In Figure 3.4 we see the application of hash functions to the subexpressions that make up the expression $\left\|(\mathrm{ft}(x+iy))\right\|_2$.

If the hash function returns a value that is already in the table, it verifies that it is the hash for the same subexpression. If it is the same then no new entry is made. Otherwise, a collision has been found and, using open addressing methodologies, the expression is rehashed to a new node number and retested.

Once the node number has been secured, the new node as well as directed edges to its children (the inputs to that node) are added to the graph. As a result, every subexpression in the DAG representation is unique, thereby achieving CSE as desired.

If two expressions are built separately, there is no guarantee that all of their common subexpressions will have hashed to the same result (this is a consequence of using open addressing). As a result, if separately built expressions are being joined, all subexpressions of the second expression must be rehashed and retested to ensure that all common subexpressions are caught.

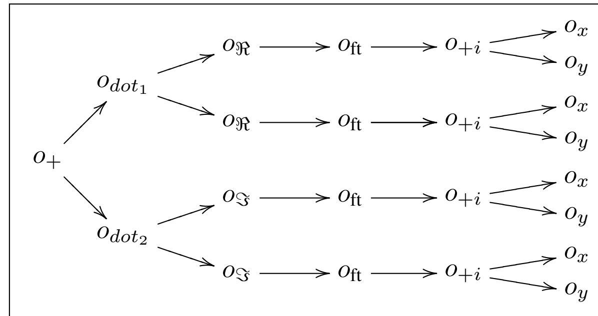


Figure 3.2: Tree of: $\left\|\mathrm{ft}(x+iy)\right\|_2$, where $(x+iy) : D \to \mathbb{C}$ is a discretization of a complex function.

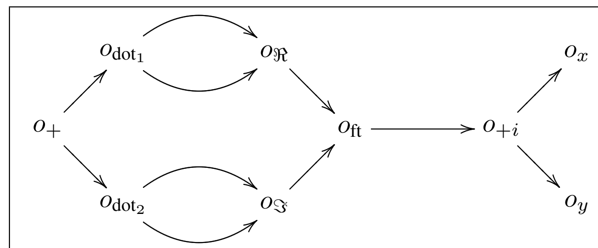


Figure 3.3: DAG of: $\left\|\mathrm{ft}(x+iy)\right\|_2$, where $(x+iy) : D \to \mathbb{C}$ is a discretization of a complex function.

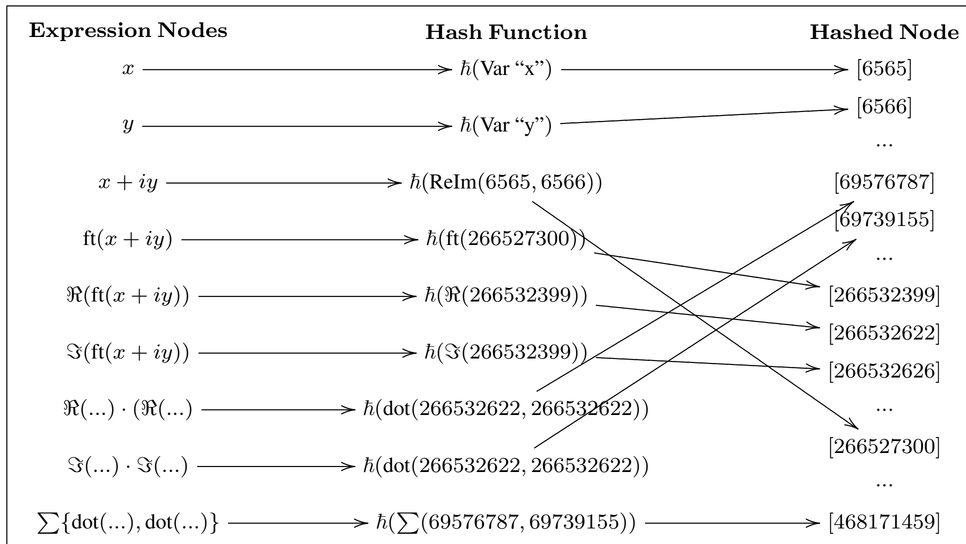


Figure 3.4: Hashing DAG nodes to maximize common subexpression elimination.

Chapter 4

Differentiation

With rapid development as one of our objectives, it is important that we provide the user with the ability to easily differentiate expressions. Derivatives are used in several different ways in scientific computing, including optimization, which we will discuss in Chapter 8.

In this chapter we will discuss various types of differentiation, our decisions for our choices, and our implementation techniques. The main novelty in this section is the goal *3. derivation with respect to vector variables*.

4.1 Background

In this section we will review symbolic and numeric techniques for differentiation.

4.1.1 Symbolic Differentiation (SD)

In symbolic computation, *Symbolic Differentiation (SD)* calculates the derivative of a expression with respect to a particular variable, and produces an expression as its output. Typically, it accomplishes this by applying standard differential calculus rules in a divide-and-conquer manner.

Since the result of SD is itself a symbolic expression, it is advantageous compared to other methods because it allows for additional (non-obvious) simplifications to be made. On the other hand, this same feature results in the evaluation of large expressions being slow relative to other alternatives, especially when taking the partial derivative with respect to many inputs. But this depends on what constitutes a variable in the expression data type, so if only scalar variables are available, complexity will grow as a function of vector size, although it would be constant for an expression language with vector variables. Another disadvantage is that SD is only applicable to expressions with a theory of differentiation (e.g., polynomials) but not for general computer programs.

Symbolic differentiation is available in all symbolic computation applications. The most wide-spread commercial examples are Maple and Mathematica. Maple has SD for scalars only. In a limited manner, one can get around this for functions that can be converted into loops [ACCM05], for example for a curve fitting problem, but it is incompatible with other vector operations, such as a Fourier transform.

4.1.2 Automatic Differentiation (AD)

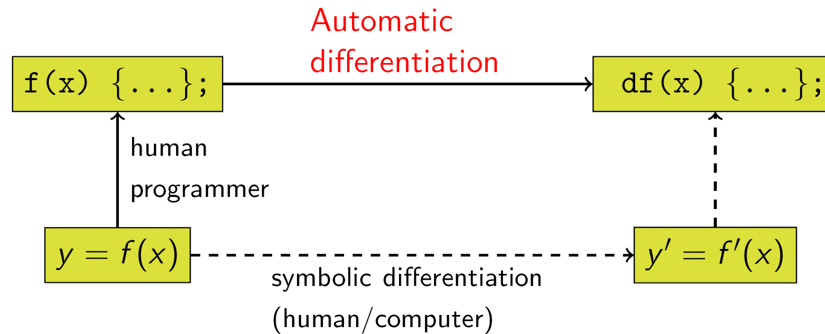


Figure 4.1: How Automatic and Symbolic Differentiation are related

Automatic Differentiation (AD), also referred to as Algorithmic Differentiation, calculates the derivative of a program in a programming language, producing a function as its output [GW08]. It does so by exploiting the fact that computer programs are, at their essence, just a series of arithmetic operations (such as additions and divisions) for which standard differential calculus rules apply. AD, therefore, derives the derivatives by continually applying the chain rule to those arithmetic operations. [MN93].

AD can, therefore, produce results for expressions that cannot be represented as a single formula. Additionally, AD preserves the efficiencies in the code as well as the sparse structure of the initial function, making it more attractive when working with massive expressions. Finally, AD (and SD) are advantageous since they do not incur truncation errors, as does finite-differencing, since it can achieve solutions in arithmetic precision [GW08, 2].

For a visual relation between AD and SD, refer to Figure 4.1 [Ber07].

4.2 Vector Differentiation in CEL

Since CEL is, at its core, an example of symbolic computation, it is not surprising the we have taken a symbolic approach to implementing differentiation. But to handle more expressions with more complicated operations, CEL differentiates implicitly rather than explicitly by applying the same standard calculus rules as it travels recursively through the DAG.

To preserve common subexpressions, differentiation is implemented by adding subexpressions to the hash table of subexpressions in the original function.

The result of implicit differentiation is the *exterior derivative*, for which there exists a formal theory which we do not exploit beyond the obvious properties. Although Maple implements the exterior derivative in the `Forms` package, that implementation was not designed for, and has not (to our knowledge) been exploited in, symbolic code generation.

The advantage of symbolic differentiation in our context is that we can quickly differentiate scalars, as well as vectors coming from 1D, 2D, 3D and 4D discretizations. Additionally, our simplification system (explained in Chapter 5) will weed out the inefficiencies that plague SD, since it identifies and preserves structures to gain efficiency in the code. This is even better than AD, which preserves efficient structures in the original program, but makes only limited further simplifications.

```
>> diff (mp ["X"]) (norm2 (ft (x +: y)))
((((Re(FT((d(X[16][16][16])+:0.0[16,16,16])))).(Re(FT((X[16][16][16]+:Y[16][16][16])))))
+ ((Re(FT((d(X[16][16][16])+:0.0[16,16,16])))).(Re(FT((X[16][16][16]+:Y[16][16][16]))))))
+ (((Im(FT((d(X[16][16][16])+:0.0[16,16,16])))).(Im(FT((X[16][16][16]+:Y[16][16][16])))))
+ ((Im(FT((d(X[16][16][16])+:0.0[16,16,16])))).(Im(FT((X[16][16][16]+:Y[16][16][16])))))))
```

Figure 4.2: CEL's differentation of $\mathrm{d}_x \left\| \mathrm{ft}(x + iy) \right\|_2$

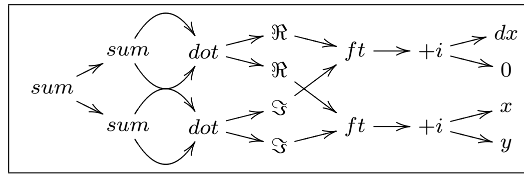


Figure 4.3: DAG of the differentation of $\mathrm{d}_x \left\| \mathrm{ft}(x + iy) \right\|_2$

In Figure 4.2, we see an example of differentiation in CEL. It takes two 3D vectors (x and y, each $16 \times 16 \times 16$), and differentiates the ℓ^2-norm of their FT,

$$\mathrm{d}_x \left\| \mathrm{ft}(x + iy) \right\|_2 .$$

The disadvantage of implicit differentiation is that it requires additional expression manipulation to isolate partial derivatives, which are required to solve the unconstrained minimization, which, in turn, is required to solve the inverse problems which motivate this thesis.

4.3 Isolating Differential Variables

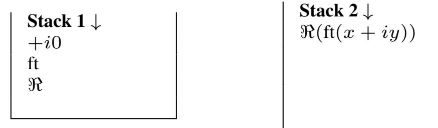
One important thing to notice in Figure 4.2 is that the differential variables, `d(X[16][16][16])` in this case, are buried deep within the expression. This happens because, as previously explained, we apply implicit differentiation recursively to our DAG, stopping at the leaves (variables and constants) and building back up the expression.

To calculate the gradient, we need to isolate these variables by having them on the left of the dot product, in the form: $\mathrm{d}x \cdot (...)$. To accomplish this we have implemented a method that utilizes stacks and adjoint operations to move around the variables and operations until we have arrived at our desired form.

As a quick reminder, a stack is a Last In First Out (LIFO) data structure with two essential operations: *push* (add an element to the top of a stack) and *pop* (remove the top item from the stack).

To isolate the differential variables we begin with two empty stacks and our symbolically differentiated DAG. When we find that there is a differential variable along one branch and not along

1. Push the part with the differential variable onto **Stack 1** and the other part onto **Stack 2**



2. When we find a differential variable, pop off **Stack 1** and push the adjoint onto **Stack 2** as adjoint

Figure 4.4: Isolating differential variable dx in the expression $\Re(\text{ft}(\text{d}x + i0))) \cdot \Re(\text{ft}(x + iy))$.

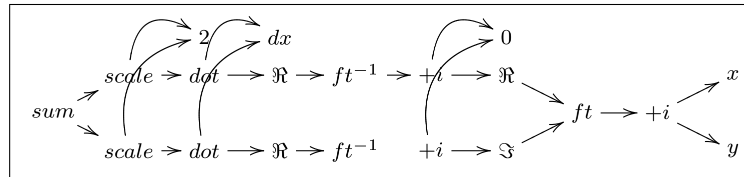


Figure 4.5: Graph rebuilt to isolate differential variable

the other, we push all of the operations that lead to the differential variable onto the first stack and the other branch of operations onto the second stack. Finally, we pop the operations from the first stack and put their adjoints onto the second stack. We then pop everything off the second stack, rebuilding the tree as we go, and we are left with a new expression that has the differential variable isolated.

An example is seen in Figure 4.4 where we see dx isolated from the expression $\Re(\text{ft}(\text{d}x + i0))) \cdot \Re(\text{ft}(x + iy))$, resulting in the new expression d$x \cdot \Re(\text{ft}^{-1}(\Re(\text{ft}(x + iy))) + i0)$. The new DAG can be seen in Figure 4.5

Chapter 5

Simplification

In this section we will explore the importance of and techniques for simplification as well as its implementations and outline what would be required to prove that this framework is sound. While the pattern replacement snippets clearly have the potential to be developed into a language, what we claim for now is a substantial down-payment on *6. an interface for computer scientists to specify simplification and parallelization operations algorithmically*.

5.1 Background

While often considered to be the more pervasive process in algebraic manipulation, simplification is also said to be the most controversial [Mos71]. While the first goal of simplification systems should be that simplified expressions are equivalent to the initial expressions [Sto11], the controversy arises due to the manner in which context redefines what it means for an expression to be simplified. For example, $(x - x)^3$ should be simplified to 0, but what should happen to $(x - y)^3$?

For CEL, the objective of applying simplification is to achieve a reduction in running time by eliminating unnecessary and redundant computations while maintaining equivalence to the initial expression.

5.2 Simplification in CEL

The objective for the simplification system in CEL is to maximize computational efficiency. To accomplish this we needed to:

1. Discard unnecessary computations
2. Replace computations for ones that are more efficient
3. Ensure that all common subexpressions are found

While a full list of the simplification rules can be found in Appendix A, let us focus on a few which can help exemplify our solutions to the three types of simplifications listed above.

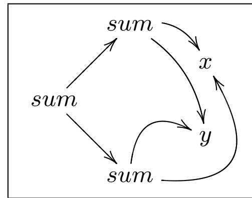
The first type of simplification is the elimination of unnecessary computations. Some of the rules are obvious, such as $x + 0 \rightarrow x$, which will save us an add operation. Some of these rules

save us thousands of floating point operations, such as removing unnecessary Fourier transforms, as in $\mathrm{ft}(\mathrm{ft}^{-1}(x)) \rightarrow x$.

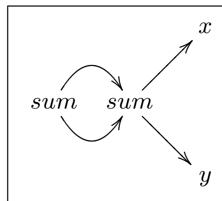
The second type of simplification is the replacement of operations with others that are more computationally efficient, as in $x + x + x + x \rightarrow scale(4, x)$.

The third type of simplification is the discovery of additional common subexpressions. While all rules help to do this in some way, operations which are invariant under certain transformations (e.g., the order of operands to commutative binary operations) can be considered equivalent by CSE and thus may induce further simplification. This is demonstrated in Figure 5.1, which shows the importance of discovering that $(x + y) + (y + x)$ is the same as $(x + y) + (x + y)$

1. With no simplification, sum hashes to a different node number for $x + y$ and $y + x$



2. At a normal form, $y + x$ is converted to $x + y$, and the CSE is exposed:



3. Additional simplifications are now exposed as well.

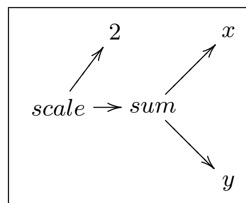


Figure 5.1: Simplifying into a normal form to expose CSEs.

To accomplish the actual simplification we have a series of simple rules that the graph falls through in a manner similar to a common `switch` statement. Some sample rules, written in Haskell, have been included in Figure 5.2, demonstrating how the fall through method is implemented and the simplicity of the rules themselves.

The fall through function `simplify` must be continually called until the graph is simplified, which raises the obvious question: "when is it safe to claim that a graph is simplified?" We found the answer to be: *when the graph stops changing*, since no more of the simplify rules apply. For our `switch` analogy, this would be like falling into the `default` case. At first this might seem obvious, but it points to an important feature of our rules, in that they must not cancel each other out or the algorithm could run indefinitely. That is to say, our system must be confluent so that our

```
-- collapse 1 scale anything
| Just (s,x) <- M.scale exprs node
, Just (1,_) <- M.const exprs (snd $ s exprs)
= x exprs
```

```
-- collapse Re ( x +: _ ) into x
| Just (x,_) <- (M.xRe `o` M.reIm) exprs node
= x exprs
```

```
-- collapse scaling of a scaling of a vector
| Just (s1,x1) <- M.scale exprs node
, Just (s2,x2) <- M.scale exprs (snd $ x1 exprs)
= ( ( s1 * s2 ) `scale` x2 ) exprs
```

```
-- collapse invft (ft (x) ) into x
| Just (x) <- (M.invFt `o` M.ft) exprs node
= x exprs
```

Figure 5.2: Three simplification code examples written in Haskell: $scale(1, x_i) \rightarrow x_i$, $\Re(x_i + ix_j) \rightarrow x_i$, $scale(c_1, scale(c_2 * x_i)) \rightarrow scale(c_1 * c_2, x_i)$, and $\text{ft}^{-1}(\text{ft}(x_i)) \rightarrow x_i$ respectively.

expressions are guaranteed to to arrive at a normal form.

The simplicity of the individual rules (a full list of which can be found in Appendix A), is powerful, since it makes them easy to reason about mathematically.

5.3 Well-definedness

There are some basic properties that need to be proven about our simplification methods to demonstrate that they are sound. By mapping our simplification rules onto a Noetherian *abstract rewriting system (ARS)* we conjecture that the properties of termination, confluence and canonical will hold. The simplification rules are fully listed in the tables of Appendix A.

5.3.1 Confluence

In Chapter 3 we discussed why it is important for our simplification algorithm to arrive at a normal form. In this section we will explore what this means and how we proved that our methodologies have this property.

The *normalisation* or *reduction* process is implemented by repeatedly applying rewrite

rules until none of them apply, meaning that the expression has arrived at a normal form. Confluence holds if and only if there exists exactly one normal form for any term [Toy05].

The Diamond Property

Proving the confluents of the majority of our rules is easily done with the help of the diamond property. In Figure 5.3 we demonstrate an example of two rules for which the diamond property holds, (1) collapsing scales of scales: $scale(s_0, scale(s_1, x_0)) \rightarrow scale(s_0 * s_1, x_0)$ and (2) a sum with a single input can be rewritten as just that element: $sum(x_0) \rightarrow x_0$.

When we substitute the second into the first, since they have a common variable, x_0, we will first rename x_0 to x_1. The substitution thus results in $scale(s_0, scale(s_1, sum(x_1)))$. The two rewrite rules result in two possible outcomes: (a) $scale(s_0{*}s_1, sum(x_1))$ or (b) $scale(s_0, scale(s1, x_1))$. Either way, they join with the result in a single step: $scale(s_0 * s_1, x_1)$. This can be visualized in Figure 5.3.

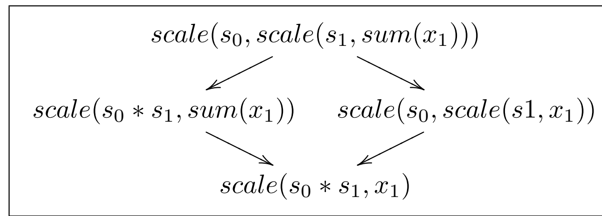


Figure 5.3: Visual demonstration of the diamond property for the rules $scale(s_0, scale(s_1, x_0)) \rightarrow scale(s_0 * s_1, x_0)$ and $sum(x_0) \rightarrow x_0$

For all pairs for which this diamond property holds, no additional work needs to be done to demonstrate that these rules are confluent. However, for those that do not have this property, additional properties must be demonstrated. We conjecture that this will hold, however the proofs are outside the scope of this thesis.

Chapter 6

Compilation, Runtime & Verification

Up to this point, we have been generating graphs that easily induce serializable code. We built two systems to run that code on a single core: a recursive Haskell interpreter and a version that generates sequential code to solve the DAG nodes in topological order. The details of these have been omitted from this thesis and we will move directly into the more interesting parallel version.

In this chapter we will explore how we produce very fast, parallelized, verifiable code to run on a multi-core system, meeting goals for *7. an efficient parallel run-time system to execute generated code,* and *8. an efficiently verifiable model of synchronization*.

6.1 Background

To provide the reader with a better understanding of the base for our design decisions when building the runtime system, we will discuss in this section some important details about the tools and computer architecture that were utilized.

6.1.1 Single Instruction, Multiple Data streams (SIMD)

First we will briefly discuss SIMD in relation to other architectural groups and then we will focus on the strengths and weaknesses of SIMD instructions.

Flynn's taxonomy: Michael J. Flynn proposed [M.J72] a classification of architecture into four groups based on the number of current instructions (single or multiple) and data streams (also single or multiple) available to them.

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Figure 6.1: Flynn's taxonomy

Very briefly, *Single Instruction, Single Data stream (SISD)* (for example, a traditional uniprocessor) operates sequentially on one operation at a time. *Single Instruction, Multiple Data streams (SIMD)* (for example, a *Graphical Processing Unit (GPU)*) does a single instruction on

multiple pieces of data (such as on multiple pieces of an array) in parallel. *Multiple Instruction, Single Data stream (MISD)* does multiple operations in parallel on a single unit of data. Finally, *Multiple Instruction, Multiple Data streams (MIMD)* has multiple processors executing different instruction on different data, such as in a distributed system. MIMD are the most popular in modern day supercomputers. See Table 6.1 to compare the four types.

SIMD Instructions: Since the 1970s, SIMD instructions have been used to manipulate vector data with a single instruction. Originally reserved for supercomputers, today SIMD processors are quickly becoming a standard, even for desktop computers. Meanwhile, supercomputers have long since moved on to MIMD. While many applications running on a modern desktop would not show significant gains from SIMD instructions since they cannot be vectorized, vector operations such as image processing (for activities like gaming) have seen significant gains from these instruction sets.

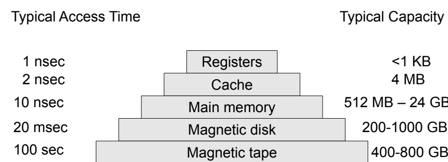


Figure 6.2: Typical memory hierarchy with rough approximations for average access times and capacity [Tan08]

Another interesting advantage of SIMD is the ability to do multiple operations while keeping data in registers. In Figure 6.2 we see a review of the tiers of memory along with their approximated access times and sizes as approximated by Tanenbaum's *Modern Operating Systems* book [Tan08]. As the access times in Figure 6.2 demonstrate, if a couple of operations can be done on data before it returns to cache, or worse, to memory, there are great reductions in the data access times.

However, SIMD instructions's disadvantages include being seen as hard to use and, more significantly, being architecture specific, leading to error prone non-portable code. Some compilers have overcome this by performing automatic *SIMDization* for the user.

6.1.2 Distributed Shared Memory

For modern CPUs with multiple cores, memory access is of important concern. *Uniform Memory Access (UMA)* and *Non-Uniform Memory Access (NUMA)*, are two different styles of memory access.

Uniform Memory Access (UMA) In UMA architectures, every core has the same access time for data in memory. [ZSLW92] In *symmetric multiprocessing (SMP)*, the most commonly used UMA architecture, a major bottleneck was discovered on the shared bus which was the only memory access point.

Non-Uniform Memory Access (NUMA) In NUMA architectures, each core has its own memory, but can access the memory of the other cores via a series of interconnected buses. As a result, the access time is relative to the distance of the core from the data that it requires, which can have an

order of magnitude effect on the access time for that data [BTK06]. NUMA has been shown to be far more scalable than UMA.

6.1.3 Westmere-EP

Previous work on Coconut [AK08] has targeted PowerPC+Altivec, Cell/B.E., and other architectures, but this was the first time that we were designing for the family of Intel® chips codenamed Westmere-EP. Our experimentation with this architecture is with the two Xeon® Processor X5690.

The processors each have 6 cores, each with hyper-threading, which share memory via Non-Uniform Memory Access (NUMA). Each core has its own L1 and L2 cache, but each 6-core node/chip shares an L3 cache.

Each processor has its own memory controller which, in data-intensive applications, will be a bottleneck. This can be alleviated by, as much as possible, storing data in the bank of memory attached to the processor on which the data will be processed.

6.2 AVOps in CEL

Coconut uses a model of parallelization centred on *Atomic Verifiable Operations (AVOps)*, indivisible components of computation and communication which are either pure computations (without side effects) or solely data transfer operations, including synchronization via signalling. In previous Coconut multi-core theory and practice concerning AVOps, signalling as a means for communication was the central research focus, with the assumption that there is native support for efficient signalling (as there is on the Cell/B.E. architecture, for example), but the current architecture uses caches and does not allow the user to control memory locality explicitly. As a result, we implemented an alternative run-time system which does not support signals, but in which synchronization is guaranteed by finite-sized ring buffers and the insertion of no-ops to guarantee the completion of previously dispatched computations prior to the commencement of the preceding computations. No-ops are needed for operations which in other parallelization frameworks would require a barrier. In many cases where it is convenient for a programmer to use a barrier, the AVOp scheduler can use pipelining instead.

The AVOps for CEL were expanded on from those in previous Coconut papers. In the context of our project, they could now be any atomic operation, so long as they kept the initial rules of being a single work-unit and of being a pure computation (with no side-effects). Included in this are all operations from the initial functions described in the list of abbreviations at the beginning, as well as those needed for the runtime (such as the previously mentioned *no-op* “operation”) in order to make guarantees about the verification (which will be further expanded on in Section 6.4).

Behind the scenes, the actual work associated with each AVOp is a single C function. So long as that operation is targeting a vector, it is written with SIMD to maximize efficiency. The simplicity of an individual AVOp, written with a single function without side effects, makes it very easy to unit test.

6.3 The Runtime System

The basic idea behind the runtime system is to have the main thread, known as the dispatcher, dedicated entirely to offloading work to a set of worker threads. Each of the worker threads operates independently and continuously to execute the AVOps that are being passed onto them. Each of

the worker threads sits on one of the inter-connected Intel® Xeon® X5690 cores, unless hyper-threading is turned on, in which case two workers share the core and work simultaneously but still independently.

The runtime system is written in C using only core libraries like Pthreads [But97], as well as Intel's architecture specific libraries. NUMA specific facilities like numactl and libnuma [DB11], were experimented with, but were discovered to give us insufficient control over the locality of the worker threads as they targetted full nodes instead of individual cores. While Pthreads, on the other hand, were designed for UMA architecture [AJR06], our experimentation with functions like `pthread_setaffinity_np` [set10] and observations of the real-time running system using top [War02] gave us confidence that we had achieved the necessary control for our model.

Each worker core is allocated a ring buffer, as is depicted in Figure 6.3. The ring buffer contains AVOps of varying sizes, depending on the number of arguments (either immediate arguments or pointers to inputs or output). The AVOps can be thought of as instructions on a virtual machine, with pointers corresponding to registers. As the worker thread finishes operations, it marks the corresponding AVOp as completed.

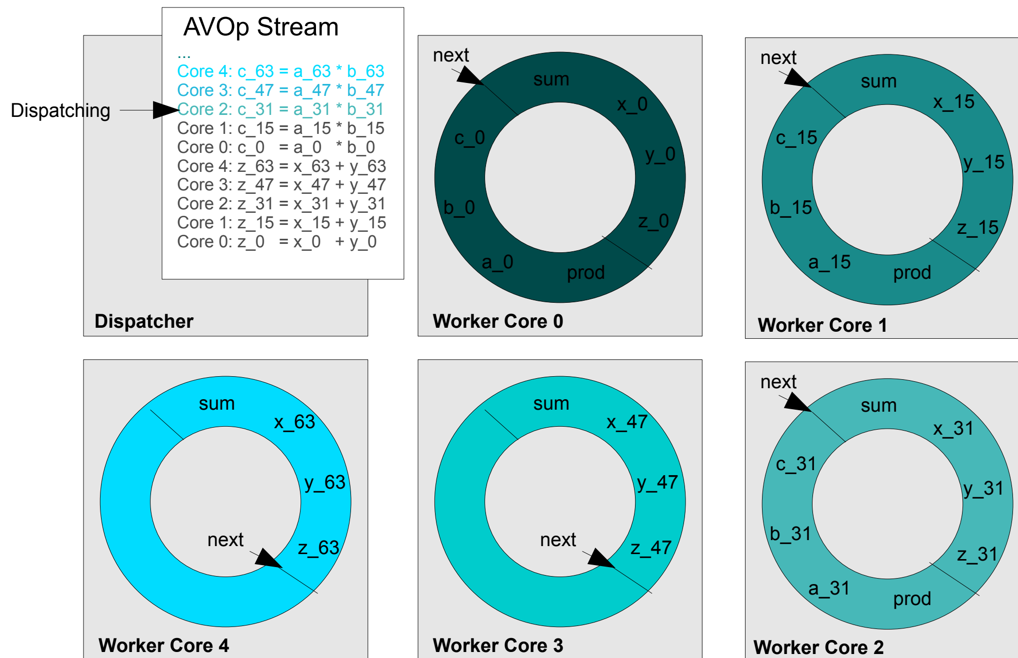


Figure 6.3: AVOps distributed onto worker core ring buffers

At `dispatch`, the dispatcher thread receives a stream of AVOps to allocate. Each AVOp has been marked with a designated core. Before providing each core with the next AVOp, the dispatcher first checks that there is sufficient room on the corresponding worker thread's ring buffer. If there is insufficient room, the dispatcher waits.

The only communication between worker threads and other threads is when indicating to the dispatcher thread that it has completed a task. Otherwise, each worker is working asynchronously, either working on a task or waiting for new work. The bus, as a result, remains clear for

the dispatcher to do its dispatching.

6.4 Verification

While the majority of operations available to AVOps are linear, and each worker works its own series of operations on its own pieces of data, there are still, inevitably, synchronization points in the AVOp streams. As the worker threads are asynchronous they cannot utilize standard locking mechanisms (such as monitors, semaphores and locks) to guard memory and avoid corruption. We, therefore, implement a verification system (in C) to check that a stream of AVOps is safe.

The three rules that we have outlined for safe reads and writes for any particular memory location are as follows:

1. Any reads and writes on the same core are safe
2. Reads on different cores are safe
3. If any core is writing then no other core may be reading or writing

These rules are enforced by running the stream of AVOps to be verified with a particular size preset for the ring buffer. The verifier builds the ring buffers of the specified size, say l, for the specified number of cores, say m. For each of the n AVOps in the stream it checks that, if there are inputs and outputs, they all obey the three rules specified above when compared to all AVOps (and their input and output pointers) currently sitting on the ring buffers of the workers. If it is safe to add that AVOp, it is added to the appropriate ring buffer (which overwrites the old AVOps and pointers as it loops around the ring buffer). If the AVOp is not safe, the verifier stops and reports why it has failed.

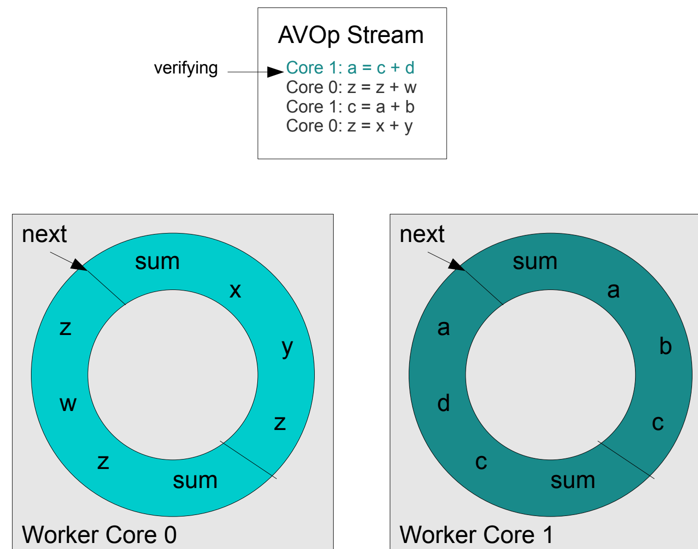


Figure 6.4: Verifying a safe AVOp stream

For example, in Figure 6.4 we see an AVOp stream for two worker cores with a ring buffer of size 8. The final AVOp, $a = c + d$, has just been added to worker core 0. The verifier then verifies that core 0 does not have any writes to a, c or d. Since there are no conflicts the stream passes.

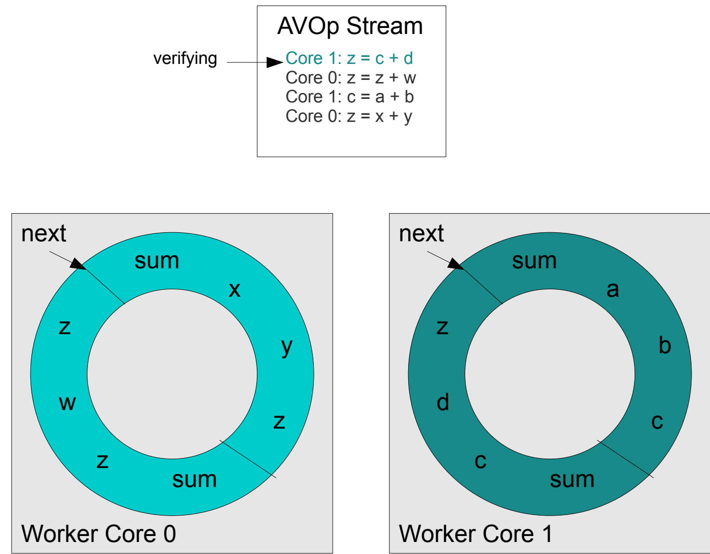


Figure 6.5: Verifying an unsafe AVOp stream

In Figure 6.5, we see the same setup of cores and ring buffers, but with a slightly different AVOp stream. After placing the final AVOp onto core 1, $z = c + d$, the verifier analyzes the buffers and finds that there is a conflict with the variable z, since both cores are attempting to write to the same variable. The verifier immediately exists with a failure.

While this might at first seem inefficient since, for a ring buffer of size l with m cores and n AVOps in the stream, the verifier runs in $O(lmn)$, it is important to note that l and m are small constants, so the verifier's execution time is linear in the order of the number of AVOps in the stream. Additionally, the verifier only needs to run once for a stream of AVOps before it is verified to be safe for a particular ring buffer size, allowing those AVOps on the asynchronous worker threads to work on arbitrary data. We expect that for safety-critical applications AVOps streams would all be generated and verified at time of software packaging, and therefore no verification would be required in the field.

6.5 Results

An important question when analyzing our system is how well it scales as we add additional cores.

Looking at the real computation CPU time of the cores, we generated code for 60 million additions, with a resolution size of 128×512. In the graph in Figure 6.6, we demonstrate how the application scales as additional worker cores are added.

Another way of looking at scalability is via the additional overhead of distributing work to cores as compared to the time spend actually doing work. We analyze this overhead by looking at the normalized execution times.

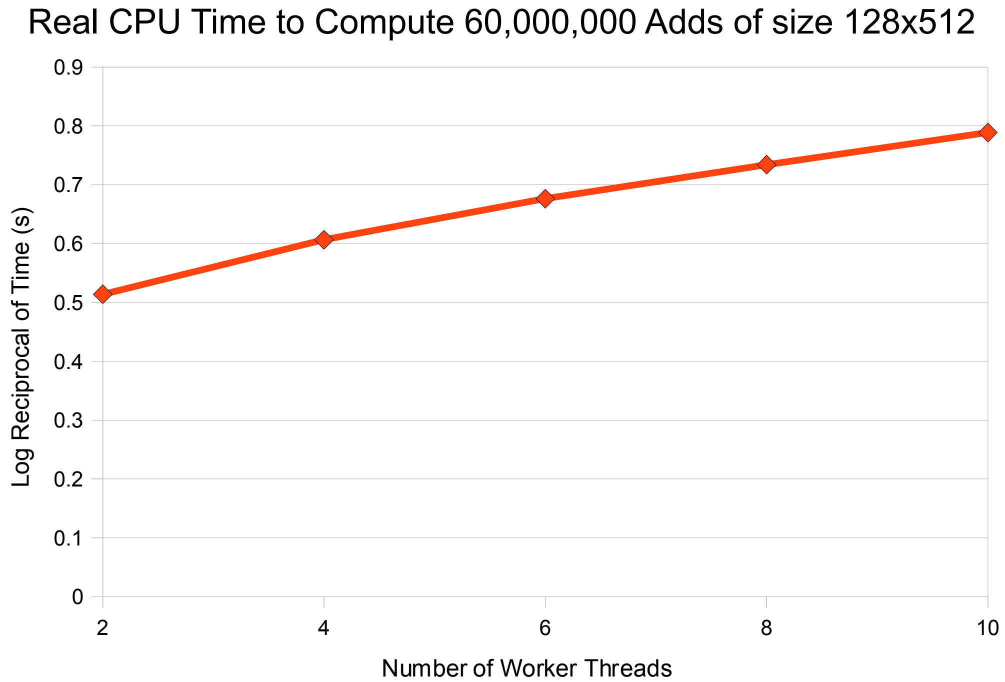


Figure 6.6: Scalability results

In Figure 6.7 we see that the overhead change from two to ten cores is less than five percent.

As additional multi-core nodes are added, increasing the distance from the dispatcher to the new cores, we expect this to increase. However, intelligence in the positioning of the new nodes and in the locality of the data will help control this.

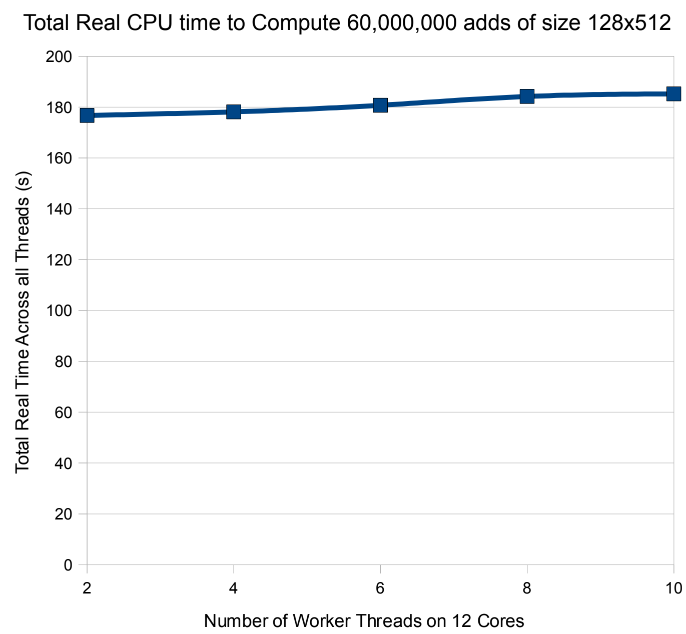


Figure 6.7: Normalized execution time results

Chapter 7

Graph Transformations for Parallelism

Some graph transformations can better prepare the graph for optimal performance on a parallel architecture such as ours. They cannot be called simplifications. In fact, they sometimes increase the size of the graph.

In this section we will explore graph transformations that expose the parallelism to the right granularity, create new nodes with cache size considerations, and permute solving order to avoid memory bottlenecks. The graph transformation methodology is the same rule-based system as explained for simplification, and therefore will not be re-addressed here. This is another demonstration that we have provided *6. an interface for computer scientists to specify simplification and parallelization operations algorithmically*.

7.1 Memory Locality and Image-Space Decomposition

It is highly advantageous if, once a worker core acquires a work unit, it does all possible work on it before moving onto another piece. Additionally, by accessing entire cache lines instead of scattered bytes, less time is wasted pulling new data into memory. With this in mind, we order the operations so that each worker core receives as large a stream of operations as possible before moving onto a new data set.

As an example, suppose we had vectors x, y and z, and the expression $(x + y) - z$. The size is such that we will need more than one iterations of our worker cores on each operation. For worker core 0, we could offer the AVOps in Figure 7.1 (note that we will be re-using memory and the solution will be found on top of the original x).

$$\begin{aligned} x_0 &:= x_0 + y_0 \\ x_l &:= x_l + y_l \\ x_0 &:= x_0 - z_0 \\ x_l &:= x_l - z_l \end{aligned}$$

Figure 7.1: First ordering of operations for core 0 of the expression

However, this would result in the x_0 leaving memory to make room for x_l and then being re-fetched

into memory. For memory locality, it is more practical to use the ordering in Figure 7.2. By performing the additional operations while the variables are still in cache we are able to save memory fetches resulting in an over all reduction in runtime.

$$\begin{aligned} x_0 &:= x_0 + y_0 \\ x_0 &:= x_0 - z_0 \\ x_l &:= x_l + y_l \\ x_l &:= x_l - z_l \end{aligned}$$

Figure 7.2: A second ordering of operations for core 0, optimized for memory locality

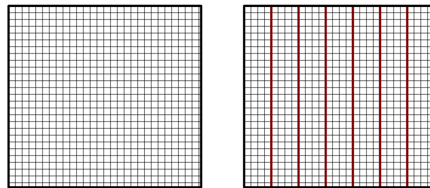


Figure 7.3: Data decomposed into strips, with the worker cores each running operations on their own strip.

It follows from these arguments that all operations should be performed on cache-lines of data. In fact, map operations, such as sum and $scale$ have the property that a subset of the worker cores can separately and asynchronously work on arbitrary pieces of the data, but other operations, particularly the Fourier transforms, need to access larger sets at once. Figure 7.3 demonstrates how the data can be split up so that each worker core only needs to focus on their on strip, and map and column-transform operations are both efficient. It is highly desirable that the strips are small enough to fit into the L1 cache, because the latency of fetching from L1 is substantially lower than L2. This is possible for most image sizes, but would not work for 2D and 3D Fourier transforms.

7.2 Operation Transformations

Some additional transformations can be done to alter operations that will help in improving performance.

7.2.1 Breaking apart Fourier transforms (FTs) that are followed by Projections:

In our MRI problems we frequently have a 2D Fourier transform (FTs), followed by a projection (the DAG for this can be found in Figure 7.4). This is because we oversample the data in different directions to prevent out-of-field artifacts or noise from appearing in the final image. Because power-of-two FTs are most efficient, this oversampling is usually a factor of two. This extra data is not required for the computations after each FT, and the projection is used to throw out the top quarter and bottom quarter of the data, leaving transformed data of half the size.

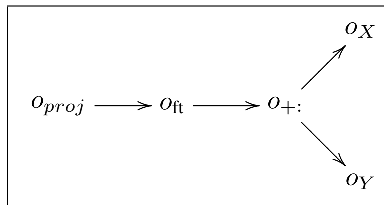


Figure 7.4: DAG for 2D Projection ∘ ft

It follows that we are able to cut the time for the second FT direction by breaking apart the FT and pulling the projection in between the two pieces (the DAG for this newly transformed series of operations can be seen in Figure 7.5). We have, therefore, cut the number of operations in half for the column portion of the FT.

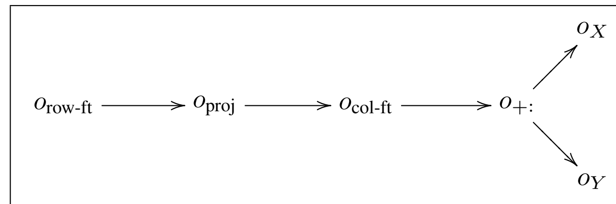


Figure 7.5: Transformation of DAG for 2D Projection ∘ ft for optimization.

However, we can do more. If we group data to take maximum advantage of SIMD instructions, as we will explain in the next section, Fourier transforms in the column direction are easier and faster than their row equivalents. Since a row FT is simply a transposed column FT, we can transform the graph to perform column FTs instead by replacing row FTs with column FTs composed with transposes before and after. The final transformed graph after all rules have been applied is shown in Figure 7.7.

7.3 Column Block Ordering

(0,0)	(0,4)	(0,8)	(0,12)	(2,0)	(2,4)	(2,8)	(2,12)
(0,1)	(0,5)	(0,9)	(0,13)	(2,1)	(2,5)	(2,9)	(2,13)
(0,2)	(0,6)	(0,10)	(0,14)	(2,2)	(2,6)	(2,10)	(2,14)
(0,3)	(0,7)	(0,11)	(0,15)	(2,3)	(2,7)	(2,11)	(2,15)
(0,16)	(0,20)	(0,24)	(0,28)	(2,16)	(2,20)	(2,24)	(2,28)
(0,17)	(0,21)	(0,25)	(0,29)	(2,17)	(2,21)	(2,25)	(2,29)
(0,18)	(0,22)	(0,29)	(0,30)	(2,18)	(2,22)	(2,29)	(2,30)
(0,19)	(0,23)	(1,17)	(0,31)	(2,19)	(2,23)	(3,17)	(2,31)
(1,0)	(1,4)	(1,8)	(1,12)	(3,0)	(3,4)	(3,8)	(3,12)
(1,1)	(1,5)	(1,9)	(1,13)	(3,1)	(3,5)	(3,9)	(3,13)
(1,2)	(1,6)	(1,10)	(1,14)	(3,2)	(3,6)	(3,10)	(3,14)
(1,3)	(1,7)	(1,11)	(1,15)	(3,3)	(3,7)	(3,11)	(3,15)
(1,16)	(1,20)	(1,24)	(1,28)	(3,16)	(3,20)	(3,24)	(3,28)
(1,17)	(1,21)	(1,25)	(1,29)	(3,17)	(3,21)	(3,25)	(3,29)
(1,18)	(1,22)	(1,26)	(1,30)	(3,18)	(3,22)	(3,26)	(3,30)
(1,19)	(1,23)	(1,27)	(1,31)	(3,19)	(3,23)	(3,27)	(3,31)

Figure 7.6: A vector with 32 rows and 4 columns in column block order

When working with linear operations, such as sum and $scale$, automatic *SIMDization* is easy to do asynchronously, so long as we know that every piece of data will see the operation eventually. However, the operation $transpose$ using SIMD requires access to a full cube of the size of SIMD to safely operate. Meanwhile, the highly optimized FT libraries would like the data in column ordering. To avoid moving data around, we rearrange the vector data at the start and end of our program into *Column Block Ordering*. In essence, the vectors are broken up into block sizes on which a transpose can be done, and they travel in columns (instead of rows). See Figure 7.6 for a sample ordering with SIMD capable of 4 simultaneous floating point operations.

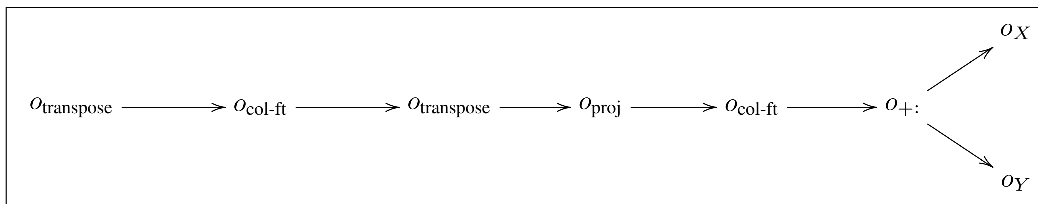


Figure 7.7: Transformation of DAG for 2D Projection ∘ ft optimized for preference of column FTs over row FTs

Chapter 8

Parallelizing L-BFGS

At the beginning of this thesis, we discussed how methods such as profiling velocity with PCA are susceptible to noisy data, but also lent themselves naturally to optimization problems. As part of our architecture specific system described in Chapter 6, we implemented and integrated a mathematical optimizer based on the algorithm for the *Limited Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS)* and optimized and parallelized it. In this chapter, we will explore the challenges we faced, and how we overcame them during that implementation, realizing *9. an efficient parallel unconstrained optimizer into which to plug generated function and gradient evaluators*.

8.1 Background

To understand why parallelizing L-BFGS was challenging, we must first understand the algorithm behind it. In this section we look at the origins and principles required for understanding the L-BFGS algorithm, starting at Newton's method for finding stationary points of functions and building our way up to the challenges of parallelization methods for L-BFGS.

8.1.1 Newton's Method

In mathematical optimization, *Newton's method* is used to find stationary points of differentiable functions. The stationary points are arrived at through an iterative method, which seeks out the roots or extrema of functions.

Starting with an initial guess x_0, it iterates by calculating

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}$$

where $f'(x_i) \neq 0$, $i \geq 0$. The iteration stops when the changes fall below a pre-set tolerance, resulting in a solution x_i, for a final iteration i.

8.1.2 Broyden-Fletcher-Goldfarb-Shanno (BFGS):

In 1970, Broyden, Fletcher, Goldfarb and Shanno each came up seperately with essentially the same algorithm (expressed as Algorithm 1 and referred to as *BFGS*) for solving non-linear optimization

problems. It is considered to be both one of the most famous and most efficient of the quasi-Newtonian methods [Dai02]. It works on problems that have a solution where the gradient is zero and have a quadratic Taylor expansion near the optimum.

Algorithm 1 The BFGS Algorithm

Require: $x_1 \in R^n$; $B_1 \in R^{n \times n}$ positive definite
Require: $tolerance \in R$ ▷ Acceptance test for gradient being zero

  $g_1 \leftarrow \nabla f(x_1)$
  if $g_1 = 0$ **then return** x_1
  end if

  $k \leftarrow 1$
  loop
    $d_k \leftarrow B_k^{-1} g_k$
    Carry out a line search along d_k
    $\alpha_k > 0$
    $x_{k+1} \leftarrow x_k + \alpha_k d_k$
    $g_{k+1} \leftarrow \nabla f(x_{k+1})$
    if $g_{k+1} < tolerance$ **then return** x_{k+1}
    end if
    $s_k \leftarrow \alpha_k d_k$
    $y_k \leftarrow g_{k+1} - g_k$
    $B_{k+1} \leftarrow B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{s_k^T} y_k$
    $k \leftarrow k + 1$
  end loop

8.1.3 Limited Memory BFGS (L-BFGS):

Limited Memory BFGS (L-BFGS) is an algorithm introduced by Nocedal in 1980 [Noc80] for approximating BFGS using less memory. It does so by only storing a few vectors that represent the approximation of the Hessian matrix implicitly. Instead of using a low rank update, it represents the low rank of the Hessian as a sum of outer products. The algorithm is defined in Algorithm 2.

8.2 Improvements

We are able to reduce the time and space complexity of this algorithm while performing a parallel version with some optimizations.

8.2.1 Skipping Copies

At each iteration of L-BFGS, the new current value and gradient values are copied onto the previous current value and gradient value so that the next iteration will be able to provide access to those values for comparison once the newer values are calculated at that iteration. Since our AVOps use offsets in memory we skip these copies at every iteration by simply using a C macro that swaps the pointers back and forth (using a simple $i \bmod 2$) to establish which is the previous and which is the

---

Algorithm 2 The L-BFGS Algorithm

Require: $x_1 \in R^n$
Require: $B_1 \in R^{n \times n}$ positive definite
Require: $M \in R$ ▷ History Size
Require: $tolerance \in R$ ▷ Acceptance test for gradient being zero

```
  g_1 ← ∇f(x_1)
  if g_1 < tolerance then return x_1
  end if

  k ← 1
  loop
      d_k ← g_k
      if iteration ≤ M then
          increment ← 0
          bound ← iteration
      else
          increment ← iteration − M
          bound ← M
      end if
      for i ← (bound − 1)..0 do
          j ← i + increment
          α_i ← ρ_j s_j^T q_{i+1}
          q_i ← q_{i+1} − α_i y_j
      end for
      r_0 ← H_0 · q_0
      for i ← 0..(bound − 1) do
          j ← i + incr
          β_j ← ρ_j y_j^T r_i
          r_{i+1} ← r_i + s_j(α_i − β_j)
      end for
      x_{k+1} ← x_k + α_k d_k
      g_{k+1} ← ∇f(x_{k+1})
      if g_{k+1} < tolerance then return x_{k+1}
      end if
      s_k ← α_k d_k
      y_k ← g_{k+1} − g_k
      k ← k + 1
  end loop
```

---

new offset to look at. This is a small optimization that saves us the overhead of moving all of the data comprising the current guess and the gradient around at each iteration.

8.2.2 Fusing Dot Products

To establish whether an optimum has been reached, the algorithm computes a series of subtractions and dot products. These are significant since the definitions of the new s_k and y_k through the subtractions feed into the dot products, and those dot products establish the next steps for the algorithm, meaning that we arrive at a synchronization point for the algorithm. As a result, it is crucial to maximize the efficiency at this point.

To increase performance of these operations, we take advantage of SIMD and do as much work as possible while the data is in registers. Each worker is provided with a piece of their share of the new and old gradient and function vectors. Working on four or eight floats at a time (depending on the system architecture), the function pulls the values for these vectors into registers. It first does the subtractions and then uses those values to do double precision sums for each of the four necessary dot products. Each of the cores stores their results to separate locations. Finally, all but one of the cores wait while the remaining core sums the totals so that they can be evaluated as if they had been done in a completely serialized manor. The body of this function is provided in Appendix B.

8.3 Results

Due to time constraints the regularizers required for the PCA experiments have not yet been fully implemented. We will therefore demonstrate results in this section for another MRI problem, known as SENSE, in which partial data sets from multiple receiver coils are combined to produce an image. Unlike the velocity problem, the signal model is linear, so the objective function is quadratic. This is an easier problem to implement first, but will give a good indication of potential speed-ups for real inverse problems.

Experiments were done using synthetic data from two coils created by taking a real single-coil brain image from a low noise experiment and multiplying point wise by a second array of complex numbers computed using the analytic solution for a dipole coil. Initial experiments are performed with synthetic data in order to have known true image, and therefore an exact measure of accuracy. The first performance test used 86% of the data at a resolution of 64×64. The first ten iterations of L-BFGS can be seen in Figure 8.2, demonstrating a remarkable improvement in image quality over those iterations.

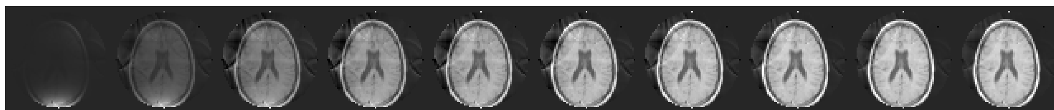


Figure 8.1: First ten iterations of the real part of the SENSE problem.

The experiment was performed first in a serial manor and then with 10 cores. The times were taken using the UNIX `time` command, returning the elapsed time between invocation and termination (Real), the user's CPU time (User) and system CPU time (System), and are expressed in Table 8.1. There is a notable 8.6 times speadup over its serial equivalent for the real time.

A second set of experiments were performed with three coils and 54% of the normal data. In this case, ten executions of the solver with fixed numbers of iterations (1, 50 and 100) were run in sequential and parallel mode. The variability in execution time was significant, indicating interference from other processes, but the minimum execution time was relatively stable. Using the ratio of the difference between the time for 100 iterations and the time for 50 iterations, in the two

	Serial Time (s)	10 Cores Time (s)
Real	2.876	0.333
User	29.692	3.357
System	0.062	0.075

Table 8.1: Times in seconds for the SENSE problem using 86% of the data from two coils.

modes, estimates a acceleration of 5 times. We conjecture that the task schedule found for the 3-coil problem is not as efficient as the task schedule for the 2-coil problem, and this is being investigated, but a discussion of task scheduling is outside the scope of this thesis.

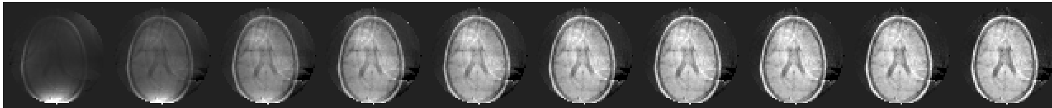


Figure 8.2: First ten iterations of the real part of the SENSE problem using 54% of the data from three coils with approximately 50% noise added.

Chapter 9

Conclusion

This thesis makes significant contributions to several goals of the Coconut project. As enumerated in the introduction, to generate efficient parallel solvers from mathematical model specifications involves many steps. In each chapter, we considered some of the steps required. The DSL provides a very abstract layer on which scientists and physicists can work without any understanding of the computer science and software engineering principles that went into it. At this stage, a variety of models can be expressed, with several examples from MRI having been so encoded. Differentiation is completely automatic, which means it is fast and, more importantly, error free. The treatment of vector variables as first-class objects whose structure is preserved by differentiation is a significant contribution to the literature on symbolic computation. Efficiency both in model manipulation and the generated code relies on subexpression elimination which is built into the expression data structure. Through additional simplification and transformation methodologies we are able to introduce additional computational efficiency at many levels, and expose structures suitable for efficient parallelization at the level of SIMD and multi-core parallelism. Finally, we modified the existing Coconut multi-core run-time framework to run on shared-memory architectures where it had been designed for distributed-memory architectures containing hardware support for signalling. The run-time framework contains two parts which were developed from scratch: a run-time library and a linear-time verification mechanism.

There is, however, much work to be done. There are some operations, specifically to do with convolution/map/zip, that require additional implementation before we are able to fully solve a broader class of MRI problems, including the PCA problem discussed in the introduction. As we add more cores, we expect that our graph scheduling will need to include additional intelligence about being node-aware by penalizing scheduling patterns that tax inter-core and inter-chip bandwidths. Also, we suspect that supplemental research into DAG graph scheduling will inspire additional intelligence being built into graph transformations leading to better runtimes. At that point we look forward to being able to validate the PCA model discussed with new regularizers and much experimentation, and proceed to clinical validation.

Appendix A

Simplification Rewrite Rules

The following table of rules are applied in order to simplify expressions.

s is used in place of scalars, x is used in place of vectors, z is used in place of complex vectors, ss is used in place of sub spaces and c is used in place of constants.

collapse scales	$scale(s_0, scale(s_1, x)) \rightarrow scale(s_0 * s_1, x)$
collapse anything ‘scale’ 0	$scale(s, 0) \rightarrow 0$
collapse 1 ‘scale’ anything	$scale(1, x) \rightarrow x$

Table A.1: Simplification rewrite rules for scale

sum with one summand collapses	$sum(x_0) \rightarrow x_0$
if a sum contains any sums, collapse them into one sum	$sum(x_0, ..., x_{n-1}, sum(x_n, ..., x_{m-1})) \rightarrow sum(x_0, ..., x_{m-1})$
if a sum contains two or more constants, collapse them into one constant	$sum(c_0, ...c_m, x_0, ..., x_n) \rightarrow sum(c_0 + c_1 + ... + c_m, x_0, ..., x_n)$
remove zeros from sum (note this doesn't change sorting of args)	$sum(0, x_0, ...x_n) \rightarrow sum(x_0, ..., x_n)$
combine identical terms in a sum	$sum(x_0, ..., x_{i_0}, x_{i_1}, ..., x_{i_m}, ..., x_n) \rightarrow sum(x_0, ..., scale(m, x_i), ..., x_n)$

Table A.2: Simplification rewrite rules for sum

product with one factor collapses	$prod(x_0) \rightarrow x_0$
remove product containing a zero	$prod(x_0, ..., x_n, 0) \rightarrow 0$
if a product contains any products, collapse them into one product	$prod(x_0, .., x_n, prod(y_0, ...y_m)) \rightarrow prod(x_0, ..., x_n, y_0, ..., y_m)$
if a product contains a zero, collapse it	$prod(x_0, ..., x_n, 0) \rightarrow 0$
if a product contains two or more constants, collapse them into one constant	$prod(c_0, ..., c_m, x_0, ..., x_n) \rightarrow prod(c_0 * ... * c_m, x_0, ..., x_n)$
if a product contains a constant 1 eliminate it	$prod(1, x_0, ..., x_n) \rightarrow prod(x_0, ..., x_n)$
distribute product inside sum	$prod(x_0, sum(x_1, x_2)) \rightarrow sum(prod(x_0, x_1), prod(x_0, x_2))$

Table A.3: Simplification rules for product

collapse $\Re(x+i_)$ into x	$\Re(x_0+ix_1) \to x_0$
collapse $\Im(_+ix)$ into x	$\Im(x_0+ix_1) \to x_1$
complex sum becomes combination of real sums	$sum((x_0+ix_1),(x_2+ix_3)) \to sum(x_0,x_1)+i\cdot sum(x_2,x_3)$
move Re Im inside projections and injections and sums	$\Re(proj(ss,z)) \to proj(ss,\Re(z))$
	$\Im(proj(ss,z)) \to proj(ss,\Im(z))$
	$\Re(inject(ss,z)) \to inject(ss,\Re(z))$
	$\Im(inject(ss,z)) \to inject(ss,\Im(z))$
	$\Re(sum(z_0,...,z_{n-1})) \to sum(\Re(z_0),...,\Re(z_{n-1}))$
	$\Im(sum(z_0,...,z_{n-1})) \to sum(\Im(z_0),...,\Im(z_{n-1}))$
pull scale outside $\Re$ and $\Im$	$\Re(scale(z)) \to scale(s,\Re(z))$
	$\Im(scale(z)) \to scale(s,\Im(z))$

Table A.4: Simplification rewrite rules for complex numbers

collapse FTs followed by their inverses	$\text{ft}(\text{ft}^{-1}(z_0)) \to z_0$
collapse inverse FTs followed by FTs	$\text{ft}^{-1}(\text{ft}(z_0)) \to z_0$
pull scale outside of the FT or its inverse.	$\text{ft}(scale(s,z_0 \to scale(s,\text{ft}(z_0))$
	$\text{ft}^{-1}(scale(s,z_0) \to scale(s,\text{ft}^{-1}(z_0))$

Table A.5: Simplification rules for Fourier transforms

turn complex products into real products	$(x_0+ix_1)*(x_2+ix_3) \to (x_0*x_1-x_1*x_2)+i(x_0*x_3-x_1*x_2)$
collapse dot with zero	$(0\cdot x) \to 0$
pull sum out of left dot	$sum(x_0,x_1)\cdot x_3 \to sum(x_0\cdot x_3,x_1\cdot x_3)$
pull scale out of left dot	$scale(s,x)\cdot y \to s*(x\cdot y)$
pull sum out of right dot	$x\cdot(y+z) \to sum(x\cdot y,x\cdot z)$
pull scale out of right dot	$x\cdot scale(s,y) \to scale(s,(x\cdot y))$
reverse vectors in dot product if any differentials are on the right	$x\cdot(fdy) \to (fdy)\cdot x$
differential only on left, now pop/push operators to the right but also simplify inside left, since this would otherwise be skipped	$(fdy)\cdot x \to dy\cdot adjfx$

Table A.6: Simplification rules for dot products

move constant scaling of SCZ into SCZ expression to enable SCZ fusion, and reduce computation downstream eliminate SCZ with constant expressions	$scz(f(r0_i,r1_i) \to c_i)[x_0,x_1] \to c$
eliminate SCZ that are the identity	$scz(f(r0_i) \to r0_i)[x_0] \to x_0$
remove unused inputs from SCZs expressions	$scz(f(r0_i,r2_i))[x_0,x_1,x_2] \to scz(f(r0_i,r1_i))[x_0,x_2]$
merge SCZ together with direct children SCZs,	$scz(f_1(r0_i,r1_i)[scz(f_2(r0_i,r1_i))[x,y],z] \to$ $scz(f_1(f_2(r0_i,r1_i)),r2_i)[x,y,z]$
push sums into SCZ's so they are exposed to further simplifications	$sum(x,scz(f(r0_i,r1_i))[y,z]) \to scz(r0_i+f(r1_i,r2_i)[x,y,z]$
simplify SCZ if some inputs are constants	$scz(f(r0_i,r1_i)[x,c] \to scz(f(r0_i,c))[x]$
simplify SCZ expression	$scz(expr)[inputs] \to$ let expr' = simplify expr in $scz(expr')[inputs]$

Table A.7: Simplification rewrite rules for sparse convolution zips

if no simplification was applied to this node, then simplify args	$fx, fin[sum,prod] \to f(\text{simplify } x)$

Table A.8: Simplification rewrite rules for inputs

Appendix B

Code Snippet: Fusing Dot Products for L-BFGS

The body of the C function described in Section 8.2.2 which computes two subtractions and four dot products in parallel using SIMD.

```
__m128  c, cn, g, gn;
__m128  s, y;
__m128d d_sum, g_sum, r_sum, a_sum;
__m128  mu;
__m128d mu1d, mu2d;

d_sum = _mm_set1_pd(0.0);
g_sum = _mm_set1_pd(0.0);
r_sum = _mm_set1_pd(0.0);
a_sum = _mm_set1_pd(0.0);

while ((size -= SIMD_SIZE) >= 0) {
  g  = _mm_load_ps(grad);
  gn = _mm_load_ps(grad_next);
  c  = _mm_load_ps(current);
  cn = _mm_load_ps(next);

  // \sigma = next - current
  s = _mm_sub_ps(cn,c);

  // y[k] = next_grad - grad
  y = _mm_sub_ps(gn,g);

  // dx_norm = \sigma[l] \dot \sigma[l]
  mu    = _mm_mul_ps(s, s);
  mu1d  = _mm_cvtps_pd(mu);
  mu2d  = _mm_cvtps_pd(_mm_shuffle_ps(mu, mu, _MM_SHUFFLE(1,0,3,2)));
  d_sum = _mm_add_pd(d_sum, mu1d);
  d_sum = _mm_add_pd(d_sum, mu2d);

  // grad_norm = gradient \dot gradient
  mu    = _mm_mul_ps(g, g);
  mu1d  = _mm_cvtps_pd(mu);

  mu2d  = _mm_cvtps_pd(_mm_shuffle_ps(mu, mu, _MM_SHUFFLE(1,0,3,2)));
  g_sum = _mm_add_pd(g_sum, mu1d);
```

```
    g_sum = _mm_add_pd(g_sum, mu2d);

    // \rho_i = (\sigma_i \dot y_i)
    mu    = _mm_mul_ps(s, y);
    mu1d  = _mm_cvtps_pd(mu);
    mu2d  = _mm_cvtps_pd(_mm_shuffle_ps(mu, mu, _MM_SHUFFLE(1,0,3,2)));
    r_sum = _mm_add_pd(r_sum, mu1d);
    r_sum = _mm_add_pd(r_sum, mu2d);


    // \a_i = \sigma_i \dot q (where q is grad on this round)
    mu    = _mm_mul_ps(s, gn);
    mu1d  = _mm_cvtps_pd(mu);
    mu2d  = _mm_cvtps_pd(_mm_shuffle_ps(mu, mu, _MM_SHUFFLE(1,0,3,2)));
    a_sum = _mm_add_pd(a_sum, mu1d);
    a_sum = _mm_add_pd(a_sum, mu2d);

    // store sigma and y
    _mm_store_ps(sigma, s);
    _mm_store_ps(y, y);

    sigma     += SIMD_SIZE;
    y         += SIMD_SIZE;
    grad      += SIMD_SIZE;
    grad_next += SIMD_SIZE;
    current   += SIMD_SIZE;
    next      += SIMD_SIZE;
  }

  // store sum vars
  _mm_store1_pd(dx, _mm_add_pd(d_sum,
                               _mm_shuffle_pd(d_sum, d_sum, _MM_SHUFFLE2(0,1))));
  _mm_store1_pd(gnorm, _mm_add_pd(g_sum,
                               _mm_shuffle_pd(g_sum, g_sum, _MM_SHUFFLE2(0,1))));
  _mm_store1_pd(rho,  _mm_add_pd(r_sum,
                               _mm_shuffle_pd(r_sum, r_sum, _MM_SHUFFLE2(0,1))));
  _mm_store1_pd(a_i,  _mm_add_pd(a_sum,
                               _mm_shuffle_pd(a_sum, a_sum, _MM_SHUFFLE2(0,1))));
```

Listing B.1: C code for computing the two subtracctions and four dot products with simdization and without leaving registers

Bibliography

[ACCM05] C. K. Anand, J. Carette, A. T. Curtis, and D. Miller. COG-PETS: Code Generation for Parameter Estimation in Time Series. *Maple Conference*, 2005.

[ACK+04] Christopher Kumar Anand, Jacques Carette, Wolfram Kahl, Cale Gibbard, and Ryan Lortie. Declarative assembler. SQRL Report 20, Software Quality Research Laboratory, McMaster University, October 2004. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[AJR06] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on numa architectures: solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *Proceedings of the 13th international conference on High Performance Computing*, HiPC'06, pages 338–352, Berlin, Heidelberg, 2006. Springer-Verlag.

[AK07a] Christopher Kumar Anand and Wolfram Kahl. A domain-specific language for the generation of optimized SIMD-parallel assembly code. SQRL Report 43, McMaster University, May 2007. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[AK07b] Christopher Kumar Anand and Wolfram Kahl. Multiloop: Efficient software pipelining for modern hardware. In *CASCON '07: Proc. 2007 Conference of the Center for Advanced Studies on Collaborative Research*, pages 260–263, New York, 2007. ACM.

[AK08] Christopher K. Anand and Wolfram Kahl. Synthesising and verifying multi-core parallelism in categories of nested code graphs. In Michael Alexander and William Gardner, editors, *Process Algebra for Parallel and Distributed Processing*. Chapman & Hall/CRC, 2008.

[Ber07] Berland. Automatic differentitation. `http://en.wikipedia.org/wiki/File:AutomaticDifferentiationNutshell.png`, January 2007.

[BTK06] Rainer Butchy, Jie Tao, and Wolfgang Karl. Automatic data locality optimization through self-optimization. In *Self-Organizing Systems*, pages 187–201. Springer Berlin / Heidelberg, 2006.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[CMP02] T. Corpetti, E. Mémin, and P. Pérez. Dense Estimation of Flud Flows. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 24(3), March 2002.

[Coc70] John Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):20–24, July 1970.

[Dai02] Yu-Hong Dai. Convergence properties of the bfgs algoritm. *SIAM J. on Optimization*, 13(3):693–701, August 2002.

[DB11] D. Domingo and L. Baily. Performance tuning guide: Optimizing subsystem throughput in red hat enterprise linux 6. `http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/main-cpu.html`, 2011.

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. second edition, Society for Industrial and Applied Mathematics, 2008.

[Hec00] A. Heck. *Introduction to Maple*. Springer-Verlag New York, Inc, 2000.

[HMB+10] D. J. Holland, D. M. Malioutov, A. Blake, A. J. Sederman, and L. F. Gladden. Reducing data acquisition times in phase-encoded velocity imaging using compressed sensing. *Journal of Magnetic Resonance*, 203:236–246, Apr 2010.

[HMH08] P. Héas, E. Mémin, and D. Heitz. Self-similar regularization of optic-flow for turbulent motion estimation. *The 1st International Workshop on Machine Learning for Vision-based Motion Analysis*, pages 1–12, 2008.

[IMB+10] B. Issa, R. J. Moore, R. W. Bowtell, P. N. Baker, I. R. Johnson, B. S. Worthington, and P. A. Gowland. Quantification of blood velocity and flow rates in the uterine vessels using echo planar imaging at 0.5 Tesla. *Journal of Magnetic Resonance Imaging*, 31:921–927, Apr 2010.

[KAC06] Wolfram Kahl, Christopher Kumar Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In Wendy MacCaull, Michael Winter, and Ivo Düntsch, editors, *RelMiCS 2005*, volume 3929 of *LNCS*, pages 147–160. Springer, 2006.

[Mar10] I. Marshall. Computational simulations and experimental studies of 3D phase-contrast imaging of fluid flow in carotid bifurcation geometries. *Journal of Magnetic Reson Imaging*, 31:928–934, Apr 2010.

[M.J72] Flynn M.J. Some computer organizations and their effectiveness. *IEEE Trans. on Comp.*, C-21:948–960, 1972.

[MN93] M. Monagan and W. M. Neuenschwander. GRADIENT: Algorithmic differentiation in Maple. *ISSAC '93 Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation*, 1993.

[Mos71] Joel Moses. Algebraic simplification: a guide for the perplexed. *Commun. ACM*, 14(8):527–537, August 1971.

[Noc80] Jorge Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35(151), pages 773–782, 1980.

[set10] Linux programmer's manual pthread_setaffinity_np(3). http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_setaffinity_np.3.html, September 2010.

[SPBL06] Raül Sirvent, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. Automatic grid workflow based on imperative programming languages. *Concurrency and Computation: Practice and Experience*, 18(10):1169–1186, 2006.

[Sto11] David R. Stoutemyer. Ten commandments for good default expression simplification. *Journal of Symbolic Computation*, 46(7):859 – 887, 2011. Special Issue in Honour of Keith Geddes on his 60th Birthday.

[TA77] A. N. Tikhonov and V. Y. Arsenin. *Solutions of Ill-Posed Problems*. Winston, 1977.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Prentice Hall, 3 edition, 2008.

[Toy05] Y. Toyama. Confluent term rewriting systems (invited talk). *Giesl, J. (ed.) RTA*, LNCS, 3467:1, 2005.

[War02] James C. Warner. Linux user's manual top(1). http://unixhelp.ed.ac.uk/CGI/man-cgi?top, September 2002.

[YKD11] A. YarkKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: Queueing and runtime for kernels, April 2011.

[ZSLW92] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Trans. Parallel Distrib. Syst.*, 3(5):540–554, 1992.