Imperial College of Science, Technology and Medicine
Department of Computing

An Active-Library Based Investigation into the Performance Optimisation of Linear Algebra and the Finite Element Method

Francis Prem Russell

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing and the Diploma of Imperial College, London, July 2011

Abstract

In this thesis, I explore an approach called "active libraries". These are libraries that take part in their own optimisation, enabling both high-performance code and the presentation of intuitive abstractions.

I investigate the use of active libraries in two domains. Firstly, dense and sparse linear algebra, particularly, the solution of linear systems of equations. Secondly, the specification and solution of finite element problems.

Extending my earlier (MEng) thesis work, I describe the modifications to my linear algebra library "Desola" required to perform sparse-matrix code generation. I show that optimisations easily applied in the dense case using code-transformation must be applied at a higher level of abstraction in the sparse case. I present performance results for sparse linear system solvers generated using Desola and compare against an implementation using the Intel Math Kernel Library. I also present improved dense linear-algebra performance results.

Next, I explore the active-library approach by developing a finite element library that captures runtime representations of basis functions, variational forms and sequences of operations between discretised operators and fields. Using captured representations of variational forms and basis functions, I demonstrate optimisations to cell-local integral assembly that this approach enables, and compare against the state of the art.

As part of my work on optimising local assembly, I extend the work of Hosangadi et al. on common sub-expression elimination and factorisation of polynomials. I improve the weight function presented by Hosangadi et al., increasing the number of factorisations found. I present an implementation of an optimised branch-and-bound algorithm inspired by reformulating the original matrix-covering problem as a maximal graph biclique search problem. I evaluate the algorithm's effectiveness on the expressions generated by our finite element solver.

Acknowledgements

I would like to express my greatest thanks and appreciation to:

- Paul Kelly, my supervisor, for his time, knowledge, patience, enthusiasm and endless discussions without which this work would never have been possible.

- Tony Field, my second supervisor, for the interesting conversations we've had.

- Graham Markall, for our many discussions about finite element assembly and other related issues.

- David Ham, for his help in understanding the numerical issues surrounding the finite element discretisation of the Navier-Stokes equations.

- William Knottenbelt and Peter Jimack, my examiners, for taking the time and effort to read my thesis and for their constructive comments and suggestions.

- Tristan Allwood, Marc Hull and Matthew Sackman for their support with special thanks to Tristan and Marc for enduring countless conversations about the many research obstacles I encountered.

- My family, for always having faith in my abilities.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Thesis Statement

This thesis argues that active libraries, libraries that self-optimise, are an essential requirement for obtaining both performance and abstraction in computational science applications.

1.2 Motivation and Objectives

Abstractions offer the ability to model a domain while hiding other concerns from the client using them. They permit more comprehensible code, and separation of domain concerns from other issues. However, this often comes at the cost of performance. In mathematical and computational science applications, speed is considered essential. Therefore, scientific code is often written in a performance-oriented manner, with clean abstractions a secondary concern.

Active libraries are are a technique whereby a library can take responsibility for the optimisation of its own code. By shifting responsibility for performance to the library, they enable library clients to write code without having to concern themselves with performance. This enables the library writer to focus on how to make the abstractions fast.

Active libraries would appear to be the perfect match for computational science applications.

They enable both the domain expert and computer scientist to focus on the problems they know how to solve without having to become an expert in each other's field.

This thesis is an investigation of active libraries in the context of computational science. First, we consider sparse linear algebra as an extension to previous work. Next, we examine active libraries as a tool to specify and optimise a solver for partial differential equations, using the finite element method.

Our objectives in this thesis are to:

- Extend previous work on dense linear algebra into the sparse domain and demonstrate how active libraries can apply optimisations to this domain that are beyond the capabilities of conventional compilers.
- Apply active libraries to a more complex domain, specifically that of the finite element method.
- Demonstrate how active libraries can be used to raise the level of abstraction in this domain, and develop our expression capture techniques beyond those that we applied to linear algebra.
- Develop optimisations for the finite element method based on domain-specific knowledge obtained through our expression capture techniques.

In aiming to achieve these, we hope to further work both on the design and implementation of active libraries and on optimisation techniques for the finite element method.

1.3 Contributions

We make the following contributions with this work:

- We present the extension of our delayed-evaluation runtime code-generation library "Desola" to sparse linear algebra.

- We show that some of the important optimisations applied to our dense linear iterative solvers are also applicable to our sparse ones. We also show that these optimisations are beyond the automatic optimisation capabilities of modern compilers and require a generative approach in order to be implemented effectively.

- We present new performance results for sparse linear iterative solvers implemented using the Intel Math Kernel Library against our extended DESOLA implementation.

- We present the design and implementation of a C++ finite element library that performs expression capture of multiple aspects of the finite element method.

- We show that expression capture of certain aspects of the finite element method enables analyses and optimisations not possible in other finite element implementations. We present an optimisation framework for the evaluation of local assembly matrices and compare against the state of the art.

- We extend the work of Hosangadi et al. [1] for common sub-expression elimination of polynomials. We present improvements to the original factorisation weighting function that further reduce the operation count of our factorised expressions. We reformulate the primary problem of finding a maximally weighted matrix covering as a graph biclique search problem, and present a branch-and-bound algorithm optimised for the presented weight function through insights into the graph structure.

1.4 Statement of Originality

I hereby declare that this document is the result of my own work as is the work it presents except where otherwise stated.

This thesis contains work on sparse extensions to DESOLA, an active linear algebra library that I developed during my MEng thesis. This work was first published as a workshop paper [2] and then revised for a special issue [3] of "Science of Computer Programming" published by Elsevier.

The sparse extensions detailed in this thesis (Section 3.8) are new work, however, content and background is incorporated from the journal paper and MEng thesis in order to provide sufficient context for both the sparse extensions and the relationship of the work to Excafé, which extends the ideas developed in Desola.

1.5 Publications

F. P. Russell, M. R. Mellor, P. H. J. Kelly, and O. Beckmann, "DESOLA: An active linear algebra library using delayed evaluation and runtime code generation," *Science of Computer Programming*, vol. 76, no. 4, pp. 227–242, 2011. Special issue on library-centric software design (LCSD 2006).

This paper is the journal version of the original Desola workshop paper [2]. It describes work I developed during my MEng thesis, with new dense linear algebra results that are also presented in this thesis.

Chapter 2

Background

In this chapter, I survey literature relevant to this research. I cover work relevant to optimising domain-specific abstractions, techniques for supporting adaptation in languages and libraries, and models and techniques for code optimisation and generation. I provide a brief mathematical description of the finite element method and review various domain-specific languages and libraries used for the specification and solution of finite element problems.

This thesis incorporates material on the active library Desola from my journal paper [3] and Masters thesis to provide context for the sparse extensions detailed in Chapter 3 in Section 3.8. Some of the code-generation and optimisation background originates from this work.

2.1 Optimising Domain-Specific Abstractions

Optimising domain-specific abstractions provides the key to achieving both high performance and comprehensible code. C++ provides mechanisms such as classes and operator overloading that make it relatively easy for developers to define their own abstractions. Significant research has been undertaken into techniques for optimising these abstractions, one of the most well-known being template metaprogramming.

2.1.1 Expression Templates and Template Metaprogramming

Expression Templates [4] are a C++ technique for passing expressions as function arguments. Conventional C-style code would involve passing the expression as a callback using function pointers. Expression templates make it possible for expressions to be passed as function arguments, and inlined into the function body. This results in less overhead, more convenient code and enables the developer to control the implementation of the code that evaluates the expression.

Expression Templates work by parsing expressions at compile time and storing the nested template arguments as an "expression type". Thus, rather than the expression `1.0+x` having the type `float`, it might have the type:

```
DExpr< DBinExprOp< DExpr<DExprLiteral>, DExpr<DExprIdentity>, DExprAdd > >
```

Expression templates give developers control over code generation for expressions. This is particularly useful for numerical applications where the naïve implementation of vector and matrix arithmetic produces code that allocates and deallocates a number of temporary arrays and uses more loops than necessary.

The Matrix Template Library [5] (MTL) is one C++ library that has successfully used expression templates and other template metaprogramming techniques [6] for producing efficient C++ code. MTL has the aim of providing the library user with appropriate C++ abstractions and high performance.

Algorithms in MTL are expressed independently of the data storage formats they may work on, using iterators to traverse the containers involved. MTL relies on the optimising abilities of the compiler to be able to remove these levels of abstraction.

MTL algorithms are built on top of a library called BLAIS [7] (Basic Linear Algebra Instruction Set) which is layered on top of FAST, the Fixed Algorithm Size Template Library. BLAIS provides similar functionality to the levels 1, 2 & 3 of BLAS (Basic Linear Algebra Subpro-

```
// STL
int len = 4;
int* x = new int[len];
int* y = new int[len];
fill(x, x+len, 1);
fill(y, y+len, 3);
std::transform(x, x+len, y, y, plus<int>());
```

(a) STL

```
// FAST
const int LEN = 4;
int* x = new int[LEN];
int* y = new int[LEN]:
fill(x, x+LEN, 1);
fill(y, y+LEN, 3);
fast::transform(x, cnt<LEN>(), y, y, plus<int>());
```

(b) FAST

Figure 2.1: Implementations of vector addition using STL and FAST. The FAST implementation requires that the vector sizes are compile time constants.

grams). FAST is effectively an implementation of the C++ Standard Template Library for computations whose size is known at compile time.

We compare STL and FAST implementations of a vector addition in Figure 2.1. Both `transform` instantiations iterate over the arrays `x` and `y`, pairwise summing them into array `y`. However, the STL instantiation of this code will involve a loop whereas the FAST implementation uses recursive templates to perform loop unrolling, resulting in inlined code.

Using template techniques, MTL is also able to provide other performance optimisations including:

- Static Polymorphism
- Lightweight Object Optimisation
- Automatic Unrolling
- Algorithmic Blocking

2.1.2 ROSE

ROSE [8] is a framework for building source-to-source translators for the high-level optimisation of scientific applications. ROSE provides the ability to recognise library-defined abstractions and leverage their semantics to provide compile-time optimisation.

User-defined abstractions may not be optimised by a conventional compiler because it does not perform sufficient global optimisation, or cannot infer the high-level semantics of user-defined abstractions in acceptable time bounds. ROSE attempts to solve this problem by providing additional semantic information through annotations. Using these annotations and transformations, ROSE can optimise arbitrary abstractions.

The ROSE front-end is responsible for parsing C++ programs. It accepts C++ source files and annotated library headers and produces an object-oriented abstract syntax tree (AST) which represents both sets of files. The AST is annotated with type information, and may additionally be annotated with information supplied by the programmer using pragmas, comments and separate annotation files. The mid-end is responsible for restructuring the AST and performing performance improving program transformations. The mid-end provides three types of operations:

AST processing These operations provide mechanisms for traversal of the AST and enable attributes to be computed for and attached to each AST node. Attributes can then be used in subsequent optimisations. Attributes may be also be *inherited* or *synthesised*, in which case they are passed down or up the AST, respectively For example, loop nesting is an inherited attribute. Annotations can be used by transformations to decide whether a particular restructuring operation can be applied safely.

Query operators These perform read-only operations on the AST and are built on top of the AST processing operators. ROSE provides a number of pre-defined queries. The query operators abstract away some of the details of the AST traversal, and can be composed to form more complex queries.

Transformation operators Transformation operators consist of two parts. Firstly, a pre-condition based on the AST annotations which holds when the transformation can be applied safely. Secondly, a sequence of AST restructuring operations.

Lastly, the ROSE back-end is responsible for unparsing the AST and generating C++ source code.

ROSE's annotation language makes it possible for the programmer to declare that certain abstractions satisfy the extended requirements for certain predefined compiler optimisations. As a result ROSE can apply a number of standard compiler transformations more effectively. For example, by annotating a user-defined array class with the declaration that it has Fortran array semantics, optimisations such as loop blocking, fusion, fission and interchange can be applied to loops with statements accessing the user-defined type.

2.1.3 Broadway Compiler

Guyer and Lin describe the Broadway [9] compiler, a source-to-source translator for C designed to support domain-specific optimisations. Domain-specific optimisation information is provided to the compiler through files written using a lightweight annotation language.

Broadway's annotation language conveys four kinds of information to the compiler: dependence information about the library interfaces, domain-specific program analysis problems, transformations that apply domain-specific optimisations and compile-time messages which are emitted based on analysis results.

Broadway's domain-specific analyses permit *properties* to be associated program values. The annotation writer defines how each library routine affects these properties. Broadway uses data flow analysis to determine how properties are associated with values at different points in the program. In a linear algebra library, for example, one property of matrices could be the form they are in (e.g. triangular or tridiagonal).

The annotation language enables the specification of library behaviour in terms of the abstract

properties. This permits special-purpose routines to be chosen when certain properties are present on objects. For example, in a linear algebra library this could be used to choose routines optimised for a particular matrix structure.

The overall Broadway optimisation model is as follows:

1. A library expert designs a set of domain-specific optimisations and encodes them using the annotation language. To do this they must identify the properties and property values relevant to the library, specify the behaviour of each routine in terms of these abstract properties and specify code transformations predicated on property values of the arguments.

2. The application programmer obtains the annotation file and passes it to Broadway.

3. During compilation, Broadway consults the annotations and determines how to manipulate library calls. To do this it solves the problem of determining what property values objects possess in the application and evaluates predicates at each function call site to determine which transformations to apply.

Guyer and Lin describe [9] how Broadway can be used to optimise a parallel linear algebra library at multiple application layers. At the highest layer, Broadway is used to perform algebraic simplification. At the middle layer, where data distribution has been made explicit, Broadway is used to eliminate empty views, unnecessary copies and make compile-time selection of specialised routines. At the lowest level, optimisations of the use of MPI have been identified, but not yet implemented.

2.2 Programming for Adaptation

In this section, I will look at programming models designed to support adaptation to different environments. Sequoia [10] uses task decomposition to facilitate mapping regular computations onto a memory hierarchy. The Themis [11] proposal suggests a run-time system which can adapt communication and scheduling in response to varying resources and component context.

2.2.1 Sequoia

Sequoia [10] is a programming language designed to facilitate the development of memory hierarchy aware parallel programs. Sequoia aims to enable such programs to remain portable across modern machines with different memory hierarchy configurations.

Sequoia introduces the notion of hierarchical memory directly into its programming model. Sequoia programs run on machines abstracted as trees of distinct memory modules. Sequoia defines *tasks* as abstractions to represent self-contained units of computation that include descriptions of communication and working sets. The programmer must define ways to decompose tasks so they can be mapped onto the memory hierarchy. Data movement between all levels of the hierarchy is explicit and computation is only allowed to occur at the leaf nodes of the hierarchy. Sequoia leaf tasks may be implemented directly in Sequoia, but may also wrap kernels written in traditional languages such as Fortran or C.

Sequoia borrows from the idea of space-limited procedures [12], proposed to encourage hierarchy-aware, parallel divide-and-conquer programs. Space-limited procedures require each function in a call chain to accept arguments occupying significantly less storage than the calling function. Sequoia tasks generalise and implement this concept to express communication and parallelism. Sequoia code does not make explicit references to machine hierarchy levels and remains oblivious to the mechanisms used to move data between memory modules.

Tasks in Sequoia allow the expression of:

Explicit Communication and Locality Communication of data is expressed by calling tasks, and is the only means of expressing data movement in Sequoia. *Virtual levels* can be used in the memory hierarchy which do not correspond to a physical machine memory. This enables modelling of communication mechanisms such as cluster interconnects.

Isolation and Parallelism Tasks operate entirely within their own private address space and have no way of communicating with other tasks other than by calling subtasks, and returning to their parent task.

Task Decomposition Sequoia provides *array blocking* and *task mapping* constructs to describe portable task decomposition.

Algorithmic Variants The programmer can specify multiple implementations of a task and specify which one to call based on context.

Parameterisation Tasks use parameterisation to preserve independence from machine constraints. Values are chosen to match the hierarchy level of the target machine.

To enable the mapping of task hierarchy onto the memory hierarchy, Sequoia allows the programmer to provide a *task mapping specification* that is created by the programmer on a per-machine basis. The task mapping specification defines how to map a task hierarchy onto the memory hierarchy and also serves as the mechanism whereby the programmer can provide optimisation and tuning directives.

2.2.2 Themis

Themis [11] is a proposed programming model and run-time library intended to support cross-component optimisation through the explicit manipulation of the computation's iteration space at runtime, Themis has not been implemented although many of the ideas have been investigated in prototype form using tools such as TaskGraph [13].

The main idea behind Themis is to combine software components with metadata describing data placement and component dependence. This metadata should permit the adaptation of software to:

Heterogeneous and varying resources There is the expectation of high performance computing resources to be heterogeneous collections of symmetric multiprocessing (SMP) clusters, linked by fast heterogeneous networks. It is necessary for components to be able to adapt themselves to these configurations. Importantly, these may vary from run to run.

The context in which components are used The data placement of components may vary, as well as the schedule for the production and consumption of a component's operands and results. Components may contend with each other for resources and may be capable of multiple levels of parallelism with different forms of communication.

The adaptive and irregular nature of problems In irregular and adaptive applications, communication and computation are focussed in specific regions of interest which may change with time.

Component metadata in THEMIS consists of two parts. Firstly, the *Component Composition Graph* data structure that represents the large-grain inter-component control graph. Secondly, the *Component Dependence Summaries* which describe each component's internal iteration space, and functions mapping points in the iteration space to the memory addresses it may use and define. In THEMIS, each component has parameters and properties. Properties of some component, P, are:

IterationSpace The n-dimensional integer space in which iterations of P's execution are enumerated.

Uses For each of the operands of P, this defines a mapping from each point in the iteration space to the set of indices which might be read by that iteration. In simple cases, these will be affine functions.

Defines For each of the results of P, defines a mapping from the iteration space to any items that might be written.

Parameters of a component are:

IterationDomain Defines the iterations that should be executed. These are composed of a set of non-intersecting *IterationRegions* each of which is a polytope contained in *IterationSpace*.

Operands A set of indexed data collections that a component may read from.

Results A set of indexed data collections that a component may write to.

The component metadata in THEMIS would allow it to solve various cross component optimisation problems. For example, given a data distribution D which specifies the subsections of an array A accessed by component P, it is possible to calculate the required placement of P's other operands and/or results. If the *Uses* or *Defines* mappings referring to A are invertible, they can be used to find the relevant iterations that access the subsections in D. Using the *Uses* and *Defines* mappings forward, the other data accessed by P can be found.

THEMIS Component metadata could also be used to derive communication plans, or perform cross-component loop fusion. The runtime nature of the system would also allow it to make decisions about how to apply these optimisations more effectively. As THEMIS would maintain extra dependence information, it would enable the use of different techniques useful for large scale simulations such as checkpointing combined with recompute-on-demand, or propagation of data dependencies backwards through a pipeline.

2.3 Library Adaptation through Empirical Techniques

In this section I will look at libraries that optimise their performance through empirical techniques. These are important examples of how to achieve performance competitive with hand-tuned libraries without having to possess expert knowledge for every architecture targeted. SPIRAL [14] is particularly interesting because of its ability to use domain-specific knowledge to perform a broad range of transformations and to target specific instruction sets. The X [15] language is not a programming model per se, but rather a way to represent multiple program versions to facilitate adaptation.

2.3.1 X Language

Donadio et al. [15] describe the *X* language. The X language represents parameterised program versions in a compact way using annotations which can be used directly by the programmer or by an empirical search tool.

Donadio et al. observe the increasing complexity of processors and architectures and the increasing difficultly of identifying optimal code sequences for them. They note that empirical search techniques can be an effective tool for searching the space of possible program versions. They describe the following features that they believe are necessary for compact representation of multiple code versions:

Elementary Transformations These are transformations that cannot be easily recast into simpler transformations. These usually target compound statements and manipulate the order of execution and control structure of the statements. Statement transformations include reordering, replication and deletion. Loop transformations include unrolling, interchange, strip-mining and fusion. Elementary transformations may require parameters.

Transformation Composition As the best version of a statement is usually the result of the application of several transformations, it should be possible to apply multiple transformations to a statement. Additionally, it should possible to perform *conditional* composition in which a condition is used to select whether a transformation should be applied. For example, unrolling an inner loop only when the size of a strip is less than a certain value.

Procedural Abstraction It should be possible to encapsulate new transformations in order to avoid duplication.

Definition mechanism for new transformations This mechanism should make it possible for the language user to define new transformations that cannot be expressed as a composition of elementary transformations. This allows the library user to define application-dependent transforms that use domain-specific knowledge of the computation semantics. Such a mechanism is most easily implemented using *transformation rules* which consist

of a code template before and after transformation. However, as this may not always be sufficient, it should also be possible to express transformations in a conventional programming language with a well defined interface to the source language.

Statement naming mechanism Sequences of transformations often require the application of transformations to one of the components of the transformed code. It is necessary to be able name components and subcomponents of statements to compose these transformations.

The X language uses C pragmas to name loops or portions of code and also to specify the transformations to apply. Syntax to name code is as follows:

```
#pragma xlang name <id> { ... }
```

Syntax to specify transformations is as follows:

```
#pragma xlang transform keyword <list-input-par> <list-output-par>
```

Figure 2.2 shows the code for a matrix-vector annotated with X directives, before and after transformation. The transformation *strip mines* loop `l1` with a tile size of 4, naming the new loop `l3` and the remainder loop `l1rem`.

Donadio et al. present results that show a matrix multiply written using X and tuned using an automatic mechanism can achieve performance superior to ATLAS routines on an Intel Itanium when using a custom memory copy routines. However, both the ATLAS and X language code variants are still outperformed by the Intel Math Kernel Library.

2.3.2 ATLAS

The Automatically Tuned Linear Algebra Software library [16] (ATLAS) is part of a research effort focussing on applying empirical techniques to provide portable performance. ATLAS uses

```
#pragma xlang name l1
for(i=0; i<N; i++) {
  #pragma xlang name l2
  for(j=0; j<M; j++) {
    c[i] = a[i][j] * b[j];
  }
}
#pragma xlang transform stripmine l1 4 l3 l1rem
```

(a) Before transformation

```
#pragma xlang name l1
for(i=0; i<(N/4)*4; i+=4) {
  #pragma xlang name l3
  for (ii=i; ii<i+4; ii++) {
    #pragma xlang name l2
    for (j=0; j<M; j++) {
      c[ii] = a[ii][j] * b[j];
    }
  }
}

#pragma xlang name l1rem
for(i=(N/4)*4; i<N; i++) {
  #pragma xlang name l2
  for(j=0; j<M; j++) {
    c[i] = a[i][j] * b[j];
  }
}
```

(b) After transformation

Figure 2.2: C code to perform a matrix-vector multiply, annotated with X pragmas. Code is shown before and after the application of the X directives.

a paradigm the authors call "Automated Empirical Optimisation of Software" (AEOS). The fundamental idea behind AEOS is that a software package provides many ways of performing the same operation and can use empirical timings to choose the best method for a given architecture. This avoids the painstaking effort required to hand optimise routines for a given architecture, which given the pace of hardware evolution, may be untenable in the long run. The ATLAS library uses these techniques to provide portably efficient routines from BLAS (Basic Linear Algebra Subprograms) and LAPACK.

A library supporting AEOS methodologies must satisfy some basic requirements:

Isolation of performance-critical routines The performance-critical sections of code need to be identified and separated into subroutines.

A method of adapting to differing environments AEOS depends on iteratively trying a number of different implementations of performance-critical routines so they must provide a way of instantiating themselves with a wide range of optimisations. This might be done by having fixed code with varying parameters or with a highly parameterised code generator.

Robust, context-sensitive timers Accurate timings are needed if the best code is to be selected. Importantly, timings need to be robust if they are carried out on a heavily loaded machine since users may not be able to guarantee they are the only user. Timings also need to reflect the context in which a routine will be called. For example, the data a routine needs may or may not already be present in the cache. This may require cache flushing or pre-loading to simulate.

Appropriate search heuristic If the search space for possible implementations is large, a heuristic is required that will prune the search tree as rapidly as possible.

ATLAS, using the AEOS methodology, makes a couple of assumptions in order to perform well:

Adequate ANSI C compiler ATLAS is written in ANSI C (with the exception of Fortran77 wrappers). ATLAS does not require an excellent compiler because it performs many of the

optimisations usually performed by compilers. However, overly aggressive compilers may convert optimal code into suboptimal code. On the other hand, compilers that cannot effectively use the underlying instruction set architecture (ISA) may produce poor results as well.

Hierarchical memory ATLAS assumes a memory hierarchy. Best results are obtained when both registers and at least an L1 cache are present.

ATLAS demonstrates that empirical optimisation techniques can be used to generate code with significantly better performance than obtained by compiler optimisation alone, and that auto-generated code can compete with hand-tuned implementations.

2.3.3 SPIRAL

SPIRAL [14] is a code generator for digital signal processing (DSP) algorithms. It uses domain-specific knowledge about the structure of the signal transformation algorithms in order to implement feedback based optimisation. SPIRAL can generate code for a domain encompassing a large number of mathematically complex algorithms. It encapsulates mathematical knowledge about the domain using a concise declarative representation suitable for computer exploration and optimisation. It also uses dynamic programming and machine learning techniques for its algorithm selection and optimisation.

The SPIRAL architecture consists of three levels:

Algorithm This level handles the breakdown and manipulation of the requested transform into a representation called SPL (Signal Processing Language).

Implementation This level takes the SPL formula, converts into Fortran or C, and performs various standard compiler optimisations.

Evaluation This level is responsible for compiling code into executables and benchmarking their performance. The evaluation provides performance information to a search mech-

anism which controls actions taken at both the algorithm and implementation levels to try to find the best implementation.

SPIRAL represents DSP transformations using matrix-vector multiplies. If x and y are input and output vectors of signal samples of length n and M is an $n \times n$ transformation matrix, the transformation can be expressed as $y = Mx$. Transforms in SPIRAL are represented by parameterised classes of matrices. For example, most transforms exist for all input sizes and will take the size value n as a parameter. In SPIRAL, the discrete Fourier transform is represented as follows:

$$\mathbf{DFT}_n = [\omega_n^{kl}]_{0 \leq k,l \leq n}, \quad \omega_n = e^{-2\pi i/n}, \quad i = \sqrt{-1}$$

As well as representing transforms as matrices, SPIRAL can also define transforms recursively or iteratively. At the time of writing of [14], SPIRAL contains 36 transforms, some of which are variants of each other.

SPIRAL uses SPL to generate and manipulate DSP algorithms. The motivation behind SPL is to identify the structure of matrix factors and make use of them. SPL is intended to allow the expression of the product of structured sparse matrices using a small set of constructs and symbols.

SPL contains a number of constructs:

Generic matrices SPL provides constructs to represent generic matrices such as diagonal or permutation matrices.

Symbols Frequently occurring classes of matrices are represented using parameterised symbols. For example, the $n \times n$ identity matrix, the zero matrix and the 2×2 rotation matrix are all represented by symbols.

Transforms These have already been described. The $\mathbf{DFT}_n$ transform is one example. Transforms differ from symbols in the important aspect that they cannot be translated into

code.

Matrix constructs These are used to form structured matrices from SPL matrices. Examples include the matrix product and matrix sum. One of the most important constructs is the Kronecker product which has been recognised as important for describing and manipulating DFT algorithms.

Conversion to real data format Complex transforms are usually implemented with real numbers. Constructs exist to convert complex arithmetic into real arithmetic, the most popular being *interleaved complex format* which represents a vector of complex numbers with alternating real and imaginary parts.

SPIRAL uses two types of rules to manipulate SPL formulae, *Breakdown* and *Manipulation*. *Breakdown rules* are used to decompose a transform into a product of structured sparse matrices that often contain other smaller transforms. *Terminal breakdown rules* form the base case, usually decomposing transforms of size 2 into formulae that contain no transforms.

Manipulation rules are matrix expressions where both sides are SPL formulae which contain no transforms. These are used to manipulate SPL formulae that have already been completely expanded. For example, one manipulation rule handles the translation of the operator used to convert complex to real arithmetic.

The *implementation level* is responsible for taking the expanded SPL formulae and converting them into code. The SPL implementation allows tags to be associated with formulae that can instruct the compiler to make specific code generation options such as performing loop unrolling. *Templates* are used to translate SPL code into C or Fortran code. Templates consist of a parameterised SPL formula construct, a set of preconditions on the formula's parameters and a C-like code fragment. Templates allow experimentation with the way formulae are mapped into code and enable the SPL code compiler to generate special instruction types such as vector instructions.

The *evaluation level* handles code compilation and performance analysis. The optimisations performed at this stage are standard compiler optimisations such as constant propagation and

array scalarisation. Benchmarking the result is done through the compilation and timing of the compiled code for a large number of iterations. This information provides feedback to a search/learning mechanism which can modify the previously generated formulae.

2.4 Code Optimisation Techniques and Models

Here I cover some of the frameworks and techniques used to optimise code. A description of a large number of standard compiler transformations is described by Bacon et al. [17]. The Polytope Model [18] seems to be particularly relevant to my work, as it provides useful mechanisms for reasoning about iteration spaces. I also cover techniques useful for runtime optimisation. The loop fusion and array contraction transformations are particularly important for optimising linear algebra.

2.4.1 Loop Fusion

Loop fusion [17] is a transformation that rewrites multiple loops and a single loop. It can improve performance by:

- Reducing loop overhead.
- Increasing instruction parallelism.
- Improving register, vector, data cache, translation lookaside buffer (TLB) or page locality, if both loops use the same data.

Loop fusion requires that the loops being fused have the same bounds. If they do not, they can sometimes be made to match by loop peeling, or introducing a conditional. Two loops with the same bounds may be fused so long as there exists no statement S_1 in the first loop and S_2 in the second loop such that S_1 has a dependence on S_2.

It is also possible for loop fusion to decrease performance. This can occur if the loop instructions can no longer fit into the instruction cache or register pressure increases to the extent that values must be "spilled" into main memory.

2.4.2 Array Contraction

Array contraction [17] is a transformation intended to optimise the memory access of a program. It allows the dimensionality of arrays to be reduced (e.g. by replacing an array with a single scalar-valued variable). It cam decrease the memory taken up by compiler generated temporaries and the number of cache lines referenced.

It is expected that this technique will be useful for removing temporary vectors and matrices created during code generation. For details of other memory access transformations, consult Bacon et al. [17].

If the iteration variable p of some loop within a loop nest is being used to index the k^{th} dimension of an array x, then dimension k may be removed from x if:

- Loop p is not in parallel.
- All distance vectors involving x have their distance for iteration variable of p equal to 0 (i.e. values do not propagate between different iterations of p via x).
- Array x is not used subsequent to the loop.

Loop transformations such as fusion and interchange are often used to facilitate array contraction.

2.4.3 Polytope Model

The Polytope model [18] is a mathematical framework for representing loop nests and loop nest transformations. The Polytope model defines geometric representations of iteration spaces, statement orderings and array accesses.

Transformations take place within *Static Control Parts* (SCoPs) which are maximal sets of consecutive statements containing no *while* loops. Within a SCoP, all loop bounds are affine and may only depend on symbolic constants and surrounding loop counters. These symbolic values are called *global parameters* of the SCoP.

The Polytope model identifies three components for representing loop nests:

Iteration Domains These are geometric representation of the bounds and strides of a loop. In the Polytope model, these are represented by convex polyhedra, which in turn can be represented by a matrix of inequalities. Each statement in the SCoP has has an associated iteration domain. By separating the iteration domain from the statement schedule, it is possible to isolate the effect of certain transformations. For example, strip mining and loop unrolling modify the iteration domain but do not affect the execution order of statements.

Affine Schedules These characterise the execution order of each statement instance. They map statement instances to multidimensional *time stamp* vectors which are ordered lexicographically. The affine schedule is stored as a matrix, which represents an affine function of loop index variables and possible parameters that generates the time-stamp vector.

The affine schedule can be decomposed in three sub-matrices. Firstly, a square *iteration ordering matrix* that orders the statements w.r.t. the surrounding loop counters. Secondly, a *statement ordering vector* that represents the ordering of all statements executed during the same iteration. Lastly, a *parameterisation matrix* that enables statements to be offset by constant or parametric amounts.

Array Access Functions These map points in the iteration domain to the array elements accessed at that point. Array access functions allow the isolation of the effect of data access transformations. For example, array privatisation does not affect either the iteration domain or statement schedule.

Array access functions are stored as a matrices that represent an affine function of loop indices, local variables and global parameters. Each row of an access function matrix

corresponds to an index used to access the associated array.

Each program statement is mapped to two sets which represent the accesses to array locations written and read by that statement. Each set contains tuples of the form (A, f) where f is the access function used to access an element of array A.

Transformations in the Polytope model correspond to recomputing the matrices of certain statements. This provides advantages over other approaches where code complexity increases with each transformation. One exception to this are transformations that add new loops or local variables, which increases the size of some matrices, however, this does not make the representation less generic.

Cohen et al. [19] describe the advantages of the Polytope model for composing program transformations. The separation of different representations allows data and control transformations to commute in general. Many transformations in the Polytope model only have the effect of modifying the matrix parameters themselves even though they may correspond to a sequence of program transformations, making it simpler to search for transformations. It is possible to characterise the exact set of all schedule, domain and access matrices associated with legal transformation sequences, so invalid transformations can be filtered quickly.

2.4.4 Runtime Data and Computation Reordering

Making efficient use of the memory hierarchy is an essential component of performance optimisation. For an algorithm to achieve high performance, it should be ensured that the processor spends as little time as possible waiting on data from memory. Techniques such as loop blocking and prefetching can be used to try to place data as close to the processor in the memory hierarchy as possible, hiding the widening gap between CPU and memory speed. These techniques work well for regular applications, but less effectively for irregular ones. Irregular applications possess poor temporal and spatial locality because they do not access data in memory with small, constant strides.

Consecutive Packing

Consecutive packing [20], also known as *first-touch data reordering*, is a data layout reordering transformation in which data is moved into adjacent locations in the order that they will first be accessed. The result is a data layout with improved spatial locality. Consecutive packing traverses an array only once, giving it a minimal time overhead. If objects are accessed at most once, consecutive packing gives optimal cache line utilisation. With repeated accesses, the problem becomes NP-complete, reducing to the G-partition problem [21].

Locality Grouping

Dynamic applications such as molecular dynamics possess locality derived from the physical model in which a particle *only interacts with its neighbours*. Given a set of objects and their interactions, *locality grouping* [20] clusters interactions involving each object in the list. Locality grouping can be done with minimal runtime overhead. One pass collects a histogram of the interactions, and another produces a sorted interaction list.

Bucket Tiling

Bucket tiling [22] is a technique to localise non-affine array references. In a bucket sort, values are first sorted into buckets, then sorted within each bucket. Bucket tiling makes use of the first step, called *bucketisation*. Bucket tiling cannot be carried out if a computation contains more than one distinct non-affine reference.

Bucket tiling consists of three tasks:

Permutation generation This task generates a new iteration order. This involves the bucketisation of a function f involved in indexing an array. The function f will be a function of one or more of the loop index variables, but need not be the array indexing expression. It might, for example, be the non-affine component of the array indexing expression.

Iterations of all loops which have their induction variables referenced by f will have their iterations mapped into a fixed number of buckets. The permutations applied to each affected loop i are stored in a permutation function σ_i. Given a bucket b and the number of times that bucket has been visited k, $\sigma_i(b, k)$ gives the value of the iteration variable of loop i.

Loop Regeneration Loop regeneration involves generating loops that iterate through the iterations assigned to each bucket. Using a mechanism similar to loop coalescing, it is possible to use two loops to iterate over the values in all buckets regardless of the number of loops in the original code. Using the σ functions computed in the previous step, it is possible to recover the original induction variables.

Data Remapping After loop regeneration, the locality of affine accesses to other arrays may have been reduced. *Data remapping* solves this problem for certain classes of array accesses. Every reference to some array a is replaced by an access to a new array $a'[b, k]$ where b and k are the bucket index and number of times the bucket has been visited, respectively.

If the loop nest consumes values from a it is necessary to perform a gather operation to populate a' from a. Similarly, if values from a are used after the loop nest, it is necessary to perform a scatter operation to form a from a'. Additionally, if a' is larger than a, the scatter operation will also involve a reduction as a' contains only partial computations. Clearly this also requires that the computation of a can be expressed as a reduction.

We consider the following simple pseudocode segment that involves a non-affine access to array A:

```
do i = 1 to N
  D[i] = C[i] + A[f(i)]
```

The loop regeneration step will produce the following code:

```
do b = 1 to numbuckets
  do k = 1 to bucketsize(b)
    i = σ_i(b, k)
    D[i] = C[i] + A[f(i)]
```

The locality of accesses to `A` have been improved but the locality of accesses to `C` and `D` have been reduced since values of `i` are no longer consecutive. Applying data remapping to array C, we would get:

```
do b = 1 to numbuckets
  do k = 1 to bucketsize(b)
    C'[b,k] = C[σ_i(b,k)]

do b = 1 to numbuckets
  do k = 1 to bucketsize(b)
    i = σ_i(b, k)
    D[i] = C'[b,k] + A[f(i)]
```

The accesses to `C'[b,k]` possess improved data locality compared to `C[i]` but the overhead is incurred of computing `C'`. The runtime overhead of both data remapping and the calculation of the σ functions must be considered when applying these transformations.

Sparse Tiling

Sparse tiling [23] is an algorithm for tiling sparse matrix computations at runtime, improving data locality. Such computations cannot be tiled at compile-time due to the existence of non-affine loop bounds and indirect memory references.

Strout et al. consider the specific case of sparse tiling applied to a Gauss-Seidel computation where unknown values and sparse matrix entries are associated with locations on the mesh.

Such matrices occur through numerical solution techniques such as the Finite Element Method (Section 2.8).

The Gauss-Seidel method is an iterative technique for solving a system of linear equations. We consider a linear problem of the form $Au = f$, where A is a sparse matrix and the vectors f and u are known and unknown, respectively. We express the Gauss-Seidel algorithm in pseudo-code as follows:

```
for i = 1 to T
  for j = 1 to R
    u_j^(i) = (1/A_jj)(f_j - Σ_{k=1}^{j-1} A_jk u_k^(i) - Σ_{k=j+1}^{n} A_jk u_k^(i-1))
```

Each iteration of the outer loop indexed by i produces a new approximation to the solution u. The inner loop indexed by j iterates over the R rows of the sparse matrix. The summations, which form the loop indexed by k, use values from the unknown vector from both the current and previous iteration of i. Each update to u can be performed in-place. The total number of iterations, T, needs to be a fixed value as the iteration space will be partitioned for a finite number of iterations.

The algorithm for sparse tiling Gauss-Seidel constructs a number of *tiles*, a set of computations that can be executed atomically. In *serial sparse tiling* the tiles must be executed in a particular order. In *parallel sparse tiling*, the tiles can be executed in parallel, but must be followed by a fill-in stage that executes computations not included in the tiles. Both strategies have the same runtime structure:

Partitioning The mesh over which the problem is being solved is partitioned. Graph partitioning is an NP-Hard problem, however, there exist many rapid heuristics for obtaining reasonable partitioning.

Tiling The iteration space for Gauss-Seidel is split into a number of atomically executable tiles. Each tile corresponds to the computations necessary to evaluate the unknowns

corresponding to a particular partition of the mesh. Each tile has layers, which correspond the to different values of i.

Tiles are constructed by partitioning the iteration space corresponding to the final iteration of loop i using the mesh partitioning as a basis. The tiles are then grown backwards to contain parts of iteration space corresponding to earlier values of i. Computations are added and removed from each tile as needed to avoid violating data dependencies.

Rescheduling Mesh nodes are renumbered so that they respect inter-tile dependencies. Within a tile, computations are scheduled by layer and within each layer.

Execution The computation is executed using the new schedule. It describes the layers within each tile and when they should be executed. The schedule may contain a tile execution ordering in the case of serial sparse tiling.

2.5 Code Generation and Optimisation Frameworks

Here I cover libraries and frameworks for multistage optimisation and code generation. LLVM maintains information throughout a program's lifetime to enable analyses and optimisation. Comparing to active libraries, this approach is also "active", but no further responsibility for performance is shifted to the library writer. Both SUIF and TaskGraph are in used our active linear algebra library Desola, described in Chapter 3.

2.5.1 LLVM

LLVM [24] (Low Level Virtual Machine) is a compiler framework designed to support transparent lifelong program analysis and transformation by providing high-level information to compiler transformations at compile-time, run-time, link-time and in the idle time between runs.

LLVM provides a number of capabilities that are useful for lifelong transformation and analysis of arbitrary programs:

Persistent program information The compilation model preserves the LLVM representation throughout the application lifetime, making it possible to perform complex optimisations at all stages.

Offline code generation It is possible to compile programs into native machine code offline, which is useful when expensive code generation techniques need to be employed.

User-based profiling and optimisation LLVM allows profiling information to be gathered at runtime so it can be applied with profiling guided transformations at run-time and idle-time.

Transparent runtime model LLVM does not enforce any kind of object model or exception handling semantics or runtime environment, making it possible to use any language or combination of languages to be compiled using it.

Uniform, whole program compilation LLVM's language independence makes it possible to compile and optimise the complete code of an application in a uniform manner including language-specific runtime libraries and system libraries.

Program representation

LLVM represents code using an abstract RISC-like instruction set but with higher level information for effective analysis. The LLVM instruction set consists of 31 opcodes, most of which are overloaded. LLVM provides an infinite set of typed virtual registers in Single Static Assignment (SSA) form, which can hold values of primitive types. Thus, each register is written by an instruction only once. The SSA representation provides a compact use-def graph that simplifies many data-flow optimisations. Non-loop transformations are simplified because they do not encounter anti or output dependencies. LLVM also makes the control flow graph of every function explicit in the representation.

Features of the LLVM program representation include:

Language independent type system Every SSA register and explicit memory object has

an associated type and obeys explicit type rules. Type information combined with the instruction opcode is used to determine exact instruction semantics, enabling a broad class of high-level transformations on low-level code. Primitive types in the LLVM system consist of ints and floats of different sizes and bools and voids. LLVM also defines four derived types sufficient to implement most high level languages: pointers, arrays, structures and functions.

The derived types capture enough information to support sophisticated language independent analyses and transformations such as call-graph construction, structure field reordering and array dependence analysis. LLVM also supports weakly typed languages. This means the declared type information in a program may not be reliable. Pointer analysis algorithms are used to determine whether the types of pointer targets are reliably known.

Explicit memory allocation LLVM instructions are provided for performing typed memory allocation. These include *malloc* and *free* which allocate and free, respectively, memory from the heap. An *alloca* instruction is provided to allocate memory on the heap which is automatically deallocated on return from a function. All l-values (addressable objects) in LLVM are explicitly allocated giving a model in which there are no implicit memory accesses, simplifying memory address analysis.

Function call and exception handling support LLVM provides a *call* instruction that abstracts away the calling convention of the target machine, simplifying program analysis. Unusually, it provides an explicit, low-level, machine independent mechanism for implementing exception handling. An *invoke* instruction is provided to transfer control to a function that might throw an exception and to specify a basic block to transfer control to, should an exception be thrown. The *unwind* instruction is used to throw the exception and will transfer control to the basic block specified by *invoke* after removing the activation record it created. This system has allowed both C's *setjmp/longjmp* calls and the C++ exception model to be implemented cleanly.

Plain-text, binary and in-memory representations The LLVM representation is a first

class language that defines equivalent textual, binary and in-memory representations. The instruction set is designed to serve as both an offline code representation and compiler internal representation without semantic conversion. This makes it much simpler to write debugging transformations and understand the in-memory representation.

Compiler Architecture

The LLVM compiler architecture is designed to enable lifelong transformation and analysis of arbitrary code. The compiler architecture has the following components:

External front-end The front-end is responsible for converting source-language programs into the LLVM instruction set. It translates source into LLVM code whilst trying to synthesise as much useful LLVM type information as possible. It may also perform language-specific optimisations or invoke LLVM inter-procedural optimisations at the module level.

The front-end need not perform SSA construction. Stack allocated variables will be converted into SSA registers if their address does not escape the enclosing function.

Linker and Inter-procedural optimiser Link-time is the first phase where most of the program is available for analysis and transformation. LLVM takes this opportunity to perform aggressive inter-procedural optimisations. LLVM contains a number of inter-procedural analyses including call graph construction mod/ref analysis and call graph construction. It can perform transformations like inlining, dead global and argument elimination, constant propagation, array bounds check elimination and structure field reordering.

At compile time, inter-procedural summaries can be computed for each program and attached to the bytecode. The link-time optimiser can use this to avoid recalculating this information from scratch, dramatically speeding up the incremental compilation of a large number of small translation units.

Native code generator Before execution, a code generator is used to translate from LLVM to native code for the target platform. This may be done at link-time or install-time and possibly use expensive code generation techniques. If a user decides to use the post-link optimisers, a copy of the LLVM bytecode is included with the executable. The code generator also inserts lightweight instrumentation code into the program to identify frequently executed regions of code. It is also possible to use a JIT code generator which translates one function at a time during execution and is capable of adding the same instrumentation as the offline code generator.

Runtime optimiser During program execution, the most frequently executed paths are identified through a combination of online and offline instrumentation. When a hot loop region is detected at runtime, a runtime instrumentation library instruments the executing native code to identify frequently taken paths. When a hot path is identified, the original LLVM code is duplicated into a trace, optimised and then regenerated into native code and placed into a trace cache. Branch instructions are inserted into the original code to the new native code.

This strategy enables LLVM to perform efficient native code generation ahead of time, enabling the runtime optimiser to receive support from the native code generator (e.g. for instrumentation) and allowing the runtime optimiser to use high-level information to perform sophisticated optimisations.

Offline optimiser The offline optimiser enables optimisation during idle-time on the end user's system. It is similar to the link-time inter-procedural optimiser but with greater emphasis on profile-driven and target-specific optimisation. Benefits include the ability to use profile information collected from runs of the system, the ability to tailor code to the architecture of a single machine and the ability to apply more aggressive optimisations due to the relaxed time constraints when operating offline.

2.5.2 SUIF

SUIF [25] (Stanford University Intermediate Format) is a C++ infrastructure for research on parallelising and optimising compilers. It has been used to perform research on topics including scalar optimisations, array data dependence analysis, loop transformations for locality and parallelism, software prefetching and instruction scheduling.

SUIF has been structured as a *kernel* plus a small *toolkit*. The kernel defines the intermediate representation between passes of the compiler while the toolkit contains compilation analyses and passes built using the kernel.

The SUIF intermediate format is a mixed level representation supporting both high level restructuring transformations and low level analyses and optimisations. The SUIF intermediate representation (IR) has a hierarchical structure and maintains high level constructs such as loops, conditionals and array access which provide enough information to allow parallelising passes to work effectively. It also provides RISC-like operations which are stored as lists.

The SUIF IR also supports annotations which allow communication between compiler passes. For example, data dependence information could be passed from a high-level analysis by annotating load and store instructions.

SUIF has a C front-end which allows parsing of C into SUIF, which along with a FORTRAN to C converter can also parse FORTRAN into SUIF with additional information stored as annotations. The SUIF IR retains enough information to allow it to be converted back to legal high-level C which makes it useful for source-to-source translation, C code generation or object code generation.

SUIF provides libraries for carrying out dependence analysis and loop transformations. SUIF's dependence analysis library provides tools for determining the dependence between array accesses inside loops. It consists of a sequence of fast exact tests, aimed towards generating exact answers for array accesses that are affine functions of the enclosing loop bounds. SUIF's loop transformation library provides tools for performing tiling as well as unimodular transformations such as loop interchange, reversal and skewing. Using SUIF's dependence analysis and

loop transformation tools, the authors have implemented a loop-level paralleliser and locality optimiser.

SUIF also defines a file format for the intermediate representation. This allow the different compilation passes to be implemented as separate programs. Although inefficient, this allows flexibility with respect to adding new passes or reordering existing ones.

2.5.3 TaskGraph

TaskGraph [13] is a SUIF-based C++ library for dynamic code generation. TaskGraph is used extensively in our linear algebra active library, DESOLA and is covered in Section 3.3.

2.6 Tensors

We make extensive use of tensor fields and notation in later chapters. We provide a brief description of tensors and the common operations defined between them.

Tensors can be viewed as geometric objects that are a generalisation of scalars and vectors. Tensors have an associated:

Rank The rank of a tensor is the number of indices required to uniquely identify a value within the tensor. For example, rank-0 tensors represent scalar values and rank-1 tensors represent vectors.

Dimension The dimension of a tensor refers to the dimensionality of the physical space in which the tensor represents a value.

The indices of a tensor can be classified as *contravariant* (upper) indices and *covariant* (lower) indices depending on how the index transforms with a change of basis. However, in a Cartesian co-ordinate system the two types of index are equivalent.

Notation for referencing tensor elements is similar to that for matrices. The expression ${a^i}_{jk}$ references the element in tensor a with contravariant index i and covariant indices j and k.

It is common to write tensor operations using *Einstein summation convention* which represents a tensor product with summation over repeated indices. For example the inner product between two n-dimensional vectors vectors $u \cdot v$ would be written as $u_i v_i$ where:

$$u_i v_i = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + \ldots + u_n v_n$$

using this notation, we can define the following operations.

The *inner product*, $a \cdot b$, of two tensors a and b with ranks r_a and r_b produces a tensor of rank $r_a + r_b - 2$. Summation is performed over the innermost index of both tensors. For example, the inner product where a and b are two rank-2 tensors can be defined as:

$$a \cdot b = T \qquad \text{where } T_{ik} = a_{ij} b_{jk}$$

The *double dot product*, $a : b$, between two tensors with ranks r_a and r_b produces a tensor with rank $r_a + r_b - 4$. Summation is performed over the two innermost indices. For example, the double dot product where a and b are rank-3 tensors can be defined as:

$$a : b = T \qquad \text{where } T_{il} = a_{ijk} b_{jkl}$$

The outer product, ab, of two tensors a and b with ranks r_a and r_b produces a tensor of rank $r_a + r_b$. It performs no summation and therefore has no repeated indices. For example, the outer product of two rank-2 tensors, a and b, can be defined as:

$$ab = T \qquad \text{where } T_{ijkl} = a_{ij} b_{kl}$$

The cross-product, $a \times b$, of two tensors a and b is only defined when a and b are both rank-1

tensors. The result is also a rank-1 tensor. It can be defined as:

$$a \times b = T \qquad \text{where } T_i = \epsilon_{ijk} a_j b_k$$

and ϵ is the Levi-Civita symbol, defined for dimension n as

$$\epsilon_{ijk} = \begin{cases} 0 & \text{when any two indices are equal} \\ +1 & \text{when } i, j, k \text{ are an even permutation of 1, 2, ..., n} \\ -1 & \text{when } i, j, k \text{ are an odd permutation of 1, 2, ..., n} \end{cases}$$

2.7 Vector Calculus Notation

In this section, I cover notation for the application of the *gradient*, *divergence* and *curl* vector differential operators, all denoted using the symbol ∇. The following definitions assume a Cartesian co-ordinate system.

The *gradient* operator applied to a tensor field f is written as ∇f. It increases the rank of the tensor field by 1. If f is a scalar field of dimension n, the result is defined as:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

The *divergence* operator applied to a tensor field f is written as $\nabla \cdot f$. It decreases the rank of the tensor field by 1 and therefore can only be applied to rank-1 or higher tensor fields. Applied to a vector field of dimension n, the result is defined as:

$$\nabla \cdot f = \frac{\partial f}{\partial x_1} + \frac{\partial f}{\partial x_2} + \ldots + \frac{\partial f}{\partial x_n}$$

The *curl* operator applied to a tensor field f is written as $\nabla \times f$. Applied to a 3 dimensional vector field, the result is defined as:

$$\nabla \times f = \left(\frac{\partial f_3}{\partial x_2} - \frac{\partial f_2}{\partial x_3}, \frac{\partial f_1}{\partial x_3} - \frac{\partial f_3}{\partial x_1}, \frac{\partial f_2}{\partial x_1} - \frac{\partial f_1}{\partial x_2} \right)$$

Finally, we define the *Laplacian* operator ∇^2 also written as Δ. It can be written using the divergence and gradient operators as $\nabla \cdot \nabla$. It transforms a tensor field to one of the same rank. Applied to a scalar field f of dimension n the result is defined as:

$$\nabla^2 f = \Delta f = \frac{\partial^2 f}{\partial x_1^2} + \frac{\partial^2 f}{\partial x_2^2} + \ldots + \frac{\partial^2 f}{\partial x_n^2}$$

2.8 The Finite Element Method

This section provides a brief overview of the finite element method. Readers unfamiliar with the finite element method are advised to consult a text such as Sherwin and Karniadakis [26] or lecture notes such as Rolf Rannacher's [27].

2.8.1 Introduction

The finite element method is a technique used to find the approximate solution to some general linear problem:

$$\mathbb{L}(u) = 0 \tag{2.1}$$

where $\mathbb{L}$ is some linear operator usually containing partial derivatives. We introduce the approximate solution u^δ and the residual R:

$$\mathbb{L}(u^\delta) = R(u^\delta) \tag{2.2}$$

The approximate solution u^δ is represented as follows:

$$u^{\delta}(x) = \sum_{i=0}^{N_{dof}} \hat{u}_i \Phi_i(x) \tag{2.3}$$

u^{δ} is represented by a linear combination of a finite number of *basis functions*. $\hat{u}_i$ represents the co-efficient for basis function i. The functions $\Phi_i(x)$ are known as *trial functions*.

2.8.2 The Method of Weighted Residuals

As the name suggests, we multiply the residual by a weight function and then integrate over the domain. Choosing g as our weight function gives us:

$$\int_{\Omega} R(u^{\delta}(x)) g(x)\, dx = 0 \tag{2.4}$$

This is known as the *weak* formulation of the problem as the approximate solution u^{δ} does not require that $R(u^{\delta}) = 0$ over the whole domain, but rather that the weighted integral of R over the domain is equal to zero.

We choose a set of functions that will act as our weighting function. These are commonly called *test functions*. Assuming N_{dof} test functions, we have a minimization problem of the form:

$$\int_{\Omega} R(u^{\delta}(x)) w_j(x)\, dx,\ j = 1, \ldots, N_{\text{dof}} \tag{2.5}$$

Different choices of w_j correspond to different projections methods. The most common is the Galerkin method where $w_j = \Phi_j$. If $w_j = \Upsilon_j$ where $\Upsilon_j \neq \Phi_j$ this is known as a Petrov-Galerkin method.

Typically, the LHS integral is a *bilinear form* (linear in the trial and test functions) and the RHS integral is a *linear form* (linear in the test function). Such systems can be assembled into a linear system of equations and solved using standard techniques. If the LHS integral is

non-linear in the trial function, it may be necessary to employ a linearisation technique or use a non-linear solver.

2.8.3 Boundary Conditions

In order for a problem to be well-posed, it may be necessary to specify boundary conditions (e.g. constraints on the value of the variable being solved for, on the edge of the domain). These may be classified as follows:

Dirichlet These specify the value of the problem variable on the edge of the domain. Sherwin et al. [26] describe an elegant mathematical formulation where the unknown solution u^δ is decomposed into a known function $u^\mathcal{D}$ that satisfies the boundary conditions and a homogeneous function $u^\mathcal{H}$ which is unknown and zero on the Dirichlet boundaries. Solution only takes place for $u^\mathcal{D}$ as $u^\mathcal{H}$ is known.

Often, imposition of Dirichlet boundary conditions is done in a less elegant manner, by altering the constructed linear system of equations. One method to do this is to use the *Payne-Irons* (or big-spring) method which scales a matrix diagonal by a large value α and a value in the RHS vector by $q\alpha$ so that a value in the unknown is strongly drawn towards the value q.

Neumann These specify restrictions on the derivative of a solution on the edge of the domain. These types of boundary conditions are dealt with implicitly as part of the formulation. Integration by parts typically results in integrals over the edge of the domain involving derivatives of the variable to be solved for. By substituting values of the variable into that term Neumann boundary conditions are applied.

For example, consider the following problem in a one-dimensional domain:

$$\int_0^1 v\mathbb{L}(u)\,dx = \int_0^1 v\left(\frac{\partial^2 u}{\partial x^2} + f\right)\,dx = 0 \tag{2.6}$$

Integrating by parts we obtain:

$$\int_0^1 \frac{\partial v}{\partial x}\frac{\partial u}{\partial x}\,dx = \int_0^1 vf\,dx + \left[v\frac{\partial u}{\partial x}\right]_0^1 \tag{2.7}$$

Substituting in values of $\frac{\partial u}{\partial x}$ at $x = 0$ and $x = 1$ imposes Neumann boundary conditions at those edges.

Robin Also known as *mixed* conditions, these refer to a combination of Dirichlet and Neumann boundary conditions.

2.8.4 Problem Discretisation

Our solution domain is partitioned into non-overlapping elements. Our basis functions are defined so that they are only non-zero over a small number of cells in physical proximity.

Typically we define our basis functions over a reference cell and transform our basis function to a global cell. We define a mapping function $\iota^k(i)$ which given the local numbering of a basis function coefficient i on some cell k returns the global numbering. We also define $\chi^k(\xi)$ which will transform a co-ordinate on some cell k specified in local co-ordinates ξ to global co-ordinates. Hence we have:

$$\Phi_{\iota^k(i)}(\chi^k(\xi)) = \phi_i(\xi) \tag{2.8}$$

where $\phi_i(\xi)$ represents basis function i defined on the reference cell, evaluated at local co-ordinate ξ.

Finite element discretisations may either be continuous or discontinuous. A continuous discretisation will have field basis function coefficients shared between neighbouring cells. A discontinuous discretisation results in fields that may take multiple values at the boundaries between cells.

Even with a continuous discretisation, our discrete function space can usually only represent

C^0 continuous fields (i.e. fields with a discontinuous derivative). As a consequence, integration by parts must often be applied to the weak form if it requires a second derivative or higher.

2.8.5 Assembly and Solution

Assembly is the process of constructing the linear system:

$$A\mathbf{u} = \mathbf{b} \tag{2.9}$$

A represents the discretised version of our linear operator, $\mathbf{b}$ represents our discretised RHS and $\mathbf{u}$ is the vector of unknown trial function coefficients. The number of rows and columns of A are equal to the number of test and trial functions respectively. Each element of A is equal to the value of the integral of the corresponding variational form over the domain so that:

$$A_{ij} = a(\Phi_j, \Psi_i)$$

Where a represents our bilinear form. Since our basis functions are only non-zero over neighbouring cells, A is a sparse matrix and can be constructed from a set of smaller matrices corresponding to integrals over individual cells. This process is called *local assembly*.

For each cell k we construct the local assembly matrix:

$$M^k_{pq} = a(\Phi_{\iota^k(q)}, \Psi_{\iota^k(p)}) \tag{2.10}$$

where p and q are the numberings of the test and trial functions local to each cell. Φ and Ψ are approximated by their local equivalents when evaluating a. This requires performing co-ordinate transforms between local and global co-ordinates and rewriting to the integral, which we do not describe here. Each contribution M^k is summed into the global system matrix A.

```
(integral-forms
  ((m u v) (* u v))
  ((k u v) (dot (gradient u) (gradient v)))
  ((d u v) (* (divergence u) (divergence v)))
  ((p u v) (+ (* alpha (m u v))
              (* nu (k u v))
              (* rho (d u v))))
)
```

Figure 2.3: The declaration of four variational forms in Analysa. The forms `m` and `k` correspond to the standard "mass" and "stiffness" forms respectively.

Once assembled, the sparse linear system can be solved by a number of methods (e.g. Krylov subspace [28]) in order to obtain $\mathbf{u}$, which is the vector of trial function coefficients used to represent u^δ.

2.9 Finite Element Libraries

In this section we describe a number of finite element implementations and domain-specific languages. In particular, we are interested in what information these systems consider necessary to specify a finite element problem, the abstractions they define and how they can use domain-specific knowledge to optimise their implementations.

2.9.1 Analysa

Analysa [29] is a problem solving environment for partial differential equations. It is written in the functional language Scheme and incorporates an embedded domain-specific language for specifying variational forms. We provide an example of the declaration of four variational forms in Analysa in Figure 2.3.

We note that Analysa provides a *projection* operator that operates on finite element function spaces. Given a discretised field (*interpolants* in Analysa terminology) and a function space, the projection operator will transform the field to that space. Depending on use, this function can act as both an extension or restriction operator on discrete fields. In particular, this seems

```
FunctionSpace {
  { Name H1; Type Form0;
    BasisFunction {
      { Name sn; NameOfCoef vn; Function BF_Node; Support D;
        Entity NodesOf[All]; }
    }
  }
}
Formulation {
  { Name Poisson; Type FemEquation;
    Quantity {
      { Name v; Type Local; NameOfSpace H1; }
    }
    Equation {
      Galerkin { [ a[] * Dof{Grad v}, {Grad v} ] ; In D; Jacobian V;
                 Integration I; }
      Galerkin { [ f[], {v} ] ; In D; Jacobian V; Integration I; }
    }
  }
}
```

Figure 2.4: Declaration of a discrete function space and system of equations for solving the weak form of Poisson's equation in GetDP's problem definition language.

to support the elegant implementation of Dirichlet boundary conditions by separating Dirichlet and Homogeneous components as described by Sherwin et al. [26].

2.9.2 GetDP

GetDP [30] is a software environment for the numeric solution of differential equations supporting multiple solution techniques including finite element methods.

Most notably, GetDP uses a *problem definition structure* that provides a formal mathematical definition of the problem in the form of a text file. We present an example of a GetDP specification for solving Poisson's equation in Figure 2.4

The generality of GetDP is of particular interest. GetDP is not embedded in some other language and provides no other means of specifying the problem and solution method other than the problem definition file. Hence, the GetDP problem definition must truly capture every aspect of the problem being solved. In contrast, language embedded finite element DSLs most

```
for (unsigned int i=0; i<dofs_per_cell; ++i)
  for (unsigned int j=0; j<dofs_per_cell; ++j)
    for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
      cell_matrix(i,j) += (fe_values.shape_grad (i, q_point) *
                           fe_values.shape_grad (j, q_point) *
                           fe_values.JxW (q_point));
```

Figure 2.5: Local assembly of the Laplace operator in deal.II. Handling of iteration over the basis functions is performed explicitly by the library client.

often have a very specific scope (commonly, just the specification of variational forms) and anything beyond that scope is handled directly in the host language.

GetDP's problem definition file is a complete problem specification including mesh topology, user defined functions, function space definitions, integration methods, linear system generation and solution and post-processing. As a consequence, the specification of GetDP's problem definition file provides a useful basis for determining what information may be needed to completely specify a general finite element solver must capture.

2.9.3 deal.II

deal.II [31] is a general purpose finite element library written in C++. It provides C++ classes for meshes, quadrature, linear systems and other finite element concepts. However, deal.II doesn't attempt to abstract finite element concepts such a variational forms or system matrix assembly. We show a code example for performing local assembly of the Laplace operator in Figure 2.5.

deal.II's interface is representative of finite element software that does not attempt to abstract away certain implementation aspects of a finite element solver. Although deal.II is of little interest to us abstraction-wise, it does provide insight into the building blocks of an efficient finite element solver implementation which may resemble what we want our abstract problem descriptions to map down to.

```
border C(t=0,2*pi){x=cos(t); y=sin(t);}
mesh Th = buildmesh (C(50));
fespace Vh(Th,P1);
Vh u,v;
func f= x*y;
real cpu=clock();
solve Poisson(u,v,solver=LU) =
  int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v))
  - int2d(Th)(f*v)
  + on(C,u=0);
plot(u);
cout << " CPU time = " << clock()-cpu << endl;
```

Figure 2.6: Code for solving Poission's equation in FreeFEM++'s C++-like language.

2.9.4 FreeFEM++

FreeFEM++ [32] is an interactive graphical environment for solving partial differential equations written in C++. It is the successor of a family of other applications, FreeFEM, FreeFEM+ and FreeFEM3D. FreeFEM++ implements an interpreter for a language that strongly resembles C++ augmented with additional finite element related types. We show an example definition of a Poisson solver in FreeFEM++ in Figure 2.6.

Although FreeFEM appears to use expression capture in their language, in choosing to incorporate C++, FreeFEM++ does not attempt to raise the level of abstraction as strongly as other languages for specifying finite element problems.

2.9.5 Sundance

Sundance [33] is a C++ library for solving partial differential equations using the finite element method. Sundance uses symbolic expressions and expression capture in order to specify problem aspects such as variational forms and boundary conditions. Sundance also supports MPI, enabling distributed memory parallelisation of data-structures and parallel solves.

Sundance makes extensive use of expression capture to the extent that it significantly raises the level of abstraction when specifying a finite element solver. We distinguish from Sundance

```
Expr phiHat = new TestFunction(new Langrange(1));
Expr phi = new UnknownFunction(new Langrange(1));
Expr v = 10.0;

Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr dz = new Derivative(2);
Expr grad = List(dx, dy, dz);

QuadratureFamily quad2 = new GuassianQuadrature(2);
Expr eqn = Integral(interior, (grad*phiHat)*(grad*phi) - v*phiHat, quad2);
```

Figure 2.7: Specification of a Poisson problem in Sundance. The code also illustrates how the `List` type can be used to construct the gradient differential operator.

by noting that one aim of our research is to capture a complete specification of a finite element solver with the aim of being to generate code from it. In contrast, Sundance's control and data flow is specified in standard C++ and therefore exists outside the scope of Sundance's expression capture.

2.9.6 FEniCS

FEniCS [34, 35] is a collection of projects and components a number of which are oriented towards the solution of partial differential equations in C, C++ and Python. Of particular interest are the FEniCS Form Compiler [36] (FFC) and the Unified Form Language [37] (UFL).

The Unified Form Language specification defines a domain-specific language for declaring discrete variational forms. This enables researchers to explore different choices of form and discretisation in a notation similar to mathematical description, without interacting with other concerns. We present an example of the UFL specification of a bilinear form for a Poisson problem in Figure 2.8.

The FEniCS Form Compiler accepts input in UFL and produces code to evaluate variational forms that can be used from C, C++ or Python. FFC development also reflects ongoing research into optimising evaluation of variational forms. FFC can generate code that performs local assembly using quadrature, but can also generate code using a tensor representation that

```
element = FiniteElement("Lagrange", triangle, 1)

v = TestFunction(element)
u = TrialFunction(element)

a = dot(grad(v), grad(u))*dx
L = v*f*dx
```

Figure 2.8: Specification of the bilinear and linear forms for the Poisson problem in UFL.

separates geometry-dependent and geometry-independent factors into different tensors which are combined using an inner product to form the final local assembly matrix. For variational forms with high-order basis functions, this can reduce the operation count considerably.

Work on optimising evaluation using tensor representation of local assembly has been undertaken by Kirby et al. [38] who have presented work on reducing the operation count needed to take the inner product between the geometry-dependent and independent tensors by exploiting redundancies in the latter.

The FEniCS optimisations demonstrate how domain-specific knowledge can be used to optimise performance. We also intend to search for domain-specific optimisation opportunities for optimisation.

2.10 Conclusion

In this chapter I covered background related to domain-specific abstractions, self-optimising libraries, code generation and optimisation which are useful in the construction of active libraries. I also covered the finite element method, domain-specific languages and libraries for specifying and solving finite element problems which will assist in understanding the design of our finite element library Excafé, detailed in later chapters.

In the next chapter, we present the sparse linear code-generation extension to my linear algebra active library Desola, in context with material from earlier work.

Chapter 3

Code Generation for Iterative Solvers of Sparse Linear Systems

3.1 Introduction

In my MEng thesis work, I explored the issues involved in performing runtime code generation for iterative solvers on dense linear systems of equations. I developed a prototype implementation of an "active" dense linear algebra library, DESOLA (Delayed Evaluation Self Optimising Linear Algebra), by delaying evaluation of expressions built using library calls, then generating code at runtime for the compositions that occurred.

This work was presented at the Library-Centric Software Design Workshop '06 [2]. During my PhD, I was invited to revise the paper for submission to a special issue of "Science of Computer Programming" published by Elsevier with extended performance results and an extensive background section [3]. This chapter is primarily intended to detail the work to extend this library to sparse linear algebra. Sections 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 and 3.9 are mainly excerpted from the journal paper and provide necessary background. Section 3.7 presents the new results collected for the journal Paper, and Section 3.8 describes work to extend DESOLA to sparse linear algebra.

"Active libraries" can be defined as libraries which play an active part in the compilation, in particular, the optimisation of their client code. The term was coined by Czarnecki et al. [39], who observed that active libraries break the abstractions common in conventional compilers. Active libraries are described in detail by Veldhuizen and Gannon [40]. In DESOLA, the key optimisations are loop fusion and array contraction. The ideas behind DESOLA are to:

Delay library call execution Calls made to the library are used to build a "recipe" for the delayed computation. When execution is finally forced by the need for a result, the recipe will often represent a complex composition of primitive calls.

Generate optimised code at runtime Code is generated at runtime to perform the operations present in the delayed recipe. In order to improve performance over a conventional library, it is important that the generated code should execute faster than a statically generated counterpart in a conventional library. To achieve this, we apply optimisations that exploit the structure, semantics and context of each library call. Compiled recipes are cached to limit overheads, but need to be executed enough times to offset the cost of the initial compilation.

This approach has a number of advantages. It does not need to analyse the client source code but is still able to optimise across statement and procedural bounds. DESOLA is implemented in standard C++ and uses a standard C compiler for runtime code compilation so the library user is not tied to a specific compiler.

One aspect of this approach is that the library interface remains isolated from the concerns of achieving high performance. The evolution of high performance numerical libraries such as BLAS [41] has been accompanied by a corresponding increase in the complexity of their interfaces. By allowing the library implementation to take more responsibility for optimisation, we aim to provide a more appropriate interface to the user, a similar goal to the Matrix Template Library [5].

Another aspect of this approach is that the code generated for a recipe is isolated from client-side code - it is not interwoven with non-library code. This is particularly important because

the structure of the code for a recipe is restricted in form, enabling us to introduce compilation passes specially targeted to achieve particular effects.

The disadvantage of this approach is the overhead of runtime compilation and the infrastructure to delay evaluation. In order to minimise the first factor, we maintain a cache of previously generated code along with the recipe used to generate it. This enables us to reuse previously optimised and compiled code when the same recipe is encountered again.

There are also more subtle disadvantages. In contrast to a compile-time solution, we are forced to make online decisions about what to evaluate, and when. Living without static analysis of the client code means we do not know, for example, which variables involved in a recipe are actually live when the recipe is forced.

3.2 Delaying Evaluation

Delayed evaluation provides the mechanism whereby we collect the sequences of operations we wish to optimise. We call the runtime information we obtain about these operations *runtime context information*.

This information may consist of values such as matrix or vector sizes, or the various relationships between successive library calls. Knowledge of dynamic values such as matrix and vector sizes allows us to improve the performance of the implementation of operations using these objects. For example, the runtime code generation system (see Section 3.3) can use this information to specialise the generated code. One specialisation we perform is with loop bounds. We incorporate dynamically known sizes of vectors and matrices as constants in the runtime-generated code.

DESOLA clients pass around handles to delayed expressions. The client can use these handles to build and assign expressions, and determine their values when needed. The library client need not be aware that delayed evaluation is occurring. Building and assigning numerical expressions with the handles causes them to be delayed by the library. Only when the client performs an

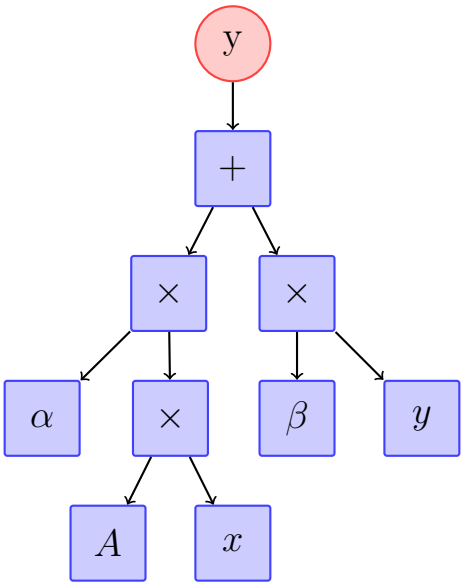


Figure 3.1: An example DAG. The circular node denotes a handle held by the library client. The expression represents the matrix-vector multiply function from Level 2 BLAS, $y = \alpha Ax + \beta y$.

operation that requires knowledge of the value of an expression is it calculated.

The delayed expressions are represented as Directed Acyclic Graphs (DAGs). Arcs in the DAG are directed in the direction of data dependence. Leaves in the DAG are data values (literals) and the other nodes represent delayed operations involving them. When the client forces evaluation of an expression node referenced by a handle, the node is replaced by a literal value. A literal value has no dependencies so previously referenced sections of the DAG may be orphaned. A simple reference counting scheme is used to determine when expression DAG nodes are no longer referenced by either other nodes or client handles, and reclaim them automatically.

An example DAG is illustrated in Figure 3.1. The circular node represents a handle held by the library client, and the other nodes represent delayed expressions. The three multiplication nodes do not have a handle referencing them. This makes them in effect, unnamed. When the expression DAG is evaluated, it is possible to optimise away the storage for these values entirely (their values are not required outside the runtime generated code). For expression DAGs involving matrix and vector operations, this enables us to reduce memory usage and improve cache utilisation.

Currently, we do not perform any form of common subexpression elimination on our expression

DAGs, although it would be possible to do so. We have not observed any of our benchmark examples computing redundant expressions. Explicit expression reuse, which does occur, is expressed directly in the structure of the constructed expression DAG.

Delayed evaluation allows DESOLA to perform cross component optimisation at runtime, and also allows us to equip it with a simple interface, such as the one required by the Iterative Template Library (ITL) set of iterative solvers.

3.3 Runtime Code Generation

Runtime code generation is performed using the TaskGraph [13]. TaskGraph is a C++ library for dynamic code generation. A TaskGraph represents a fragment of code which can be constructed and manipulated at runtime, compiled, dynamically linked back into the host application and executed. TaskGraph enables optimisation with respect to:

Runtime Parameters Code can be specialised to its parameters and other runtime contextual information.

Platform SUIF-1, the Stanford University Intermediate Format is used as an internal representation in TaskGraph, making a large set of dependence analysis and restructuring passes available for code optimisation.

Characteristics of the TaskGraph approach include:

Simple Language Design TaskGraph is implemented in C++ enabling it to be compiled with a number of widely available compilers.

Explicit Specification of Dynamic Code TaskGraph requires the application programmer to construct the code explicitly as a data structure, as opposed to annotation of code or automated analysis.

```
void TG_mm_ijk(unsigned int sz[2], TaskGraph &t)
{
  taskgraph(t) {
    tParameter(tArrayFromList(float, A, 2, sz));
    tParameter(tArrayFromList(float, B, 2, sz));
    tParameter(tArrayFromList(float, C, 2, sz));
    tVar(int, i); tVar(int, j); tVar(int, k);

    tFor(i, 0, sz[0]-1)
      tFor(j, 0, sz[1]-1)
        tFor(k, 0, sz[0] -1)
          C[i][j] += A[i][k] * B[k][j];
  }
}
```

Figure 3.2: A C++ function for constructing a TaskGraph that performs a dense matrix multiplication. The matrix sizes are passed in through the array `sz` and incorporated as constants inside the TaskGraph.

Simplified C-like Sub-language Dynamic code is specified with the TaskGraph library via an embedded sub-language similar to C. This language is implemented through extensive use of macros and C++ operator overloading. The language has first-class arrays, which facilitates dependence analysis.

We show a C++ function that constructs a matrix multiplication in the TaskGraph sub-language in Figure 3.2. The library is designed so that TaskGraph construction resembles C code. The generated code is specialised to the matrix dimensions stored in the array `sz`. The matrix parameters `A`, `B`, and `C` are supplied when the code is executed.

A code generation visitor visits nodes from the delayed expression DAG in reverse topological order to generate TaskGraph code. In order to calculate the value of the node that has been forced, the only nodes we need to evaluate are those that the forced node transitively depends on. However, as we wish to maximise the opportunities for optimisation, we evaluate all nodes transitively connected to the one being evaluated (i.e. we traverse the dependence edges in both directions). This heuristic is intended to maximise optimisation opportunities by evaluating all expressions that use the same data at the same time, possibly allowing loop fusion and array contraction to occur between loops using the same data.

Code generated by DESOLA is specialised to matrix and vector sizes as demonstrated in Figure 3.2. The constant loop bounds and array sizes make it much simpler to apply our loop fusion and array contraction optimisations. These are described in Section 3.5.

3.4 Code Caching

As the cost of compiling the runtime generated code is extremely high (compiler execution time is in the order of tenths of a second) it is important that this overhead be minimised.

Related work by Beckmann [42], on the efficient placement of data in a parallel linear algebra library, cached execution plans in order to improve performance. We adopt a similar strategy in order to reuse previously compiled code. We maintain a cache of previously encountered recipes along with the compiled code required to execute them. As any caching system will be invoked at every force point within a program using the library, it is essential that checking for cache hits is as computationally inexpensive as possible.

As previously described, delayed recipes are represented in the form of directed acyclic graphs. In order to allow the fast resolution of possible cache hits, all previously cached recipes are associated with a hash value. If recipes already exist in the cache with the same hash value, a full check is then performed to see if the recipes match.

Time and space constraints were of paramount importance in the development of the caching strategy and certain concessions were made in order that it could be performed quickly. The primary concession was that both hash calculation and isomorphism checking occur on flattened forms of the delayed expression DAG, ordered using a topological sort.

This causes two limitations:

- We do not detect the situation where the presence of commutative operations allow two differently structured delayed expression DAGs to be used in place of each other.

- As there can be more than one valid topological sort of a DAG, it is possible for multiple identically structured expression DAGs to exist in the code cache.

As we will see later, neither of these limitations significantly affects the usefulness of the cache, but first we will briefly describe the hashing and isomorphism algorithms.

Hashing occurs as follows:

- Each DAG node in the sorted list is assigned a value corresponding to its position in the list.
- A hash value is calculated for each node with references to other nodes encoded using the values assigned to them in the previous step.
- The hash values of all the nodes in the list are combined together in list order using a non-commutative function.

Isomorphism checking works similarly:

- Nodes in the sorted lists for each graph are assigned a value corresponding to their location in their list.
- Both lists are checked to be the same size.
- The corresponding nodes from both lists are checked to be the same type, and any nodes they reference are checked to see if they have been assigned the same numerical value.

Isomorphism checking in this manner does not require that a mapping be found between nodes in the two DAGs involved (it is implied by each node's location in the sorted list for each graph). It only requires determining whether the mapping is valid.

As the maximum number of nodes a node can depend on is bounded (maximum of two for a library with only unary and binary operators) then both hashing and isomorphism checking

between delayed expression DAGs can be performed in linear time with respect to the number of nodes in the DAG.

We previously stated that the limitations imposed by using a flattened representation of an expression DAG do not significantly affect the usefulness of the code cache. We expect the code cache to be at its most useful when the same sequence of library calls are repeatedly encountered (as in a loop). In this case, the generated DAGs will have identical structures, and the ability to detect non-identical DAGs that compute the same operation provides no benefit.

The second limitation, the need for identical DAGs matched by the caching mechanism to also have the same topological sort is more important. To ensure this, we store the dependency information held at each DAG node using lists rather than sets. By using lists, we can guarantee that when two DAGs are constructed in the same order they will also be traversed in the same order. Thus, when we come to perform our topological sort, the nodes from both DAGs will be sorted identically.

The code caching mechanism discussed, while it cannot recognise all opportunities for reuse, is well suited for detecting repeatedly generated recipes from client code. For the benchmark set of iterative solvers, compilation time becomes a constant overhead, regardless of the number of iterations executed.

3.5 Loop Fusion and Array Contraction

We implemented two optimisations using the TaskGraph back-end, SUIF [25]. Both loop fusion and array contraction are applied to the runtime generated code. As loop fusion often facilitates array contraction, the loop fusion pass precedes the array contraction pass.

Loop fusion [17] can lead to an improvement in performance when the fused loops use the same data. As the data is only loaded into the cache once, the fused loops take less time to execute than the sequential loops. Alternatively, if the fused loops use different data, it can lead to poorer performance, as the data used by the fused loop displace each other in the cache. Loop

```
for (int i=0; i<100; ++i)
  a[i] = b[i] + c[i];

for(int i=0; i<100; ++i)
  e[i] = a[i] + d[i];
```

(a) Before loop fusion

```
for (int i=0; i<100; ++i) {
  a[i] = b[i] + c[i];
  e[i] = a[i] + d[i];
}
```

(b) After loop fusion

Figure 3.3: Vector addition before and after loop fusion. The computed values assigned to `a[i]` are reused immediately, leading to improved temporal locality.

fusion is described in more detail in Section 2.4.1.

We provide an example of loop fusion in Figure 3.3. After fusion, the value stored in `a[i]` is reused immediately for the calculation of `e[i]`.

In DESOLA, we use a rather simple loop fusion algorithm which does not take into account cache locality and could be improved (although the fusions are always correct). We require that the loop bounds of the loops to be fused are constant but this does not limit us since our runtime generated code has already been specialised with loop bound information.

As discussed in Section 3.3, we employ a heuristic to generate code where loops are likely to reference the same data. Visual inspection of the code generated during execution of the iterative solvers indicates that the fused loops commonly use the same data. We believe this is likely due to the structure of the dependencies involved in the operations required for the iterative solvers.

We follow the loop fusion transformation by array contraction [17]. Array contraction allows the size of arrays to be reduced, possibly to a scalar value, when dependencies permit. This results in a reduction in memory, cache lines referenced, etc. Array contraction is described in more detail in Section 2.4.2. We also provide results on the number of array contractions we perform on our benchmarks in Section 3.7.

We provide an example of array contraction in Figure 3.4. The array `a` can be reduced to a scalar value so long as it is not required by any code following the two fused loops.

We use this technique to optimise away temporary matrices or vectors in the runtime generated

```
double a[100], b[100], c[100] d[100];

for (int i=0; i<100; ++i) {
  a[i] = b[i] + c[i];
  e[i] = a[i] + d[i];
}
```

(a) Before array contraction

```
double a, b[100], c[100] d[100];

for (int i=0; i<100; ++i) {
  a = b[i] + c[i];
  e[i] = a + d[i];
}
```

(b) After array contraction

Figure 3.4: Vector addition before and after array contraction. The array `a` is replaced by a scalar value, reducing memory requirements.

code. This is important because the DAG representation of the delayed operations does not hold information on what memory can be reused. However, we can determine whether or not each node in the DAG is referenced by the client code, and if it is not, it can be allocated locally to the runtime generated code and possibly optimised away. For details of other memory access transformations, consult Bacon et al. [17].

3.6 Liveness Analysis

Upon visual inspection of the runtime generated code produced by our benchmark suite of iterative solvers, it became apparent that a large number of vectors were being passed in as parameters. Their initial values were not used by the runtime generated code, instead, they were being assigned inside the runtime generated code so that their values could be propagated out.

Further investigation showed that that the values of these vectors were not used outside the runtime generated code either. In other words, these vectors could be made completely local to the runtime generated code. Unfortunately, our library could not determine this because handles to the vectors were still held by the iterative solver. As a consequence, these vectors were not susceptible to our array contraction pass. We realised that by designing a system to recover runtime information, we had lost the ability to use static information, in particular, the liveness properties of variables.

```
void printScaledCrossProduct(const Vector<float>& a,
                             const Vector<float>& b,
                             const Scalar<float>& alpha)
{
  const Vector<float> product = cross(a, b);
  const Vector<float> scaled = mul(product, alpha);
  print(scaled);
}
```

Figure 3.5: A C++ function that computes and prints the scaled value of a cross product using DESOLA vector and scalar handles passed as parameters.

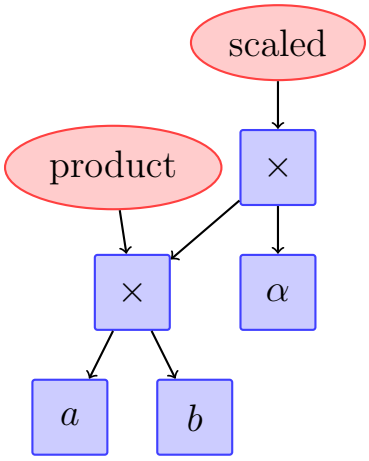


Figure 3.6: A DAG representing the computation built in Figure 3.5. The client holds handles to both the cross product and scaling operations (handles are denoted with elliptical nodes). When evaluation of the handle *scaled* is forced, the client still possesses the handle *product*, so the cross-product result cannot be allocated locally to the runtime generated code.

Consider the code in Figure 3.5 that takes the cross product of two vectors, scales the result and prints it. This operation can be represented with the DAG in Figure 3.6. The value pointed to by the handle *product* is never required by the client code. From the client's perspective the value is dead, but the library must assume that any value which has a handle may be required later on. Values required by the library client cannot be allocated locally to the runtime generated code, and therefore cannot be optimised away through techniques such as array contraction. Runtime liveness analysis permits the library to make informed guesses about the liveness of nodes in repeatedly executed DAGs, and allow them to be allocated locally to runtime generated code if it is suspected they are dead, regardless of whether they have a handle.

Having already developed a system for recognising repeatedly executed delayed expression

DAGs, we developed a similar mechanism for associating collected liveness information with expression DAGs.

Nodes in each generated expression DAG are instrumented and information collected on whether the values are live or dead. The next time the same DAG is encountered, the previously collected information is used to annotate each node in the DAG with an expectation with regards to whether it is live or dead. As the same DAG is repeatedly encountered, statistical information about the liveness of each node is built up.

If an expression DAG node is believed to be dead, then it can be allocated locally to the runtime generated code and possibly optimised away. This could lead to a possible performance improvement. Alternatively, it is also possible that the expression DAG node is not dead, and its value is required by the library client at a later time. As the value was not saved the first time it was computed, the value must be computed again. This could result in a performance decrease of the client application if such a situation occurs repeatedly.

3.7 Performance Evaluation for Dense Linear Algebra

We evaluated the performance of DESOLA using solvers from the Iterative Template Library (ITL) set of templated iterative solvers running on dense asymmetric matrices of different sizes. The ITL provides templated classes and methods for the iterative solution of linear systems, but not an implementation of the linear algebra operations themselves. ITL is capable of utilising a number of numerical libraries, requiring only the use of an appropriate header file to map the templated types and methods ITL uses to those specific to a particular library. ITL was modified to use DESOLA through the addition of a header file and other minor modifications.

We compare the performance of our library against the Matrix Template Library [5] (MTL), Intel's Math Kernel Library (IMKL) and ATLAS [16]. We compare against MTL because it has a similar goal of trying to provide high performance code in C++ with an elegant interface. Comparisons against ATLAS and Intel's MKL are provided as a performance baseline.

ITL already provides support for using MTL as its numerical library. We adapted ITL's existing interface for Sparse BLAS to dense BLAS allowing the ITL solvers to work with ATLAS and Intel's Math Kernel Library. We analysed the performance of five iterative solvers suitable for asymmetric matrices, namely Conjugate Gradient Squared, BiConjugate Gradient, BiConjugate Gradient Stabilised, Quasi-Minimal Residual and Transpose Free Quasi-Minimal Residual. The Chebyshev Iteration and Preconditioned Richardson solvers were not benchmarked due to the need to generate coefficient matrices with appropriate spectral properties.

We did not benchmark the Generalised Conjugate Residual (GCR) and Generalised Minimum Residual (GMRES) solvers because they required additional functionality from the matrix abstraction that we did not have time to implement (e.g. access to matrix columns as vectors). Additionally, all benchmarks used the identity preconditioner since all other ITL preconditioners were coded specifically to MTL. This is due to the fact that the ITL-specified interface provides insufficient functionality to implement them directly (e.g. decomposition into diagonal, upper and strictly lower parts for the Symmetric Successive Overrelaxation preconditioner).

The version of ITL used was 4.0.0-1. The version of MTL used was 2.1.2-23 of MTL2. Version 10.1 of the Intel C and C++ Compilers was used for both the runtime compiled code generated by our library and the compilation of the MTL benchmarks, respectively. The options passed to the Intel C and C++ compilers are described in Table 3.2. The version of Intel's MKL library was 10.0.2.018. Version 3.8.1 of ATLAS was used on architecture 1 (Xeon, see below). On architecture 2 (Pentium IV, see below) we took the slightly unusual step of comparing against ATLAS 3.6.0 with pre-collected tuning results from the Ubuntu Linux distribution for the Pentium IV with SSE2 support. We did this because we found that it outperformed any locally tuned ATLAS we could build. Both ATLAS builds used GCC 4.2.1 for kernel code compilation. We note that at the time of writing the online ATLAS installation guide advocates the use of GCC 4.2 over previous series 4 and 3 versions and also advises against the use of Intel's C Compiler for compiling ATLAS kernel code.

In order to show trends more clearly, we show throughput in terms of floating point operations per second. The number of floating point operations required have been estimated from the ITL

implementation of the iterative solvers. It is important to note that our library requires that we invoke a compiler at runtime and hence incur a compilation overhead. We omit this from the throughput graphs as the relative effect of this overhead is dependent on numerous factors including the size of the matrices involved and the number of iterations the solvers are run for. An indication of the compilation overhead for one of our architectures is given in Table 3.1.

The number of compiler invocations in our benchmarks is independent of the number solver iterations (when ≥ 3) as compiled code fragments are reused each iteration. The number of invocations corresponds to the number of unique expression DAGs generated. Two interrelated factors affect this value:

Force Points Any point in the execution of solver where the value of an expression is required forces evaluation of its associated expression DAG. Larger numbers of force points tend to result in more unique expression DAGs and therefore more compiler invocations. Within the main loop of many of the iterative solvers, force points exist both for checking the convergence condition and for checking various conditions to detect solver breakdowns.

Control Flow Changes in a program's control flow may cause it to construct novel expression DAGs. In our solvers, the expression DAGs constructed on the first and second iterations of the main loop differ due to the code that preceded each iteration. It is only on the third iteration of the loop that all generated expression DAGs can be evaluated using previously compiled code fragments.

We will discuss the observed effects of the different optimisation methods we implemented, and we conclude with a comparison against the same benchmarks using MTL, ATLAS and Intel's MKL. We evaluated the performance of the solvers on two architectures. All the solvers are single threaded and use double precision.

1. Intel Xeon "Clovertown" processor running at 2.66GHz, 4096 KB L2 cache with 4 GB RAM running 64-bit Ubuntu 7.04.

Solver	Number of Compiler Invocations	Total Compilation Time	Total Execution Time (size 500)	Total Execution Time (size 5000)
bicg	9	0.929	0.999	20.340
bicgstab	10	0.942	1.033	36.170
cgs	9	0.930	1.025	35.977
qmr	12	1.120	1.237	20.659
tfqmr	9	0.887	1.056	36.292

Table 3.1: The number of compiler invocations in each iterative solver, the total compiler overhead in seconds and total execution time (including compilation) for 256 iterations of each solver with a problem size of 500 and 5000 for architecture 1. Liveness analysis (Section 3.6) was disabled since it consistently lowered performance. This was due to the increased number of compiler invocations required.

Option	Description
-O3	Enables the most aggressive level of optimisation including loop and memory access transformations, and prefetching.
-restrict	Enables the use of the *restrict* keyword for disambiguating pointers. The *restrict* keyword is not used in MTL but is used in our runtime generated code.
-ansi-alias	Allows icc to perform more aggressive optimisations if the program adheres to the ISO C aliasing rules.
-xT	Generate code specialised for Intel Core2 Duo (for architecture 1).
-xP	Generate code specialised for Intel Pentium 4 with SSE3 (for architecture 2).

Table 3.2: The options supplied to Intel C/C++ compilers and their meanings.

2. Pentium IV "Prescott" processor running at 3.2GHz with Hyper-threading disabled, 2048 KB L2 cache with 2 GB RAM running 32-bit Ubuntu 7.04.

The first optimisation implemented was loop fusion. Three of five of the benchmarks did not show any noticeable improvement with this optimisation. Visual inspection of the runtime generated code showed multiple loop fusions had occurred between vector-vector operations but not between matrix-vector operations. As we were working with dense matrices, these fusions were unable to contribute any significant performance effects given that vector-vector operations are $O(n)$ and the matrix-vector multiplies present in each solver are $O(n^2)$.

We obtained significant speedups from loop fusion between matrix-vector multiply operations on two benchmarks, the BiConjugate Gradient solver and the Quasi-Minimal Residual solver. The first required no modification to ITL, but the latter[1] required minor changes to ITL. We observed that the checks for QMR breakdown forced the evaluation of a matrix-vector

[1] the QMR matrix-vector multiply fusion result was not yet achieved in the LCSD'06 paper [2].

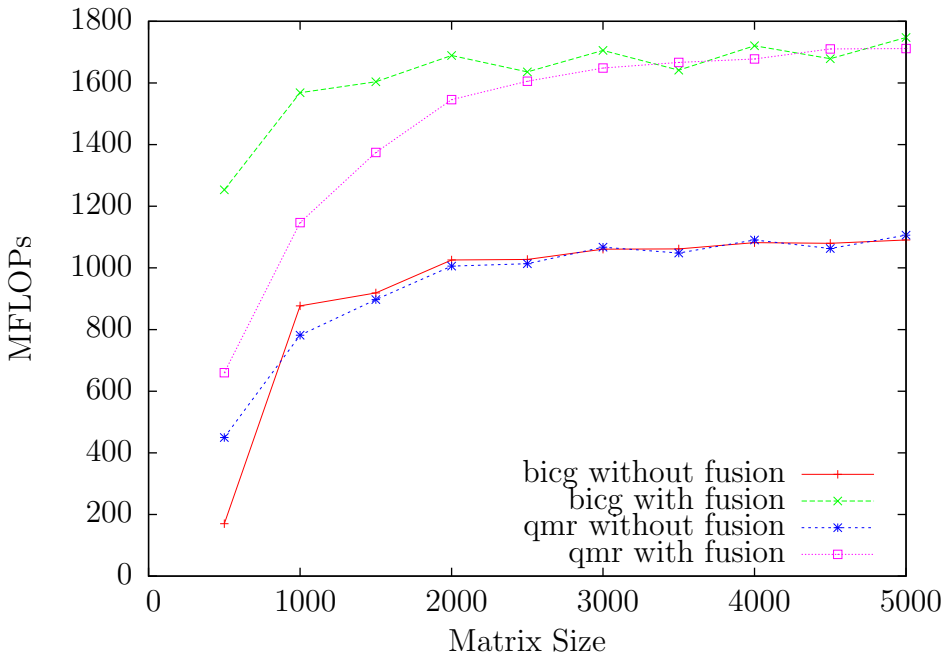


Figure 3.7: Throughput of the BiConjugate Gradient (BiCG) and Quasi-Minimal Residual solvers running on architecture 2 with and without loop fusion.

multiply before a second matrix-vector multiply, making fusion impossible. Data dependencies permitted moving the second (transpose) matrix-vector multiply to be adjacent to the first, enabling fusion. We note that as this move disregarded the control dependence between the transpose matrix-vector multiply and the QMR breakdown check, our changes also had the minor effect of the multiply being executed unnecessarily in the instance of the breakdown of the solver.

In both these cases the loop fuser was able to fuse a matrix-vector multiply and a transpose matrix-vector multiply with the result that the matrix involved was only iterated over once for both operations. A graph of the speedup obtained across matrix sizes is shown in Figure 3.7.

The second optimisation implemented was array contraction. We only evaluated this in the presence of loop fusion as the former is facilitated by the latter. The array contraction pass did not show any noticeable improvement on any of the benchmarks applications. On visual inspection of the runtime generated code we found that the array contractions had occurred in all the iterative solvers. The number of array contractions for each iterative solver are listed in Table 3.3. However, these only occurred on vectors, and affected only vector-vector operations. This is not surprising since only one matrix is used during the execution of the linear solvers

Solver	Total array contractions	Array contractions in repeatedly executed code
bicg	13	5
bicgstab	12	7
cgs	17	7
qmr	10	9
tfqmr	13	10

Table 3.3: Number of array contractions occurring in each iterative solver. *Total array contractions* refers to the number of array contractions performed in code generated during the execution of the solver. *Array contractions in repeatedly executed code* refers to the number of array contractions that occurred in code executed by the solver each iteration. Liveness analysis (Section 3.6) was disabled since it consistently lowered performance. This was due to the increased number of compiler invocations required.

and as it was required for all iterations, could not be optimised away. Section 3.8 describes the extension of our library to sparse matrices, which makes the effect of array contraction more apparent since the cost of the matrix-vector multiply operations becomes $O(n)$ instead of $O(n^2)$.

The last technique we implemented was runtime liveness analysis. This was used to try to recognise which expression DAG nodes were dead to allow them to be allocated locally to runtime generated code.

The runtime liveness analysis mechanism was able to find vectors in three of the five iterative solvers that could be allocated locally to the runtime generated code. The three solvers had an average of two vectors that could be optimised away, located in repeatedly executed code. Unfortunately, usage of the liveness analysis mechanism resulted in an overall decrease in performance. We discovered this to be because the liveness mechanism resulted in extra constant overhead due to more compiler invocations at the start of the iterative solver. This was due to the statistical nature of the liveness prediction, and the fact that as it changed its beliefs with regard to whether a value was live or dead, a greater number of runtime generated code fragments had to be compiled.

We also compared Desola against the Matrix Template Library (MTL), Intel's Math Kernel Library (IMKL) and ATLAS, running the same benchmarks. We enabled the loop fusion and array contraction optimisations, but did not enable the runtime liveness analysis mechanism

because of the overhead previously discussed.

We observe that on the BiCG and QMR benchmarks, on which we were able to perform matrix-vector loop fusion, we outperform the other implementations on both architectures at matrix sizes above 1500. We show the performance of the different implementations with the BiCG benchmark running on architecture 1 in Figure 3.8 and QMR on architecture 2 in Figure 3.9. Performance results for the QMR and BiCG benchmarks are similar on both architectures.

On architecture 1, we note that IMKL and ATLAS appear to perform particularly well with matrix sizes smaller than 1000. On the BiCGSTAB, TFQMR and CGS benchmarks, we can outperform MTL and ATLAS for matrix sizes above 2000, but do not achieve the performance of IMKL. Performance comparisons for the TFQMR benchmark are shown in Figure 3.10. Results for the BiCGSTAB and CGS benchmarks are similar.

On architecture 2, on the BiCGSTAB, TFQMR and CGS benchmarks, we outperform IMKL and MTL at matrix sizes above 1500. ATLAS consistently outperforms all other implementations at all matrix sizes. Performance comparisons for the TFQMR benchmark are shown in Figure 3.11. Results for the BiCGSTAB and CGS benchmarks are similar.

Once again, we stress that these graphs ignore the compilation overhead which is a constant in the case of the iterative solvers (see Table 3.1). Therefore, the relative effects of this overhead on performance will vary depending on the size of the problem, and the number of iterations required to meet convergence. We also note that mechanisms such as a persistent code cache could allow the compilation overheads to be significantly reduced. These overheads are discussed in Section 3.9.

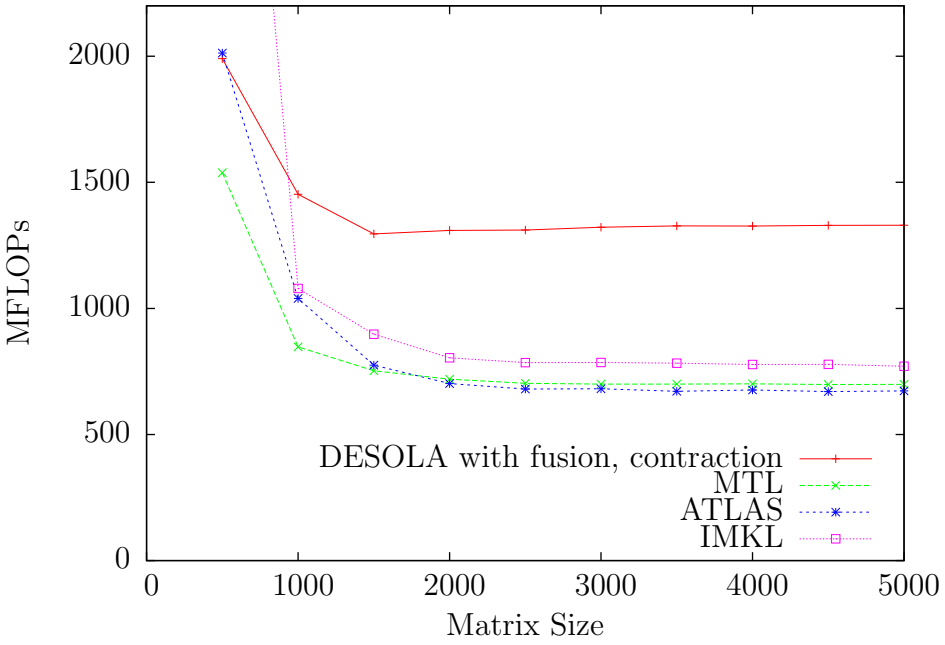


Figure 3.8: Throughput of the BiConjugate Gradient (BiCG) solver using DESOLA, MTL, ATLAS and IMKL running on architecture 1. Estimated throughput for IMKL at matrix size 500 is 4504 MFLOPs. Throughput for DESOLA ignores the constant compilation overhead (see Table 3.1).

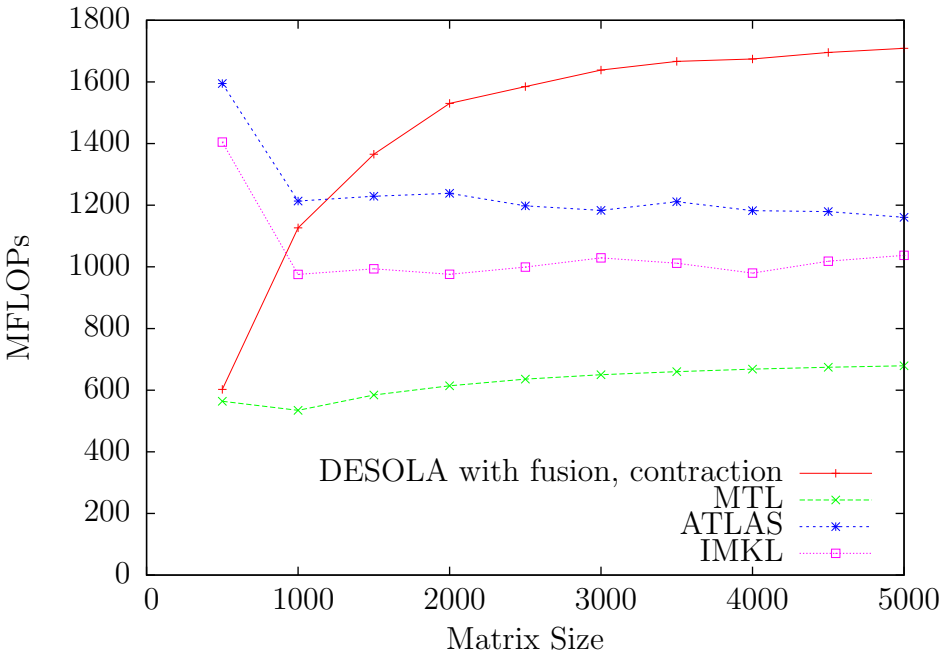


Figure 3.9: Throughput of the Quasi-Minimal Residual (QMR) solver using DESOLA, MTL, ATLAS and IMKL running on architecture 2. Throughput for DESOLA ignores the constant compilation overhead (see Table 3.1).

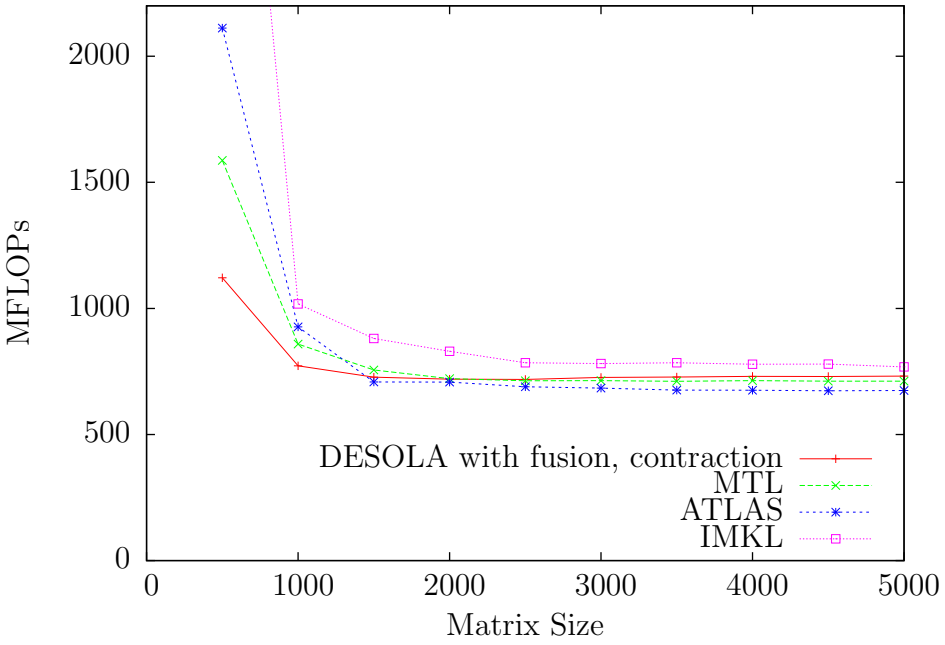


Figure 3.10: Throughput of the Transpose Free Quasi-Minimal Residual (TFQMR) solver using DESOLA, MTL, ATLAS and IMKL running on architecture 1. Estimated throughput for IMKL at matrix size 500 is 4233 MFLOPs. Throughput for DESOLA ignores the constant compilation overhead (see Table 3.1).

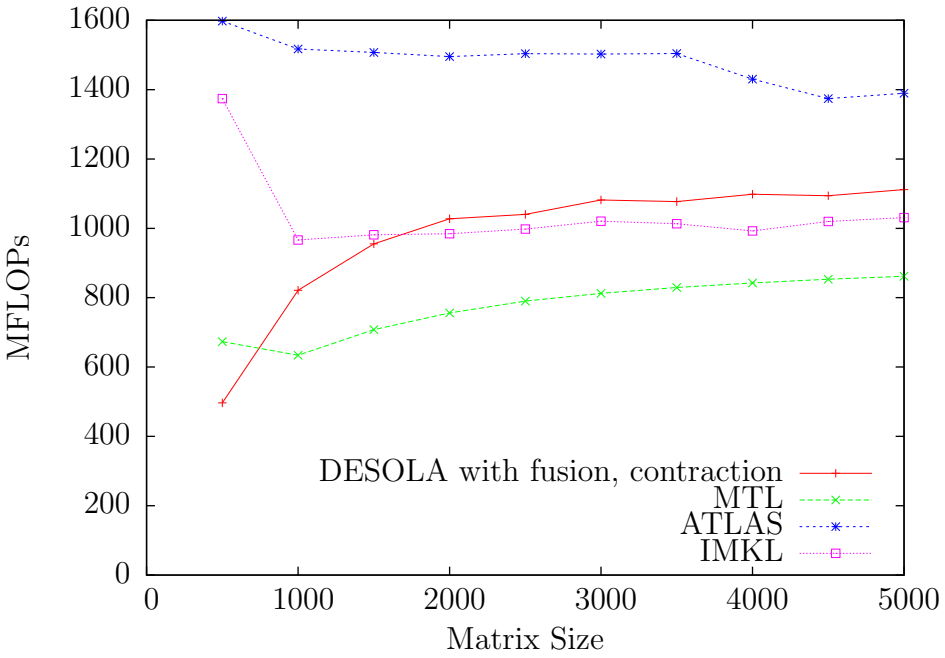


Figure 3.11: Throughput of the Transpose Free Quasi-Minimal Residual (TFQMR) solver using DESOLA, MTL, ATLAS and IMKL running on architecture 2. Throughput for DESOLA ignores the constant compilation overhead (see Table 3.1).

3.8 Extension to Sparse Matrices

DESOLA was originally restricted to dense linear algebra as this provided a simple domain in which to investigate and understand the various aspects of delayed-evaluation and runtime code generation. However, our benchmark suite, a set of iterative solvers for linear systems of equations is only of limited usefulness when employed with dense linear algebra. This is because direct techniques, such as Gaussian elimination, can be used to solve these systems in $O(n^3)$ operations.

Direct techniques are less applicable to sparse systems, because they typically cause large increases in the number of non-zeros in the matrix, causing it to become dense and making it extremely memory inefficient and impractical to store. The primary benefit of iterative methods is that they do not have this issue and thus can be used to solve sparse linear systems. We note that research into implementing direct sparse solvers that do not suffer from fill-in is an active research topic, with scalable parallelisation being an important issue [43, 44].

3.8.1 Sparse Matrix Storage

Compressed Row Storage [28] and Compressed Column Storage are two of the most common data structures for representing sparse matrices. I added support for CRS storage and two different implementations of sparse matrix-vector multiply to DESOLA. An example of a small sparse matrix in CRS representation is given in Figure 3.12.

3.8.2 Code Generation

We expect that the optimisations we apply to the runtime generated code may have different behaviours on the sparse solvers compared to the dense case. In particular:

Performance is matrix-dependent. In a matrix-vector multiply, the sparsity pattern of the matrix determines the access pattern to the vector. As the cache access pattern for the

$$\begin{pmatrix} 1 & 0 & 0 & 7 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 5 & 8 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 7 & 0 & 2 \end{pmatrix} \tag{3.1}$$

val	1	7	2	8	9	6	5	8	4	7	2
col_ind	1	4	2	6	5	4	1	2	5	4	6

row_ptr	1	3	5	6	7	10	12

Figure 3.12: A Compressed Row Storage representation of the matrix in Equation 3.1 is held by the arrays *val*, *col_ind* and *row_ptr*. Array *val* contains the non-zero values in the matrix. Array *col_ind* holds the corresponding columns of the non-zero values. Array *row_ptr* holds the indices at which each row starts in the *col_ind* and *row_ptr* arrays. This example uses one-based indexing, however zero-based indexing may be used as well.

vector is matrix-dependent, a matrix-vector multiply that performs well on one sparse matrix may perform poorly on another.

Sparse matrix-vector multiply is $O(n)$. We apply loop fusion and array contraction optimisations to the runtime generated code. This has the effect of reducing the memory usage and improving locality for a number of $O(n)$ vector-vector operations. However, dense matrix-vector multiply is an $O(n^2)$ operation so these optimisations have little observable effect except for the loop fusions between dense matrix-vector multiplies. Many sparse matrices from scientific applications have an upper bound on the number of non-zeros per row that is independent of system size, making the sparse matrix-vector multiply an $O(n)$ operation. As a consequence, we expect that the benefits of these optimisations should be more apparent.

Loop fusion is more difficult to achieve. In order to perform loop fusion between compatible loops, our optimisation pass only required that the loop bounds were identical and constant. This worked for dense matrix-vector multiply, but is harder in the sparse case as loop bounds are dependent on the matrix structure.

Vectorisation is difficult to achieve. As accesses into the left-hand side vector are no longer contiguous it is difficult to make effective use of vector instructions.

```
currentRow = 0

for valIndex = 0 to length(val)
{
  while row_ptr[currentRow+1] == valIndex
    currentRow += 1

  b[currentRow] += val[valIndex] * x[col_ind[valIndex]]
}
```

Figure 3.13: Pseudo-code implementation of CRS sparse matrix-vector multiply using an inner while-loop. The outer for loop iterates along all the non-zero values in the matrix whilst the inner while loop is used to update the corresponding row.

```
for row = 0 to numRows
{
  valStartIndex = row_ptr[row]
  valEndIndex   = row_ptr[row+1]

  for valIndex = valStartIndex to valEndIndex
    b[row] += val[valIndex] * x[col_ind[valIndex]]
}
```

Figure 3.14: Pseudo-code implementation of CRS sparse matrix-vector multiply using an inner for-loop. The outer for-loop iterates over the matrix rows whilst the inner for-loop is used to iterate over the non-zero values for each row.

Sparse Matrix Iteration

We developed two different implementations of sparse matrix-vector multiply. The version in Figure 3.13 uses a for-loop to iterate over all the non-zero values of the matrix. An inner while loop is used to increment a variable storing the current row by detecting when the offset into the *val* array is equal to index of the next row. A while-loop rather than an if-statement is required for when there are rows with no non-zero values.

The implementation in Figure 3.14 uses two nested for loops. The outer loop iterates over each row of the sparse matrix and the inner loop iterates over the range of the *val* array corresponding to the non-zero values in that row.

High-Level Fusion

Our double for-loop implementation of sparse matrix iteration cannot be fully fused by our SUIF loop fusion pass. This is due to the fact that the inner loop has non-constant bounds that our pass cannot handle. Our single for-loop implementation also cannot be fused, because our fuser cannot reason about the dependence implications of the inner while-loop.

In order to overcome these problems, we wrote a pass specifically for performing sparse matrix-vector multiply fusion. Rather than attempting to perform this optimisation on the AST of the generated code, we do it at the level of the TaskGraph DAG. The TaskGraph DAG still resembles the high-level expression tree, but contains information specific to code-generation such as the names of variables to be bound and the data storage formats of expressions represented in the DAG.

We add a new node type to the TaskGraph DAG which represents a number of matrix-vector and transpose matrix-vector multiplies involving the same matrix but different vectors. We also add a pass that takes individual (possibly transpose) matrix-vector multiplies and combines them if they operate on the same matrix and there are no dependencies between them.

This has the benefit that we are always able to generated fused code for matrix-vector multiplies regardless of the complexity of the code for iterating over the matrix. We also avoid performing any redundant index computations that might have been done had we simply performed loop fusion. The differences between the code generated after fusion as performed by SUIF and by the high-level fusion pass can be seen by comparing Figures 3.15 and 3.16, respectively.

Row-Length Specialisation

We observe that for the majority of sparse matrices representing linear systems, the number of non-zeros in each row tends to fall into a small set of values. We use this property to specialise the code for iterating over a sparse matrix to the most frequent row lengths.

For the sparse matrix iteration implementation using two nested for-loops, we generate a num-

```
for row = 0 to numRows
{
  valStartIndex_1 = row_ptr[row]
  valEndIndex_1   = row_ptr[row+1]

  for valIndex_1 = valStartIndex_1 to valEndIndex_1
    b_1[row] += val[valIndex_1] * x_1[col_ind[valIndex_1]]

  valStartIndex_2 = row_ptr[row]
  valEndIndex_2   = row_ptr[row+1]

  for valIndex_2 = valStartIndex_2 to valEndIndex_2
    b_2[col_ind[valIndex_2]] += val[valIndex_2] * x_2[row]
}
```

Figure 3.15: Pseudo-code implementation of CRS sparse matrix-vector multiply and transpose matrix-vector multiply after application of the SUIF loop fusion pass. The pass is unable to fuse the inner loops and redundant index calculations are performed.

```
for row = 0 to numRows
{
  valStartIndex = row_ptr[row]
  valEndIndex   = row_ptr[row+1]

  for valIndex = valStartIndex to valEndIndex {
    b_1[row] += val[valIndex] * x_1[col_ind[valIndex]]
    b_2[col_ind[valIndex]] += val[valIndex] * x_2[row]
  }
}
```

Figure 3.16: Pseudo-code implementation of a fused CRS sparse matrix-vector multiply and transpose matrix-vector multiply. The code is generated completely fused and there are no redundant calculations.

```
for (row_0 = 0u; row_0 <= 4690001u; row_0++) {
  valPtrStart_0 = rowPtr_0[row_0];
  valPtrEnd_0 = rowPtr_0[row_0 + 1];
  rowLength_0 = valPtrEnd_0 - valPtrStart_0;

  if (rowLength_0 == 5u) {
    for (valOffset_0 = 0; valOffset_0 <= 4u; valOffset_0++) {
      (*convVector_2)[row_0] = (*convVector_2)[row_0] +
        val_0[valPtrStart_0 + valOffset_0] *
        (*convVector_0)[colInd_0[valPtrStart_0 + valOffset_0]];
    }
    continue;
  }

  if (rowLength_0 == 3u) {
    for (valOffset_0 = 0; valOffset_0 <= 2u; valOffset_0++) {
      (*convVector_2)[row_0] = (*convVector_2)[row_0] +
        val_0[valPtrStart_0 + valOffset_0] *
        (*convVector_0)[colInd_0[valPtrStart_0 + valOffset_0]];
    }
    continue;
  }

  for (valPtr_0 = valPtrStart_0; valPtr_0 <= valPtrEnd_0 - 1; valPtr_0++) {
    (*convVector_2)[row_0] = (*convVector_2)[row_0] + val_0[valPtr_0] *
      (*convVector_0)[colInd_0[valPtr_0]];
  }
}
```

Figure 3.17: Specialised C generated by the DESOLA BiConjugate Gradient Stabilised solver for a matrix-vector multiply with the matrix rajat31. The `u` suffix on some integer values is C syntax for declaring an integer literal to be unsigned.

ber of variants of the inner loop specialised to the number of non-zeros in that row. These are ordered by frequency so that the most commonly executed loops require fewer conditionals to be checked. To avoid generating code for rows with infrequent numbers of non-zeros, we specialise to 95% of the matrix rows. The remaining rows are iterated over using the unspecialised version of the inner loop. Through this specialisation, we hope that the compiler can use techniques such as loop unrolling to improve the performance of the code.

An example of the specialised code is given in Figure 3.17.

3.8.3 Matrices

For our dense benchmark suite, we generated matrices of various sizes with appropriate numerical properties. For our sparse benchmark suite, this is no longer appropriate as performance is matrix-dependent.

We chose the following matrices from the University of Florida Sparse Matrix Collection:

Name	Rows	Columns	Non-zeros	Non-zeros per row	Structure	Problem kind
ASIC_680k	682,862	682,862	2,638,997	3.86	asymmetric	circuit simulation
Chebyshev4	68,121	68,121	5,377,761	78.94	asymmetric	structural
ex11	16,614	16,614	1,096,948	66.03	asymmetric	computation fluid dynamics
rajat26	51,032	51,032	247,528	4.85	asymmetric	circuit simulation
rajat31	4,690,002	4,690,002	20,316,253	4.33	asymmetric	circuit simulation
torso1	116,158	116,158	8,516,500	73.32	asymmetric	2D/3D problem

3.8.4 Results

We benchmarked Desola with different optimisations and code-generation options enabled against MTL and Intel's MKL. All benchmarks were single threaded and used double precision. The sparse matrices were represented using the Compressed Sparse Row storage format in the Desola and IMKL benchmarks and MTL's own internal format in the MTL benchmarks as a MTL CSR implementation was not available.

The version of ITL used was 4.0.0-1. The version of MTL used was 2.1.2-23 of MTL2. Version 11.0 of the Intel C and C++ Compilers was used for both the runtime compiled code generated by our library and the compilation of the MTL benchmarks, respectively. The options passed to the Intel C and C++ compilers are described in Table 3.2. The version of Intel's MKL library was 10.1.1.019.

The two architectures used for benchmarking were:

1. Intel Xeon "Clovertown" processor running at 2.66GHz, 4096 KB L2 cache with 4 GB RAM running 64-bit Ubuntu 8.10.

2. Pentium IV "Prescott" processor running at 3.2GHz with Hyper-threading disabled, 2048 KB L2 cache with 2 GB RAM running 32-bit Ubuntu 8.10.

Our complete benchmark results are presented in Appendix B. We summarize them here for clarity.

We observe that our double for-loop implementation of sparse matrix-vector multiply consistently performs better than our other implementation that uses an inner while loop. For this reason we do not show the for-while based implementation in conjunction with other optimisations.

The high-level fusion optimisation has a significant performance impact on the BiConjugate Gradient and Quasi-Minimal Residual solvers. Both of these solvers are the ones on which matrix-vector multiply fusion provided significant performance improvements in the dense case. We note that this performance improvement is most significant on sparse matrices with high numbers of non-zeros per row. An example of this is illustrated in Figure 3.18.

We have not done analysis as to why the for-for implementation outperforms the for-while one. We note that the variable storing the current matrix row is more susceptible to induction analysis in the for-for implementation. We also note that it is possible the compiler has performed loop unrolling based on the dynamically determined row-lengths in the for-for implementation. This is consistent with the observation that our sparse matrix-vector multiply seems to perform best on matrices with high numbers of non-zeros per row.

Our array contraction also improves performance in the majority of benchmarks and never decreases it. In our dense benchmarks, we theorised that the array contraction optimisation had no observable effect because the execution time was being dominated by the $O(n^2)$ sparse matrix-vector multiply whereas the array contractions affected $O(n)$ vector-vector operations. In our sparse benchmarks, the matrix-vector multiply operations are also $O(n)$, allowing the effects of the contraction optimisations to produce observable effects.

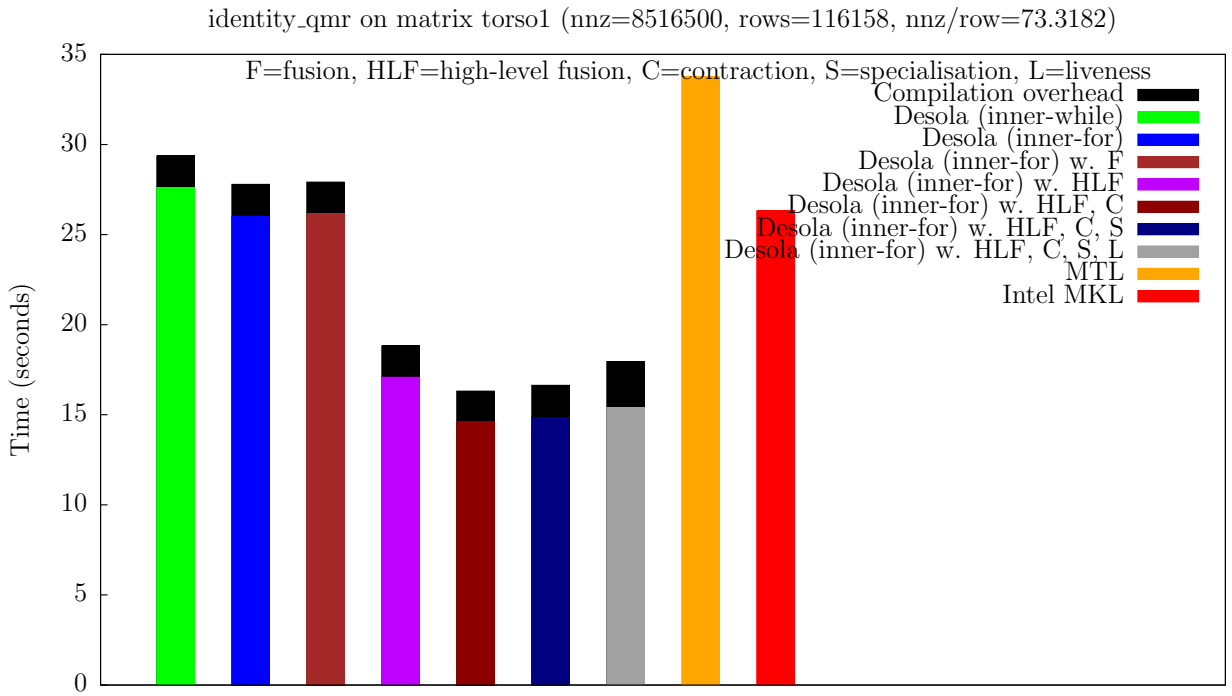


Figure 3.18: Time to execute 256 iterations of the Quasi-Minimal Residual solver with matrix torso1 on architecture 1.

Unfortunately, our row-length specialisation optimisation does not produce a consistent performance impact across our benchmarks and in a number of cases significantly increases the compilation overhead of some of the benchmarks. For this reason, we do not present performance results for this transformation.

Desola's performance in relation to our Intel MKL implementation varies with respect to both the solver and the matrix used. For example, we consistently outperform the IMKL implementation for all benchmarks on matrix ASIC_680k and rajat31. On other matrices, the performance of our solvers are less consistent.

The factors which affect the performance of Desola against the IMKL implementation is unclear. Both matrices ASIC_680k and rajat31 have low (< 5) numbers of non-zeros per row. However, matrix rajat26 also has a low number of non-zeros per row, but has performance similar to that of the IMKL implementation.

3.9 Conclusions and Further Work

We have presented DESOLA, a prototype library that allows the composition and optimisation of arbitrary sequences of kernels. The experimental results from Section 3.7 show that for larger sizes of dense matrices we match or exceed the performance of MTL, and when fusion of matrix operations occurs, we exceed the performance of ATLAS and IMKL. For sparse matrices, our experimental results from Section 3.8.4 show that in some cases we can significantly improve upon the performance of IMKL.

Furthermore, we do this while providing a relatively simple library interface, because by handling kernel composition at runtime, the library user is not required to assist the library by mapping their application onto a specific set of kernels.

Numerical libraries such as BLAS have had to adopt a complex interface to obtain the performance they provide. Libraries such as MTL use unconventional techniques to work around the limitations of conventional libraries to provide both simplicity and performance. The library we developed also uses unconventional techniques, namely delayed evaluation and runtime code generation, to work around these limitations. The effectiveness of this approach provides more compelling evidence towards the benefits of Active Libraries [39].

We have shown how a framework based on delayed evaluation and runtime code generation can achieve high performance on certain sets of applications. We have also shown that this framework permits optimisations such as loop fusion and array contraction to be performed on numerical code where it would not be possible otherwise, due to either compiler limitations or the difficulty of performing these optimisations across procedural boundaries.

We have shown that loop fusion can be an effective technique for optimising both dense and sparse linear algebra. However, the additional improvements provided by our high-level fusion pass suggest that optimising the AST of the generated code is insufficient for high-performance code generation. The loop fusion pass is limited by its ability to fuse what they can prove to be correct but by recognising kernels that use the same pattern of iteration, we can generate fused kernels, regardless of their complexity.

We have also shown the performance improvements array contraction can provide for sparse linear algebra, by removing temporary vectors that would have been difficult or impossible to contract in a BLAS based implementation.

While we have concentrated on the benefits such a framework can provide, we have paid less attention to the situations in which it can perform poorly. The overhead of the delayed evaluation framework, expression DAG caching and matching and runtime compiler invocation will be particularly significant for programs which have a large number of force points, and/or use small sized matrices and vectors. Some of these overheads can be reduced. Methods include:

Persistent compiled code caching This would allow cached compiled code fragments to persist across multiple executions of the same program and avoid compilation overheads on future runs. The extent to which this would benefit performance would be dependent on how likely subsequent program executions were to generate the same expression DAGs and how specialised the generated code fragments were to values such as vector sizes and sparsity patterns. The more specialised the code-fragment to problem-specific values, the less likely it is that it would be able to be reused.

Evaluation using BLAS or static code Evaluation of the delayed expression DAG using BLAS or a statically compiled code variant would allow the overhead of runtime code generation to be avoided when it is believed that runtime code generation would provide no benefit.

Investigation of other applications using numerical linear algebra would be required before the effectiveness of these techniques can be evaluated.

We also note that while we aim for our system to be transparent to the library user, the placement of force points can have a significant effect on performance. As we observed in the Quasi-Minimal Residual solver, force points place a barrier between the operations that may be optimised together, possibly resulting in lost optimisation opportunities. One method to combat this would be to perform speculative evaluation based on detecting repeated evaluated sequences of expressions.

Other plans for future work include:

Client Level Algorithms Currently, all delayed operations correspond to nodes of specific types in the delayed expression DAG. Any library client needing to perform an operation not present in the library would either need to extend it (difficult), or implement it using element level access to the matrices or vectors involved (poor performance). The ability of the client to specify algorithms to be delayed would significantly improve the usefulness of this approach.

Improved Optimisations We implemented restricted methods of loop fusion and array contraction. Improvements to these optimisations or applying others such as skewing or tiling could improve the compiled code's performance further. It could also reduce dependence on the of the quality of the runtime-generated code and effectiveness of the vendor compiler used.

Parallelisation This provides a number of interesting research topics. Loop fusion can inhibit parallelisation when sequential and parallel loops are fused. We need to able to choose when to fuse and when to parallelise taking into account these interactions. In addition, there is the issue of data alignment when parallelisation is considered in a distributed memory setting. Work by Beckmann and Kelly [42] has already investigated these issues in the context of delayed evaluation.

In the following chapters, we continue our exploration of active libraries. We have chosen the domain of finite element solvers for partial differential equations. This domain is a superset of our work on sparse linear algebra. As well as including the solution of sparse linear systems, it requires capture of other domain-specific expressions. We also attempt to extend our abstractions further, incorporating knowledge of iteration into our captured expressions.

Chapter 4

Excafé: An Expression Capturing Finite Element Library

To continue our exploration of active libraries, we needed a domain which would benefit from the abstraction and optimisation abilities that active libraries offer and would also enable us to push our investigation further by requiring new, more complex abstractions and facilitating interesting domain-specific optimisations.

4.1 The Finite Element Method

The domain we chose was the solution of partial differential equations using the finite element method. As a tool for performing this investigation, we have developed an active finite element library in C++ that we call Excafé.

In contrast to Desola which was specifically designed to perform expression capture transparently to the client, Excafé does this explicitly, requiring the client to build an object that represents all the information required to solve a given finite element problem.

We assume familiarity with the finite element method and its implementation. Otherwise, readers are encouraged to consult Sherwin et al. [26] or Logg's paper on automating the finite

element method [34].

Finite element solvers typically solve partial differential equations over some physical domain to find the value of unknown scalar and/or vector fields. Any solution to the equations is approximate and represented by a linear combination of so called *basis functions*. Any approximated field can be represented by a list of coefficients which are used to weight the basis functions before summation. These lists of coefficients are usually stored in vectors and determined as the solution of a sparse linear system of equations.

The system of equations is constructed using the partial differential equations governing the system after being re-written into an appropriate form (including linearisation of non-linear terms if necessary). System matrices and vectors are constructed from *bilinear and linear forms* respectively. Both correspond to descriptions of integrals over the mesh, defined using linear differential operators applied to *basis functions*.

4.2 Our Investigation

Excafé performs expression capture at the level of basis functions, bilinear forms, and matrices and vectors corresponding to discretised operators and tensor fields. Our investigation focusses on two main areas:

Abstractions and Expression Capture

Desola performed expression capture and optimisation transparently to the library client. The technique of building DAGs to represent delayed computations then optimising and executing them on demand worked well, but also demonstrated two important limitations:

1. The on-demand evaluation of expressions that was needed to perform optimisation transparently also meant that Desola never saw a complete representation of the problem being solved. Conditionals on delayed expressions caused the delayed computations to be evaluated since capture of conditionals could not be implemented

transparently. This resulted in research into the development of heuristics to estimate what expression DAG nodes contained live or dead values (Section 3.6).

2. The lack of ability to capture (or represent) loops in DESOLA required the development of low-cost expression DAG comparisons that could be used to detect repeatedly constructed expression DAGs and reuse previously optimised code.

In EXCAFÉ, we have attempted to take expression capture to a higher level. Firstly, we have made expression capture explicit, as only by doing this can we capture a complete view of the finite element problem to be solved. Secondly, we have developed a mechanism for specifying iteration within our captured expressions.

Most importantly, our iteration capture does not work imperatively as in TaskGraph [13] but uses a declarative syntax inspired by mathematical subscript notation. EXCAFÉ infers the loop structure itself and the programmer is freed from manually managing constructs such as cyclic buffers.

Local Assembly Optimisation

Using expressions captured from our basis functions and bilinear forms, we search for domain-specific optimisations for reducing the operation count required to perform local assembly (the process of constructing the small dense matrices that are summed into the large sparse system matrix).

In doing so we build upon existing work on polynomial factorisation [1] by improving the scoring function for factorisations and developing an algorithm to find maximum scoring factorisations whilst also scaling to the large problem sizes we deal with.

Work on local assembly optimisation is still ongoing and has not yet reached that stage where the effectiveness of the approach can be evaluated effectively.

We emphasise that despite being designed to capture enough information to perform code generation for the majority of a finite element solver, EXCAFÉ does not currently perform any code generation. Our work has focussed on the development of EXCAFÉ's expression capture

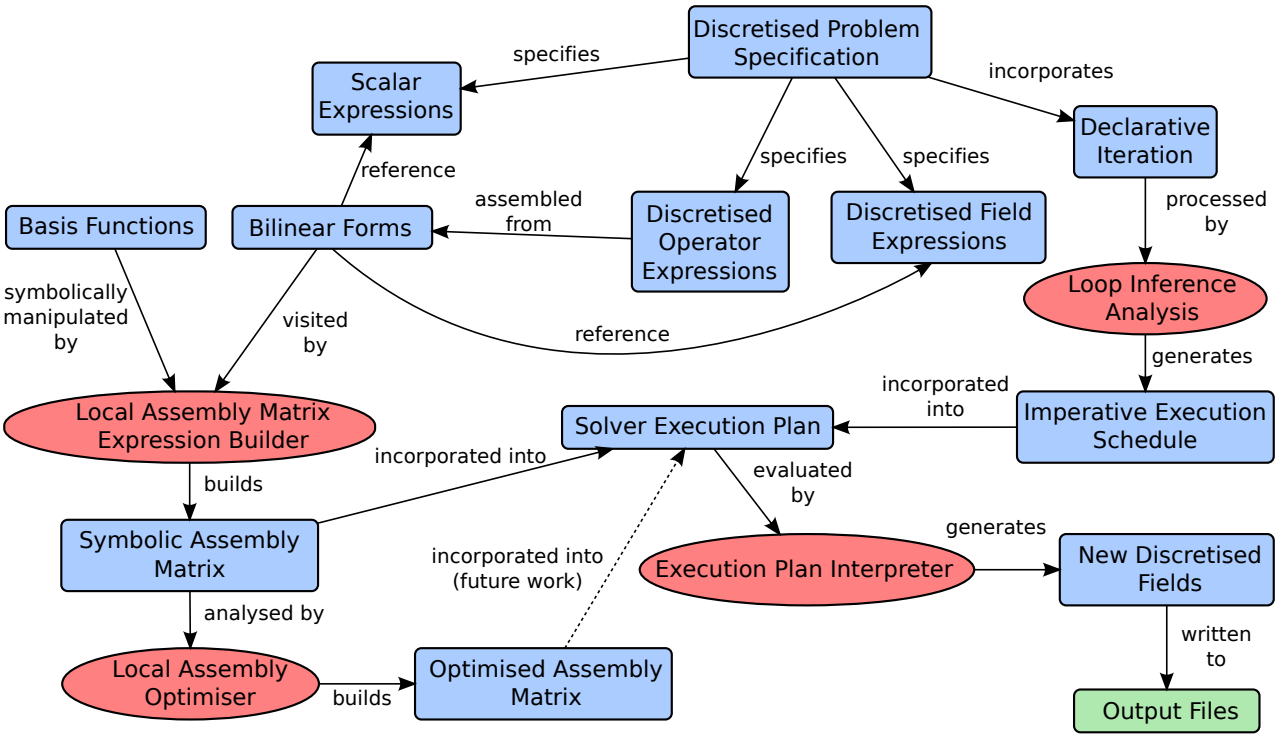


Figure 4.1: Data flow in Excafé. Rectangular nodes represent data being manipulated. Elliptical nodes represent processes that operate on the data. The arcs show the relationships between the data and processes.

capabilities and our local assembly optimisations, the operation count of which we can evaluate without code generation.

In the next section, we provide a high-level overview of how the various expressions captured by Excafé are analysed and manipulated to produce an optimised execution plan.

4.3 Data Flow in Excafé

We present a diagram of high-level data flow in Excafé in Figure 4.1.

The user initially provides a specification of operations between discretised fields, discretised operators and scalars which are used to produce a new state of the discretised system. Details of how the client specifies this are provided in Section 6.3 and details of the data structures involved in Section D.3.

The directed acyclic graph (DAG) of discrete expressions incorporates references to indices

used to declare iteration. The DAG is processed by a loop inference pass that constructs an imperative execution schedule. Details of how the client specifies iteration are provided in Section 6.4. Details of the loop inference analysis are provided in Chapter 7.

Within the discrete expression DAG, it is likely that there will exist operators assembled from the specification of a bilinear form. The mechanism of bilinear form specification is described in Section 6.5 with details of the data structures involved in Section D.4. The bilinear form can reference both fields and scalar values from the discrete expression DAG.

Combining the specification of the bilinear form with the symbolic representation of basis functions allows Excafé to construct a symbolic representation of the scalar expressions contained within the local assembly matrix. How this is performed is described in Chapter 9. Details of the scalar expression representation are provided in Section D.5.

The symbolic representation of the local assembly matrix is subjected to an optimisation pass designed to exploit common subexpressions and factorisation to reduce the cost of performing assembly for each cell. These optimisations are discussed in Chapters 9 and 10.

The imperative execution schedule and the symbolic assembly matrix representation (although not yet the optimised one) are used to evaluate the discrete expression DAG and compute the new state of the discrete system. The data structures used to actually store the values computed by Excafé are described in Section D.2.

In the next section, we describe the current state of Excafé in representing certain aspects and variations of the finite element method.

4.4 Representation Capabilities of Excafé

In this section, I discuss the extent of Excafé's support for representing certain aspects of the finite element method. I also describe what would be required to extend Excafé to support the aspects it currently cannot handle.

Boundary Conditions

Dirichlet boundary conditions are currently specified by attaching constant tensor values to mesh facets. This could easily be extended to non-constant values by attaching position-dependent functions.

Currently, boundary conditions must be represented by a discretised field i.e. they must be discretised using a finite element approximation. Although we have not implemented the specification of boundary conditions using data from a file, it would be possible to construct a discretised field in this manner.

We have not yet considered the case of when boundary conditions are specified using time-dependent functions or time-dependent data from a file. Both these cases typically require that quadrature be used for efficiency (as discretising the function each time-step would be too expensive). This would require an extension of our symbolic representation to represent call-backs to the source of the boundary condition data.

Neumann boundary conditions are always incorporated as boundary integrals which are described next.

Boundary Integrals

We permit specification of boundary integrals, although support for them is incomplete in Excafé. This was primarily due to time constraints since handling edge-normal and Jacobian calculations in a symbolic manner is more complex in the case of facet integrals.

Discontinuous Elements

Excafé does not currently support discontinuous elements. In order to do so would need to extend our symbolic representation of assembly to handle referencing degrees of freedom associated with neighbouring elements.

True Tensor-Valued Elements

Although all currently implemented tensor-valued elements in Excafé are implemented by repeating scalar basis functions, Excafé supports true tensor-valued elements.

Constructing tensor-valued basis functions by repeating scalar-valued ones is sometimes hard-coded into finite element libraries in order to exploit sparsity properties. Excafé's

symbolic representation naturally specialises to these cases so there is no need for this restriction.

Higher-Degree Basis Functions

Excafé currently implements linear and quadratic polynomial basis functions. However, Excafé can represent any basis function that can be expressed symbolically in its expression capture framework. Experimentation has been performed with different symbolic representation back-ends including a GiNaC [45] based one, and an alternative internal representation that is restricted to rational functions. Since Excafé needs to be able to perform symbolic integration of these functions, arbitrary order polynomials are the best supported (and most basis functions can be approximated by them). If using the GiNaC based representation, it would also be possible to directly represent trigonometric based ones (e.g. sin, cos).

Curved Elements

Excafé currently uses a custom data structure to hold mesh geometry information and uses a linear basis to interpolate global co-ordinates. However, Excafé's representation would permit an arbitrary vector field to be used to describe the mesh geometry, facilitating curved elements.

However, we do note that curved elements result in the need to be able to perform integrals of expressions that are no longer polynomial in the local co-ordinate system. This may be non-trivial to do symbolically, meaning that it may be necessary to resort back to quadrature based approaches. This in turn could hinder our ability to optimise expressions.

We also note that it is often beneficial to only use curved elements on mesh boundaries, and use linear elements elsewhere. We have not yet considered how to handle this in Excafé.

4.5 Overview of the Excafé Chapters

The chapters on Excafé are organised as follows:

In Chapter 5 we present a heat solver implemented using Excafé. This serves as an introduction to the library and its client visible expression capture mechanisms.

In Chapter 6 we provide a description of Excafé's expression capture facilities and discuss what information this provides to the library. In particular, we describe Excafé's declarative loop syntax that we use to perform iteration within a solver step.

In Chapter 7 we describe how we may infer an imperative loop structure and nesting using a specification captured as a DAG.

In Chapter 8 we describe the implementation of an incompressible Navier-Stokes solver demonstrating Excafé solving a non-trivial problem. It incorporates many different aspects including:

1. Linearisation of a term using declarative loop syntax to perform Picard iteration within each time-step.
2. Global assembly matrices constructed from multiple bilinear forms.
3. Use of composite function spaces and fields and the *projection* operator to extend and restrict those fields.
4. Multiple assemblies within a time-step. Assembly is performed to construct both the LHS of a linear system, and an operator used to transform a field to produce the RHS.

The local assembly matrices produced by our incompressible Navier-Stokes solver also serve as the basis on which we are evaluating our assembly optimisations.

In Chapter 9 we describe how Excafé constructs its representation of a local assembly matrix from the information previously captured. We discuss our approach to optimising evaluation and compare it against the optimisations developed in the FEniCS Form Compiler [38].

In Chapter 10 we discuss our improvements to the factorisation algorithm presented by Hosangadi et al. [1] and the construction of an algorithm to search for optimal matrix coverings utilising properties of bipartite graphs.

Chapter 5

Heat Solver Example in Excafé

In order to explore the optimisation opportunities present in finite element method implementations, I constructed an active finite element library in C++ named Excafé. In this chapter, I describe how a finite element problem can be specified and solved using Excafé.

Many of the design decisions in Excafé are similar to those in other finite element libraries. For this reason, they are detailed in Appendix C. This allows us to concentrate on the more important aspects of the library in this chapter.

5.1 A Simple Heat Conduction Problem

To illustrate the library, we show how to specify and solve a simple test problem. We choose a time-dependent heat conduction problem, the layout of which is shown in Figure 5.1.

Excafé includes an interface to the triangle library [46] which can be used to construct simple two-dimensional meshes. We construct the mesh and central object with the code in Figure 5.2. The central diamond is associated with the label 5. This label will be used to identify its edges when applying Dirichlet boundary conditions.

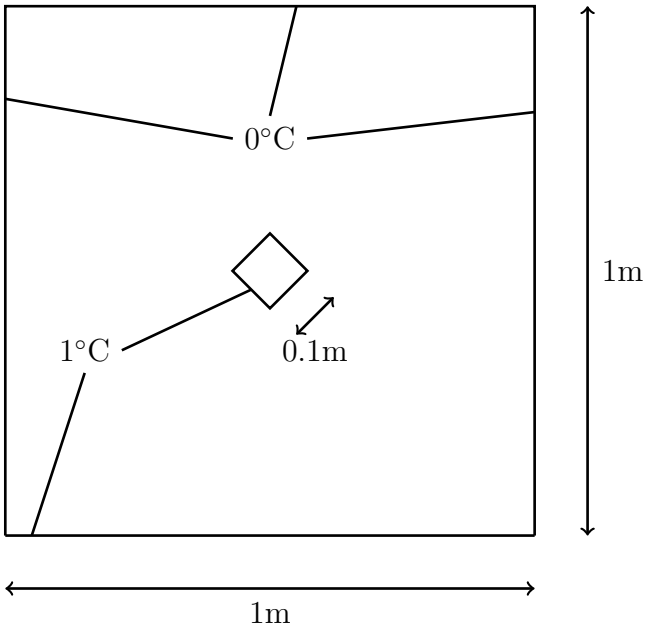


Figure 5.1: We specify a temperature of 0°C on the top, left and right boundaries and a temperature of 1°C on the bottom edge and the central diamond.

```
static const std::size_t dimension = TriangularMeshBuilder::cell_dimension;
static const double maxCellArea = 1.0/5000;
static const double polySize = 0.1;
static const std::size_t polyEdges = 4;
static const double polyLabel = 5;
static const double polyRotation = 0.0;

TriangularMeshBuilder meshBuilder(1.0, 1.0, maxCellArea);
const Polygon poly(vertex<2>(0.5, 0.5), polyEdges, polySize, polyRotation);
meshBuilder.addPolygon(poly, polyLabel);
Mesh<dimension> mesh(meshBuilder.buildMesh());
```

Figure 5.2: The construction of the discretised domain used for out heat solver problem in Excafé. The mesh builder is instructed to place a 4-sided polygon in the centre of the domain and label its edges with the value 5.

5.2 Discretisation of the Heat Equation

In order to solve our problem using the finite element method, we must take the partial differential equation describing heat conduction and re-write in a form suitable for the finite element method. We start with the heat equation with no external source:

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0 \tag{5.1}$$

where:

u is the temperature at some point.

$\frac{\partial u}{\partial t}$ represents the rate-of-change of temperature at some point w.r.t. time.

α is a scalar representing thermal diffusivity.

∇^2 is the Laplacian operator and represents the divergence of the gradient or $\nabla \cdot \nabla$.

We assume that we are solving this problem in a bounded domain Ω. In order to solve this problem using the finite element method, we need to convert it to the "weak form". We multiply with a function ϕ (the test function) and integrate over the domain Ω. We assume ϕ is zero on the boundaries on which we apply Dirichlet conditions.

$$\int_\Omega (\frac{\partial u}{\partial t} - \alpha \nabla^2 u)\phi \, dx = 0 \tag{5.2}$$

We discretise in time using the backward Euler method and introduce u^n and u^{n-1} to refer to the heat field at the current and previous time-steps, respectively:

$$\frac{1}{k}\int_\Omega (u^n - u^{n-1})\phi \, dx - \int_\Omega \alpha \nabla^2 u^n \phi \, dx = 0 \tag{5.3}$$

We apply integration by parts to remove the Laplacian. This is required because our approximation of u is of differentiability class C^0, that is, the derivative may be discontinuous so we

cannot take the second derivative.

$$\frac{1}{k}\int_\Omega (u^n - u^{n-1})\phi\, dx + \int_\Omega \alpha \nabla u^n \cdot \nabla \phi\, dx - \int_{\partial\Omega} \alpha(\nabla u^n \cdot n)\phi\, dS = 0 \tag{5.4}$$

This introduces a boundary integral. Here, $\partial\Omega$ is the boundary of our domain over which we integrate this term, and n is the outward pointing surface normal. To solve these equations, we need to re-arrange into a form where the left-hand side is a bilinear form in u and ϕ, and the right-hand side is a linear form in ϕ.

$$\int_\Omega u^n\phi\, dx + k\int_\Omega \alpha \nabla u^n \cdot \nabla \phi\, dx - k\int_{\partial\Omega} \alpha(\nabla u^n \cdot n)\phi\, dS = \int_\Omega u^{n-1}\phi\, dx \tag{5.5}$$

Since we have no non-Dirichlet boundary conditions and we assume ϕ is zero on all Dirichlet boundaries, our edge integral vanishes.

$$\int_\Omega u^n\phi\, dx + k\int_\Omega \alpha \nabla u^n \cdot \nabla \phi\, dx = \int_\Omega u^{n-1}\phi\, dx \tag{5.6}$$

We use the finite element discretisation to form a linear system of equations. The linear form on the RHS becomes a vector, and the bilinear form on the LHS becomes the application of an operator to our unknown u^n. This operator is represented by a sparse matrix. In discretised form, we can write our problem as:

$$M\mathbf{u}^n + k\alpha A\mathbf{u}^n = M\mathbf{u}^{n-1} \tag{5.7}$$

We solve this system of equations (with appropriate boundary conditions) each time-step of our simulation. Next we describe how we specify this problem to EXCAFÉ.

```
// Define scenario over the given mesh
Scenario<dimension> scenario(mesh);

// Define the basis we will use for our temperature field
Element temperature = scenario.addElement(new LagrangeTriangleLinear<0>());

// Define the discretisation of our temperature field
FunctionSpace temperatureSpace = scenario.defineFunctionSpace(temperature, mesh);

// Add the temperature field
temperatureField = scenario.defineNamedField("temperature", temperatureSpace);
```

Figure 5.3: The construction of an Excafé `Scenario` object. The finite elements and function spaces used in the solver are registered with the `Scenario` to provide handles that will be used during expression capture. Fields that are persistent state of the problem (in this example, the temperature field) must also be specified.

5.3 Specifying the Problem Context to Excafé

In order to solve any finite element problem in Excafé, we need to construct a `Scenario` object. The `Scenario` object contains references to the mesh, function spaces and finite elements which describe how the problem has been discretised. Any Dirichlet boundary conditions to be used must also be added to the `Scenario` object. Additionally, it contains discretised tensor fields that constitute the state of the simulation (persistent fields) and expressions to calculate the next state from the existing values.

In Figure 5.3 we show the code to construct the `Scenario` object for a given mesh. After construction, the finite elements, function spaces and persistent fields of the problem are added to the scenario.

We also need to register the Dirichlet boundary conditions with the `Scenario` object. We show the code to do this in Figure 5.4. The `BoundaryConditionList` class allows us to specify a prioritised list of boundary conditions. The `BoundaryConditionTrivial` class allows us to specify a constant tensor value that will be applied to all facets of our mesh with a specified label. Here, the edges of our domain have been labelled 1 to 4, and `polyLabel` is the label we gave to the object at the centre of our mesh.

```
Tensor<dimension> cold(0);
cold = 0.0;

Tensor<dimension> hot(0);
hot = 1.0;

BoundaryConditionList<dimension> boundaryConditionList(0);
boundaryConditionList.add(BoundaryConditionTrivial<dimension>(1, hot));
boundaryConditionList.add(BoundaryConditionTrivial<dimension>(2, cold));
boundaryConditionList.add(BoundaryConditionTrivial<dimension>(3, cold));
boundaryConditionList.add(BoundaryConditionTrivial<dimension>(4, cold));
boundaryConditionList.add(BoundaryConditionTrivial<dimension>(polyLabel, hot));

BoundaryCondition boundaryCondition =
  scenario.addBoundaryCondition(temperatureSpace, boundaryConditionList);
```

Figure 5.4: The registration of boundary conditions with an Excafé `Scenario` object. Constant tensor values are associated with integer valued labels used to identify facets on the boundary of the domain.

5.4 Specifying the Solution Steps to Excafé

Now we have specified the problem context to Excafé, we need to tell it how to advance the system by a time-step. As Excafé is an expression capture library, the code written by the user is only executed once when building the description. Excafé is then free to implement the solution steps in whatever way it sees fit.

Specification of the evolution of a `Scenario` is done through the construction of a `SolveOperation` object. We show the code to do this in Figure 5.5. The `SolveOperation` object associates persistent fields with a discrete expression DAGs used to compute their value in the next step of the simulation.

Our syntax for specifying bilinear forms is similar to that of the Unified Form Language [37]. However, we do not explicitly denote basis functions as trial or test. Instead, we use the `B()` method to construct the inner product[1] between the trial and test parts of the form. The basis functions used to form the first and second parameters to `B()` become the trial and test spaces, respectively, for that bilinear form.

[1]This may be a scalar multiplication, inner product or double inner product depending on the rank of the operands. The result is always a scalar.

```
SolveOperation solve = scenario.newSolveOperation();

// Define alpha and k.
Scalar alpha = 1e-4;
Scalar k  = 10.0;

// Construct the mass matrix.
Operator massMatrix(temperatureSpace, temperatureSpace);
massMatrix = B(temperature, temperature)*dx;

// Apply mass matrix to temperature field from previous time-step to transform
// to test-space.
Field rhs = massMatrix * temperatureField;

// Define the LHS bilinear form.
const BilinearFormIntegralSum lhsForm =
  B(temperature, temperature)*dx +
  B(alpha*k*grad(temperature), grad(temperature))*dx;

// Construct a linear system.
LinearSystem system =
  assembleGalerkinSystem(temperatureSpace,  // Function space
                         lhsForm,           // Linear operator
                         rhs,               // Field in test-space
                         boundaryCondition, // Boundary conditions
                         temperatureField); // Initial guess

// Define new value of temperature field.
solve.setNewValue(temperatureField, system.getSolution());

// Signal that construction of the SolveOperation object is complete and
// analyses can be applied.
solve.finish();
```

Figure 5.5: Construction of a `SolveOperation` object in Excafé. We explicitly construct the mass matrix and multiply it with the previous temperature field to obtain the RHS of our linear system. We do not explicitly construct the system matrix. This is handled by the method `assembleGalerkinSystem` which takes the bilinear form used to assemble the system matrix, as well as the boundary conditions to apply to it and the RHS vector.

```
for(int i=0; i<200; ++i)
{
  std::cout << "Starting timestep " << i << "..." << std::endl;
  solve.execute();
  std::ostringstream filename;
  filename << "./heat_" << boost::format("%|04|") % i << ".vtk";
  solve.outputFieldsToFile(filename.str());
}
```

Figure 5.6: The loop used to drive our heat solver simulation. It repeatedly calls the `SolveOperation` object to advance the state of the discretised system then outputs the fields to a VTK file.

We also use UFL syntax for specifying the region over which a bilinear form should be integrated. An integral over the domain interior is denoted by a multiplication by `dx`. Exterior and interior facet integrals are denoted by multiplications by `ds` and `dS` respectively. Currently, only cell integrals are fully implemented in Excafé.

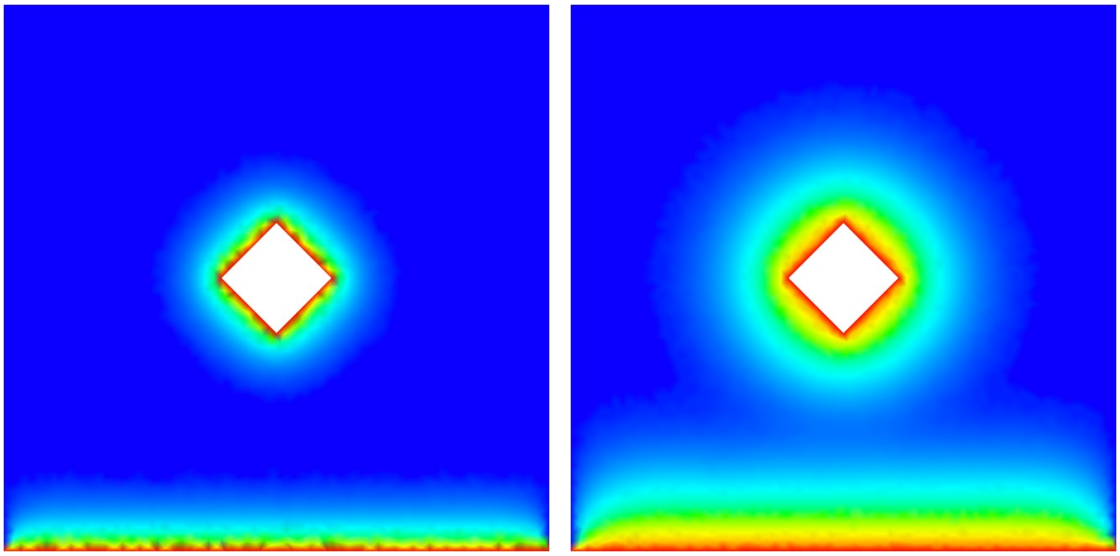
5.5 Executing the solver

Excafé does not yet implement capture of the time-stepping loop so we use a standard *for* loop to drive time-stepping. We show the code to do this in Figure 5.6.

Excafé provides a syntax that permits nested iteration to be specified within a given time-step using a declarative notation. This is described in Section 6.4. The time-stepping loop could be described with our declarative iteration syntax, however, we have not yet considered issues such as how to specify when fields should be written to a file, or how to generate useful diagnostic output from within a declaratively declared loop.

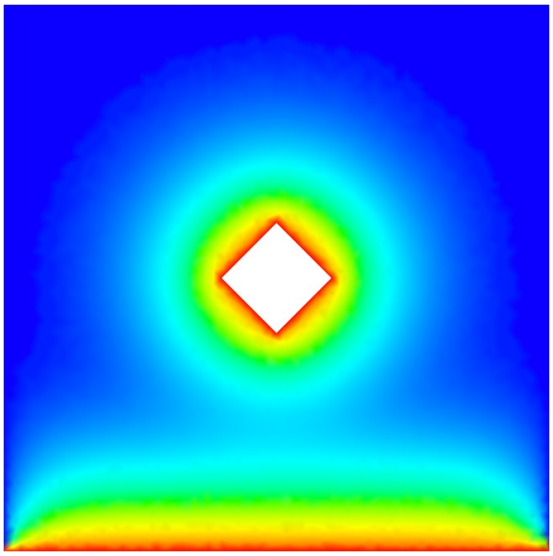
5.6 Generated Output

We show the scalar temperature fields generated by solving our problem in Figure 5.7. The complete source of the heat solver example is provided in Appendix E.

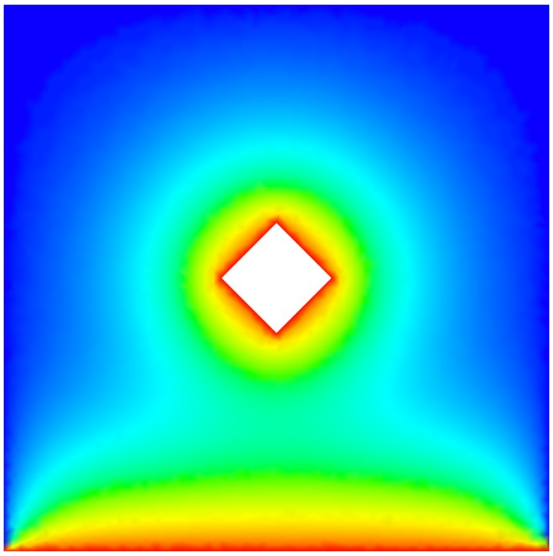


(a) Time-step 0 (0 seconds)

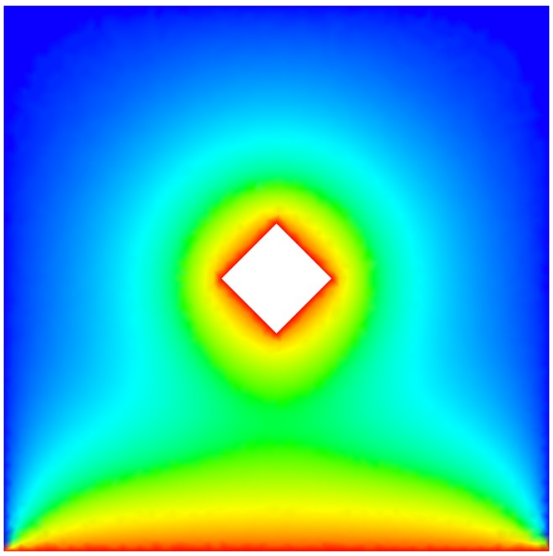
(b) Time-step 5 (50 seconds)



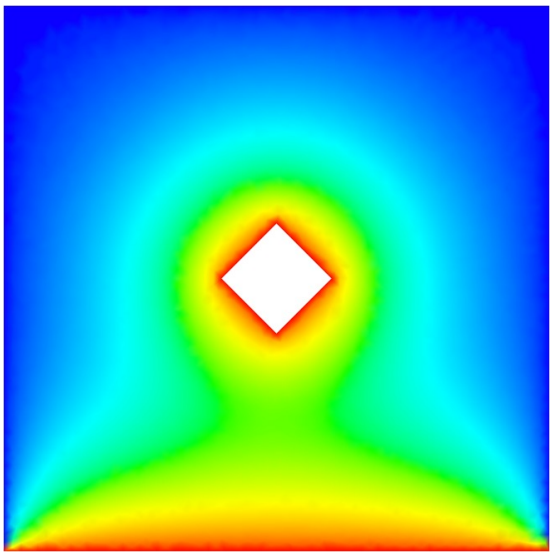
(c) Time-step 10 (100 seconds)



(d) Time-step 20 (200 seconds)



(e) Time-step 30 (300 seconds)



(f) Time-step 50 (500 seconds)

Figure 5.7: Our example heat conduction problem at different time-steps. Thermal diffusivity $\alpha = 10^{-4} m^2 s^{-1}$ and time-step duration $k = 10s$.

5.7 Conclusion

In this chapter, we have shown how to specify and solve problems using Excafé. In doing so, we have demonstrated Excafé's expression capture capabilities applied to variational forms and operations between discretised fields and operators. In the next chapter, we cover Excafé's expression capture in more detail, looking at the types of optimisations that we may be able to perform with the captured representations.

Chapter 6

Expression Capture in Excafé

In this chapter, we describe Excafé's expression capture in more detail. In particular, we describe how capture is performed, the types of optimisations we hope to apply and the advantages and limitations of our approach. We also compare our approach with that taken in the FEniCS Form Compiler [36], a code-generator for variational forms that incorporates research into the optimisations of local matrix assembly. Further information on the data structures used for our expression capture can be found in Appendix D.

6.1 Overview

The finite element method requires that we can rewrite our problem into the following form:

$$a(u, v) = L(v) \tag{6.1}$$

where a is linear in u and v and L is a linear in v. a and L are called bilinear and linear forms, respectively. They are both variational forms. In the finite element method, the LHS and RHS are integrals over our mesh. For example, for our heat conduction problem from Chapter 5, we let:

$$a(u,\phi) = \int_\Omega u\phi\,dx + k\int_\Omega \alpha\nabla u\cdot\nabla\phi\,dx$$
$$L(\phi) = \int_\Omega u^{n-1}\phi\,dx$$

this description, combined with boundary conditions and discretisation choices for u and ϕ is sufficient information to construct the discretised system:

$$A\mathbf{x} = \mathbf{b}$$

Matrix A is constructed from the bilinear form, and due to the finite-element discretisation, is sparse. Vector $\mathbf{b}$ can be assembled from the linear form. $\mathbf{x}$ (the discretised value of u) can be found using a linear system solver. As the basis functions are cell-local, the matrix A is sparse. For small systems a direct solver can be applied, but for larger ones Krylov subspace methods [28] are typically used as these avoid matrix fill-in.

Even though sparse linear algebra is invariably required during the solution of a finite element problem, there is no need for the solution specification to explicitly refer to the discrete representations of the field and operators i.e. the vectors and sparse matrices involved. Despite this, EXCAFÉ provides distinct levels of expression capture for:

Variational forms We provide a syntax for expressing variational forms in a manner similar to the Unified Form Language [37]. Variational forms describe integrals over cells used to construct our discretised fields and operators.

Discrete Fields and Operators We allow specification of operations between discrete fields and operators. Naturally, the syntax is similar to that of a linear algebra library.

We provide these distinct levels of expression capture for two reasons:

1. It may still be necessary for the library user to specify parameters that directly affect the discretised operations. These include the tolerances for the linear solvers, preconditioning methods and other aspects that require direct knowledge of the linear system.

2. The majority of finite element libraries treat the specification of linear and bilinear forms as a means to performing assembly and not as a subset of a mathematical expression for the entire problem. As Excafé developed from a more conventional (non-expression capture) implementation, it also inherited this distinction.

Hence, although we make a distinction between expression capture of variational forms and discrete operations we expect that these two syntaxes can be unified, with the discrete level less apparent to the library client.

6.2 Problem Context Construction

In order to solve a problem in Excafé (as demonstrated in Chapter 5), we first need to specify the physical domain, the fields we wish to solve for and our choice of discretisation for those fields.

The `Scenario` class stores all state related to a particular finite element problem as well as descriptions of the steps required to solve or advance the problem. `Scenario` instances are templated by the physical dimension of the problem, and are constructed using a supplied `Mesh` instance that will be used for spatial discretisation. An example of a declaration of a `Scenario` over a mesh is given in Figure 6.1.

6.2.1 Specifying Tensor Field Discretisation

Next we need to define the fields that we wish to solve for. However, before we can do this, we must choose the basis functions used to approximate the fields, and the sub-domain of the mesh that the fields' function spaces will cover.

```
// A 2D problem
static const std::size_t dimension = 2;

// Define a mesh over the unit square with a maximum cell area of 0.01
TriangularMeshBuilder meshBuilder(1.0, 1.0, 0.01);
Mesh<dimension> mesh(meshBuilder.buildMesh());

// Declare finite element problem over the mesh
Scenario<dimension> scenario(mesh);
```

Figure 6.1: Declaration of the context for a 2D finite element problem.

```
Element velocity = scenario.addElement(new LagrangeTriangleQuadratic<1>());
Element pressure = scenario.addElement(new LagrangeTriangleLinear<0>());

FunctionSpace velocitySpace = scenario.defineFunctionSpace(velocity, mesh);
FunctionSpace pressureSpace = scenario.defineFunctionSpace(pressure, mesh);
FunctionSpace coupledSpace = velocitySpace + pressureSpace;
```

Figure 6.2: Declaring a coupled velocity-pressure function space using a linear scalar pressure space and quadratic vector velocity space. The field ranks are specified through the template parameters used to instantiate the basis function classes.

Excafé currently implements linear and quadratic basis functions over 2D triangular cells with the `LagrangeTriangleLinear` and `LagrangeTriangleQuadratic` types, both which implement the `FiniteElement` interface. Both types are templated by the rank of the tensor field they are approximating. Adding user-defined basis functions can be achieved by providing other implementations of the `FiniteElement` interface.

To avoid mixing implementation with expression capture types, classes implementing the `FiniteElement` interface must be instantiated then added to the `Scenario` using the `addElement` method. The `Scenario` returns an `Element` object that can then be used during expression capture. Registration of basis functions with a `Scenario` is shown in Figure 6.2.

The `FunctionSpace` type is used to reference discrete finite element function spaces. They are constructed from a choice of finite element basis and a subset of the physical domain. Hence, they correspond to the collection of functions that can be represented as a linear combination of the basis-functions located in a sub-domain of our mesh.

We permit addition and subtraction of `FunctionSpace` instances (the latter only works as our

```
NamedField velocityField = scenario.defineNamedField("velocity", velocitySpace);
NamedField pressureField = scenario.defineNamedField("pressure", pressureSpace);
```

Figure 6.3: Declaring the desired fields to solve for to a `Scenario`. Each field is discretised using the supplied `FunctionSpace` and becomes persistent state of the `Scenario`.

function spaces are discretised). Thus, we can construct composite and disjoint function spaces. These can be used for:

Coupled pressure-velocity solvers We can construct a function space that represents the various possible joint pressure-velocity fields.

Boundary condition application When applying boundary conditions, we can form disjoint function spaces corresponding to the fixed (Dirichlet) and non-fixed (homogeneous) degrees of freedom. This enables boundary conditions to be applied in a mathematically elegant form. However, we do not currently do this in Excafé since applying boundary conditions directly to the system matrix is more convenient from an implementation perspective.

`FunctionSpace` construction and addition is shown in Figure 6.2.

As a `FunctionSpace` completely describes the discretisation of a field, it is used in the definition of the fields we wish to solve for (persistent fields). Persistent fields are referenced using the `NamedField` type as shown in Figure 6.3. Each `NamedField` becomes part of the state of the `Scenario` (as opposed to temporary fields calculating during a solve). Persistent fields are named so they can be identified when written to an output file.

6.2.2 Boundary Conditions

In order for our problems to be well-posed, we need to be able to specify boundary conditions.

Neumann boundary conditions (restrictions on the normal component of the gradient of a field) can be incorporated in the finite element formulation as additional integrals over the boundaries of the physical domain and do not require additional library support.

```
// Define a couple of rank-0 (scalar-valued) tensors
Tensor<dimension> one(0), zero(0);
one  = 1.0;
zero = 0.0

// Set a scalar-field to 0.0 on mesh facets labelled 1 and to 1.0
// on mesh facets labelled 2.
BoundaryConditionList<dimension> boundaryConditionList(0);
boundaryConditionList.add(BoundaryConditionTrivial<dimension>(1, zero));
boundaryConditionList.add(BoundaryConditionTrivial<dimension>(2, one));

// Construct a handle to be used during expression capture
BoundaryCondition bc =
  scenario.addBoundaryCondition(temperatureSpace, boundaryConditionList);
```

Figure 6.4: Construction of Dirichlet boundary conditions attached to certain mesh facets for a scalar field in Excafé.

Dirichlet boundary conditions (specification of the value of a field) constrain specific degrees of freedom and require library support in order to define and apply. Our current implementation stores Dirichlet boundary conditions as a field only defined on edges where boundary conditions have been applied. During mesh construction, the generator attaches integer valued labels to the different facets in our mesh. We provide a simple means to associate constant tensor values to facets with a particular label.

The library client constructs a priority ordered list of tensor field values and the label of the facets they should be attached to. When registered with the `Scenario` object, all boundary values are evaluated and a `BoundaryCondition` object is returned that can be used to reference them in Excafé captured expressions. We show the construction of Dirichlet boundary conditions and the returned handle to them in Figure 6.4.

We describe how the `BoundaryCondition` handle can be used to apply Dirichlet boundary conditions when solving for a field in the next section.

6.3 Discrete Expression Capture

The previous section described how to define a discretised physical domain, and the fields to be represented over that domain. In this section, I describe how the steps required to solve or advance the problem state provided to Excafé. The majority of Excafé's expression capture occurs during this phase. Additional information on the data structures built can be found in Section D.3.

As mentioned in Section 6.1, Excafé provides separate levels of expression capture for discrete operations and for variational forms. We describe discrete expression capture first.

6.3.1 Handle Types

Discretised tensor fields and operators can be represented by vectors and sparse matrices, respectively. Our strategy for expression capture is similar to that used in Desola for linear-algebra expression capture (i.e. we build a DAG representing the computation to be executed by providing handle types to the library client to manipulate). We define three main handle types.

Scalar

The `Scalar` type represents references to scalar values that we compute during the execution of a solve. They are distinct scalar values such as parameters of our PDEs or the residual of a linear solve. They are unrelated to scalar-fields which are scalar values defined over our physical domain.

Using C++ operator overloading, we provide addition, subtraction, division and multiplication as well as comparison operators. We show valid syntax in Figure 6.5. We have not defined a boolean type in our discrete expression capture, so scalar comparisons return a zero or non-zero scalar to represent false or true, respectively.

```
Scalar a = 5.0; Scalar b = 2.0;
a=b;

// Arithmetic operations
a+=b; a-=b; a/=b; a*=b;
a+b;  a-b;  a/b;  a*b;

// Comparison operations
a<b; a<=b; a==b; a>b; a>b;
```

Figure 6.5: Valid operations on handles to scalar expressions.

```
// Declaring zero valued fields
Field pressureA(pressureSpace), pressureB(pressureSpace);

// Valid Expressions
pressureA+pressureB; pressureA+=pressureB;
pressureA-pressureB; pressureA-=pressureB;

// Combining a pressure and velocity field
Field composite = project(velocityField, coupledSpace) +
  project(pressureField, coupledSpace);

// Extracting a velocity field
Field extracted = project(composite, velocitySpace);
```

Figure 6.6: Discretised fields can be added and subtracted if they are defined on the same function space. The `project` function can be used to perform sub-field extraction and create composite fields.

Field

The `Field` type provides a handle to tensor fields defined over the mesh. All `Field`s possess a reference to a `FunctionSpace` which determines how they are discretised. Fields can be added and subtracted provided that they have the same function space. We also provide a `project` function (quite possibly a misnomer) that given a new function space, will copy the subset of the field that shares the same basis and spatial discretisation into a field with that function space. In a coupled Navier-Stokes solver for example, we can use this to extract the velocity and pressure fields from a coupled solution.

Figure 6.6 shows valid field expressions, the construction of a composite field and extraction of a sub-field.

```
// Define trial and test elements and function spaces
Element trial = scenario.addElement(new LagrangeTriangleLinear<0>());
Element test = scenario.addElement(new LagrangeTriangleLinear<0>());

FunctionSpace trialSpace = scenario.defineFunctionSpace(trial, mesh);
FunctionSpace testSpace = scenario.defineFunctionSpace(test, mesh);

// Construction of a mass matrix
Operator massMatrix(trialSpace, testSpace);
massMatrix = B(trial, test)*dx;

// Operator application
Field trialField(trialSpace);
Field testField = massMatrix * trialField;
```

Figure 6.7: Construction of the mass matrix and application to a field in Excafé. In this example, we have constructed different function spaces to represent the trial and test spaces. However, it is possible to use the same space for both as we do not require that function spaces be designated as trial or test.

Operator

The `Operator` type is a handle to a discretised linear operator. Typically, a discretised operator is represented as a sparse matrix. Each `Operator` possesses a reference to two `FunctionSpace`s. The first is the `FunctionSpace` of the `Field` the `Operator` can be used to transform. The second is the `FunctionSpace` of the result `Field`. In the finite element method, these are typically the trial and test spaces. By associating this information with the matrix, we can verify the operator isn't used in invalid ways during expression capture.

Most importantly, `Operator` handles can have bilinear forms assigned to them which causes the operator to be populated with the result of the assembly. We show the syntax for assembling and application of an `Operator` in Figure 6.7. As mentioned in Section 5.3, we borrow UFL's syntax for specifying the mesh region to integrate over and provide the `B()` method to declare a bilinear form where the first and second parameters are functions in the trial and test spaces respectively.

In addition to the `Scalar`, `Field` and `Operator` expression handle types, we provide a helper class to hide some of the details of solving our discretised system.

6.3.2 Linear System Solution

We provide a `LinearSystem` type that represents a linear system solver. Currently, construction is only possible through a call to the `assembleGalerkinSystem` function. The constructed `LinearSystem` transparently generates expression DAG nodes that represent the following:

1. An system matrix assembled from a bilinear form.
2. The system matrix with boundary conditions applied.
3. The system RHS with boundary conditions applied.
4. The result field (currently calculated using a PETSc linear system solver).

We do this partly for simplicity and also because we do not wish to directly expose the operators we use for applying boundary conditions to a linear system.

The operators we generate do not correspond to standard algebraic operations and may change in future. Currently, we apply boundary conditions in the following manner:

1. Zero rows in the system matrix corresponding to constrained degrees of freedom, except for the diagonals which are set to one.
2. Update values in the RHS vector to the boundary condition values.

We do however expose mechanisms for accessing the modified system matrix and RHS for the purposes of calculating the residual. We show the construction and use of a `LinearSystem` instance in Figure 6.8.

6.3.3 Registering Solution Steps

The discrete expression DAGs we construct must be registered with our `Scenario` object. We do this through the construction of a `SolveOperation` object, which is a handle to a sequence

```
LinearSystem system = assembleGalerkinSystem(
  functionSpace,      // The function space of the field being transfomed and
                      // the result. We assume that they are both the same in
                      // the Galerkin case.

  lhs,                // The bilinear form to be assembled.

  load,               // The RHS of the system (the desired field after
                      // application of the operator).

  boundaryConditions, // Dirichlet boundary conditions to apply to the system.

  initialGuess);      // A field representing initial guess at the solution of
                      // the system.

// Retriving the solution to the system
Field solution = system.getSolution();

// Retrieving the operator with boundary conditions applied.
Operator modifiedSystem = system.getConstrainedSystem();

// Retrieving the RHS with boundary conditions applied.
Field modifiedRHS = system.getConstrainedLoad();

// Calculating the residual
Scalar residual = (modifiedSystem*solution - modifiedRHS).two_norm();
```

Figure 6.8: Solution of for a linear operator with Dirichlet boundary conditions in Excafé. It is possible to retrieve the LHS operator and RHS field with boundary conditions applied, so the linear system's residual can be determined.

```
FunctionSpace functionSpace = scenario.defineFunctionSpace(element, mesh);
NamedField field = scenario.defineNamedField("field", functionSpace);
SolveOperation solve = scenario.newSolveOperation();

Operator op(functionSpace, functionSpace);
op += B(element, element)*dx;

Field newField = op * field;

// Define new value of temperature field.
solve.setNewValue(field, newField);

// Complete construction of SolveOperation.
solve.finish();
```

Figure 6.9: Creation of a `SolveOperation` (with little practical application) that updates a field by multiplying it by the mass-matrix.

of operations that solve (or advance by a time-step) the finite element problem. We support multiple `SolveOperation` instances to account for that possibility of needing to advance the simulation in different ways (e.g. progressing the system towards an acceptable initial state before performing solution steps).

After constructing a `SolveOperation` object using a given `Scenario`, we associate each `NamedField` with an expression DAG that describes how to calculate its new value using the `setNewValueMethod`. Finally, the `finish` method must be called on the `SolveOperation` in order to initiate analysis and optimisation. A trivial example demonstrating syntax is shown in Figure 6.9.

6.4 Declarative Iteration Capture

In the last section, we described how a DAG was built in order to represent the steps required to perform a solution step for a given finite element problem. This was sufficient for our heat equation. However, for more complex problems, we may need to be able to perform iteration within a time-step.

Excafé provides a mechanism to perform expression capture of iteration using a declarative syntax. Before describing this we present a motivating example, solving the non-linear convective acceleration term of the Navier-Stokes equations.

6.4.1 Linearising the Convective Acceleration Term of Navier-Stokes

We consider the convective acceleration term of the Navier-Stokes equations:

$$\mathbf{u} \cdot \nabla \mathbf{u}$$

The finite element method requires that the discretised operator is linear in the trial function if the system of equations is to be solved using a linear solver. Unfortunately, the convective acceleration term is not linear in $\mathbf{u}$. We linearise this term using the Picard iteration by approximating it as follows:

$$\mathbf{u}^i \cdot \nabla \mathbf{u}^i \approx \mathbf{u}^{i-1} \cdot \nabla \mathbf{u}^i$$

That is, we replace one instance of $\mathbf{u}$ with $\mathbf{u}^{i-1}$ so that our term is now linear in $\mathbf{u}$. At each iteration i, we substitute the value of $\mathbf{u}$ from the previous iteration as $\mathbf{u}^{i-1}$ and construct a new approximate solution for $\mathbf{u}$. In our example, we will solve repeatedly until we find a value of $\mathbf{u}^i$ such that letting $\mathbf{u} \cdot \nabla \mathbf{u} = \mathbf{u}^i \cdot \nabla \mathbf{u}^i$ in our original equation satisfies the convergence requirement. For our initial guess of $\mathbf{u}^{i-1}$ we use $\mathbf{u}$ from the previous time-step.

Figure 6.10 shows an Excafé implementation of the solution of the Navier-Stokes convective acceleration term using Picard iteration. We declare the `TemporalIndex` variable `i` which will be the index variable of our Picard iteration loop. We use this to define an `IndexedField` named `unknown` which has a distinct value for each value of `i`.

Expressions that access `unknown` using `i` as part of the index expression implicitly become part of the loop. The loop will iterate until the termination condition attached to `i` becomes true.

Accessing the final values `unknown` took is done by using index expressions that are offsets against the special variable `final`.

Finally, we note that solving this term in isolation from other terms with a zero RHS field is not a useful exercise and we use it merely to demonstrate Excafé's indexing syntax. An implementation of an incompressible Navier-Stokes solver including the Picard iteration linearisation of the convective acceleration term is discussed in Chapter 8, and the corresponding code is presented in Appendix F.

6.4.2 C++ Syntax

Our syntax for the declarative declaration of loops is inspired by mathematical subscript (or sometimes superscript) notation to denote different values of the same symbolic variable.

Excafé provides the following handle types to represent this notation:

TemporalIndex

which represents a mathematical subscript or superscript used to refer to different values of a variable calculated during the repeated application of a formula.

IndexedScalar, IndexedField & IndexedOperator

which are indexed versions of the `Scalar`, `Field` and `Operator` handles. They are used to assign values of the variable at the current iteration and access variables from previous iterations.

Declaration of a `TemporalIndex` and each indexed type is shown in Figure 6.11. Each indexed type instance is associated with a particular `TemporalIndex` at construction. For example, `indexed_field` as declared in Figure 6.11 will take a unique value for each value of `i`.

At this point, we stress that although the library client associates each instance of an indexed type with only one `TemporalIndex`, this does not prevent that variable taking unique values with respect to multiple index variables. In other words, our declarative loop declaration syntax

```
// Declare a temporal index i. Expressions referencing i will implicitly be
// placed in a loop with i as the index variable.
TemporalIndex i;

// Declare a vector field for the unknown indexable by i.
IndexedField unknown(i);

// Declare the load vector (zero in this example).
Field load(velocitySpace);

// Set the initial value of the unknown guess to velocity field from the
// previous time-step.
unknown[-1] = velocityField;

// Declare a bilinear form dependent on the value of unknown at the
// previous iteration.
const forms::BilinearFormIntegralSum lhsForm =
  B(inner(unknown[i-1], grad(velocity)), velocity)*dx;

// Declare a linear system that assembles a matrix using the declared bilinear
// form. As both the form and guess at the solution involve an expression
// dependent on i, the linear system is placed in a loop.
LinearSystem system = assembleGalerkinSystem(velocitySpace,
                                             lhsForm,
                                             load,
                                             velocityConditions,
                                             unknown[i-1]);

// Get the discretised operator with boundary conditions applied.
Operator linearisedSystem = system.getConstrainedSystem();

// Set the next value of unknown to be the solution from this iteration.
unknown[i] = system.getSolution();

// Calculate the residual using the solution from the *previous* time-step. As
// our operator was assembled using unknown[i-1], we must compute the
// residual as the norm of the operator applied to unknown[i-1].
Scalar residual = (linearisedSystem*unknown[i-1] -
                   system.getConstrainedLoad()).two_norm();

// Describe how to terminate the loop with index i.
i.setTermination(residual < 1e-3);

// Set the value of the velocity field for this time-step.
s.setNewValue(velocityField, unknown[final-1]);
```

Figure 6.10: Declarative construction of a loop to linearise the convective acceleration term of the Navier-Stokes equations using Picard iteration. This example exists to demonstrate the indexing syntax and does not solve a useful problem. See Appendix F for the code of an incompressible Navier-Stokes implementation using this linearisation.

```
TemporalIndex   i;

IndexedScalar   indexed_scalar(i);
IndexedField    indexed_field(i);
IndexedOperator indexed_operator(i);
```

Figure 6.11: Declaration of of a `TemporalIndex` and indexed scalar, field and operator types. Each indexed type instance is associated with a particular `TemporalIndex` at construction.

```
// Given C++ variables of the following types:
TemporalIndex i;
Field f;
IndexedField u;

// Indexed value initialisation
u[-2] = f;
u[-1] = f;

// Setting a value of u within an iteration
u[i] = f;

// Retrieving a previous value of u within an iteration
f = u[i-1];

// Accessing value of u relative to to the final value
f = u[final];
f = u[final-1];
```

Figure 6.12: Given C++ variables of the specified types, examples of construction of valid expressions using the `IndexedField` type. Identical syntax is valid for the `IndexedScalar` and `IndexedOperator` types.

in *composable*, otherwise we would not be able to support nested iteration. Before we describe how composability is achieved, we first consider declarative loop construction involving a single `TemporalIndex`.

We give an example of the use of the `IndexedField` type in Figure 6.12. For our indexed types, we have overloaded C++'s array indexing operator in order to represent our subscript notation.

For some indexed type associated with a `TemporalIndex i` we can assign values at negative constant indices, which correspond to initial values required before an iteration. We can also assign at index `i`, which describes the value assigned to the variable at each iteration.

```
TemporalIndex i;
Field a = ...
IndexedField b(i)

Field c = a + b[i];
```

Figure 6.13: Assignment of the expression `a + b[i]`, which is only valid inside a loop indexed by `i`, to the non-indexed handle `c`. The expression held by handle `c` is only valid within loop `i` as well, but it is not possible to use indexing with the handle `c`.

At any given iteration, we can retrieve values of the variable at index `i-n`, for any natural number `n`. Once an iteration is complete, we can also use indices of the form `final-n`, where `final` is a special value that indicates the final value of an index.

What happens when we assign a value that is only defined within an iteration to a non-indexed type as in Figure 6.13?

The expression `a + b[i]` is only valid within the scope of the iteration corresponding to `TemporalIndex i`. Hence, `c` is only valid within the same scope. In order for assignments like this to be valid, we require that expressions inherit indices associated with their operands. Thus, the set of indices associated with an expression describe the scope in which that expression is valid.

The propagation of `TemporalIndex` instances through expressions also provides the mechanism whereby we achieve nested iteration. An expression of the form `a[i] + b[j]` is only valid if loops `i` and `j` are contained within each other. However, it does not tell us if loop `j` is nested in loop `i` or vice-versa. We describe how this is determined in the next chapter.

In Chapter 7, I describe how `TemporalIndex` instances are associated with each node in the expression DAG and how this is used to infer a valid loop nesting structure required to evaluate expressions involving declarative iteration.

```
TemporalIndex i;
IndexedScalar f(i);

f[-2] = 0.0;
f[-1] = 1.0;
f[i] = f[i-1] + f[i-2];

i.setTermination(f[i] > 100.0);
```

(a) Excafé

```
double f[3];
f[0] = 0.0;
f[1] = 1.0;

int i=1;

do
{
  ++i;
  f[i%3] = f[(i+2) % 3] + f[(i+1) % 3];
}
while (f[i%3] <= 100.0)
```

(b) C

Figure 6.14: Excafé syntax for describing an iterative construction of Fibonacci values (in floating point) and a corresponding C implementation. The C implementation requires more complex indexing to access `f` which is used as a cyclic buffer.

6.4.3 Comparison to Imperative Form

Consider the declaratively declared loop in Figure 6.14 that iterates until `f` reaches the first Fibonacci number greater than 100[1].

The advantages of the declarative syntax are apparent. How to store `f` is not a consideration in the Excafé implementation whilst in the C implementation we must explicitly manage `f` as a cyclic buffer of the previously computed values. In addition, we do not need to specify how large the buffer to store `f` should be. The size of the buffer can be automatically determined from the sizes of the offsets used to access `f`.

6.4.4 Future Development

We previously showed how the indexing notation and discrete expression syntax could be used to linearise the convective acceleration term of the Navier-Stokes equations.

We note that this syntax is ideally suited to description of iterative linear system solvers. Currently, we call PETSc [47] to perform our linear system solve, invoked by a linear-system

[1] This is a contrived example with no purpose other than to demonstrate the benefits of Excafé's indexing syntax.

solver node in our discrete expression DAG. However, we can see no reason why these solvers could not be represented directly in our syntax.

6.5 Variational Form Capture

Similarly to other finite element libraries, we provide a mechanism for expressing variational forms. Our syntax is deliberately similar to that of the variational form notation of the Unified Form Language [37]. However, we only implement the most common vector calculus operators required for the forms we investigate. We have not implemented the more general indexed representation that UFL uses to represent its tensor operations. Additional information on the data structures used during variational form capture can be found in Section D.4.

Figure 6.15 shows the declaration of a bilinear form for Poisson's equation in UFL and Excafé. Unlike UFL, we do not specifically denote certain function spaces as trial and test. In our example, we have used the same one for both the trial and test space (as is always the case with Galerkin methods). However, it is possible to declare the same function space again, making it possible to distinguish between the two. This allows us to raise an error if mathematical expressions are declared involving incompatible function spaces.

The Excafé expression capture representation has the expected DAG structure as presented in Figure 6.16. We do not perform any optimisation on the DAG representation, instead it is visited during the construction of the local assembly matrix (as detailed in Chapter 7) and optimisations performed on the resulting expressions.

We draw attention to the fact that our captured bilinear forms can reference any discretised fields defined in our discrete expression DAG (as is needed in time-dependent problems). As a consequence, it is possible for an optimiser to see the structure of the discretised fields (i.e. a summation of basis functions multiplied by coefficients). There is the possibility that this may lead to redundancy optimisations if the basis functions used to represent our discretised field are already being used for the trial and test functions, as is likely.

```
element = FiniteElement("Lagrange", triangle, 1)

v = TestFunction(element)
u = TrialFunction(element)

a = dot(grad(v), grad(u))*dx
L = v*f*dx
```

(a) UFL

```
u = scenario.addElement(new LagrangeTriangleQuadratic<1>());

const BilinearFormIntegralSum form = B(grad(u), grad(u))*dx;
```

(b) Excafé

Figure 6.15: A Bilinear form for the LHS of the finite element discretisation of Poisson's equation in UFL and Excafé. We use the same function space for the trial and test spaces in the Excafé example. The order of parameters to the `B` method implies whether `u` is being used in the context of a trial or test function.

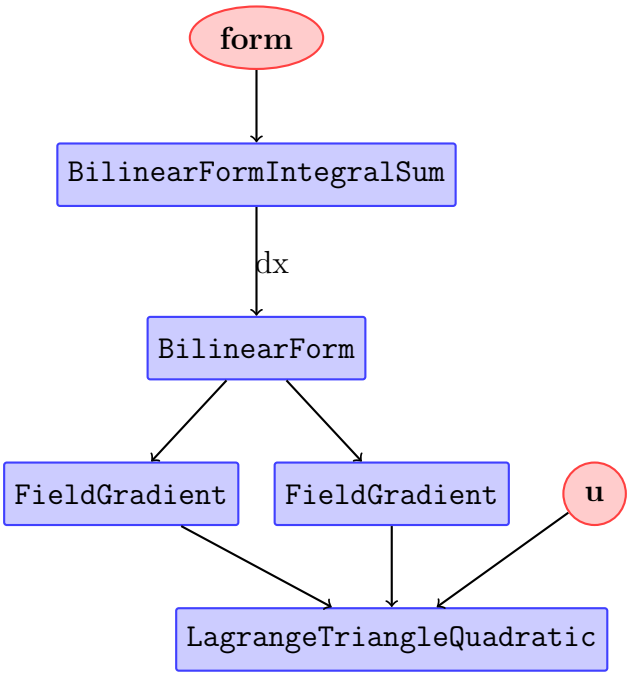
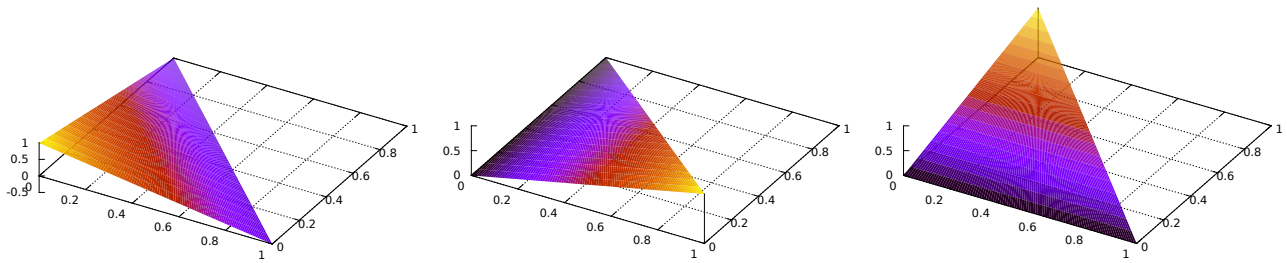


Figure 6.16: The DAG we construct for the bilinear form $\int_\Omega \nabla u \cdot \nabla u \, dx$. It is used to assemble an operator required during the calculation of the RHS of our heat solver example. The assembled operator is applied to the temperature field from the previous time-step in order the obtain the RHS vector.



(a) $1 - x - y$ (b) x (c) y

Figure 6.17: Linear Lagrange basis functions defined over the reference triangle.

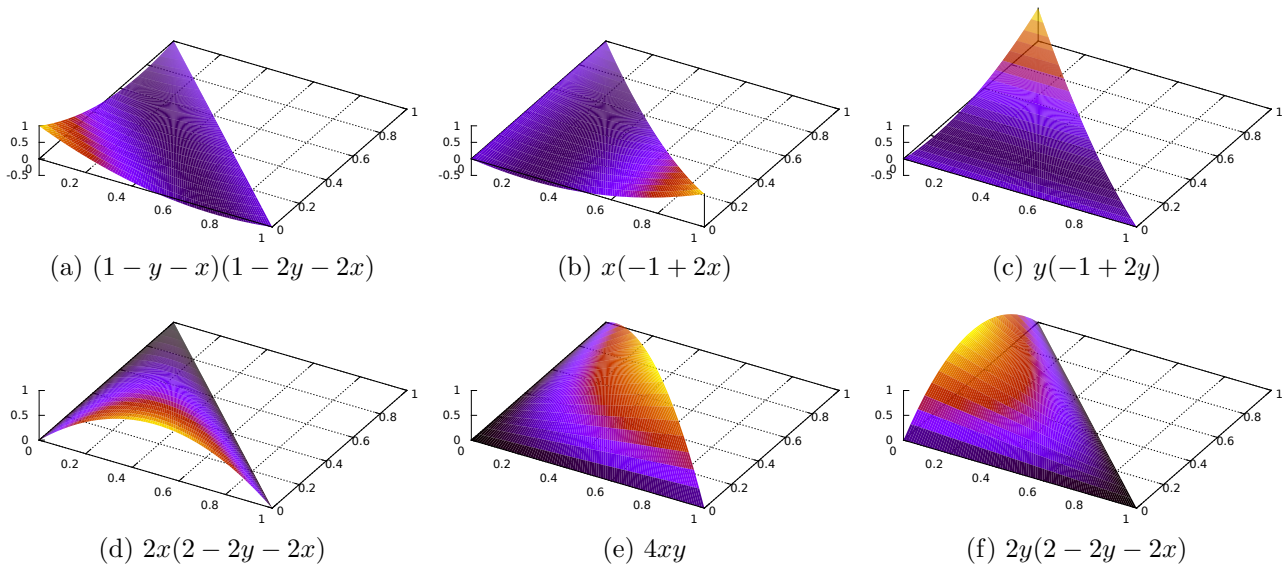
For example, in our Navier-Stokes problem (described in Chapter 8), all velocity fields are discretised with the same basis functions. We wish to explore the possibility that when assembling bilinear forms involving previously calculated fields, knowledge of the field discretisation can assist in optimising local assembly.

We contrast with the FEniCS Form Compiler [36] which also has knowledge of the discretisations of the referenced fields, but performs tensor contraction optimisations using a numeric representation of the local assembly matrix, where redundancy relationships due to basis function reuse may be significantly more difficult or impossible to infer.

6.6 Basis Function Capture

Excafé implements expression capture of basis functions. Currently, Excafé has implementations of linear and quadratic Lagrangian basis functions over triangular cells. We show plots of the linear and quadratic basis functions in Figures 6.17 and 6.18 respectively. Additional information on the data structures used to represent scalar basis function expressions in Excafé can be found in Section D.5.

We show a code excerpt from Excafé that performs expression capture for the Lagrange linear basis functions in Figure 6.19. The code doesn't directly resemble the formulae because we exploit the rotational symmetry of the basis functions so that we only define one formula. In addition, we make use of a helper function in order to generalise our scalar basis to a set of arbitrary tensor-valued basis functions.



(a) $(1-y-x)(1-2y-2x)$ (b) $x(-1+2x)$ (c) $y(-1+2y)$

(d) $2x(2-2y-2x)$ (e) $4xy$ (f) $2y(2-2y-2x)$

Figure 6.18: Quadratic Lagrange basis functions defined over the reference triangle.

Unlike the rest of our expression capture facilities, the interface for basis function definition has not yet been optimised for easy use by library clients, though it is possible for clients to define other basis functions. The primary purpose of our basis function capture is to give our library representations that it can analyse and optimise, rather than to expose an interface to the library client.

The most common way to construct vector or arbitrary tensor-valued basis functions is to repeat the same set of scalar basis functions for each element of the tensor. This permits efficient implementations since they can be hard-coded to the sparsity of the basis functions. However, these optimisations can prohibit the use of truly vector-valued basis functions such as the Raviart-Thomas [48] element.

Excafé supports arbitrary tensor-valued basis functions, although all current tensor-valued bases currently repeat scalar-valued ones. As Excafé has access to run-time representations of basis functions, this permits analysis to determine when sparsity optimisations are appropriate. We discuss how the captured basis function representations are optimised in Chapter 9.

As is common in finite-element implementations [31, 34], our basis functions are defined over a reference cell. However, we require the ability to evaluate integrals of bilinear forms involving our basis functions and their derivatives over arbitrary cells. We can do this provided we have:

```
tensor_expr_t
getBasis(const std::size_t i, const detail::PositionPlaceholder& v) const
{
  using namespace detail;

  if (i >= spaceDimension())
    CFD_EXCEPTION("Requested invalid basis function.");

  const TensorSize tensorSize(rank, dimension);
  tensor_expr_t bases(tensorSize);

  const DofAssociation dofAssociation = dofNumbering.getLocalAssociation(i);

  assert(dofAssociation.getEntityDimension() == 0);
  const unsigned node_on_cell = dofAssociation.getEntityIndex();
  const unsigned index_into_tensor = dofNumbering.getTensorIndex(i);

  const int ip1 = (node_on_cell+1) % 3;
  const int ip2 = (node_on_cell+2) % 3;

  const TensorIndex tensorIndex =
    TensorIndex::unflatten(tensorSize, index_into_tensor, row_major_tag());

  bases[tensorIndex] =
   (referenceCell->getLocalVertex(ip2)[0]-referenceCell->getLocalVertex(ip1)[0])*
   (v[1] - referenceCell->getLocalVertex(ip1)[1]) -
   (referenceCell->getLocalVertex(ip2)[1]-referenceCell->getLocalVertex(ip1)[1])*
   (v[0] - referenceCell->getLocalVertex(ip1)[0]);

  return bases;
}
```

Figure 6.19: Excafé implementation of expression capture for the linear Lagrange basis functions over a triangular element. We exploit rotational symmetry and use a helper class that simplifies the generalisation of the scalar basis to arbitrary tensor-valued bases.

1. The ability to transform a gradient of a function on our reference cell to that of the function on an arbitrary cell.

2. The ability to transform an integral of a function over our reference cell to that of the integral of the function over an arbitrary cell.

The gradient of a function u in the elemental region $x \in \Omega_e$ can be calculated from the gradient in a standard region $\xi \in \Omega_{st}$ using the chain rule. For example, in 2-D space:

$$\frac{\partial u}{\partial x_1} = \frac{\partial u}{\partial \xi_1}\frac{\partial \xi_1}{\partial x_1} + \frac{\partial u}{\partial \xi_2}\frac{\partial \xi_2}{\partial x_1} \tag{6.2}$$

$$\frac{\partial u}{\partial x_2} = \frac{\partial u}{\partial \xi_1}\frac{\partial \xi_1}{\partial x_2} + \frac{\partial u}{\partial \xi_2}\frac{\partial \xi_2}{\partial x_2} \tag{6.3}$$

Integration within the general element can be expressed in terms of an integral over the standard element via a change of co-ordinates:

$$\int_{\Omega_e} u_e(x_1, x_2)\, dx_1\, dx_2 = \int_{\Omega_{st}} u(\xi_1, \xi_2)|J_{2D}|\, d\xi_1\, d\xi_2 \tag{6.4}$$

where u_e is the function u defined in terms of global co-ordinates, and $|J_{2D}|$ is the determinant of the Jacobian of the local-to-global co-ordinate transformation:

$$|J_{2D}| = \begin{vmatrix} \frac{\partial x_1}{\partial \xi_1} & \frac{\partial x_1}{\partial \xi_2} \\ \frac{\partial x_2}{\partial \xi_1} & \frac{\partial x_2}{\partial \xi_2} \end{vmatrix} = \frac{\partial x_1}{\partial \xi_1}\frac{\partial x_2}{\partial \xi_2} - \frac{\partial x_1}{\partial \xi_2}\frac{\partial x_2}{\partial \xi_1} \tag{6.5}$$

In order to evaluate these, we need to define a mapping between the reference space ξ and the elemental space x. We define the mapping χ so that:

$$x_1 = \chi_1(\xi_1, \xi_2) \tag{6.6}$$

$$x_2 = \chi_2(\xi_1, \xi_2) \tag{6.7}$$

For our triangular cell, we define χ using the linear basis ϕ already defined.

$$\chi_i(\xi_1, \xi_2) = \sum_{q=0}^{n} \hat{x}_q^i \phi_q(\xi_1, \xi_2) \tag{6.8}$$

where $\hat{x}_q^i$ is component i of vertex q of the cell. In other words, we have represented the coordinate transformation from the reference cell to an arbitrary cell as an interpolation between cell vertices using our set of linear basis functions. Our framework's symbolic nature permits the possibility of using an arbitrary vector field to specify mesh geometry, enabling curved elements. However, we have not yet extended Excafé to do this.

Hence, in capturing our basis functions we not only have the representations of our basis functions over our reference cells, but a representation of the geometric transformation required to represent the integrands of integrals over an arbitrary cell analytically.

We note that having an expression capture representation of our basis functions may not appear useful at first glance. As the basis functions are defined over the reference cell, finite element implementations typically evaluate the basis functions and their gradients at quadrature points defined over the reference cell. These values are then stored in tables and the gradients transformed as appropriate when calculating integrals over the cells in the mesh.

Fringe benefits of capturing our basis functions include:

1. Automatic calculation of derivatives.

2. Automatic determination of degree of quadrature for integrals.

However, our primary hope is to expose patterns between elements of our assembly matrices that can only be determined using a representation of our basis functions. We believe such optimisations should be a superset of those possible with the topological optimisations developed by the FEniCS project [38] which operates on local assembly matrices (on the reference cell) where all elements have already been evaluated.

6.7 Conclusion

In this chapter, we presented Excafé's expression capture features. Specifically, Excafé captures run-time representations of basis functions, variational forms and a declarative representation of operations between scalars, discretised fields and operators. We conclude with an overview of the most important aspects.

We contrast with code generators such as FFC that optimise assembly without an expression representation of basis functions. We hope to use our basis function representations to explore optimisations that arise from repeated use of, and redundancies between, basis functions.

In capturing basis functions over the reference cell, we also have the ability to construct expressions representing the geometric transformation between the reference cell and an arbitrary cell. Typically this is done using the linear basis functions over that cell, but future work could include other basis functions or discretised vector fields used to represent curved cells.

Above the level of basis functions, we capture representations of bilinear forms. These are functionals that are linear in the functions used for the trial and test spaces, built from vector calculus operators. We observe that in many finite element problems, our bilinear forms often make reference to fields defined earlier during the solver execution.

By capturing the symbolic representations of the trial and test functions, along with the basis functions of any other discretised fields, we have the opportunity to detect optimisations that may arise from commonalities between those discretisations. FFC's local assembly optimisations operate on a numeric representation of the local assembly matrix, where these relationships

are no longer apparent. Hence, we hope this will expose optimisation opportunities not visible to FFC.

Our highest level of expression capture captures operations between scalars, discretised fields and operators. For the most part, these resemble traditional linear algebra, but with some additional operators and additional constraints on valid operations dictated by choices about discretisation.

Rather than choosing to capture these operations in an imperative form (e.g. in the manner used by TaskGraph [13]) we have chosen to explore a declarative mechanism. Most notably, this mechanism includes support for specification of iteration, needed within some more complex problems. The resulting syntax is similar to mathematical subscript notation and avoids the need to specify mathematically unrelated concerns such as cyclic buffer handling. We provide a description of how we convert this representation to an imperative form in Chapter 7.

In this next chapter we present our implementation of an incompressible Navier-Stokes solver in Excafé. We use this as an example problem to motivate the optimisations we wish to explore.

Chapter 7

Imperative Loop Inference

We described a syntax that permits declarative specification of iteration in Section 6.4. This syntax enables Excafé to build a representation of the computation to be performed in the form of a directed acyclic graph (DAG). In Desola, which could not capture iteration, there was a direct correspondence between nodes in the expression DAG and values in the computation. In Excafé, the relationship is less apparent, since nodes in our expression DAG may correspond to expressions that are evaluated within a loop nest, and therefore take multiple values.

In this section, we describe how we take an expression DAG with references to indexed values and infer an imperative structure that can be used to execute the computation. We then look at how it is possible for a library client to construct an invalid specification and how this may be detected.

7.1 The Problem

To consider the issues raised when trying to evaluate an expression DAG involving indexed expressions, we will consider our linearisation example from Section 6.4.1. Code for this example was presented in Figure 6.10.

The example shows the solution of a non-linear system by repeatedly solving a linear approximation, constructed using the approximate solution for system computed during the previous iteration. The example makes use of one indexed field, called `unknown`. We show the discrete expression DAG that is constructed in Figure 7.1.

The DAG describes a computation that repeatedly computes values for the field `unknown`, which represents the solution to a non-linear system. The approximate solution from the previous iteration, `unknown[i-1]` is used to assemble a linear operator so that it can be used to approximate the non-linear one. The previous value of `unknown` is also used as the initial guess for the next approximate solution. The nodes `OperatorApplyBC` and `DiscreteFieldApplyBC` are responsible for applying boundary conditions to the LHS and RHS, respectively, of our linear system.

To determine when to terminate, the true residual of our non-linear system (as opposed to the approximate one) is compared against a tolerance value. We multiply the operator by the value of `unknown` from the *previous* iteration, subtract the RHS, take the l^2-norm and compare to our tolerance value. As the previous value of `unknown` was also used to assemble the operator, this gives us the true residual. This is why the final solution is taken as `unknown[final-1]` and not `unknown[final]`.

A handle labelled u_{next} is held by Excafé to the result of the computation. This represents the new value of the field u. Nodes representing accesses to the indexed field `unknown` do not maintain any references. It is the responsibility of the `IndexableValue` internal handle class corresponding to `unknown` to keep references to the values assigned to it. Lastly, each `TemporalIndexValue` instance, which represents an index variable, must also maintain a reference to its termination condition.

7.2 Algorithm

Our algorithm to infer an imperative execution strategy computes a set of possibly-nested loop scopes and an execution order. It consists of three steps:

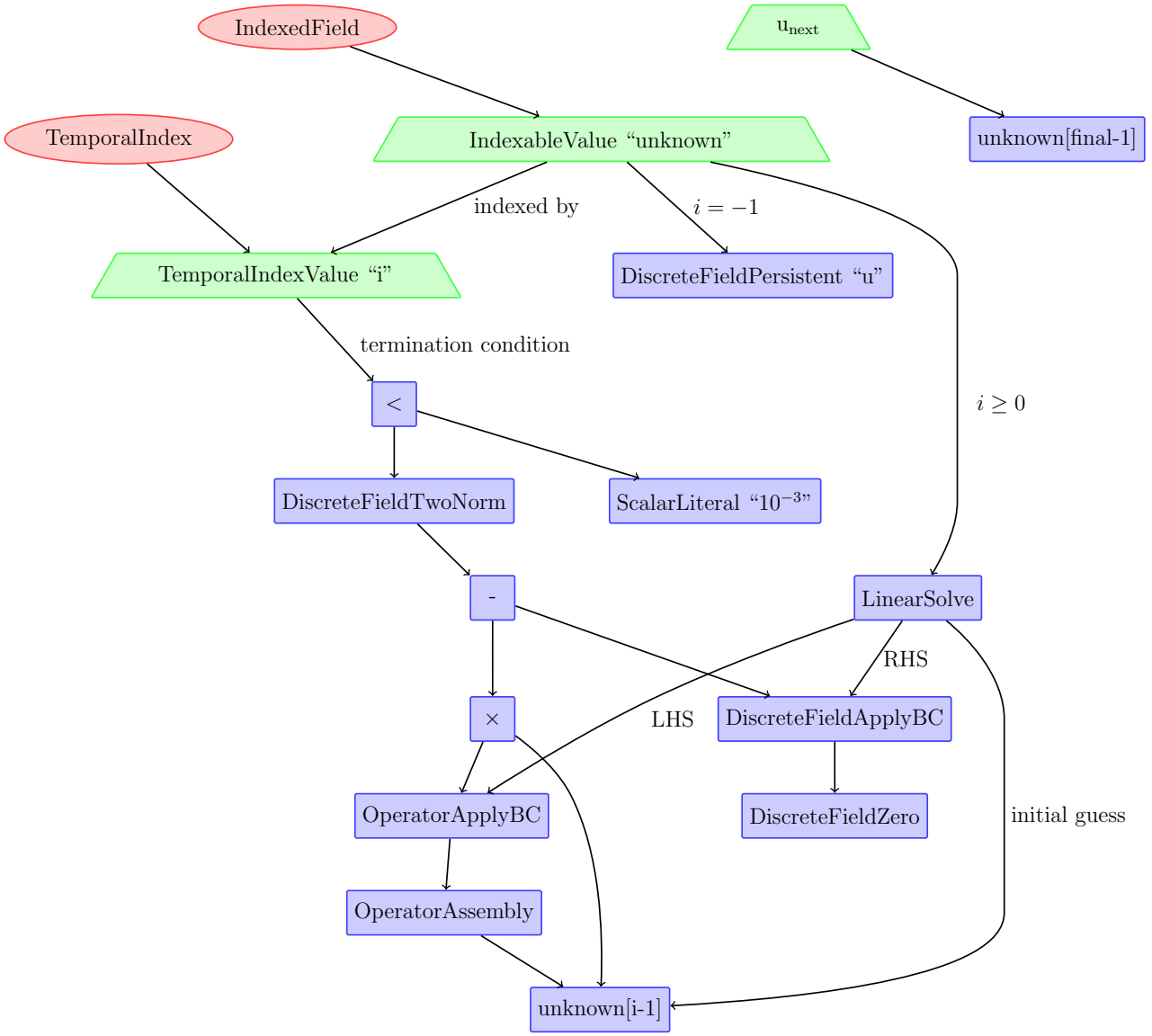


Figure 7.1: The data structure built during execution of the linearisation example from Figure 6.10. The elliptical nodes correspond to handles held by the library client. The trapezoid shaped nodes correspond to Excafé internal classes that also need to maintain handles to parts of the discrete expression DAG. The rectangular nodes correspond to expressions that are either scalar, discrete field or discrete operator valued. These are the nodes that form our (discrete) *expression DAG*. The handle u_{next} holds a reference to `unknown[final-1]`, the expression that will become the next value of u. However, `unknown[final-1]` has no dependencies on other expressions. This is due to the fact that our expression DAG structure no longer fully reflects data dependencies when dealing with indexed values.

Expression DAG Annotation We associate sets of index variables with each node in the expression DAG. The expression corresponding to the node is only well-defined within all loops referenced by the index set, implying that they are nested. The order of nesting is not yet known.

Nesting Order Inference Using the annotations, we infer the loop nesting structure of the imperative code.

Topological Sorting The nodes and loops within every scope must be sorted with respect to dependencies in order to construct a valid execution order.

We describe these steps in detail in the next sub-sections.

7.2.1 Expression DAG Annotation

In a conventional expression DAG, each node corresponds to a single expression. In our expression DAGs, it is possible for a node to correspond to a syntactic expression that takes multiple values, if it is computed iteratively.

In our example, the expression `unknown[i-1]` is clearly only valid within the scope of loop i since it has a dependency on the value of i. Similarly, any expressions that use `unknown[i-1]` directly or indirectly are only well-defined within loop i.

In order to determine which expression DAG nodes are associated with which indices, we construct a new graph in which edges represent the inheritance of indices (`TemporalIndexValue` instances) from a node's operands. We show the graph corresponding to our linearisation example in Figure 7.2.

In Figure 7.2 the arcs show the propagation from indices from expression DAG nodes to their operands. Indices must also propagate from values assigned to `unknown` to all uses of `unknown`. These rules aren't reflected in the structure of Figure 7.1 since the values being used are computed in previous iterations of i. We show these propagation rules using dashed arcs since they are in a sense, *implicit*.

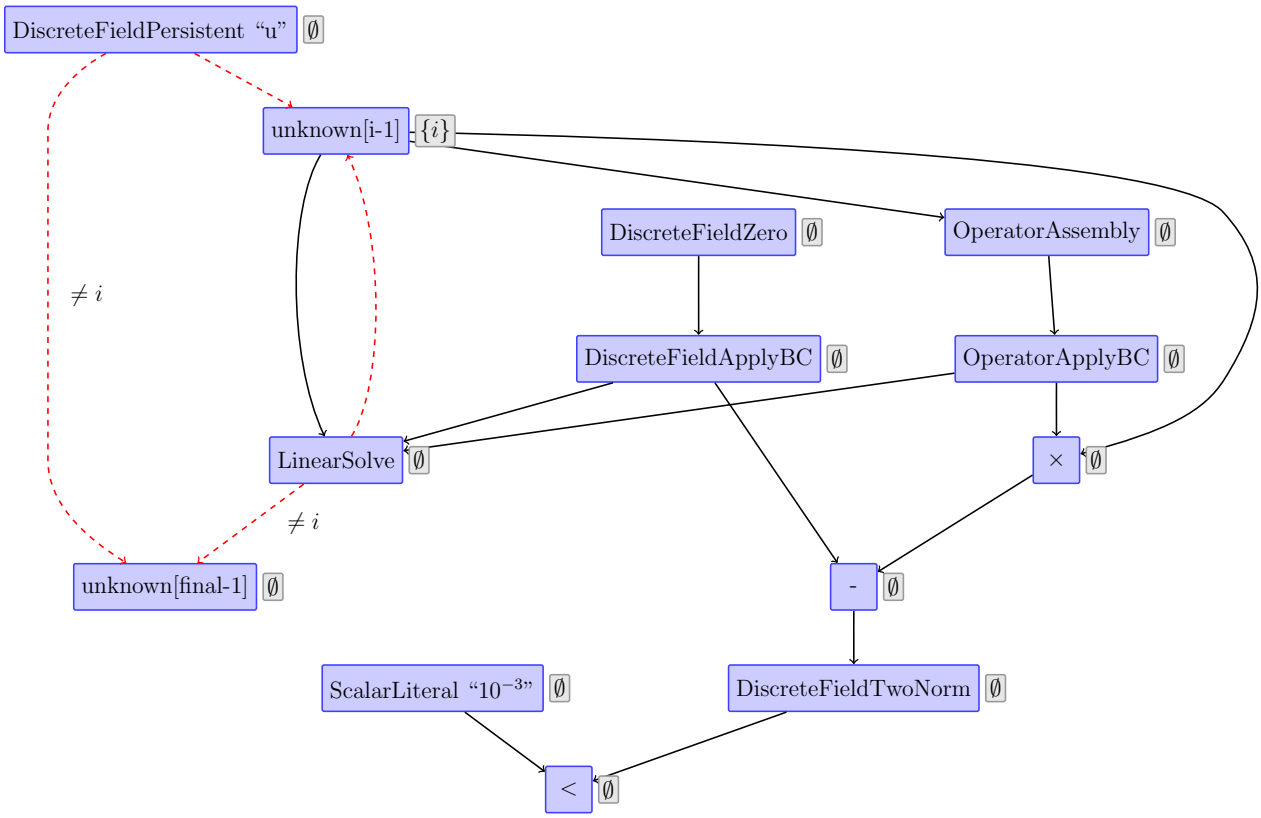


Figure 7.2: The example discrete expression DAG with arcs between nodes in the direction of index propagation. The dashed arcs denote index propagation from expressions assigned to `unknown` to all uses of `unknown`. We show the indices associated with each node at the *start* of the propagation phase. The index variable i is not allowed to propagate along the two arcs labelled $\neq i$ since the expression `unknown[final-1]` is only valid *outside* of loop i.

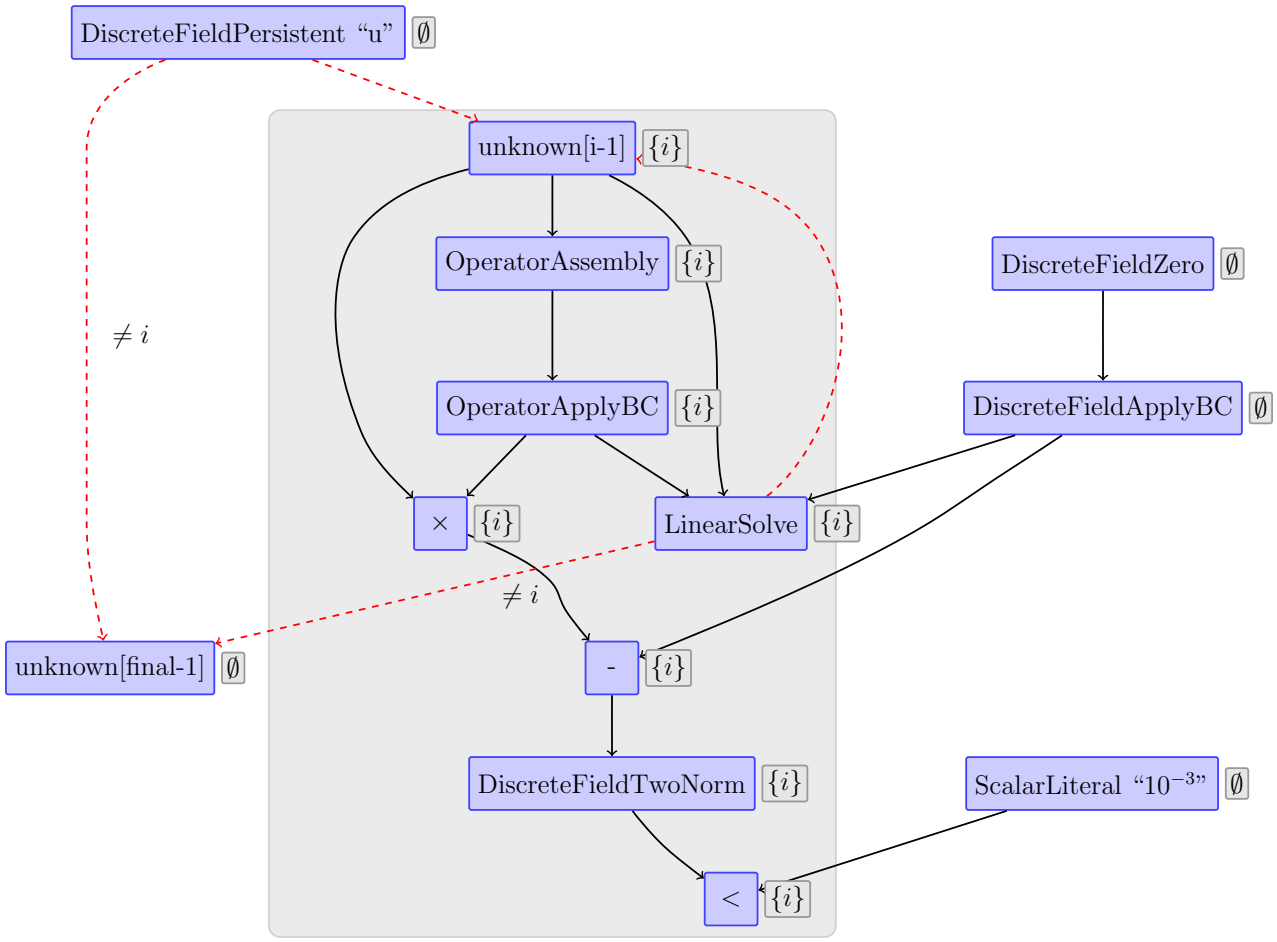


Figure 7.3: Our expression DAG after index propagation. Each node is annotated with the indices of the loops it is contained in. All expression DAG nodes inferred to be in loop i have been placed in the rectangular region.

Two of our implicitly defined arcs are labelled $\neq i$. These arcs are forbidden from propagating the index variable i. Since the expression `unknown[final-1]` refers to the penultimate value assigned to `unknown` by loop i, it must be in the enclosing scope of loop i. Therefore, it inherits all indices of the values assigned to `unknown`, except i.

At the start of our index propagation, only nodes that explicitly make use of an index are associated with it. In Figure 7.2, the only node that refers to i is `unknown[i-1]`. We show the result of the index propagation pass for our linearisation example in Figure 7.3.

The implicitly defined edges used for our index propagation result in a problem defined over a graph that may no longer be acyclic. As a consequence, index propagation is an iterative process. However, it is guaranteed to terminate since our graph is finite and each propagation step either makes progress, or marks the end of the algorithm.

We recall that the `unknown[final-1]` expression DAG node is the value that will become the next value of the persistent field u. Therefore, it cannot be associated with any indices, otherwise its value would not be well defined.

7.2.2 Determining Loop Nesting

Associating each node in our discrete expression DAG with a set of `TemporalIndexValue` instances tells us the loops in which that node must be contained. However, it does not tell us the loop nesting structure.

For example, if node n is associated with `TemporalIndexValue` instances i, j, we know that n is an expression that is calculated within loops i and j but we do not know if loop i contains loop j or vice-versa.

Loop nesting is determined as follows:

1. Assign all expression DAG nodes associated with no `TemporalIndexValue` instances to the global scope.

2. For each expression DAG node associated with a `TemporalIndexValue` set $\{i\}$ of size 1, create a scope for the new loop i and place the node inside that scope.

3. Continue the process for expression DAG nodes associated with `TemporalIndexValue` sets of increasing size. Scopes for at least all but one `TemporalIndexValue` instances in the set should already have been created, so it is possible to unambiguously identify the scope in which any new loop scope should be created.

For an example, assume we are given nodes A, B, C, D and E associated with sets of indices as follows:

$A \to \{i, k, l\}$
$B \to \{i, j\}$

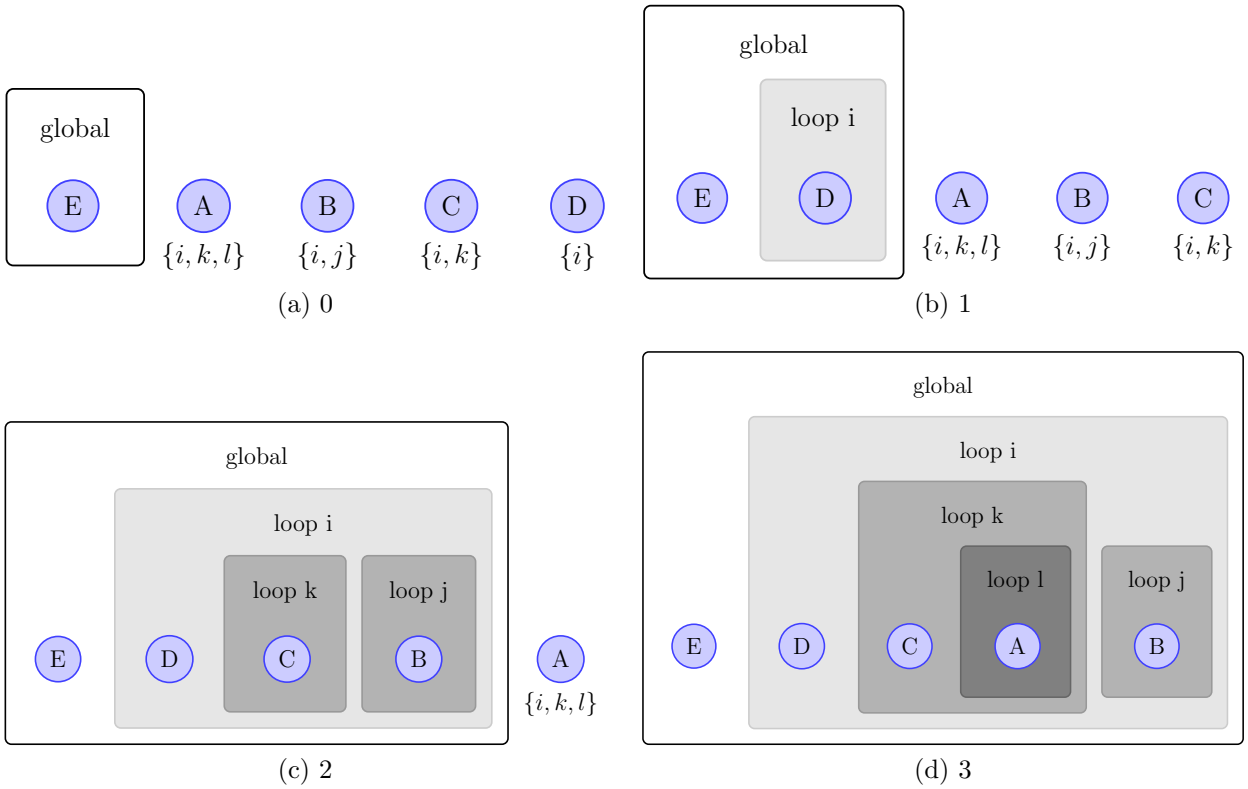


Figure 7.4: The loop nesting construction at different iterations of our algorithm. At each step, nodes for which the all but one of the indices belong to a known scope are used to construct new nested scopes.

$C \to \{i, k\}$

$D \to \{i\}$

$E \to \emptyset$

We show the inference of the corresponding loop nesting in Figure 7.4. The complete scope of a node can be determined once all but one of the scopes of its indices are known.

Our algorithm uses the property that for any node n associated with a `TemporalIndexValue` set S of size $|S| > 0$, there exists another node p associated with a `TemporalIndexValue` set T of size $|T| = |S| - 1$ such that $T \subset S$.

The single `TemporalIndexValue` in the set $S - T$ corresponds to the loop scope that n is in, but p is not. If no node p exits, the loop nesting is under-specified and ambiguous. If there is more that one candidate for p, $S - T$ should be identical for all of them otherwise, the nesting has been over-specified and cannot be satisfied.

We discuss whether it is possible for a library client to under or over-specify loop nesting and how these situations can be detected in Section 7.3.

7.2.3 Topological Sort

Once each expression DAG node has been assigned to a loop, the remaining step is to determine an execution order. Instead of applying the topological sort to the whole expression DAG, as would be the case with an ordinary DAG, the sort is applied to each loop scope independently.

Within each loop scope, expression DAG nodes and sub-loops are sorted together. Any loop scope S will have the following dependencies:

- The initialisation values of any `IndexedScalar`, `IndexedField` or `IndexedOperator` associated with `TemporalIndex` of the loop.
- Dependencies of all expression DAG nodes in S that are not found in S.
- Dependencies of all sub-scopes of S that are not found in S.

Within a scope, the root elements used to perform a topological sort are:

- The expression DAG node representing the loop termination condition.
- All values assigned to indexable scalars, fields or operators during the current iteration. Only indexed types can be used to retrieve a value from a loop or have their values used by the next iteration.

Once the topological sort is complete, we have a complete loop nesting structure and an execution order for the global scope and all loop scopes.

```
TemporalIndex i;
IndexedField f(i);
i.setTermination(f[i].two_norm() < 1e-4);

solve.setNewValue(u, f[i]);
```

Figure 7.5: A code fragment showing the construction of an invalid specification for updating the value of u. Since the expression `f[i]` is only valid within the context of a loop, it cannot be used to define the new value of u.

7.3 Declarative Specification Validation

Our algorithm assumes that we have a valid unambiguous declarative loop specification. However, there are multiple ways to construct invalid loop specifications. We consider invalid specifications and how construction is either prevented or detected.

7.3.1 Updating persistent values with incorrectly scoped expressions

Our expression DAG is used to define new values for persistent scalars, discretised fields or discretised operators. The nodes that represent the new values must exist at the global scope (i.e. they cannot exist within a loop).

Figure 7.5 shows a code fragment that tries to use an incorrectly scoped value to update the value of the persistent field u. This incorrect usage can easily be detected. After application of our index propagation algorithm, we verify that the expression DAG node used to define the new value for a persistent variable is not associated with any index variables.

7.3.2 Under-specification of loop nesting

An under-specified loop nesting means that we do not have sufficient ordering relations between index variables to construct an unambiguous loop nesting. Provided that the expressions used

to update our persistent variables are correctly scoped, it is impossible to construct an under-specified loop nesting.

We prove this by considering the index propagation graph of any valid discrete expression DAG (e.g. Figure 7.3).

1. We select an arbitrary node n from our discrete expression DAG. We will call the set of indices associated with a node the *index set*. We denote the index set of n as $\theta(n)$. We will show that for any n, it is possible to determine the scoping of the loops whose induction variables are contained in $\theta(n)$.

2. Since n is required to evaluate the final result (or results) of the expression DAG, there exists at least one path to some result node r from n.

3. As we require that our result nodes are correctly scoped, the index set of any r is the empty set so that $\theta(r) = \emptyset$.

4. If $\theta(n) = \emptyset$, we already know that n is in the global scope. If $\theta(n) \neq \emptyset$, then we know that n must be inside at least one loop.

5. As loop indices propagate along the edges of our graph, in order for $\theta(r)$ to be empty, the path from n to r must contain edges such that each loop index in $\theta(n)$ cannot propagate along at least one of them.

6. By definition, our edges can either propagate all indices, or propagate all except one index.

7. By only restricting one index variable at a time, the path from n to r specifies a total nesting relationship for all loops referenced by $\theta(n)$.

Hence, we can always find at least one containment ordering between a set of nested loops. However, different paths between n and r may specify different orderings. This is an over-specification of loop nesting and we discuss how we detect it in the next sub-section.

```
TemporalIndex i, j;

IndexedScalar a(i), b(j);
IndexedScalar c(j), d(i);

a[-1] = 1;
b[-1] = 2;

// Implies that loop i is contained in loop j
a[i] = a[i-1] + b[j-1];
c[j] = a[final];

// Implies that loop j is contained in loop i
b[j] = a[i-1] + b[j-1];
d[i] = b[final];

solve.setNewValue(u, c[final] + d[final]);
```

Figure 7.6: A declarative loop specification that cannot be satisfied. Both `a` and `b` are only valid inside the nested scopes of loops `i` and `j`. However, it is impossible to infer whether $i \subset j$ or vice-versa. We note that the expression assigned to u has no indices associated with it, and is therefore scoped correctly according to our definition.

7.3.3 Over-specification of loop nesting

It is possible to over-specify loop nesting i.e. declare conflicting constraints on how loops should be nested. We show an example of this in Figure 7.6.

As both `a` and `b` have expressions assigned to them using both indices i and j, loops i and j must be nested. However, the first use of `final` implies that `i` is the innermost loop, and the second use implies that `j` is.

Our algorithm for determining loop nestings will handle this case by constructing multiple scopes for identical loop variables, corresponding to the different conflicting loop nestings. We can detect this case at run-time and raise an error to inform the library client.

7.3.4 Invalid dependencies

We consider the attempt to construct a value that has a cyclic dependency in Figure 7.7. Unlike our `Scalar`, `Field` and `Operator` handles, which can be assigned expressions involving

```
TemporalIndex i;
IndexedScalar s[i];
IndexedScalar t[i];

s[i] = t[i] + 1;
t[i] = s[i] + 1;
```

Figure 7.7: Construction of invalid values in Excafé. This will raise an exception.

themselves to progressively build up an expression, indexed types can only be assigned once during an iteration. We prevent invalid constructions like the one in Figure 7.7 by requiring that all indexed values used on the RHS of an assignment must come from a previous iteration.

7.4 Conclusion

In this chapter, we have described how we infer an imperative loop structure from an iteration specification represented in the form of a DAG. This enables recipes for iteratively constructed values to be specified to Excafé in a form far more similar to mathematical notation than using imperative constructs such as *for* or *while* loops. In addition to an algorithm for inferring an imperative representation, we have also described how it is possible to construct an invalid declarative specification and how to detect such situations.

Chapter 8

Incompressible Navier-Stokes Solver

In this section, we describe the implementation of an incompressible Navier-Stokes solver in Excafé. We chose this problem in order explore optimisations for a non-trivial (i.e. time-dependent, multiple term, non-linear and multiple discretisations) test case. Before describing our implementation, we present our derivation of the finite element discretisation of the incompressible Navier-Stokes equations. The complete source of our incompressible Navier-Stokes solver is available in Appendix F.

8.1 The Test Problem

Our chosen problem is the time-dependent, two-dimensional fluid flow around a cylinder in a rectangular channel. This test case is commonly used in the literature [49]. We present the physical layout of our test problem in Figure 8.1. Fluid enters through the left edge Γ_{in}, around the cylinder and exits through Γ_{out}. Fluid velocity is 0 at the edges Γ_{edge} and on the surface of the cylinder.

More formally, we apply the following Dirichlet boundary conditions:

At Γ_{edge} and $\Gamma_{cylinder}$, fluid velocity is (0, 0) ms^{-1}.
At Γ_{in}, fluid velocity is (5, 0) ms^{-1}.

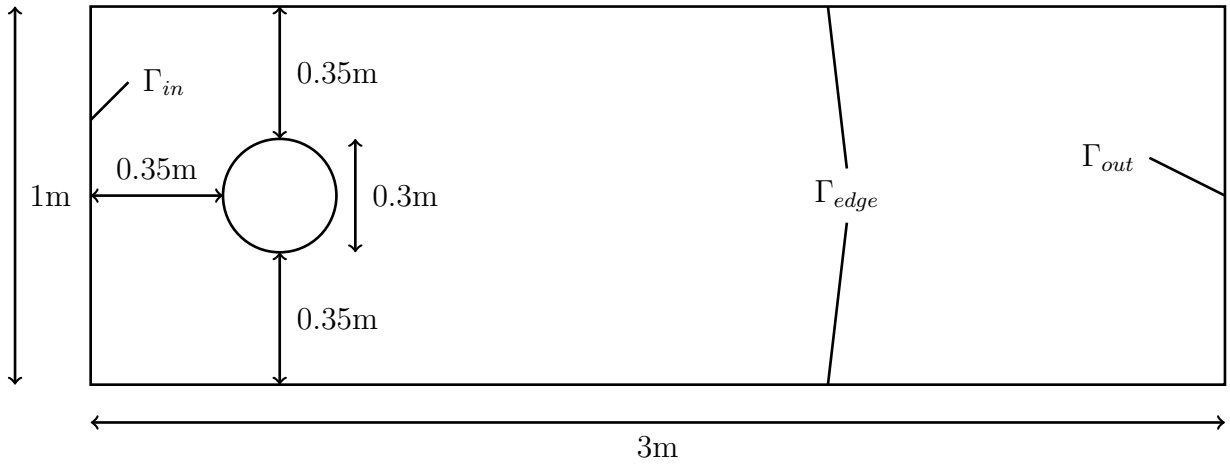


Figure 8.1: The physical layout of our test problem. A cylinder of diameter 0.3m is positioned at (0.5,0.5)m. Fluid enters through the edge Γ_{in} and exits through Γ_{out}.

On the outflow edge Γ_{out}, we will apply the "natural" outflow boundary condition. We describe this in Section 8.4.

Under certain conditions, vortices will be periodically shed from either side of the cylinder. This is known as a *Kármán vortex street*. In order to observe this effect we require that the *Reynolds number*, which describes the ratio of inertial to viscous forces in the fluid, to be within a certain range. We note that the Reynolds number, which primarily characterises the type of flow, is a *dimensionless* value. We have only made use of units when describing our test case to ease comprehension.

The Reynolds number, Re may be defined as [49]:

$$Re = \frac{\bar{U}d}{\nu} \tag{8.1}$$

where $\bar{U}$ is the average fluid velocity upstream of the cylinder,

d is the diameter of the cylinder and

ν is the kinematic viscosity of the fluid.

We let $\nu = 1 \div 250 = 0.004$ in our test problem. Substituting into Equation 8.1 we get:

$$Re = \frac{5 \times 0.3}{0.004} = 375 \tag{8.2}$$

Our system should exhibit vortex shedding behaviour at this Reynolds number.

8.2 Our model

Our derivation is based around that described by Ralf Rannacher's lecture notes [27]. Equations 8.3a, 8.3b and 8.3c describe conservation of momentum, mass and energy, respectively.

$$\partial_t(\rho v) + \rho v \cdot \nabla v - \nabla \cdot (\mu \nabla v + \frac{1}{3}\mu \nabla \cdot vI) + \nabla p = \rho f \tag{8.3a}$$

$$\partial_t \rho + \nabla \cdot (\rho v) = 0 \tag{8.3b}$$

$$\partial_t(c_p \rho T) + c_p \rho v \cdot \nabla T - \nabla \cdot (\lambda \nabla T) = h \tag{8.3c}$$

Our unknowns are the fluid velocity v, fluid pressure p, fluid density ρ and fluid temperature T. Parameters describing our fluid are the heat capacity c_p, heat conductivity λ and dynamic viscosity μ. f and h are our volume forcing and heat source terms, respectively, and are given.

Firstly, we assume that our flow is isothermal. This causes Equation 8.3c to decouple from our system and we do not consider it further. Secondly, we assume that the density of our fluid $\rho = \rho_0$, a constant. This enables us to rewrite Equation 8.3b as follows:

$$\nabla \cdot v = 0 \tag{8.4}$$

Equation 8.4 states that our fluid is incompressible and is called the *incompressibility constraint*. Substituting Equation 8.4 back into Equation 8.3a, letting $\rho_0 = 1$ and $\nu = \mu/\rho_0$ we get the form of the incompressible Navier-Stokes equations that we will discretise with the finite element

method.

$$\overbrace{\partial_t v}^{\substack{\text{unsteady}\\ \text{acceleration}}} + \overbrace{v \cdot \nabla v}^{\substack{\text{convective}\\ \text{acceleration}}} - \overbrace{\nu \nabla^2 v}^{\text{viscosity}} + \overbrace{\nabla p}^{\substack{\text{pressure}\\ \text{gradient}}} = \overbrace{f}^{\substack{\text{volume}\\ \text{force}}} \tag{8.5a}$$

$$\underbrace{\nabla \cdot v}_{\substack{\text{incompressibility}\\ \text{constraint}}} = 0 \tag{8.5b}$$

8.3 Variational Formulation

We let v and p be our velocity and pressure trial functions, respectively. We introduce φ and χ as our test functions in the same function spaces as v and p, respectively.

We also introduce the following notation, where Equations 8.6, 8.7, 8.8 show the shorthand for integrals over the domain involving scalar multiplication, vector inner products and double inner products, respectively:

$$(u, v) = \int_\Omega uv \, dx \tag{8.6}$$

$$\langle u, v \rangle = \int_\Omega u \cdot v \, dx \tag{8.7}$$

$$[u, v] = \int_\Omega u : v \, dx \tag{8.8}$$

Multiplying by our test functions and integrating over our domain Ω:

$$\langle \partial_t v, \varphi \rangle + \langle v \cdot \nabla v, \varphi \rangle - \nu \langle \nabla^2 v, \varphi \rangle + \langle \nabla p, \varphi \rangle = \langle f, \varphi \rangle \tag{8.9}$$

$$(\nabla \cdot v, \chi) = 0 \tag{8.10}$$

As our finite element approximation of v is only of differentiability class C^0 (i.e. we can only

take the first derivative) we apply Green's first identity to the viscosity term, reducing the order of the derivative:

$$\nu\langle\nabla^2 v, \varphi\rangle = -\nu[\nabla v, \nabla\varphi] + \nu\int_{\partial\Omega}(\varphi\cdot(\nabla v\cdot n))\,dS \tag{8.11}$$

We also apply the Divergence theorem to the pressure gradient term:

$$\langle\nabla p, \varphi\rangle = -(p, \nabla\cdot\varphi) + \int_{\partial\Omega} p(\varphi\cdot n)\,dS \tag{8.12}$$

We rewrite our system as follows, excluding the edge integrals.

$$\langle\partial_t v, \varphi\rangle + \langle v\cdot\nabla v, \varphi\rangle + \nu[\nabla v, \nabla\varphi] - (p, \nabla\cdot\varphi) = \langle f, \varphi\rangle \tag{8.13}$$

$$(\nabla\cdot v, \chi) = 0 \tag{8.14}$$

Excluding the edge integrals implicitly gives us boundary conditions on the outflow, which we describe in the next section.

8.4 Outflow Boundary Condition

From Equations 8.11 and 8.12, it follows that:

$$\begin{aligned}\nu[\nabla v, \nabla\varphi] - (p, \nabla\cdot\varphi) = \langle\nabla p, \varphi\rangle - \nu\langle\nabla^2 v, \varphi\rangle \\ - \int_{\partial\Omega} p(\varphi\cdot n)\,dS + \nu\int_{\partial\Omega}\varphi\cdot(\nabla v\cdot n)\,dS\end{aligned} \tag{8.15}$$

Integration by parts of the pressure gradient and viscosity terms gives us boundary integrals that we use to impose Neumann boundary conditions. We impose the "do nothing" boundary

condition by setting these terms to zero on the outflow Γ_{out}. Hence, our outflow boundary condition is:

$$\int_{\Gamma_{out}} (\nu(\varphi \cdot (\nabla v \cdot n)) - p(\varphi \cdot n))\ dS = 0 \tag{8.16}$$

These have been observed to produce satisfactory results for essentially parallel flow [49].

8.5 Temporal Discretisation

We define the following matrices that we will use to define our linear system:

$$\begin{aligned} M_{i,j} &= (\varphi_j^\delta, \varphi_i^\delta)_{i,j=1}^{N_v} \\ A_{i,j} &= \nu(\nabla\varphi_j^\delta, \nabla\varphi_i^\delta)_{i,j=1}^{N_v} \\ N(x)_{i,j} &= \left(\sum_{k=0}^{N_v} x_k \varphi_k^\delta \cdot \nabla\varphi_j^\delta, \varphi_i^\delta\right)_{i,j=1}^{N_v} \\ B_{i,j} &= (\chi_j^\delta, \nabla \cdot \varphi_i^\delta)_{i,j=1}^{N_v,N_p} \end{aligned}$$

where:

φ_i^δ, $1 \leq i \leq N_v$ is the set of discrete basis functions used to approximate v and φ.

χ_i^δ, $1 \leq i \leq N_p$ is the set of the discrete basis function i uses to approximate p and χ.

We ignore our forcing term f, which for our example problem will be 0. We write our system of ordinary differential equations as follows:

$$M\dot{x}^n + Ax^n + N(x^n)x^n - By^n = 0 \tag{8.17}$$

$$B^T x^n = 0 \tag{8.18}$$

where x^n and y^n represent vectors of discrete basis function coefficients used to approximate v and p respectively at time-step n.

We apply a one-step-θ time-stepping scheme:

$$[M + \theta k(A + N(x^n))]x^n - By^n = [M - (1-\theta)k(A + N(x^{n-1}))]x^{n-1} \tag{8.19}$$

$$B^T x^n = 0 \tag{8.20}$$

where k is the time-step length from $t_{n-1} \rightarrow t_n$. Special cases of this scheme include the forward Euler scheme ($\theta = 0$, first-order explicit), the backward Euler scheme ($\theta = 1$, first-order implicit, strongly A-stable) and the Crank-Nicolson scheme ($\theta = \frac{1}{2}$, second-order implicit, A-stable). We adopt the Crank-Nicolson scheme for our solver.

8.6 Linearising the Convective Acceleration Term

To be able to use a linear system solver, all of our operators must be linear (i.e. they must correspond to a bilinear form). However, we have the term $N(x^n)x^n$ which corresponds to the non-linear operator $(v \cdot \nabla v)$.

We linearise our system using the Picard iteration. We approximate the non-linear term by assuming that:

$$v^i \cdot \nabla v^i \approx v^{i-1} \cdot \nabla v^i$$

and repeatedly solve for v^i until our residual becomes acceptable. Rewriting our discretised term, we approximate our discrete non-linear operator as:

$$N(x^n)x^n \approx N(x^{n,i-1})x^{n,i}$$

```
static const std::size_t dimension = 2;

TriangularMeshBuilder meshBuilder(3.0, 1.0, 1.0/900.0);
meshBuilder.addPolygon(Polygon(vertex<2>(0.5, 0.5), 16, 0.15, 0), 5);
Mesh<dimension> mesh(meshBuilder.buildMesh());
```

Figure 8.2: Construction of of a 3x1 mesh in Excafé with a 16-sided regular polygon located at (0.5, 0.5).

Hence, we solve a linear system assembled using successive approximations to the non-linear term, multiple times within each time-step.

8.7 Implementation in Excafé

We briefly describe the implementation of our solver.

8.7.1 Mesh Construction

First, we construct our rectangular domain and add our cylindrical barrier. Since we can only model shapes with straight edges, we approximate the cylinder using a 16-sided regular polygon. We show the code to construct the mesh in Excafé in Figure 8.2.

For mesh generation we use the triangle library [46]. The mesh we generate is shown in Figure 8.3. In our implementation, we have restricted the maximum cell area to $900^{-1}m^2$.

8.7.2 Discretisation

For our solver to be numerically stable, we require that our choice of element satisfies the *Babuška-Brezzi* (BB) condition [49] a.k.a. the *inf-sup* condition. The BB condition prescribes restrictions on the choice of the velocity and pressure discrete function spaces in order that the solution to be stable and unique.

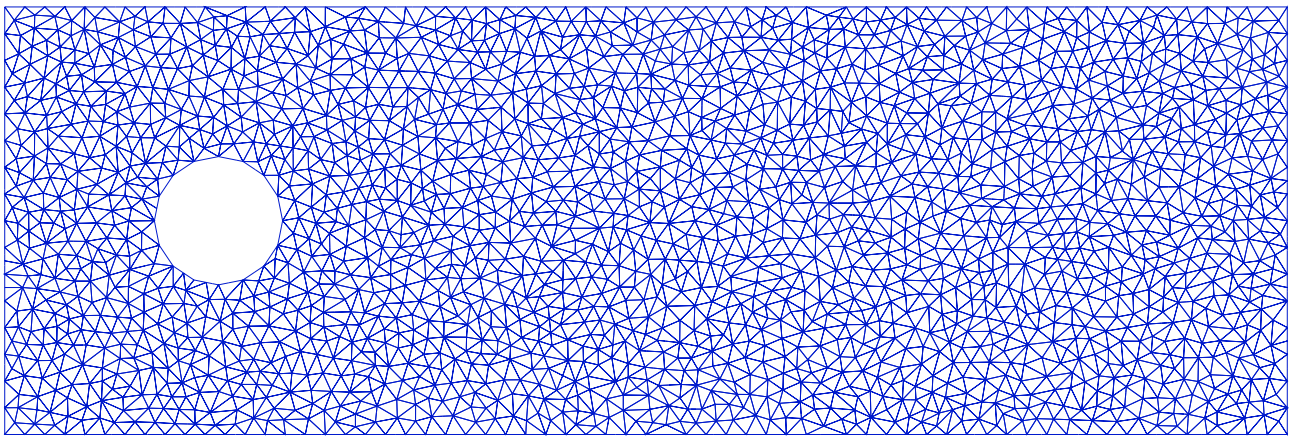


Figure 8.3: The mesh we use for our incompressible Navier-Stokes test problem. Mesh size is 3x1 and maximum cell area is 900^{-1}.

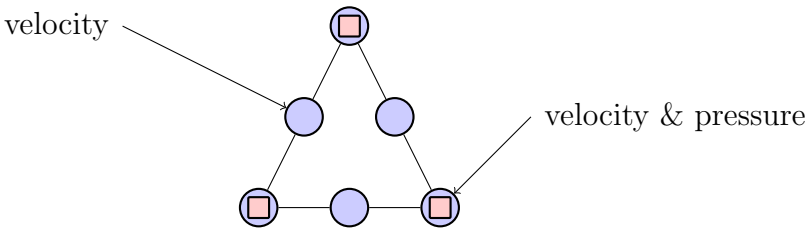


Figure 8.4: A P2-P1 Taylor-Hood element is a two-dimensional triangular element with pressure nodes located at the cell vertices (rectangular nodes) and velocity nodes located at both the cell vertices and edge midpoints (circular nodes). Consequently, pressure is a linear field and velocity is a quadratic field.

We satisfy the BB condition by using the *Taylor-Hood P2-P1* element (shown in Figure 8.4) which is known to satisfy these requirements [49]. We construct our element by using a linear basis for pressure and a quadratic basis for velocity. We show the code we use to declare our elements, function spaces and fields in Figure 8.5. We declare a composite function space that can be used to construct a field that contains both velocity and pressure components.

8.7.3 Boundary Conditions

We recall the Dirichlet boundary conditions we specified in Section 8.1. Boundary conditions in Excafé are currently built by associating mesh facets with constant tensor values. We show the code to do this in Excafé in Figure 8.6 and the resulting labelling in Figure 8.7. Labels 1 to 4 are automatically assigned to the edges of our mesh. Label 5 was specified to the mesh generator for the edges constructed when adding our cylinder approximation to the mesh.

```
Scenario<dimension> scenario(mesh);

Element velocity = scenario.addElement(new LagrangeTriangleQuadratic<1>());
Element pressure = scenario.addElement(new LagrangeTriangleLinear<0>());

FunctionSpace velocitySpace = scenario.defineFunctionSpace(velocity, mesh);
FunctionSpace pressureSpace = scenario.defineFunctionSpace(pressure, mesh);
FunctionSpace coupledSpace = velocitySpace + pressureSpace;

NamedField velocityField = scenario.defineNamedField("velocity", velocitySpace);
NamedField pressureField = scenario.defineNamedField("pressure", pressureSpace);
```

Figure 8.5: Declaration of vector-valued quadratic and scalar-valued linear function spaces. We define the fields that will approximate velocity and pressure using the vector-valued and scalar-valued fields respectively. We also define a function space that can be used to approximate a coupled velocity-pressure field.

```
const Tensor<dimension> zero(1);

Tensor<dimension> inflow(1);
inflow(0) = 5.0;

BoundaryConditionList<dimension> velocityConditionList(1);
velocityConditionList.add(BoundaryConditionTrivial<dimension>(1, zero));
velocityConditionList.add(BoundaryConditionTrivial<dimension>(3, zero));
velocityConditionList.add(BoundaryConditionTrivial<dimension>(4, inflow));
velocityConditionList.add(BoundaryConditionTrivial<dimension>(5, zero));

BoundaryCondition velocityConditions =
  scenario.addBoundaryCondition(velocitySpace, velocityConditionList);
```

Figure 8.6: Construction of rank-1 (vector) valued Dirichlet boundary conditions for the velocity field. Vector-valued values are attached to mesh facets using labels provided by the mesh generator. We do not attach a Dirichlet boundary condition to label 2 since this is the outflow boundary.

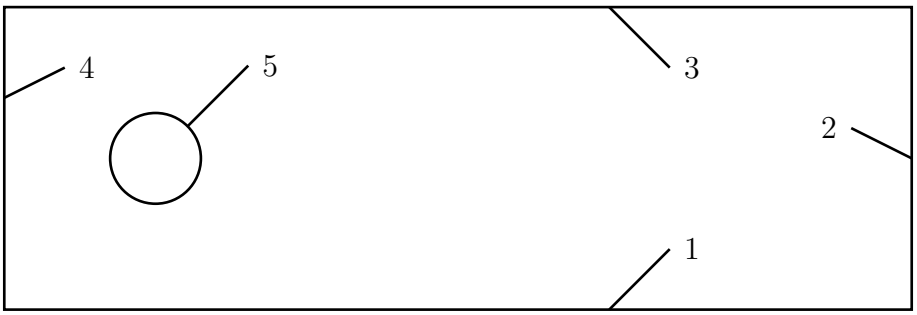


Figure 8.7: The boundary facet labelling used for our test problem. Labels $1-4$ are assigned automatically by the mesh generator and the cylinder was declared to have label 5.

8.7.4 Solver Specification

Specification of a sequence of operations required to solve, or advance, a finite element problem is done through the construction of a `SolveOperation` instance. For our example, we use a coupled solver that solves for velocity and pressure simultaneously.

Our bilinear form `lhsForm` is used to construct the system matrix in the following problem:

$$\begin{pmatrix} M + \theta k(A + N(x^{n,i-1})) & -B \\ B^T & 0 \end{pmatrix} \begin{pmatrix} x^n \\ y^n \end{pmatrix} = \begin{pmatrix} Du^{n-1} \\ 0 \end{pmatrix} \tag{8.21}$$

where D represents the other operator we construct, `nonLinearRhs`.

$$D = \Big(M - (1-\theta)k(A + N(x^{n-1})) \Big) \tag{8.22}$$

The system in Equation 8.21 is repeatedly solved until the residual of the system using the actual non-linear term $N(x^{n,i})$ instead of our approximation $N(x^{n,i-1})$ is less than an acceptable value.

8.7.5 Solver Execution

To execute our solver for multiple time-steps, we execute the `SolveOperation` repeatedly and dump the scenario's tensor fields to VTK file at each time-step. This is shown in Figure 8.9. In future, we hope to also capture the time-stepping loop using our declarative indexing syntax.

8.8 Generated Output

We present the vector fields generated by the Excafé on our example incompressible Navier-Stokes problem in Figures 8.10 and 8.11.

```
SolveOperation step = scenario.newSolveOperation();

Scalar theta = 0.5;
Scalar k  = 0.01;
Scalar kinematic_viscosity = 1.0/250;

Operator nonLinearRhs(velocitySpace, velocitySpace);
nonLinearRhs =
  B(velocity, velocity)*dx +
  B(-(1.0-theta)*k * kinematic_viscosity * grad(velocity), grad(velocity))*dx +
  B(-(1.0-theta)*k * inner(velocityField, grad(velocity)), velocity)*dx;

Field velocityRhs = nonLinearRhs * velocityField;
Field load(project(velocityRhs, coupledSpace));

TemporalIndex i;
IndexedField unknownGuess(i);

unknownGuess[-1] = project(velocityField, coupledSpace) +
                   project(pressureField, coupledSpace);

const forms::BilinearFormIntegralSum lhsForm =
  B(velocity, velocity)*dx +
  B(theta * k * kinematic_viscosity * grad(velocity), grad(velocity))*dx +
  B(-1.0 * k * pressure, div(velocity))*dx +
  B(div(velocity), pressure)*dx +
  B(theta * k * inner(project(unknownGuess[i-1], velocitySpace),
    grad(velocity)), velocity)*dx;

LinearSystem system = assembleGalerkinSystem(coupledSpace, lhsForm, load,
                                             velocityConditions, unknownGuess[i-1]);
Operator linearisedSystem = system.getConstrainedSystem();
unknownGuess[i] = system.getSolution();

Scalar residual = ((linearisedSystem * unknownGuess[i-1]) -
                   system.getConstrainedLoad()).two_norm();

i.setTermination(residual < 1e-3);

step.setNewValue(velocityField, project(unknownGuess[final-1], velocitySpace));
step.setNewValue(pressureField, project(unknownGuess[final-1], pressureSpace));

step.finish();
```

Figure 8.8: Construction of `SolveOperation` object that describes how to advance the simulation by a single time-step. The solution of velocity and pressure components is done simultaneously, producing a field from which the velocity and pressure components are then extracted.

```
for(int i=0; i<6000; ++i)
{
  std::cout << "Starting timestep " << i << "..." << std::endl;
  step.execute();
  std::ostringstream filename;
  filename << "./navier_stokes_" << boost::format("%|04|") % i << ".vtk";
  scenario.outputFieldsToFile(filename.str());
}
```

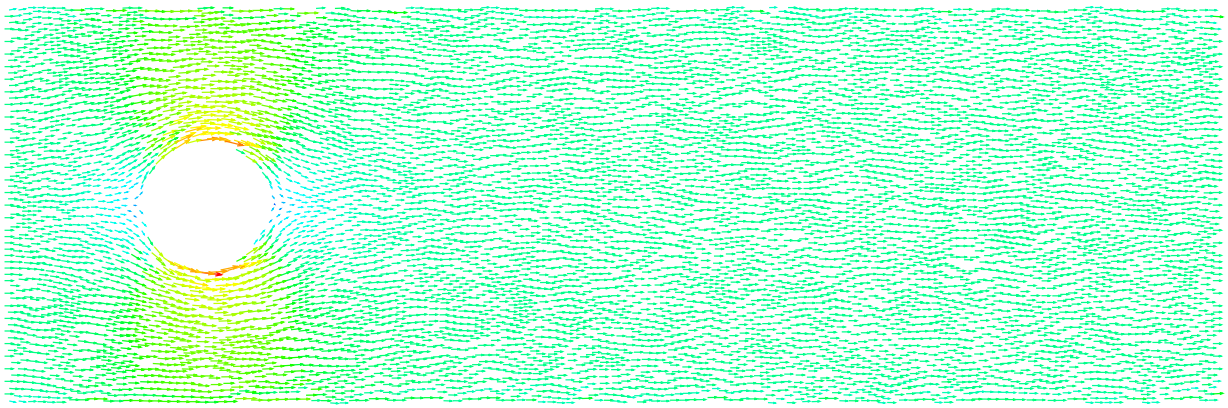
Figure 8.9: The time-stepping loop of our Navier-Stokes solver.

8.9 Conclusion

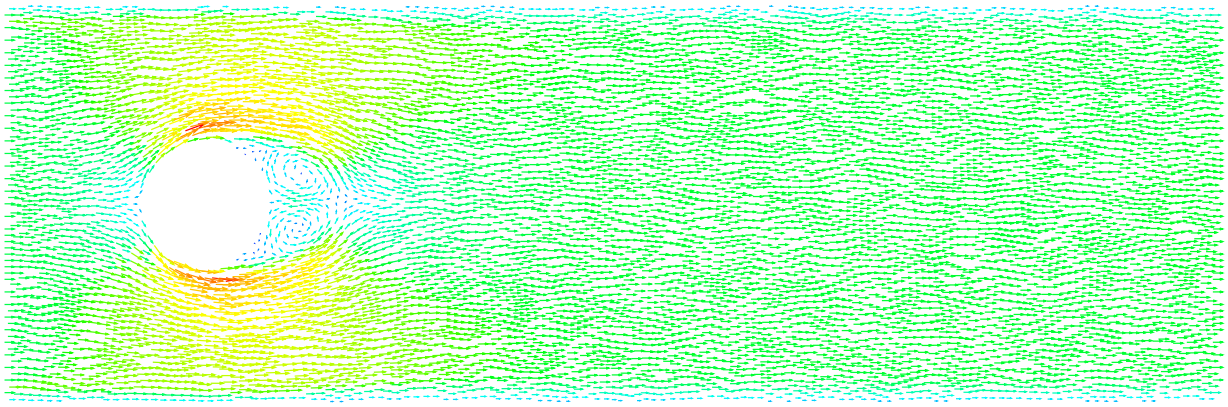
In this chapter we presented the implementation of an incompressible Navier-Stokes solver using Excafé. It also demonstrated an application of the declarative loop syntax, which was used to implement Picard iteration for linearising the convective acceleration term.

The problem presents many interesting aspects that we wish to explore with our local assembly optimisations. These include the use of previously computed fields during assembly, assembly using multiple bilinear forms and reuse of basis functions and their gradients across bilinear forms.

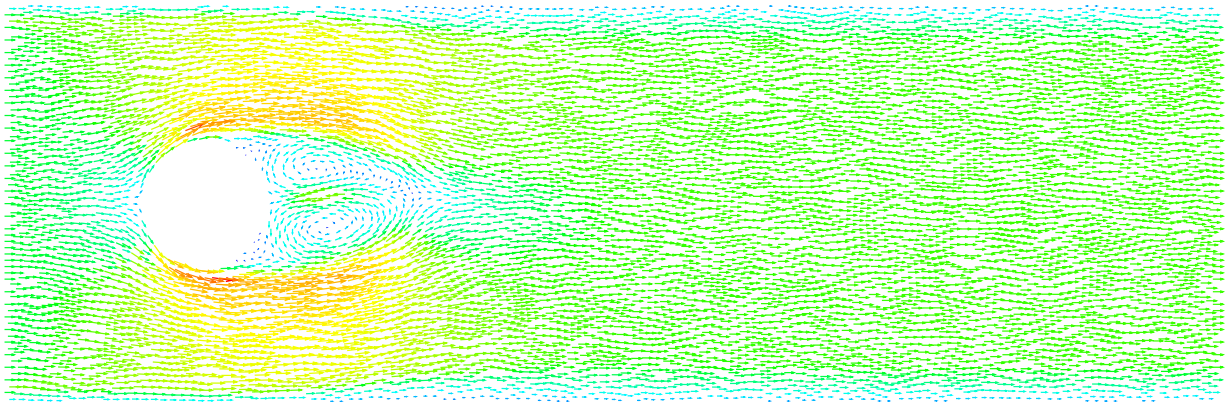
In the next chapter we discuss how Excafé constructs a representation of the local assembly matrix amenable to optimisation and how an optimiser might be able to take advantage of these properties.



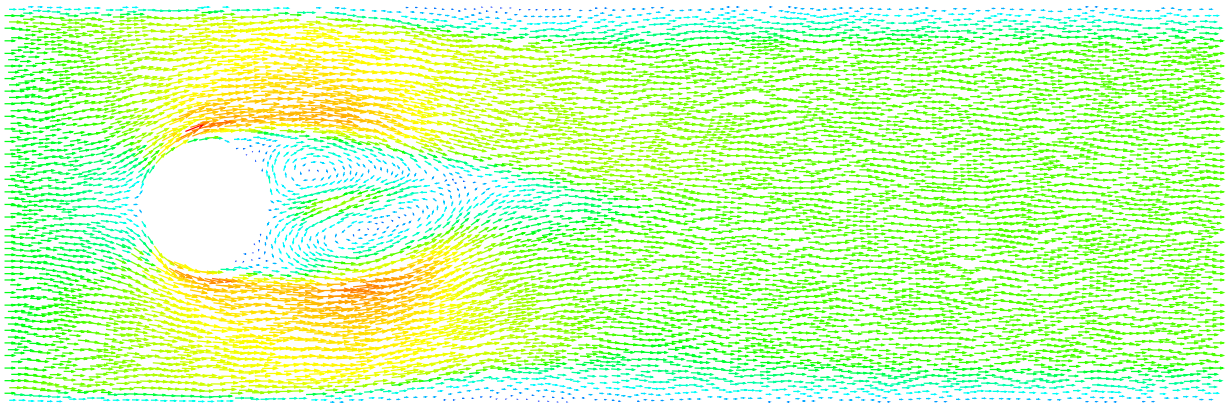
(a) Time-step 0 (0 seconds)



(b) Time-step 9 (0.09 seconds)

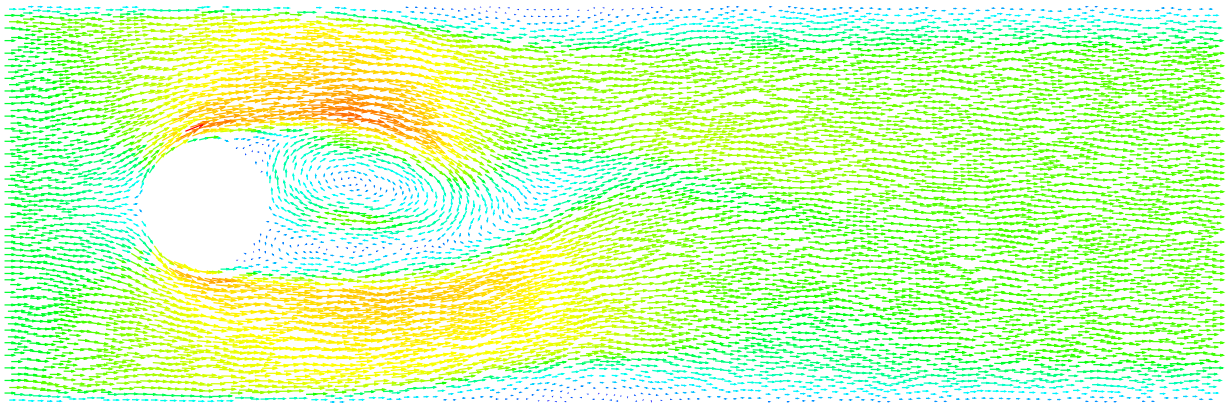


(c) Time-step 18 (0.18 seconds)

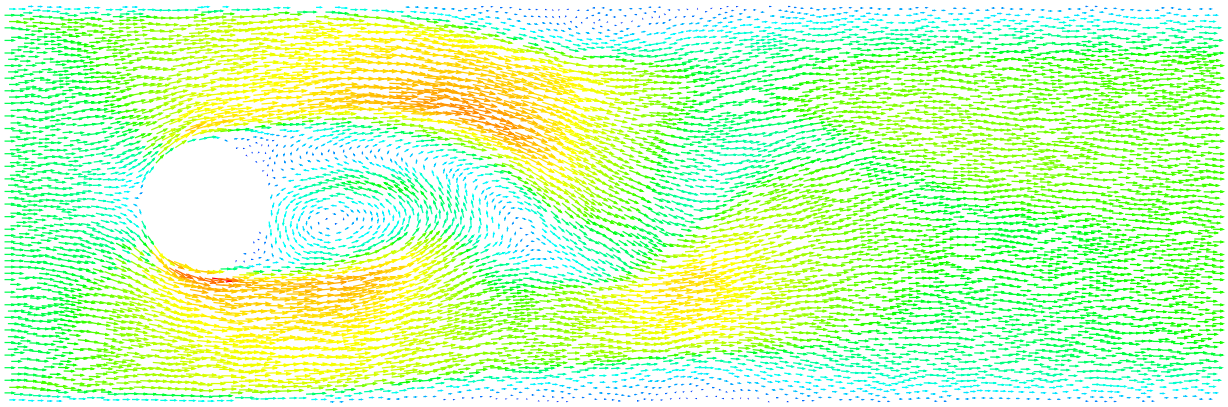


(d) Time-step 27 (0.27 seconds)

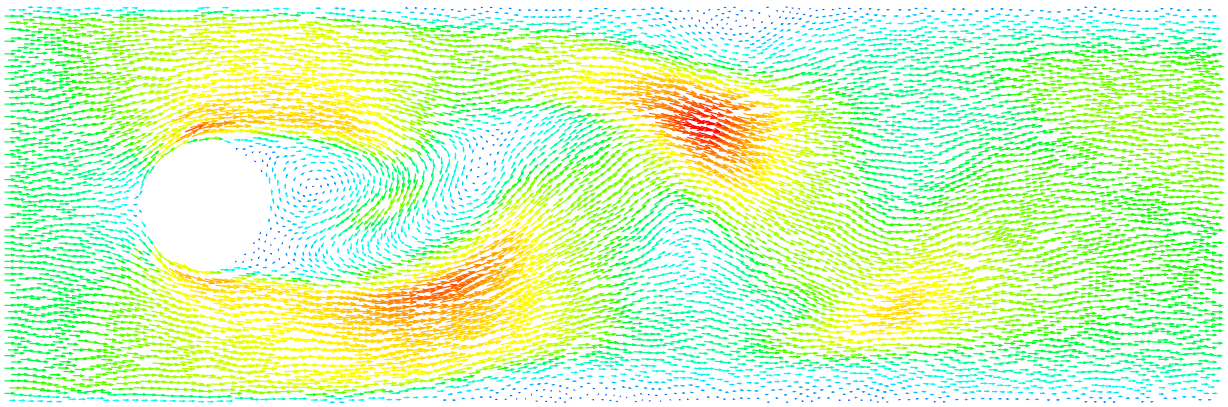
Figure 8.10: The velocity vector fields produced by our incompressible Navier-Stokes solver for the time interval $0 \rightarrow 0.27$ seconds.



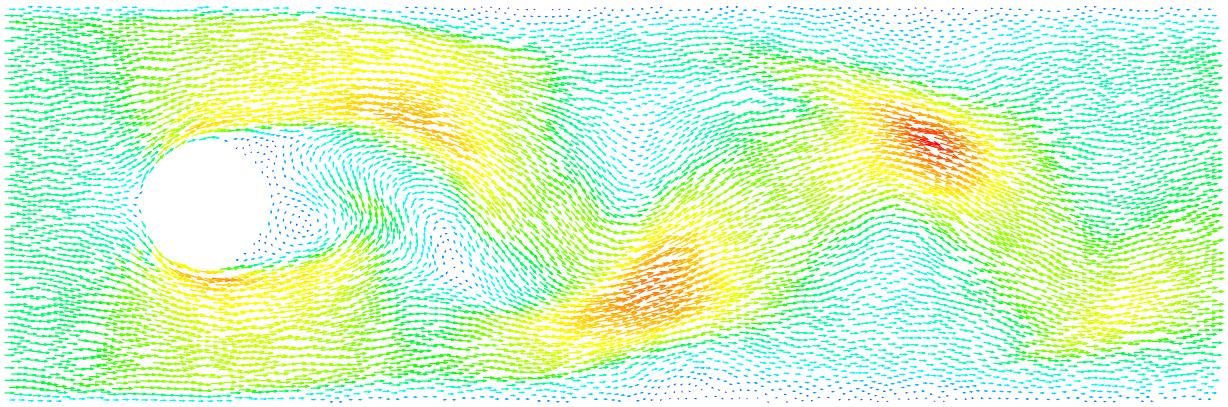
(a) Time-step 36 (0.36 seconds)



(b) Time-step 45 (0.45 seconds)



(c) Time-step 54 (0.54 seconds)



(d) Time-step 63 (0.63 seconds)

Figure 8.11: The velocity vector fields produced by our incompressible Navier-Stokes solver for the time interval $0.36 \rightarrow 0.63$ seconds.

Chapter 9

Local Assembly Construction and Optimisation

In this chapter I describe how Excafé performs *local assembly*, the construction of small dense matrices from cell-local information that are used to form the full sparse linear system of equations. I also discuss the space of optimisations that Excafé's expression capture enables us to explore and compare them to those performed by the FEniCS Form Compiler [36] (FFC), a code generator for variational forms.

In Section 9.1 we provide an overview of the process of local assembly and introduce the notation we use. In Section 9.2 we compare the approaches taken by other finite element libraries to implementing and optimising local assembly to our own. In Sections 9.3 and 9.4 we describe how we construct and optimise a representation of the local assembly matrix in Excafé. In Section 9.5 we compare our approach to the assembly optimisations developed in FFC [38], which are the subject of ongoing research. In Sections 9.6 and 9.7, we present future work then conclude.

9.1 Local Assembly Overview

The finite element method involves rewriting our problem into the weak formulation. This can be described as follows where both u and v are functions of the global co-ordinate x:

find $u \in \mathcal{X}$ such that

$$a(u, v) = L(v), \ \forall v \in \mathcal{V} \tag{9.1}$$

where a and L are bilinear and linear forms respectively. The function spaces $\mathcal{X}$ and $\mathcal{V}$ are called the trial and test spaces, respectively. Both $\mathcal{X}$ and $\mathcal{V}$ contain an infinite set of functions. We approximate them with the discrete function spaces $\mathcal{X}^\delta$ and $\mathcal{V}^\delta$ using the basis sets Φ and Ψ, respectively. Our problem now reads:

find $u^\delta \in \mathcal{X}^\delta$ such that

$$a(u^\delta, v^\delta) = L(v^\delta), \ \forall v^\delta \in \mathcal{V}^\delta \tag{9.2}$$

Our solution u^δ is expressed in terms of a linear combination of basis functions as follows:

$$u^\delta(x) = \sum_{j=0}^{|\Phi|} \hat{u}_j \Phi_j(x) \tag{9.3}$$

We define our discrete linear system as follows:

$$A\mathbf{x} = b \tag{9.4}$$

A and b are the discretised versions of a and L respectively, defined as a matrix and vector as follows:

$$A_{ij} = a(\Phi_j(x), \Psi_i(x)) \tag{9.5}$$

$$b_i = L(\Psi_i(x)) \tag{9.6}$$

and the unknown $\mathbf{x}$ is the vector of basis function coefficients so that:

$$\mathbf{x}_j = \hat{u}_j \tag{9.7}$$

Fortunately, in order to compute A we do not need to evaluate each integral over the entire domain. Our basis functions are defined so that they are only non-zero over neighbouring cells. Hence, it is possible to compute A from a set of cell-local contributions. Calculating the contribution of a single cell is called *local assembly*.

Before we can define the local contribution, called the *local assembly matrix*, we must first define the following:

K, the set of cells that our domain Ω is partitioned into.

$\chi^k(\xi)$, a function that transforms a local co-ordinate ξ on the reference cell to the global co-ordinate x on the cell k.

$\iota^k(i)$, a function that returns the global numbering of the local basis function i defined on cell k.

ϕ and ψ, the local versions of the basis function sets Φ and Ψ respectively. For all local basis functions $1 \leq p \leq |\phi|$, $1 \leq q \leq |\psi|$ on any cell $k \in K$:

$$\Phi_{\iota^k(p)}(\chi^k(\xi)) = \phi_p(\xi) \tag{9.8}$$

$$\Psi_{\iota^k(q)}(\chi^k(\xi)) = \psi_q(\xi) \tag{9.9}$$

We define our local assembly matrix for cell k as follows:

$$M^k_{qp} = a(\Phi_{\iota^k(p)}, \Psi_{\iota^k(q)}) \tag{9.10}$$

The functional a corresponds to an integral over cell k. However, via a change of variables, it is possible to rewrite a so that the integration occurs over a reference cell, Ω_{st}. The integration is now performed w.r.t local co-ordinates and references the local basis sets ϕ and ψ instead of the global ones Φ and Ψ.

The change of variables requires that the integrand be multiplied by the term $|J(\chi^k(\xi))|$. This is the determinant of the Jacobian of the co-ordinate transformation from the reference cell to cell k, which acts as a scaling factor.

The functionals a and L may also contain derivatives of the basis functions with respect to global co-ordinates. Handling these requires use of the chain rule:

$$\frac{d}{dx} = \frac{d}{d\xi}\frac{d\xi}{dx} \tag{9.11}$$

This allows us to compute derivatives of our global basis functions with respect to global co-ordinates in terms of local ones:

$$\nabla\Phi_{\iota^k(p)}(\chi^k(\xi)) = (\nabla\chi^k(\xi))^{-1} \cdot \nabla\phi_p(\xi) \tag{9.12}$$

$$\nabla\Psi_{\iota^k(p)}(\chi^k(\xi)) = (\nabla\chi^k(\xi))^{-1} \cdot \nabla\psi_p(\xi) \tag{9.13}$$

Rewriting the integral so that it is performed over the reference cell means that evaluating it only requires knowledge of how to compute values and derivatives of functions contained in the local basis sets rather than the global ones.

Lastly, we need a method to evaluate our integrals. Typically, this is done numerically rather than analytically, through the use of *quadrature*. A quadrature rule defines an approximation of an integral as a finite summation:

$$\int_{\Omega_{st}} u(\xi)\, d\xi \approx \sum_{i=0}^{Q-1} w_i u(\xi_i) \tag{9.14}$$

The function is evaluated at Q points over the integration region and a weighted sum is constructed. In practice, *Gaussian* quadrature is almost always used since it permits construction of rules that will exactly integrate a polynomial of a given order.

9.2 Implementation Approaches to Local Assembly

We briefly describe the different approaches taken to performing local assembly, including those taken by the FEniCS Form Compiler and Excafé. We consider how to evaluate the local assembly matrix for the Laplacian operator:

$$a(u, v) = \int_{\Omega} \nabla u(x) \cdot \nabla v(x)\, dx \tag{9.15}$$

9.2.1 Quadrature

The standard approach to local assembly is through the use of quadrature [26]. We compute our local assembly matrix for some cell k as follows:

$$M^k_{qp} = \sum_{i=0}^{Q-1} w_i (\nabla \chi^k)^{-1} \cdot \nabla \phi_p \cdot (\nabla \chi^k)^{-1} \cdot \nabla \psi_q |J(\chi^k)| \tag{9.16}$$

The values of $\nabla \phi_p(\xi)$ and $\nabla \psi_p(\xi)$ are only dependent on the local co-ordinate ξ and so can be tabulated at the quadrature points and reused for all cell integrals. The gradient transform $(\nabla \chi^k)^{-1}$ and the scaling factor $|J(\chi^k)|$ are cell-dependent and therefore need to be re-computed for each cell being integrated over. However, if the local to global co-ordinate transform is affine, these values do not vary across the element and can be computed only once for each cell, instead of at every quadrature point.

9.2.2 Tensor Representation (FEniCS)

The FEniCS project has investigated the representation of local assembly as a tensor contraction [36] between a geometry independent *reference tensor* A^0 and a *geometry tensor* G_k which is computed for each cell k. Assembly (for affine elements) is performed using a double inner product between these two tensors:

$$M^k = A^0 : G_k \tag{9.17}$$

In the case of our Laplacian operator, the tensors can be defined as follows:

$$A^0_{qp\alpha\beta} = \int_{\Omega_{st}} \frac{\partial \phi_p}{\partial \xi_\alpha} \frac{\partial \psi_q}{\partial \xi_\beta} \, d\xi \tag{9.18}$$

$$G^{\alpha\beta}_k = |J(\chi^k)| \sum_{\gamma=0}^{d} \frac{\partial \xi_\alpha}{\partial x_\gamma} \frac{\partial \xi_\beta}{\partial x_\gamma} \tag{9.19}$$

Here, α, β and γ are co-ordinate directions that we take partial derivatives with respect to. Our trial and test basis function numberings are p and q respectively. It is possible to extend this representation to non-affine mappings if the rank of the reference and geometry tensors are increased.

The cost of assembly is the number of operations required to evaluate the geometry tensor and the contraction between the reference and geometry tensors. In some cases, this is a significant improvement.

Further work on performing assembly using tensor representation has looked at reducing the operation count required to perform the contraction between the geometry and reference tensors [38]. The optimisations involve forming a complexity reducing relation based on Hamming distance and co-linearity between vectors in the reference tensor.

Both tensor and quadrature based assembly have been implemented in FFC, the FEniCS Form

Compiler [36]. FFC is the result of ongoing development and research and provides a basis for comparison with our work.

9.2.3 Excafé

The approach to optimising assembly in Excafé is inspired by the numerous patterns and redundancies observed in the assembly of forms in different finite element implementations. These include:

1. Tensor-valued basis functions constructed from repeating scalar-valued basis functions leading to sparse tensors.
2. Basis functions defined over a one-dimensional region multiplied together to form basis functions for higher dimensions.
3. Quadrature over a one-dimensional region multiplied together to form quadrature for higher dimensions.
4. Variations in when different terms may be partially evaluated based on whether the element is affine.
5. The use of the same basis functions for trial and test functions (always the case in Galerkin methods).
6. The use of the same basis functions to discretise fields as used for the trial and test functions.

To exploit these patterns (if they are exploitable) requires a representation in which they can be exposed. These representations are often specific to the particular optimisation being performed so it is unclear what a general representation for exploring these optimisations should look like.

In Excafé, we use expression capture to construct a representation of assembly that can be analysed symbolically (as opposed to numerically). We observe that there are opportunities

for optimisations at the levels of bilinear forms, and within and between basis functions at the assembly level.

Our optimisation framework is based around representing each element of the local assembly matrix as a scalar-valued function. Therefore, we lose the loop structure that characterises many operations in local assembly. However, we have a representation that should allow any redundancies and form-specific optimisations to be detected.

We expect that such a representation should enable detection of form-specific optimisations that would typically have needed hand coding. Such optimisations would usually have come at the cost of the structure of the written code, as this would also be highly form-specific.

9.3 Assembly in Excafé

An assembly matrix in Excafé is represented by a matrix of scalar-valued expressions. Each expression may contain symbolic values whose actual values are unknown because they are either not yet calculated (e.g. scalar expressions from the discrete expression DAG) or are cell-dependent (e.g. vertex co-ordinates and basis function co-efficients). Further information on Excafé's scalar expression representation can be found in Section D.5.

We describe how Excafé constructs this set of expressions from a bilinear form and basis function representations. Again, we consider the Laplacian operator:

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx \tag{9.20}$$

where u is the trial function, v is the test function and both are scalar valued. We assume we wish to form the assembly matrix for a two-dimensional triangular cell.

9.3.1 Assembly matrix overview

We choose to represent our set of scalar trial and test functions over the reference cell with the symbols ϕ and ψ respectively. We use the linear Lagrange basis functions over the reference triangle, of which there are three. Therefore our assembly matrix is 3x3 where each element corresponds to an integral over the cell using different basis functions. Using Φ and Ψ to represent the global versions of our trial and test functions we wish to compute the following matrix of expressions:

$$M = \begin{pmatrix} \int_k \nabla\Phi_{\iota^k(0)} \cdot \nabla\Psi_{\iota^k(0)}\, dx & \int_k \nabla\Phi_{\iota^k(1)} \cdot \nabla\Psi_{\iota^k(0)}\, dx & \int_k \nabla\Phi_{\iota^k(2)} \cdot \nabla\Psi_{\iota^k(0)}\, dx \\ \int_k \nabla\Phi_{\iota^k(0)} \cdot \nabla\Psi_{\iota^k(1)}\, dx & \int_k \nabla\Phi_{\iota^k(1)} \cdot \nabla\Psi_{\iota^k(1)}\, dx & \int_k \nabla\Phi_{\iota^k(2)} \cdot \nabla\Psi_{\iota^k(1)}\, dx \\ \int_k \nabla\Phi_{\iota^k(0)} \cdot \nabla\Psi_{\iota^k(2)}\, dx & \int_k \nabla\Phi_{\iota^k(1)} \cdot \nabla\Psi_{\iota^k(2)}\, dx & \int_k \nabla\Phi_{\iota^k(2)} \cdot \nabla\Psi_{\iota^k(2)}\, dx \end{pmatrix} \tag{9.21}$$

Unlike M^k, the local assembly matrix for a given cell k, M is a matrix of expressions that can be used to compute M^k, the local assembly matrix for any cell k. Each row corresponds to a different test function and each column to a different trial function. To evaluate M^k from M for some cell k, the actual values of all symbolic values in M must be known. Before we consider integration, we first form a matrix of expressions that represent the value of the bilinear form at some arbitrary point on the cell k in terms of the local co-ordinate system ξ. We name this matrix of expressions E so that:

$$M_{qp} = \int_k E_{qp}(\xi) |J(\chi^k)|\, d\xi \tag{9.22}$$

where $|J(\chi^k)|$ is the scaling factor from our co-ordinate transformation. We can decompose E into the trial and test contributions as follows:

$$E_{qp} = (A_p \cdot B_q) \tag{9.23}$$

where A and B are vectors of expressions defined as:

$$A = \begin{pmatrix} \nabla\Phi_{\iota^k(0)} \\ \nabla\Phi_{\iota^k(1)} \\ \nabla\Phi_{\iota^k(2)} \end{pmatrix} \qquad B = \begin{pmatrix} \nabla\Psi_{\iota^k(0)} \\ \nabla\Psi_{\iota^k(1)} \\ \nabla\Psi_{\iota^k(2)} \end{pmatrix} \tag{9.24}$$

and the elements of both vectors are tensor-valued expressions for the trial and test portions of the bilinear form to be integrated.

9.3.2 Bilinear form expression construction

We recall the basis function definitions from Section 6.6 which we use to define ϕ and ψ, the basis functions defined over our reference cell:

$$\phi_0(\xi_1, \xi_2) = \psi_0(\xi_1, \xi_2) = 1 - \xi_1 - \xi_2 \tag{9.25}$$

$$\phi_1(\xi_1, \xi_2) = \psi_1(\xi_1, \xi_2) = \xi_1 \tag{9.26}$$

$$\phi_2(\xi_1, \xi_2) = \psi_2(\xi_1, \xi_2) = \xi_2 \tag{9.27}$$

We now show how we construct the vector A of expressions corresponding to $\nabla\Phi$. We start with the vector of expressions representing ϕ:

$$\begin{pmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \end{pmatrix} = \begin{pmatrix} 1 - \xi_1 - \xi_2 \\ \xi_1 \\ \xi_2 \end{pmatrix} \tag{9.28}$$

To compute $\nabla\phi$, we analytically differentiate each entry. We write the result as a vector of rank-1 tensors:

$$\begin{pmatrix} \nabla\phi_0 \\ \nabla\phi_1 \\ \nabla\phi_2 \end{pmatrix} = \begin{pmatrix} (-1,-1) \\ (1,0) \\ (0,1) \end{pmatrix} \tag{9.29}$$

We note that all references to the local co-ordinates ξ_1 and ξ_2 have vanished. This is often not the case if we do not take derivatives (e.g. a mass-matrix) or use higher order basis functions (e.g. quadratic Lagrange).

Before we can transform our gradients to the global cell, we first need to define our local to global co-ordinate transformation χ. χ is defined using the already captured linear Lagrange basis functions:

$$\chi(\xi_1, \xi_2) = \begin{pmatrix} (1-\xi_1-\xi_2)v_x^0 + \xi_1 v_x^1 + \xi_2 v_x^2 \\ (1-\xi_1-\xi_2)v_y^0 + \xi_1 v_y^1 + \xi_2 v_y^2 \end{pmatrix} \tag{9.30}$$

where v_x^α and v_y^α represent the x and y components, respectively, of vertex α of some arbitrary cell. As they are cell-dependent, they are also manipulated symbolically.

Applying the gradient operator to find $\nabla\chi$:

$$\nabla\chi(\xi_1, \xi_2) = \begin{pmatrix} v_x^1 - v_x^0 & v_y^1 - v_y^0 \\ v_x^2 - v_x^0 & v_y^2 - v_y^0 \end{pmatrix} \tag{9.31}$$

Our local co-ordinates have vanished due to the fact that our transformation is affine, and therefore constant across the entire cell. We invert to find $(\nabla\chi)^{-1}$:

$$(\nabla\chi(\xi_1,\xi_2))^{-1} = \frac{1}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)}\begin{pmatrix} v_y^2-v_y^0 & v_y^0-v_y^1 \\ v_x^0-v_x^2 & v_x^1-v_x^0 \end{pmatrix} \tag{9.32}$$

$$= \begin{pmatrix} \frac{v_y^0-v_y^2}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{v_y^1-v_y^0}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)} \\ \frac{v_x^2-v_x^0}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{v_x^0-v_x^1}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)} \end{pmatrix} \tag{9.33}$$

We can now form $\nabla\Phi$ by transforming $\nabla\phi$ since:

$$\nabla\Phi = (\nabla\chi)^{-1}\cdot\nabla\phi \tag{9.34}$$

We take the inner product between each element $\nabla\phi_p$ of our vector of gradients and the gradient of the inverse co-ordinate transformation $(\nabla\chi)^{-1}$:

$$A_p = \begin{pmatrix} \frac{v_y^0-v_y^2}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{v_y^1-v_y^0}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)} \\ \frac{v_x^2-v_x^0}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{v_x^0-v_x^1}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)} \end{pmatrix}\cdot\begin{pmatrix}(-1,-1)\\(1,0)\\(0,1)\end{pmatrix}_p \tag{9.35}$$

$$A = \begin{pmatrix} (\frac{v_y^2-v_y^1}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)}, \frac{-v_x^2+v_x^1}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)}) \\ (\frac{v_y^0-v_y^2}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)}, \frac{v_x^2-v_x^0}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)}) \\ (\frac{v_y^1-v_y^0}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)}, \frac{v_x^0-v_x^1}{(v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)}) \end{pmatrix} \tag{9.36}$$

Each element A_p, of A, is the expression for the rank-1 tensor representing the gradient of ϕ_p w.r.t. global co-ordinates, $\nabla\Phi_p$. Since the Laplacian operator is symmetric, and we use the same set of basis functions for our trial and test spaces, we know that $A = B$. For this reason we do not show the derivation of B. We can form the matrix of scalar expressions $E_{qp} = A_p \cdot B_q = A_p \cdot A_q$:

$$E = \begin{pmatrix} \frac{(v_y^2-v_y^1)^2+(-v_x^2+v_x^1)^2}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} & \frac{(v_y^2-v_y^1)(v_y^0-v_y^2)+(-v_x^2+v_x^1)(v_x^2-v_x^0)}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} & \frac{(v_y^2-v_y^1)(v_y^1-v_y^0)+(-v_x^2+v_x^1)(v_x^0-v_x^1)}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} \\ \frac{(v_y^2-v_y^1)(v_y^0-v_y^2)+(-v_x^2+v_x^1)(v_x^2-v_x^0)}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} & \frac{(v_y^0-v_y^2)^2+(v_x^2-v_x^0)^2}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} & \frac{(v_y^0-v_y^2)(v_y^1-v_y^0)+(v_x^2-v_x^0)(v_x^0-v_x^1)}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} \\ \frac{(v_y^2-v_y^1)(v_y^1-v_y^0)+(-v_x^2+v_x^1)(v_x^0-v_x^1)}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} & \frac{(v_y^0-v_y^2)(v_y^1-v_y^0)+(v_x^2-v_x^0)(v_x^0-v_x^1)}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} & \frac{(v_y^1-v_y^0)^2+(v_x^0-v_x^1)^2}{\left((v_x^1-v_x^0)(v_y^2-v_y^0)-(v_y^1-v_y^0)(v_x^2-v_x^0)\right)^2} \end{pmatrix} \tag{9.37}$$

The matrix of expressions E represents the evaluation of the Laplacian at an arbitrary location on the reference cell. It needs to be integrated in order to form the set of expressions that constitute our representation of the local assembly matrix.

9.3.3 Integration

Integration involves two steps. First, we need to multiply by the determinant of the Jacobian of our co-ordinate transform $|J(\chi)|$. Then we need to integrate our expressions over the reference element. Since we have no references to the local co-ordinates ξ_1 and ξ_2, this is simply a multiplication by the area of the reference triangle, 0.5.

$$M = \begin{pmatrix} \frac{(v_y^2-v_y^1)^2+(-v_x^2+v_x^1)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{(v_y^2-v_y^1)(v_y^0-v_y^2)+(-v_x^2+v_x^1)(v_x^2-v_x^0)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{(v_y^2-v_y^1)(v_y^1-v_y^0)+(-v_x^2+v_x^1)(v_x^0-v_x^1)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} \\ \frac{(v_y^2-v_y^1)(v_y^0-v_y^2)+(-v_x^2+v_x^1)(v_x^2-v_x^0)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{(v_y^0-v_y^2)^2+(v_x^2-v_x^0)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{(v_y^0-v_y^2)(v_y^1-v_y^0)+(v_x^2-v_x^0)(v_x^0-v_x^1)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} \\ \frac{(v_y^2-v_y^1)(v_y^1-v_y^0)+(-v_x^2+v_x^1)(v_x^0-v_x^1)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{(v_y^0-v_y^2)(v_y^1-v_y^0)+(v_x^2-v_x^0)(v_x^0-v_x^1)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} & \frac{(v_y^1-v_y^0)^2+(v_x^0-v_x^1)}{2(v_x^1-v_x^0)(v_y^2-v_y^0)-2(v_y^1-v_y^0)(v_x^2-v_x^0)} \end{pmatrix} \tag{9.38}$$

Multiplication by the determinant of the Jacobian actually reduces the order of the quotient in our example. This is due to the fact that the transformation of the basis function gradients involved a multiplication by the inverse of the determinant of the Jacobian. Our resulting matrix of expressions M is a representation of the local assembly matrix for the Laplacian operator for an arbitrary element. In order to evaluate it, we simply need to substitute cell co-ordinate values.

In our example, the local co-ordinates ξ_1 and ξ_2 vanished due to taking the gradient of the basis

functions. In other forms this is not the case so we need to have a way to integrate our matrix of expressions with respect to the local co-ordinate system ξ.

Integration over the reference cell can either be done entirely symbolically, or via the substitution of quadrature points and weights. Symbolic integration is preferred as our local assembly optimiser is also entirely symbolic. Quadrature based integration is less likely to work well with our symbolic framework, since the partial evaluation of expressions after quadrature substitution may result in different floating point coefficients (due to floating point inaccuracies) for terms that were symbolically identical.

We note that both integration techniques, symbolic integration and partially evaluating expressions after quadrature substitution, cause the cost of evaluating the assembly matrix for an arbitrary *linear* cell to become constant. As with FEniCS's tensor representation, the cost of computing an integral over a linear cell becomes independent of the polynomial degree of the basis functions used (but not the number of basis functions). This is a consequence of the Jacobian of the local to global co-ordinate transformation being constant for linear cells.

9.3.4 Referencing other fields and scalars

In many finite element problems, it is necessary to be able to reference other fields. For example, in the case of our incompressible Navier-Stokes problem from Chapter 8, we construct bilinear forms that include the velocity field from the previous time-step, as well as multiple approximations to the velocity at the current time-step due to linearisation of the convective acceleration term using the Picard iteration.

When we reference other tensor fields from our bilinear forms, we also possess complete knowledge about how they have been discretised. Therefore, we can also incorporate the basis functions used to approximate these fields and their coefficients directly into our assembly matrix expressions.

For example, say we have some vector field f represented using linear Lagrange basis functions, designated ϕ, on a 2D triangular element. The three linear basis functions are repeated for

each element of f giving 6 basis functions. The expression for f is derived as follows:

$$f(\xi_1, \xi_2) = \sum_{p=0}^{5} \hat{f}_p \phi_p(\xi_1, \xi_2) \tag{9.39}$$

$$= \begin{pmatrix} (1 - \xi_1 - \xi_2)\hat{f}_0 + \xi_1 \hat{f}_1 + \xi_2 \hat{f}_2 \\ (1 - \xi_1 - \xi_2)\hat{f}_3 + \xi_1 \hat{f}_4 + \xi_2 \hat{f}_5 \end{pmatrix} \tag{9.40}$$

where $\hat{f}_\alpha$ represents basis function co-efficient α for field f. Note that we automatically exploit the sparsity of our basis functions as only three of the six basis functions are non-zero for each vector component.

Lastly, in addition to basis function coefficients, we can also reference any scalar computed inside our discrete expression DAG. This may be useful if one wants to dynamically adjust time-step duration, for example. Like basis function coefficients, references to scalar values are incorporated as symbolic values in our expressions.

9.3.5 Summary

In this section we described the construction of a local assembly matrix in which each element is represented by a scalar-valued rational expression. We construct our expressions using captured basis function representations and apply gradient transformations using co-ordinate transformations constructed from linear basis functions.

We form a set of expressions that have variables corresponding to the local co-ordinates, cell vertex positions and possibly discrete field basis function coefficients and scalar expressions referenced from the discrete expression DAG.

We perform integration on this set of rational expressions over the reference cell. This eliminates all references to our local co-ordinates and gives a recipe for the local assembly matrix over an arbitrary cell in terms of cell vertex positions and any referenced discrete field coefficients or

scalar values from our discrete expression DAG.

In the next section, I explain our approach to optimising this representation.

9.4 Optimisation

After constructing our representation of a local assembly matrix for an arbitrary cell, we need to find an optimised strategy for evaluating it. Each element of our local assembly matrix is represented by a rational function (i.e. the quotient of two multivariate polynomials).

We were unable to find any relevant work into the efficient evaluation of multivariate rational functions. In contrast, there has been much work on the efficient evaluation of multivariate polynomials [50, 51, 52]. Therefore we choose to optimise the polynomial expressions we use to form our rational functions. Unfortunately, there has been significantly less work on how to optimise sets of independent polynomial expressions, which is our goal here. Our optimiser is based on the work of Hosangadi et al. [1]. We present improvements to the Hosangadi et al. common sub-expression elimination and factorisation scheme in Chapter 10.

It is important to note that our work on optimising assembly matrix expressions is currently a work in progress and we do not yet have an effective optimisation strategy. We discuss the current state of this work and observations about what needs to be done to produce an efficient evaluation strategy.

9.4.1 Expression Representation and Complexity

We currently represent rational expressions as a quotient between two expanded polynomials. For our Navier-Stokes assembly matrices, this leads to large expansions in the size of the expressions for the numerator and denominator. In order to counter this, we use the rational function normalisation routines in GiNaC [45] to simplify the local assembly matrix expressions. Unfortunately, multivariate greatest common divisor is a particularly computationally expensive process for large expressions and must often performed heuristically [53].

Our local assembly matrix evaluator currently makes use of the non-factorised rational expressions for each matrix entry when evaluating. We observed significant numerical differences between the computed values of local assembly matrices depending on whether we performed rational function simplification before evaluation (using 64-bit double precision). In particular, our linear solver did not converge on the resulting global system of equations when constructed from local assembly matrices evaluated using non-simplified expressions. We confirmed these issues were entirely numerical by evaluating the same expressions using an arbitrary precision floating point library.

Normalising the local assembly expressions appears to be a requirement for numerical accuracy, but is so computationally expensive we wish to avoid it. In addition, rational functions cannot be simplified by our factorisation pass as it only works on the component polynomials. As a consequence, we intend to modify our polynomial representation so that we can choose when to expand the numerator and denominator of our captured fractions into a sum-of-monomials representation. By doing this, it should be trivial to spot common multipliers on the top and bottom of the fraction before expansion.

Despite fraction simplification, the size of our expressions has led to a need for algorithms that can scale to large sizes and still perform effectively. In particular, in Chapter 10 we present improvements to part of the Hosangadi et al. algorithm for finding an efficient evaluation strategy for a set of polynomials, that enables scaling to larger sets of more complex expressions.

9.4.2 Polynomial Factorisation Effectiveness

Our optimisation of polynomial evaluations is based on the work of Hosangadi et al. [1]. We take the weighting function for factorisations given in [1] and improve its behaviour. In addition, we present a branch and bound algorithm for the matrix covering problem problem presented by Hosangadi et al., in the form of an algorithm for solving a variant of the maximal biclique problem.

Our algorithm successfully scales to the problem sizes we encounter in our Navier-Stokes solver

whilst still finding the optimal matrix covering. In contrast, Hosangadi et al. do not present an algorithm for finding the best matrix covering, simply stating that they choose the "best prime rectangle". We describe our weighting function and search algorithm in detail in Chapter 10.

Although, the matrix coverings we find are optimal with respect to the factorisation weighting function, the algorithm is still greedy. This is due to the fact that it chooses the highest scoring matrix covering at each iteration, which is only a locally optimal choice.

The factorisations produced by our optimisation algorithm are not yet competitive with FFC with respect to floating point operation count. Since the optimisation algorithm is greedy, it is unlikely to be able to find a particularly efficient evaluation strategy when provided with such a large search space.

Our strategy to handle this has not yet been implemented and is part of future work. Our aim is to avoid fully expanding our polynomials, instead explicitly factorising out our geometry related terms before performing common sub-expression elimination. This is not a dissimilar strategy from FEniCS which applies optimisations to the structure of the reference tensor independent from the geometry tensor, which does not possess enough complexity to be a candidate for optimisation. Hence our common sub-expression elimination pass will work mostly on the basis function terms, without the additional complexity of the expanded geometry terms. This search space should be significantly smaller, enabling the factorisation algorithm to perform more effectively.

9.5 Comparison with the FEniCS Form Compiler

Given that the development of our optimisations are still in progress, we cannot yet make an effectiveness based comparison with FFC. However, since the optimisations available in FFC such as tensor contraction based assembly [36] and topological optimisation of assembly matrices [38] are at the leading edge of finite element research, we consider it important to relate the similarities and differences between our approach and the FEniCS approach to assembly optimisation.

We consider the tensor formulation of local matrix assembly to be a particular choice of factorisation influenced by the structure of the computation. We can represent (and intend to implement) a similar factorisation that enables us to analyse the structure of the assembly matrix whilst mainly ignoring the structure of the geometry transformation.

In a similar manner, we consider the transformation to remove quadrature an instance of partial evaluation of the assembly matrix. We effectively eliminate quadrature in Excafé when we integrate our matrix of bilinear expressions.

In contrast, Excafé loses the structure of assembly when it flattens its expressions. As a consequence, the code we generate for assembly will have a rather arbitrary structure, even if it shares the same floating point operation count as a FFC generated assembly. As FFC particularly targets this optimisation and its structure, the code to evaluate an assembly matrix using FEniCS's tensor contraction representation will be extremely succinct.

Our search for assembly optimisations in Excafé is done without prior knowledge of a specific structure to exploit. Hence, we consider FEniCS's topological optimisations of the reference tensor to have more in common with our approach than the representation of assembly as a tensor inner product, as it is dependent on the bilinear form being assembled for.

The FEniCS topological optimisations make use of the structure of the tensor contraction between the reference and geometry tensor. It can be considered as a set of Euclidean inner products between a single vector formed from the geometry tensor and a number of vectors formed from the reference tensor.

If two vectors from the reference tensor v and w are scaled versions of the other such that $v = \alpha w$, the inner product of one with the geometry vector g can be derived from the inner product with the other simply by multiplying the result by a constant value e.g. $v \cdot g = \alpha(w \cdot g)$. This is called the *co-linearity* optimisation.

If two vectors v and w are identical except for ρ entries, the value of the inner product of one with the geometry vector can be computed from the other by subtracting a partial inner product using only ρ entries. This is called the *Hamming distance* optimisation.

Both the co-linearity and Hamming distance optimisations can be considered special cases of factorisation on scalar expressions representing an inner product. Hence, we have the ability to represent each in our framework. We do not need to restrict ourself to these optimisations and can exploit factorisations that might correspond to a co-linearity optimisation with a partial inner product, for example. However, our detection of factorisations is purely symbolic.

We believe we can represent a superset of the FEniCS topological optimisations. However, Kirby et al. state that their use of a minimal spanning tree results in an optimal computation [38] with respect to the optimisations they can represent. This may be problematic to improve upon since we currently use a greedy algorithm for our search.

Work by Ølgaard and Wells [54] shows examples where the operation count to evaluate a variational form using a tensor representation can use hundreds of times more operations than a quadrature one as well as vice-versa. This suggests that there is still significant scope for improvement of local assembly strategies.

Both quadrature and tensor based evaluation of local assembly matrices enforce a certain structure on evaluation strategy. We theorise that this structure comes at the cost of certain lower bounds on operation count that make evaluating certain classes of variational forms inefficient. By searching for an evaluation strategy without enforcing any other structure, we hope to be able to find evaluation strategies that are not representable with other approaches.

9.6 Future Work

We have an optimiser for a local assembly expression matrix that optimises evaluation of the elements by performing CSE and factorisation on the numerator and denominators of all constituent rational expressions simultaneously.

Currently, both the numerator and the denominator of these expressions are represented as expanded polynomials. We intend to modify our expression representation to perform selective expansion so that we can avoid expanding out geometry related terms and move them into

separate variables.

Once this is complete, we expect to be able to collect results comparing the number of floating point operations required to evaluate common forms using quadrature, tensor and Excafé approaches to assembly. We may also investigate the behaviour of our factorisations with respect to the problem cases described by Ølgaard et al. [54]

9.7 Conclusion

In this chapter, we have described how we can construct a representation of an assembly matrix as a set of rational scalar functions of vertex co-ordinates, discrete field coefficients and referenced scalar values.

Our research into optimising this representation is still being developed. We have a framework that performs common sub-expression elimination and factorisation on the polynomials that make up these expressions, described in further detail in Chapter 10.

Currently, our results are not competitive with code generated by the FEniCS form compiler. We have identified why we believe this is the case and have a strategy for reducing the complexity of the expressions we pass to our optimiser, which we hope will produce improved results on the smaller search space.

We have compared our optimisation approach to that taken by the FEniCS project for optimising tensor-based evaluation. We have chosen an approach that enforces less structure on the representation being optimised, which should enable us to find optimisations not representable in other frameworks. However, it also requires that we implement algorithms capable of handling the size of the search space we create, and generate a result of acceptable quality. Therefore, our future research may also involve improving the algorithms described in Chapter 10.

Chapter 10

Polynomial Common Sub-expression Elimination and Factorisation

In Chapter 9, we mentioned that we search for an optimised evaluation strategy for the set of rational expressions representing local matrix assembly. To do this, we construct of an optimised evaluation strategy for the set of polynomial expressions used within those rational expressions.

To optimise evaluation of our set of polynomial functions, we extend the work of Hosangadi et al. [1]. The work by Hosangadi et al. was chosen as a basis for our optimisations as it was particularly tailored to polynomial expressions and could handle multiple polynomials simultaneously. As this work is applicable outside of the finite element method, we have chosen to cover it in a dedicated chapter.

10.1 The algorithm

We start by giving an overview of the factorisation algorithm presented by Hosangadi et al. [1] for the optimising the evaluation of sets of polynomial expressions.

We require the following definitions:

Literal A constant value (e.g. 2.6) or variable (e.g. x).

Cube A product of variables raised to a non-negative integer power (e.g. x^2y). This is the terminology used by Hosangadi et al., and we continue to use it here despite the fact that a Cube appears to correspond to the conventional definition of a *monomial*.

SOP An expression represented as a sum of cubes (e.g. $3a^2b + 2ac^2$).

Cube-Free A SOP is cube-free if the only cube of that can divide every cube in the sum is the cube "1.0" (e.g. $a^2b + cd$ is cube-free however, $a^2b + ac$ is not as it is divisible by the cube a).

Kernel For some polynomial P and cube c, an expression P/c is a kernel if it is cube-free and it has at least two terms. (e.g. letting $P = a^2bc + ac$ makes $P/(ac) = ab + 1$ a kernel).

Co-Kernel The cube dividing the polynomial in a kernel expression. In the above example, this is ac.

We present the algorithm using a worked example. Suppose we wish to optimise simultaneous evaluation of the following two polynomials (the subscripts denote term numberings):

$$ab_{\,(1)} + ac_{\,(2)} \tag{10.1a}$$

$$a^2{}_{\,(3)} + ab_{\,(4)} + d_{\,(5)} \tag{10.1b}$$

The first algorithm step computes all kernels of each polynomial:

$$(ab + ac)/1 = ab + ac \tag{10.2a}$$

$$(ab + ac)/a = b + c \tag{10.2b}$$

$$(a^2 + ab + d)/1 = a^2 + ab + d \tag{10.2c}$$

$$(a^2 + ab + d)/a = a + b \tag{10.2d}$$

	a	a^2	ab	ac	b	c	d
1	0	0	$1_{(1)}$	$1_{(2)}$	0	0	0
a	0	0	0	0	$1_{(1)}$	$1_{(2)}$	0
1	0	$1_{(3)}$	$1_{(4)}$	0	0	0	$1_{(5)}$
a	$1_{(3)}$	0	0	0	$1_{(4)}$	0	0

Figure 10.1: The Kernel Co-Kernel Matrix (KCM) corresponding to the kernels in Equation List 10.2. Each one in the matrix corresponds to a particular way of evaluating the cube identified by the subscript in brackets.

Each kernel represents a factorisation of a polynomial P into the form $C * F_1 + F_2$ where C is a cube and F_1 and F_2 are sums of cubes (F_1 must be non-zero). Equation 10.2d, for example, corresponds to the factorisation $a^2 + ab + d = a(a + b) + d$.

We do not describe the algorithm for kernel and co-kernel extraction here, instead we refer the reader to [1] for full details. Hosangadi et al. show that all minimal factorisations of a polynomial expression can be obtained from the kernels and co-kernels of the expressions. Hosangadi et al. define a factorisation as minimal if F_1 is cube-free.

The next algorithm step reformulates the list of kernels and co-kernels in matrix form. This is called the Kernel-Cube matrix or KCM. Each row corresponds to a co-kernel and each column to a unique cube. The same co-kernel may appear multiple times, if present in kernels from different polynomials. A matrix entry is equal to one if multiplying the entry's associated cube and co-kernel results in a term from the original set of expressions. All other entries are zero.

We show the KCM for our example polynomials in Figure 10.1. Each one in the matrix corresponds to the term in the original set of polynomials denoted by the subscript in brackets. The term can be calculated by multiplying the cubes in the row and column labels. Hence, the KCM matrix describes the different ways in which each term may be evaluated.

A valid factorisation is any sub-matrix formed from a subset of rows and columns of the KCM matrix, where every entry in the sub-matrix is equal to one. Alternatively, we can imagine a factorisation as any block of ones in the KCM matrix that can be formed through an arbitrary permutation of its rows and columns.

	a	a^2	ab	ac	b	c	d
1	0	0	$1_{(1)}$	$1_{(2)}$	0	0	0
a	0	0	0	0	$1_{(1)}$	$1_{(2)}$	0
1	0	$1_{(3)}$	$1_{(4)}$	0	0	0	$1_{(5)}$
a	$1_{(3)}$	0	0	0	$1_{(4)}$	0	0

(a) A KCM covering

The rewritten polynomials:

$$e + ac$$
$$a^2 + e + d$$

where:

$$e = ab$$

(b) The corresponding factorisation

Figure 10.2: A factorisation of our example polynomials from Equations 10.1 that avoids evaluating ab twice.

Having chosen a block of ones, a new variable is defined as the sum of all the cubes (columns) the block covers. This new variable multiplied by the appropriate co-kernel is substituted into every polynomial associated with each kernel (row) the block covers.

We show the sub-matrix in Figure 10.2 that corresponds to the factorisation of our example polynomials. Terms 1 and 4 are replaced by the common expression e.

In order to select the best matrix covering, Hosangadi et al. provide a scoring function for intersections that attempts to describe the number of multiplications and additions saved.

$$score = m * \left((C-1) * \left(R + \sum_{i=0}^{R} M(R_i) \right) + (R-1) * \sum_{j=0}^{C} M(C_j) \right) + (R-1) * (C-1) \tag{10.4a}$$

where:

m is a weighting factor for the number of multiplies,

R is the number of rows (kernels),

C is the number of columns (cubes),

$M(R_i)$ is the number of multiplications in co-kernel i,

$M(C_j)$ is the number of multiplications in cube j

We describe our improvements to this scoring function in Section 10.3.

In order to find the best factorisation, a search must be performed to find the covering that scores the highest with respect to the presented scoring function. Hosangadi et al. do not describe an algorithm for achieving this, only stating that "we pick the best prime rectangle in each iteration". This is then used in an algorithm which greedily extracts intersections from the KCM and removes the factorised terms, only rebuilding the KCM when no further intersections can be found from the current one. We present a branch and bound algorithm for finding intersections that appears to scale effectively to large numbers of cubes and kernels in Section 10.2.

The final step of the algorithm performs single-term common sub-expression elimination (i.e. CSE between individual cubes). This is required because the KCM-based optimisation can only find multiple-term common sub-expressions. This optimisation problem is extremely similar to that faced by conventional compiler CSE frameworks and does not have the same scaling issues as multiple-term CSE. Therefore we have not expended effort on attempting to optimise it.

10.2 Optimised branch and bound algorithm

We present the branch and bound algorithm that we use to find the maximum scoring covering of the KCM matrix.

Firstly, we observe that it is possible to represent the KCM in undirected graph form, treating the KCM as if it were an adjacency matrix. The rows (kernels with co-kernels) and columns (cubes) become vertices of the graph and the ones in the matrix become edges. We show the graph representation of the KCM from Figure 10.1 in Figure 10.3.

As the graph is derived from an adjacency matrix, it is bipartite. Our problem of finding the maximum scoring covering translates to one of finding the maximum scoring complete bipartite graph, or *biclique*. This type of problem is known as a *maximum biclique problem*.

The complexity of finding a maximal biclique is directly dependent on the function used to

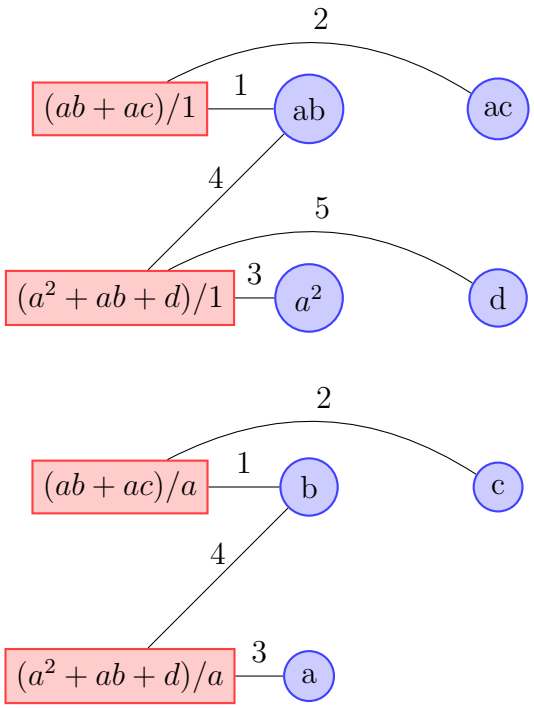


Figure 10.3: The KCM from Figure 10.1 expressed as a graph. Rectangular nodes denote row-vertices (kernels) and circular nodes denote columns (cubes). The edges are annotated with their corresponding term numbers.

.

score the biclique. For example, for a bipartite graph, if the aim is simply to maximise the number of vertices in the biclique, the algorithm is polynomial time [55]. Finding the biclique with the maximum number of edges in a bipartite graph is NP-complete [56].

10.2.1 Definitions and Propositions

Before we describe our search algorithm, we require a number of propositions and definitions. We use the those provided by Liu et al. [57] in their paper on mining large maximal vertex bicliques.

We use the following definitions:

Undirected graph We define an undirected graph G as as a pair (V, E) where V is a set of vertices and E is a set of edges between the vertices.

Adjacent Two vertices are adjacent if there exists an edge between them.

Adjacency List The adjacency list of a vertex v in $G = (V, E)$ is designated as $\Gamma(v, G)$. It is defined as the set of vertices adjacent to v $\{u \in V : (u, v) \in E\}$. We also define the adjacency list of a set of vertices X in $G(V, E)$ such that $\Gamma(X, G) = \{u \in V : \forall v \in X, (u, v) \in E\}$. This requires that each vertex u in $\Gamma(X, G)$ is adjacent to every vertex in X. As a consequence the adjacency list has an anti-monotone property.

Bipartite Graph A graph $G = (V, E)$ is bipartite if its vertex set V can be partitioned into two disjoint non-empty sets V_1 and V_2 such that no edge in E connects two vertices in V_1 or V_2. We denote this $G = (V_1, V_2, E)$.

Biclique A bipartite graph $G = (V_1, V_2, E)$ is biclique if for every $v_1 \in V_1$ and $v_2 \in V_2$ there exists an edge between them. As all edges can be determined from the graph vertices, we also denote a biclique G as $G = (V_1, V_2)$.

We state the following propositions:

Proposition 1. If V_1 and V_2 are both sets of vertices in $G = (V, E)$ and $V_1 \subseteq V_2$ then $\Gamma(V_2) \subseteq \Gamma(V_1)$. This is a consequence of the anti-monotone property of the adjacency list operator.

Proposition 2. If $G = (V_1, V_2, E')$ is a biclique subgraph of $G = (V, E)$ then $V_1 \subseteq \Gamma(V_2, G)$ and $V_2 \subseteq \Gamma(V_1, G)$.

10.2.2 Problem Definition

We first quantify our search problem as follows:

- We have a bipartite graph $G = (K, C, T)$ which is the adjacency list representation of our KCM. K represents all kernel vertices, C represents all cube vertices, and T represents the edges corresponding to terms.
- We have a weight function W which given a biclique (V_1, V_2) computes a score as function of the vertex set sizes $|V_1|$, $|V_2|$ and the multiplication costs of the vertices in each set

$M(v)$ for v in $V_1 \cup V_2$. Our weight function corresponds to the number of numerical operations saved so we wish to maximise it. The function W is monotonically increasing so that given two bicliques $B_1 = (V_1, V_2)$ and $B_2 = (V_3, V_4)$ where $B_1 \subseteq B_2$ we have $W(B_1) \leq W(B_2)$.

- We wish to find a biclique subgraph $B = (V_1, V_2)$ of G such that $W(B)$ is greater than or equal to any other biclique subgraph of G.

We observe that:

1. Given a set of vertices $V \subseteq V_1$ or $V \subseteq V_2$, where the bipartite graph $G = (V_1, V_2)$, $(V, \Gamma(V))$ is a biclique subgraph by construction.

2. The biclique subgraph $(V_1, \Gamma(V_1))$ is a superset of all other biclique subgraphs (V_1, V_2) for any valid V_2.

3. Due to the monotonicity property of W, $W((V_1, \Gamma(V_1)))$ will be greater than or equal to $W((V_1, V_2))$ for any valid V_2.

As a consequence, if (V_1, V_2) is a maximally scoring biclique then $(V_1, \Gamma(V_1))$ will be a biclique of the same score. Hence, we can specify our search problem as the search for the vertex set V_1 where the value of V_2 is implied.

The set of vertices we need to find could either be from the kernels or cubes of our bipartite graph. We have chosen to search for the set of cube vertices that imply our biclique as we find it more intuitive although there is no reason why we could not search for a set of kernels instead.

Given our KCM in adjacency list form $G = (K, C, T)$, our search space is all sets of vertices $C' \subseteq C$. Hence, we have an exponentially-sized search space of $2^{|C|}$ possible bicliques.

10.2.3 Algorithm

Our search algorithm is a best-first search similar to A* [58] with an admissible heuristic. We maintain a priority queue of search spaces being explored.

We require that a total ordering be defined on the cube vertices so that $v_1 < v_2$ or $v_2 < v_1$ for any two non-identical vertices v_1, v_2.

We define a search space as a triple:

$< C, K, S >$

where C is a set of cube vertices.

K is a set of kernel vertices and (C, K) is always the maximal biclique derivable from C. As we stated earlier, K can always be derived as the value of $\Gamma(C)$ however, we maintain K explicitly for efficiency reasons.

S is a cube vertex that we refer to as a *split-point*. We call it such because this vertex describes how the triple will be split into two new search spaces when expanded.

Our biclique search algorithm is presented in Algorithm 10.1. For each cube c in our KCM graph, we construct an initial priority queue of biclique search spaces using c and $\Gamma(c)$. Our search algorithm functions by adding more cube-vertices to each maximal biclique candidate, which in turn reduces the number of kernel-vertices in the biclique. To keep track of our progress exploring the search space, we also need a way to represent which cube-vertices we have considered adding to the biclique and which ones are still candidates.

The KCMs produced by our finite-element local assembly optimiser can contain in excess of tens of thousands of cubes. As our priority list of search spaces may also grow quite large, we cannot afford to explicitly store the set of cubes that may be added to our biclique and those we have discounted. Instead, we handle this implicitly:

1. Cube-vertices are considered for addition to the biclique in the order specified by the total ordering defined earlier on cube-vertices.

Algorithm 10.1 FIND_MAXIMAL_BICLIQUE(kcm_graph)

$(K, C, T) \leftarrow kcm\_graph$
$search\_spaces \leftarrow$ EMPTY_PRIORITY_QUEUE()
for all $c \in C$ **do**
  $search\_space = (c, \Gamma(c),$ FIND_SPLIT_POINT$(\Gamma(c), c))$
  $maximum\_score =$ FIND_MAXIMUM_SCORE$(search_space)$
  INSERT_INTO_PRIORITY_QUEUE$(search_spaces, search_space, maximum_score)$
end for
$best\_biclique \leftarrow (\emptyset, \emptyset)$
while $search\_spaces$ is not empty **do**
  $search\_space \leftarrow$ POP$(search_spaces)$
  if FIND_MAXIMUM_SCORE$(search_space) >$ SCORE$(best_biclique)$ **then**
    if SPLITTABLE$(search_space)$ **then**
      $(C, K, S) \leftarrow search\_space$
      $C' \leftarrow C \cup \{S\}$
      $K' \leftarrow K \cap \Gamma(S)$
      $new\_space_1 \leftarrow (C, K,$ FIND_SPLIT_POINT$((K, S))$
      $new\_space_2 \leftarrow (C', K',$ FIND_SPLIT_POINT$((K', S))$
      $new\_max\_score_1 \leftarrow$ FIND_MAXIMUM_SCORE(new_space_1)
      $new\_max\_score_2 \leftarrow$ FIND_MAXIMUM_SCORE(new_space_2)
      if $K' \neq K$ **then**
        INSERT_INTO_PRIORITY_QUEUE$(search_spaces, new_space_1, new_max_score_1)$
      end if
      INSERT_INTO_PRIORITY_QUEUE$(search_spaces, new_space_2, new_max_score_2)$
    end if
    if SCORE$(best_biclique) <$ SCORE$((C, K))$ **then**
      $best\_biclique \leftarrow (C, K)$
    end if
  end if
end while
return $best\_biclique$

2. Cube-vertices less than the split-point vertex are assumed to have already been considered for addition to the biclique. Cube-vertices greater than or equal to the split point are candidates for addition to the biclique.

Hence, we only need to store a single cube-vertex, the *split-point*, in order to know which cube-vertices are candidates for addition to a given biclique. Each time we examine a biclique we form two new bicliques, one where the split-point has been added to the biclique, and one where it isn't. Each time we add a new cube-vertex c to our biclique's cube-vertex set C, we do not need to recalculate $\Gamma(C)$ to find our set of kernel-vertices K. Instead K can be updated incrementally, by taking the intersection with $\Gamma(c)$.

Due to the anti-monotone property of Γ, each cube-vertex added to a biclique causes the set of kernel-vertices to either remain the same size, or grow smaller. If adding a cube-vertex c to a biclique B does not cause the set of kernel-vertices to become smaller, we do not need to consider the search space for bicliques derived from B where we choose not to add c. This is due to the fact that adding c to to the biclique does impose any additional restrictions on our search space, and we always prefer to form the larger biclique. In matrix-covering terms, if adding a column to an existing covering doesn't reduce the number of rows, there is no need to consider excluding that column. Algorithm 10.1 shows how the choice of adding a cube-vertex to a biclique may result in either one or two new search spaces.

The KCM graphs we generate in some of our finite-element problems have tens or hundreds of thousands of kernel and cube vertices but sparse connectivity. Hence, even though there may be thousands of candidate cube-vertices for addition to a given biclique, only a significantly smaller proportion of them will not cause the biclique's set of kernel-vertices to collapse. Considering the cube-vertices that do cause the biclique to collapse is computationally inefficient.

The only cube-vertices that can be added to a biclique without causing it to collapse are cube-vertices already adjacent to at least one vertex in the biclique's kernel-vertex set. Hence, the only cube vertices worth considering are those that are a member of $\Gamma(k)$ for some $k \in K$ where K is the biclique's set of kernel-vertices. In Algorithm 10.2 we show how we traverse

each cube-vertex greater than the split-point and adjacent to at least one kernel-vertex when finding the next split-point. In this way, we trim our biclique search spaces of cubes that cause them to collapse.

Algorithm 10.2 FIND_SPLIT_POINT(kernels, old_split_point)

```
split_point ← null vertex
for all kernel ∈ kernels do
  for all cube ∈ Γ(kernel) do
    if split_point = null vertex or old_split_point = null vertex or (c < split_point and
    c > old_split_point) then
      split_point = c
    end if
  end for
end for
if split_point ≠ null vertex then
  return split_point
else
  error "search space exhausted"
end if
```

Our FIND_MAXIMUM_SCORE function is the equivalent of an admissible heuristic in an A* search. We need to always exactly or over-estimate the score of the biclique we may derive from a biclique being grown in order for our algorithm to find the maximal biclique. We also want this estimate to be as close to the true value as possible so we can prune search spaces if the maximum possible score we can obtain is smaller than that of a biclique we've already found.

In order to estimate the maximum score of the biclique that can be derived from an existing biclique, we require the sizes of the kernel-vertex and cube-vertex sets, and the multiplication costs of those vertices. By definition, our kernel-vertex sets either remain the same size or grow smaller as the biclique is grown, so they provide a sufficient upper bound for those values. As for the cube-vertex set, any cube-vertex adjacent to one of the vertices in the biclique's kernel-vertex set could be present in the final biclique.

We already observed that the only cube-vertices worth adding to the biclique are adjacent to at least one of the biclique's kernel-vertices. In addition, we observe that for some biclique (C, K)

$$C = \{c \mid \forall k \in K, c \in \Gamma(k)\}$$

i.e. the cube-vertex set is equal to the intersection of the adjacency sets of each kernel-vertex (the reciprocal is also true). This implies that for any biclique (C', K') derived from (C, K), there is at least one $k \in K$ such that $C' \subseteq \Gamma(k)$ as $K' \subseteq K$. As we do not know which members of K will be in the final biclique, we estimate C' as:

$$C \cup \{c \mid c \in \Gamma(k), c \geq S\}$$

where S is the split-point of the biclique being grown. We do this for each $k \in K$ and take the maximum value. We show this in Algorithm 10.3.

Algorithm 10.3 FIND_MAXIMUM_SCORE(search_space)

```
maximum_score ← 0
(C, K, S) ← search_space
for all kernel ∈ K do
  candidate_cubes ← {c | c ∈ Γ(kernel), c ≥ S}
  local_score ← SCORE(C ∪ candidate_cubes, K)
  if local_score > maximum_score then
    maximum_score ← local_score
  end if
end for
```

10.3 Improved weighting function

In this section we describe how the weight function presented by Hosangadi et al. [1] can be improved further. We start by describing the construction of the original weight function from Equation 10.4.

	$\dots$	c_1	c_2	$\dots$	c_C	$\dots$
$\vdots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
k_1	$\dots$	c_1k_1	c_2k_1	$\dots$	c_Ck_1	$\dots$
k_2	$\dots$	c_1k_2	c_2k_2	$\dots$	c_Ck_2	$\dots$
$\vdots$	$\dots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\dots$
k_R	$\dots$	c_1k_R	c_2k_R	$\dots$	c_Ck_R	$\dots$
$\vdots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

Figure 10.4: An abstract Kernel Co-Kernel matrix with a covering of R co-kernels (rows) and C cubes (columns). The computed expressions are shown inside the intersection.

10.3.1 The original weighting function

We consider the KCM in Figure 10.4. The biclique represents a number of subexpressions of an expanded polynomial:

$$c_1k_1 + c_2k_1 + \ldots + c_{C-1}k_1 + c_Ck_1 \tag{10.5a}$$

$$c_1k_2 + c_2k_2 + \ldots + c_{C-1}k_2 + c_Ck_2 \tag{10.5b}$$

$$\ldots \tag{10.5c}$$

$$c_1k_R + c_2k_R + \ldots + c_{C-1}k_R + c_Ck_R \tag{10.5d}$$

extracting the common factor as f gives:

$$k_1f \tag{10.6a}$$

$$k_2f \tag{10.6b}$$

$$\ldots \tag{10.6c}$$

$$k_Rf \tag{10.6d}$$

$$f = c_1 + c_2 + \ldots + c_{C-1} + c_C \tag{10.6e}$$

We can see that the evaluation of the non-factorised version requires:

$\sum_{i=0}^{R} M(k_i)$ multiplies to evaluate all co-kernels, repeated for every column giving $C * \sum_{i=0}^{R} M(k_i)$ multiplies.

$\sum_{j=0}^{C} M(c_j)$ multiplies to evaluate all cubes, repeated for each row giving $R * \sum_{j=0}^{C} M(c_j)$ multiplies.

$R * C$ multiplies to multiply each cube by each co-kernel.

$C - 1$ additions for each row giving $R * (C - 1)$ additions in total.

giving:

$$flops_{\text{unfactorised}} = C * (R + \sum_{i=0}^{R} M(k_i)) + R * \sum_{j=0}^{C} M(c_j) + R * (C - 1)$$

floating point operations to evaluate the non-factorised intersection. After factorisation, we require:

$\sum_{i=0}^{R} M(k_i)$ multiplies to evaluate all co-kernels, only performed once.

$\sum_{j=0}^{C} M(c_j)$ multiplies to evaluate all cubes, only performed once.

R multiplies to multiply the value of f by each co-kernel.

$C - 1$ additions to evaluate the new term f.

giving:

$$flops_{\text{factorised}} = R + \sum_{i=0}^{R} M(k_i) + \sum_{j=0}^{C} M(c_j) + (C - 1)$$

The *score* function represents the number of floating point operations saved:

$$score = flops_{\text{unfactorised}} - flops_{\text{factorised}}$$

10.3.2 Improving the weighting function

In this section, I show the problems with the weighting function presented by Hosangadi et al. and how it may be improved.

	a	b
1.0	1	1

Figure 10.5: The KCM for the expression $a + b$. The covering corresponds to the useless factorisation $1.0(a + b)$ as it is impossible to reduce the operation count of this expression.

	a^2	ab	a	b
1.0	1	1	0	0
a	0	0	1	1

Figure 10.6: The KCM for the expression $a^2 + ab$. The covering shown corresponds to the factorisation $a(a + b)$.

We consider the non-factorisable expression $a+b$. We show a covering corresponding to factorisation in Figure 10.5. Clearly, this factorisation will not reduce the operation count. However, according to the weight function presented by Hosangadi et al., it reduces the operation count by one.

The problem is that according to our weight function, the factorisation reduces the operation count by avoiding one multiplication with the co-kernel, 1.0. Instead, the factorisation will simply create a new variable equal to the original expression. Furthermore, the weight function will also show that it can reduce the operation count of this expression, and factorisation will continue forever.

There are two obvious ways to solve this problem:

1. Avoid considering co-kernels whose value is 1.0.

2. Adjust our weighting function for this case so that it reflects the true number of operations saved, 0.

The first solution is problematic since it prevents factorisations involving any kernels whose co-kernel is one, resulting in missed factorisations. We will show that second solution is also problematic, since fixing the scoring function for this case will also result in missed factorisations.

		Kernel		
		one	constant	variable
Co-Kernel	one	0	0	0
	constant	0	0	1
	variable	0	1	1

Figure 10.7: Number of multiples required to evaluate a term from its cube and co-kernel when they are either 1.0, any other constant value or a variable.

We show the KCM for the expression $a^2 + ab$ in Figure 10.6. This time, we have a desirable factorisation. However, the covering has an identical score to that from Figure 10.5. Any modification to the scoring function that prevents infinite factorisations on the expression $a + b$ will miss the factorisation of $a^2 + ab$.

It is impossible to construct any scoring function that meets these criteria unless the terms 1.0 and a are treated differently. The Hosangadi et al. algorithm treats them identically (i.e. literals that require no multiplies to evaluate). By distinguishing between the 1.0 literal and other cases, we can construct a weighting function that more accurately reflects the number of multiplies saved. However, we can take this even further by identifying cases where the cube and co-kernel are both constant values. The majority of compilers will perform this at compile-time (or the expression can be rewritten) so that no multiply operation is performed at all.

In order to fix the weighting function, we can no longer assume that evaluating a term from its cube and co-kernel always requires one multiply. Figure 10.7 shows the number of multiplies required to evaluate a term from its cube and co-kernel. By comparison, the Hosangadi et al. weighing function is equivalent to all entries in the table being equal to 1.

In unfactorised form, we assumed that the number of multiplies required to evaluate our terms from their kernels and co-kernels was $C * R$ (the number of entries in the covering). Instead, we approximate this as:

$$C_{\text{variable}} * R_{\text{variable}} + C_{\text{constant}} * R_{\text{variable}} + C_{\text{variable}} * R_{\text{constant}}$$

In factorised form, we assumed that R multiplies were required to multiply the factorised term f by each co-kernel. Instead, this becomes:

$R_{\text{constant}} + R_{\text{variable}}$

Our improved weight function:

$$\begin{aligned} score_{\text{improved}} =& (C-1) * \sum_{i=0}^{R} M(R_i) + (R-1) * \sum_{j=0}^{C} M(C_j) + \\ & (R-1) * (C-1) + \\ & R_{\text{variable}}(C_{\text{variable}} + C_{\text{constant}} - 1) + R_{\text{constant}} * (C_{\text{variable}} - 1) \end{aligned} \tag{10.7}$$

10.4 Experimental Results

As the performance of our factorisation pass is highly dependent on the structure of the expressions operated on, we do not currently have a way of objectively quantifying its performance. Instead, we present a rough idea of its performance behaviour using the local assembly matrices we have generated in our Navier-Stokes solver.

Our Navier-Stokes assembly matrix is for a coupled system using a quadratic velocity field (6 nodes x 2 dimensions = 12 DOFs) and a linear pressure field (3 DOFs) giving an assembly matrix of 15x15=225 elements. Each element is a rational function represented using two polynomials, giving 450 polynomials to factorise initially. We note that many of the denominators will be identical.

Our initial set of expressions has 1611 literals, 21 of which are variables, the rest being constants. Of the 21 variables, 6 are vertex positions, 3 are scalars and 12 are basis function coefficients from the previous velocity field.

We present statistics in Table 10.1 for our factoriser. We show the number of kernels, cubes and non-zero values (terms) in the KCM matrix at each iteration. We also show the number of floating point operations required to evaluate our polynomials before factorisation, the number of floating point operations saved by our choice of factorisation and the time taken to find it. Each factorisation results in a new term being added to the set of polynomials, causing the

Iteration	Polynomial Count	Kernels in KCM	Cubes in KCM	Terms in KCM	Total FLOPs to evaluate polynomials	FLOPs saved by factorisation	Time to find factorisation (seconds)
1	450	50787	87466	333241	80834	8484	1.02835
2	451	41137	90504	271729	72350	8104	0.91792
3	452	33439	93545	217639	64246	3776	0.754398
4	453	30392	95091	195233	60470	3754	0.693411
5	454	27173	96690	173100	56716	3698	0.634616
6	455	22687	98202	149460	53018	3698	0.504207
7	456	19680	99718	130144	49320	3434	0.46729
8	457	17088	101414	113189	45886	3154	0.421542
9	458	15109	103034	100422	42732	1321	0.389409
10	459	14413	103056	97187	41411	755	0.41561
11	460	13682	103078	91054	40656	657	0.416083
12	461	13527	103080	89166	39999	657	0.411966
13	462	13372	103082	87287	39342	643	0.410865
14	463	13218	103084	85640	38699	636	0.405262
15	464	13060	103086	84026	38063	629	0.397605

Table 10.1: Time taken for our algorithm to find the maximum scoring covering for the first 15 iterations of our factoriser. We also present statistics on the dimensions and sparsity of the kernel cube matrix (KCM) and the reduction in floating point operations each iteration.

number of polynomials to increase each iteration. We present a graph showing the decrease in FLOP count for the first 500 iterations of the factoriser in Figure 10.8.

For our Navier-Stokes assembly matrix, total time to execute the factoriser including overheads (such as rebuilding the KCM each iteration) was 225 seconds. Factorisation finishes on iteration 1287 giving 1736 polynomials for this instance and a final FLOPs value of 8147. This might seem surprising given that it is a reduction of almost 10x over the initial FLOPs value of 80834, however, we note that expanded polynomial form is an extremely inefficient way to evaluate expressions. We also note that since our vertex numbering is non-deterministic, different factorisations may occur on different runs when two bicliques score the same value.

Our test system had a 3.6 Ghz Pentium 4 processor with 2MB L2 cache and 2GB RAM. It was running Ubuntu 10.04 (Lucid Lynx), 32-bit. Our factoriser (as well as the rest of Excafé) was compiled with g++ 4.4.3 with '-O2' optimisation.

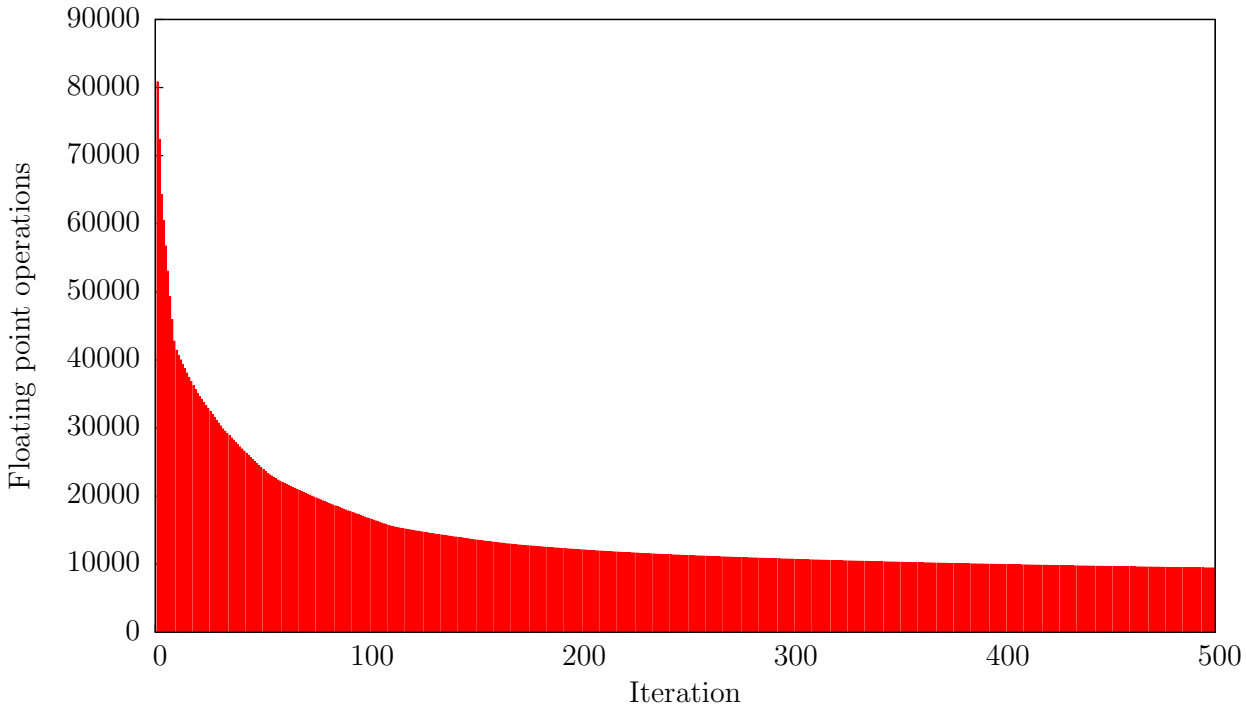


Figure 10.8: The number of floating point operations required to evaluate the local assembly matrix polynomials generated by the Excafé Navier-Stokes solver, for the first 500 iterations of our polynomial CSE pass. It does not include the division operations required to evaluate the rational functions from their constituent polynomials (225).

10.5 Conclusion

We have presented an extension to the work of Hosangadi et al. for performing common sub-expression and factorisation on sets of polynomial expressions. We show how the covering scoring function presented by Hosangadi et al. may lead to infinite recursion due to misestimation of expression operation count and that a truly representative function requires the ability to distinguish between different types of literal.

We also describe an algorithm for finding a maximum scoring covering of the kernel cube matrix. It reformulates the covering problem by representing the KCM as a bipartite graph and searching for the maximal scoring biclique. Properties of bipartite graphs are used to place bounds on the maximum score an existing biclique may be grown to and the candidate vertices to be added to the biclique, allowing the search space to be pruned.

10.6 Future Work

Despite our improvements to the covering scoring function and biclique search algorithm, the Hosangadi et al. algorithm is still greedy. Further work is required to see if it is possible to produce an algorithm with better heuristics or guarantees on the quality of factorisation. Such as algorithm might be derived from the Hosangadi et al. work, or a generalisation of other work on developing efficient evaluation strategies for individual polynomials.

Chapter 11

Conclusion

We conclude in this chapter by examining how this thesis supports the contributions it claims, followed by a discussion of what we consider important and novel about our investigation and the insights that can be drawn. We close with a discussion of future work.

11.1 Summary of Thesis Achievements

We review our contributions from Section 1.3.

- *We present the extension of our delayed-evaluation runtime code-generation library "*Desola*" to sparse linear algebra.*

  We present our extension of Desola to sparse linear algebra in Section 3.8. In Section 3.8.2 we present two variants of code generation for performing iteration over the elements of a sparse matrix stored in CSR format. Desola was modified to generate both these variants. We also explore specialisation of our generated code to the most frequent matrix row lengths.

- *We show that some of the important optimisations applied to our dense linear iterative solvers are also applicable to our sparse ones. We also show that these optimisations*

are beyond the automatic optimisation capabilities of modern compilers and require a generative approach in order to be implemented effectively.

In Section 3.8.2 we show a code fragment to illustrate the effect of our SUIF-level loop fusion pass on two parse matrix-vector multiplies. The pass successfully fuses the outer loop, but is unable to handle the inner loop as it has non-constant bounds.

We discuss the implementation of an optimisation pass specifically designed to fuse matrix-vector multiplies together and a new node type in the DESOLA expression DAG designed to represent them. After applying this pass, we can generate code for the fused sparse matrix-vector multiplies directly, showing how we have used knowledge of sparse matrix iteration to generate code containing optimisations beyond that of a conventional compiler.

- *We present new performance results of sparse linear iterative solvers implemented using the Intel Math Kernel Library against our extended* DESOLA *implementation.*

  We present results from our set of sparse linear iterative solvers running on matrices from the University of Florida sparse matrix collection in Section 3.8.4. We collect results for multiple variants of code-generation as well as for the same Iterative Template Library solvers using the Intel Math Kernel Library for all computational kernels. As solver performance is directly dependent on sparse matrix structure, we require different graphs for each solver-matrix combination. We provide full results in Appendix B.

- *We present the design and implementation of a C++ finite element library that performs expression capture of multiple aspects of the finite element method.*

  We provide an overview of EXCAFÉ in Chapter 4 and describe in detail the more novel aspects of EXCAFÉ's expression capture in Chapter 6. We discuss EXCAFÉ's handling of local assembly in detail in Chapter 9. We also discuss some of EXCAFÉ's design decisions in Appendix C and document some of the data structures it uses in Appendix D.

- *We show that expression capture of certain aspects of the finite element method enables analyses and optimisations not possible in other finite element implementations. We*

present an optimisation framework for the evaluation of local assembly matrices and compare against the state of the art.

We describe Excafé's optimisation strategy and implementation for local assembly matrices in Chapter 9. Our expression capture has allowed us to analyse local assembly matrices in terms of rational scalar functions of cell vertices and other independent variables as well as observing the effect our basis functions have on this representation. We also compare the optimisations we expect should be possible with this framework to those explored by the FEniCS project [38]. We have not yet identified any optimisations unique to this framework however, research is ongoing.

- *We extend the work of Hosangadi et al. [1] for common sub-expression elimination of polynomials. We present improvements to the original factorisation weighting function that further reduce the operation count of our factorised expressions. We reformulate the primary problem of finding a maximally weighted matrix covering as a graph biclique search problem and present a branch-and-bound algorithm optimised for the presented weight function through insights into the graph structure.*

  We present our work on extending polynomial factorisation and common sub-expression elimination of polynomials as the mostly self-contained Chapter 10. Hosangadi et al. did not present an algorithm for finding the maximal KCM matrix covering so we constructed one that remains optimal but also scales to the problem sizes we wish to deal with. We give some idea of those sizes and our algorithm's performance in Section 10.4.

  We also show that the covering scoring function presented by Hosangadi et al. can lead to infinite recursion. We present a new scoring function that more accurately reflects the number of floating point operations saved by a factorisation and show that it is only possible to define this function if additional information about whether the expression literals are actually variables or constant values is known.

11.2 Discussion

One insight from extending DESOLA to support sparse code generation was the importance of the representation at which optimisations are performed. From a more abstract perspective, it is obvious that two or more matrix-vector multiplies using the same matrix should be able to be carried out simultaneously with reuse of matrix elements regardless of its storage format.

However, when performing these optimisations at lower level, that of the code generator, finding these optimisations becomes harder. The dense matrix-vector multiply is trivial for our SUIF pass to optimise, but it was unable to completely fuse a sparse matrix-vector multiply. Given some other matrix representation, it might have been impossible. This provides evidence that a *generative* approach is required for effectively achieving optimisations using active libraries.

A major distinction between the design of DESOLA and EXCAFÉ is that EXCAFÉ explicitly makes it clear that it is performing expression capture. In trying to remain transparent, DESOLA was forced to work with incomplete information about the problem being solved, and could only use low-overhead analyses. By explicitly capturing all the information needed to specify a finite element problem, EXCAFÉ inherits none of these restrictions.

Another notable distinction is EXCAFÉ's declarative loop syntax. Many C++ embedded expression capture schemes will tend to resemble code that could have been written directly in C++ to do the same thing, albeit less efficiently. In contrast, EXCAFÉ's declarative loop syntax provides an abstraction that otherwise would completely exist outside the scope of a C++ program. Hence, EXCAFÉ has used expression capture not just to obtain a description of a computation, but as a tool to provide a level of abstraction that would otherwise be impossible.

It is debatable whether we have chosen an appropriate level of representation for our investigation into the optimisation of local assembly matrix computation. After all, by representing each element as an independent scalar rational function, we have lost all loop structure related to the various tensor products that are used during the computation of the bilinear forms. However, work by Ølgaard and Wells [54] seems to show that the structures of tensor-based and quadrature-based computations inhibit optimisations in certain instances.

Kirby et al. [38] demonstrate an algorithm that is optimal in some sense with respect to the optimised execution strategy it can produce for evaluating a local assembly matrix. Yet, work by Ølgaard and Wells shows that tensor based evaluation may require hundreds of times more operations than a quadrature based assembly in some cases.

In choosing such a flexible representation, we have set ourselves the opposite problem. We should be able to represent a superset of other optimisations in our representation, but as it is so unstructured, the challenge is the construction of an algorithm that can effectively restructure the representation to an efficient evaluation plan.

We consider how this investigation should progress in the next section.

11.3 Future Work

The declarative loop syntax provided by Excafé is sufficient for our Navier-Stokes problem, but should be tested for other examples to gauge applicability. We also intend to extend Excafé so that the declarative iteration syntax can also be used for time-stepping. Additionally, we see no reason why the same syntax cannot be used for the specification of iterative linear solvers.

Although we have shown algorithms for inferring loop nesting, and detecting invalid iteration specifications, we do not have a formal specification of how the syntax should behave nor proofs of correctness. These would be of great benefit towards providing confidence that the inferred loop structure is actually the one desired.

Our other future work revolves around optimisation of the evaluation of the local assembly matrix. We have theorised that by reducing the size of of our local assembly expressions, through avoiding the full expansion of cell-geometry related terms, we can reduce our optimisation search space. If our factorisation algorithms perform effectively on this representation, we have a basis with which to investigate further optimisations.

Should our factorisation algorithms not perform effectively, it is possible that further improvements can be made to increase the quality of the result, or that existing optimisation algorithms

for evaluating single multi-variate polynomials can be extended.

Lastly, we would like to consider a more structured approach to optimisation, that still provides more flexibility than basing evaluations around a particular rearrangement of variational forms. It may be possible to apply techniques and models from the *Tensor Contraction Engine* [59], which can synthesise high-performance code for evaluating tensor contractions, to the tensor operations used to evaluate local assembly terms.

Bibliography

[1] A. Hosangadi, F. Fallah, and R. Kastner, "Optimizing polynomial expressions by algebraic factorization and common subexpression elimination," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2012–2022, 2006.

[2] F. P. Russell, M. R. Mellor, P. H. J. Kelly, and O. Beckmann, "An active linear algebra library using delayed evaluation and runtime code generation," in *Proceedings of the Second International Workshop on Library-Centric Software Design (LCSD'06)*, pp. 5–13, Oct. 2006.

[3] F. P. Russell, M. R. Mellor, P. H. J. Kelly, and O. Beckmann, "DESOLA: An active linear algebra library using delayed evaluation and runtime code generation," *Science of Computer Programming*, vol. 76, no. 4, pp. 227–242, 2011. Special issue on library-centric software design (LCSD 2006).

[4] T. L. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, pp. 26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[5] J. G. Siek and A. Lumsdaine, "The matrix template library: A generic programming approach to high performance numerical linear algebra," in *ISCOPE* (D. Caromel, R. R. Oldehoeft, and M. Tholburn, eds.), vol. 1505 of *Lecture Notes in Computer Science*, pp. 59–70, Springer, 1998.

[6] T. Veldhuizen, "Using C++ template metaprograms," *C++ Report*, vol. 7, pp. 36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[7] J. G. Siek and A. Lumsdaine, "A rational approach to portable high performance: The basic linear algebra instruction set (BLAIS) and the fixed algorithm size template (FAST) library.," in *ECOOP Workshops* (S. Demeyer and J. Bosch, eds.), vol. 1543 of *Lecture Notes in Computer Science*, pp. 468–469, Springer, 1998.

[8] D. J. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen, "Classification and utilization of abstractions for optimization.," in *ISoLA* (T. Margaria and B. Steffen, eds.), vol. 4313 of *Lecture Notes in Computer Science*, pp. 57–73, Springer, 2004.

[9] S. Z. Guyer and C. Lin, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proceedings of the IEEE*, vol. 93, no. 2, 2005. special issue on "Program Generation, Optimization, and Adaptation".

[10] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[11] P. H. J. Kelly, O. Beckmann, T. Field, and S. B. Baden, "THEMIS: Component dependence metadata in adaptive parallel applications," *Parallel Processing Letters*, vol. 11, no. 4, pp. 455–470, 2001.

[12] B. Alpern, L. Carter, and J. Ferrante, "Space-limited procedures: A methodology for portable High-Performance," in *Working Conference on Massively Parallel Programming Models, MPPM-95, Suitability, Realization, Performance.*, (Berlin), 1995.

[13] O. Beckmann, A. Houghton, M. Mellor, and P. H. J. Kelly, "Runtime code generation in C++ as a foundation for domain-specific optimisation.," in *Domain-Specific Program Generation* (C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, eds.), vol. 3016 of *Lecture Notes in Computer Science*, pp. 291–306, Springer, 2003.

[14] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232–275, 2005.

[15] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzaran, D. Padua, and K. Pingali, "A language for the compact representation of multiple program versions."

[16] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Comput.*, vol. 27, no. 1–2, pp. 3–25, 2001.

[17] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, 1994.

[18] A. Cohen, S. Girbal, and O. Temam, "A polyhedral approach to ease the composition of program transformations," in *EuroPar'04, Pisa, Italy, Aug. 2004*, LNCS, Springer-Verlag, 2004.

[19] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache, "Facilitating the search for compositions of program transformations," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, (New York, NY, USA), pp. 151–160, ACM Press, 2005.

[20] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, (New York, NY, USA), pp. 229–241, ACM Press, 1999.

[21] D. G. Kirkpatrick and P. Hell, "On the completeness of a generalized matching problem," in *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 240–245, ACM Press, 1978.

[22] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), p. 192, IEEE Computer Society, 1999.

[23] M. M. Strout, L. Carter, and J. Ferrante, "Rescheduling for locality in sparse matrix computations," in *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, (London, UK), pp. 137–148, Springer-Verlag, 2001.

[24] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*, (Washington, DC, USA), p. 75, IEEE Computer Society, 2004.

[25] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[26] S. J. Sherwin and G. E. Karniadakis, *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2005.

[27] R. Rannacher, "Finite element methods for the incompressible Naver-Stokes equations," Institut für Angewandte Mathematik, Universität Heidelberg, 1999.

[28] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadalphia: Society for Industrial and Applied Mathematics. Also available as postscript file on http://www.netlib.org/templates/Templates.html, 1994.

[29] B. Bagheri and R. Scott, "About analysa," Tech. Rep. TR-2004-09, The University of Chicago Department of Computer Science, 2004.

[30] P. Dular, C. Geuzaine, F. Henrotte, and W. Legros, "A general environment for the treatment of discrete problems and its application to the finite element method," *IEEE Transactions on Magnetics*, vol. 34, pp. 3395–3398, Sept. 1998.

[31] W. Bangerth, R. Hartmann, and G. Kanschat, "deal.II – a general purpose object oriented finite element library," *ACM Trans. Math. Softw.*, vol. 33, no. 4, p. 24, 2007.

[32] O. Pironneau, F. Hecht, A. L. Hyaric, and J. Morice, "FreeFEM++," 2010. URL: http://www.freefem.org/ff++/.

[33] K. R. Long, R. C. Kirby, and B. van Bloemen Waanders, "Unified embedded parallel finite element computations via software-based frechet differentiation," *SIAM J. Scientific Computing (to appear)*.

[34] A. Logg, "Automating the finite element method," *Arch. Comput. Methods Eng.*, vol. 14, no. 2, pp. 93–138, 2007.

[35] FEniCS, "FEniCS project." URL: http://www.fenics.org/, 2009.

[36] R. C. Kirby and A. Logg, "A compiler for variational forms," *ACM Trans. Math. Softw.*, vol. 32, no. 3, pp. 417–444, 2006.

[37] M. S. Alnæs and A. Logg, "UFL," 2009. URL: http://www.fenics.org/ufl/.

[38] R. C. Kirby, A. Logg, L. R. Scott, and A. R. Terrel, "Topological optimization of the evaluation of finite element matrices," *SIAM J. Sci. Comput.*, vol. 28, no. 1, pp. 224–240, 2006.

[39] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen, "Generative programming and active libraries," in *Selected Papers from the International Seminar on Generic Programming*, no. 1766 in LNCS, pp. 25–39, Springer-Verlag, 2000.

[40] T. L. Veldhuizen and D. Gannon, "Active libraries: Rethinking the roles of compilers and libraries," in *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98)*, SIAM Press, 1998.

[41] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.

[42] O. Beckmann and P. H. J. Kelly, "Efficient interprocedural data placement optimisation in a parallel library," in *LCR98: Languages, Compilers and Run-time Systems for Scalable Computers*, no. 1511 in LNCS, pp. 123–138, Springer-Verlag, May 1998.

[43] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Mathematical Software*, vol. 29, pp. 110–140, June 2003.

[44] L. Grigori, J. W. Demmel, and X. S. Li, "Parallel symbolic factorization for sparse LU with static pivoting," *SIAM J. Scientific Computing*, vol. 29, no. 3, pp. 1289–1314, 2007.

[45] C. Bauer, A. Frink, and R. Kreckel, "Introduction to the GiNaC framework for symbolic computation within the C++ programming language," *Journal of Symbolic Computation*, vol. 33, no. 1, pp. 1–12, 2002.

[46] J. R. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," in *Applied Computational Geometry: Towards Geometric Engineering* (M. C. Lin and D. Manocha, eds.), vol. 1148 of *Lecture Notes in Computer Science*, pp. 203–222, Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[47] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc users manual," Tech. Rep. ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2009.

[48] P. A. Raviart and J. M. Thomas, "A mixed finite element method for 2-nd order elliptic problems," in *Mathematical Aspects of Finite Element Methods* (A. Dold and B. Eckmann, eds.), Springer, 1975. Lecture Notes of Mathematics, Volume 606.

[49] S. Turek, *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*. Springer, Berlin, 1999.

[50] W. C. Rheinboldt, C. K. Mesztenyi, and J. M. Fitzgerald, "On the evaluation of multivariate polynomials and their derivatives," *BIT Numerical Mathematics*, vol. 17, pp. 437–450, 1977. 10.1007/BF01933453.

[51] S. K. Lodha and R. Goldman, "A unified approach to evaluation algorithms for multivariate polynomials," *Math. Comput.*, vol. 66, no. 220, pp. 1521–1553, 1997.

[52] M. Kojima, "Efficient evaluation of polynomials and their partial derivatives in homotopy continuation methods," *Journal of the Operations Research Society of Japan*, vol. 51, no. 1, pp. 29–54, 2008.

[53] H.-C. P. Liao and R. J. Fateman, "Evaluation of the heuristic polynomial GCD," in *Proc. of Int'l Symp. on Symbolic and Algebraic Computation (ACM Press) (ISSAC-95) Montreal CA*, pp. 240–247, ACM Press, 1995.

[54] K. B. Ølgaard and G. N. Wells, "Optimisations for quadrature representations of finite element tensors through automated code generation," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, 2010.

[55] M. Yannakakis, "Node-deletion problems on bipartite graphs," *SIAM J. Comput.*, vol. 10, no. 2, pp. 310–327, 1981.

[56] R. Peeters, "The maximum edge biclique problem is NP-complete," *Discrete Applied Mathematics*, vol. 131, no. 3, pp. 651–654, 2003.

[57] G. Liu, K. Sim, and J. Li, "Efficient mining of large maximal bicliques," in *Data Warehousing and Knowledge Discovery* (A. Tjoa and J. Trujillo, eds.), vol. 4081 of *Lecture Notes in Computer Science*, pp. 437–448, Springer Berlin / Heidelberg, 2006.

[58] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for heuristic determination of minimum path cost," *IEEE Trans. on SSC*, vol. 4, p. 100, 1968.

[59] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. chung Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, A. Sibiryakov, D. E. Bernholdt, A. Bibireata, D. Cociorva, X. Gao, S. Krishnamoorthy, and S. Krishnan, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," in *Proceedings of the IEEE*, p. 2005, 2005.

[60] W. Bangerth and O. Kayser-Herold, "Data structures and requirements for hp finite element software," *ACM Trans. Math. Softw.*, vol. 36, no. 1, pp. 1–31, 2009.

[61] A. Logg, "Efficient representation of computational meshes," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 4, pp. 283–295, 2009.

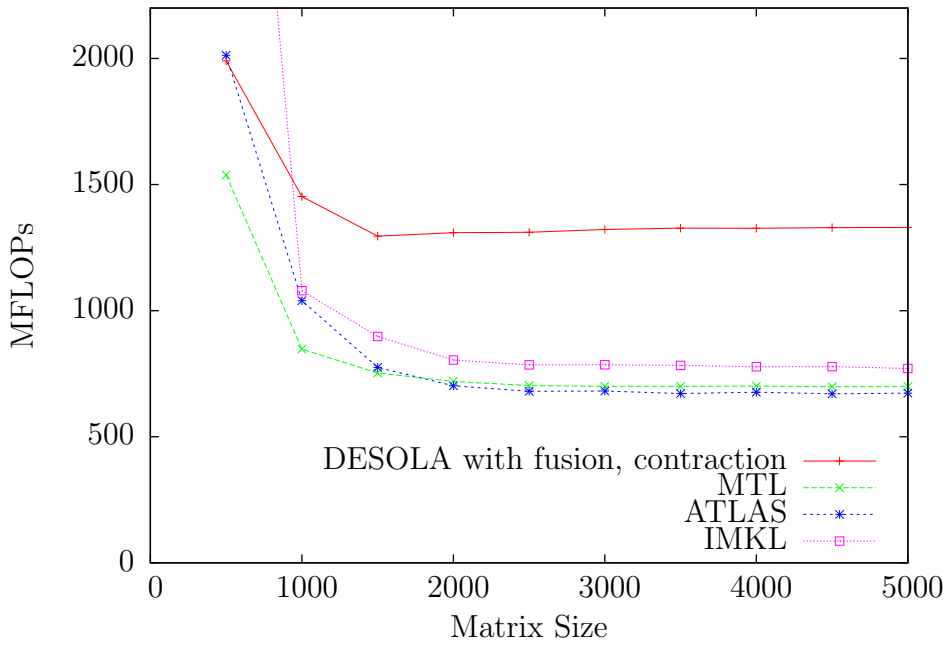
Appendix A

DESOLA Dense Performance Results

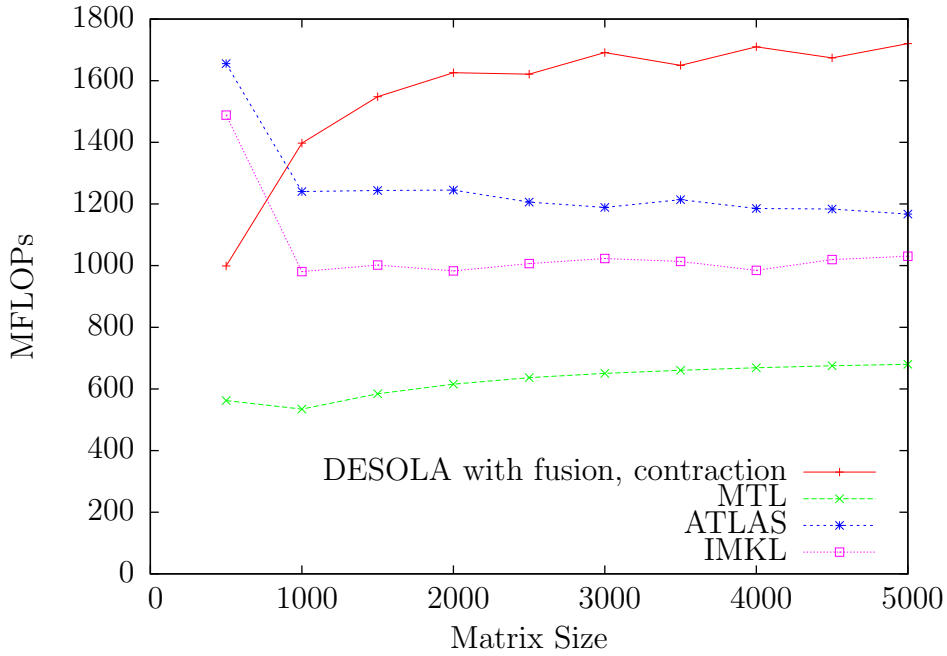
The benchmark architectures were:

1. Intel Xeon "Clovertown" processor running at 2.66GHz, 4096 KB L2 cache with 4 GB RAM running 64-bit Ubuntu 8.10.

2. Pentium IV "Prescott" processor running at 3.2GHz with Hyper-threading disabled, 2048 KB L2 cache with 2 GB RAM running 32-bit Ubuntu 8.10.

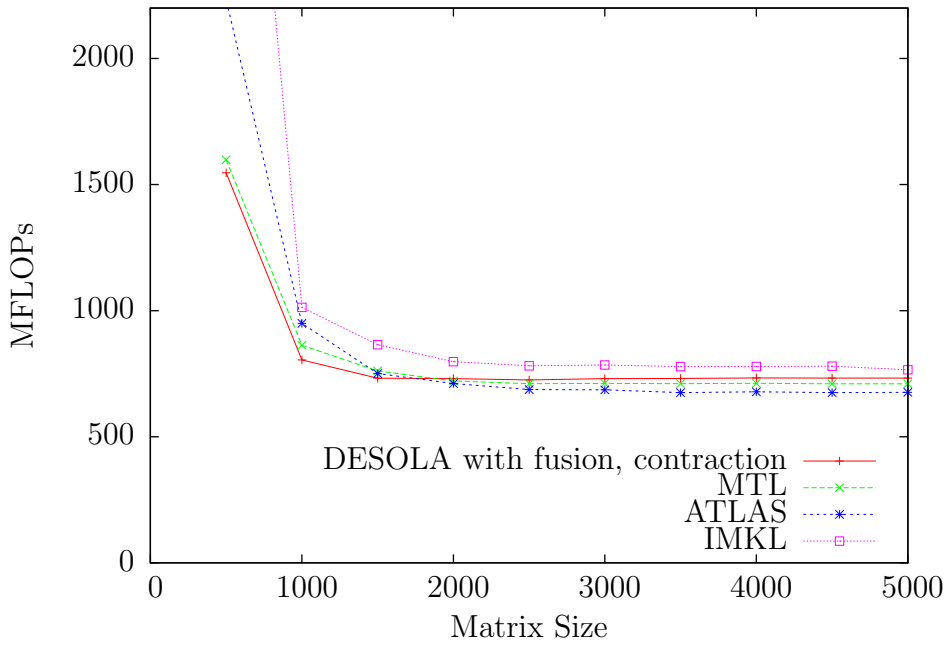


(a) architecture 1

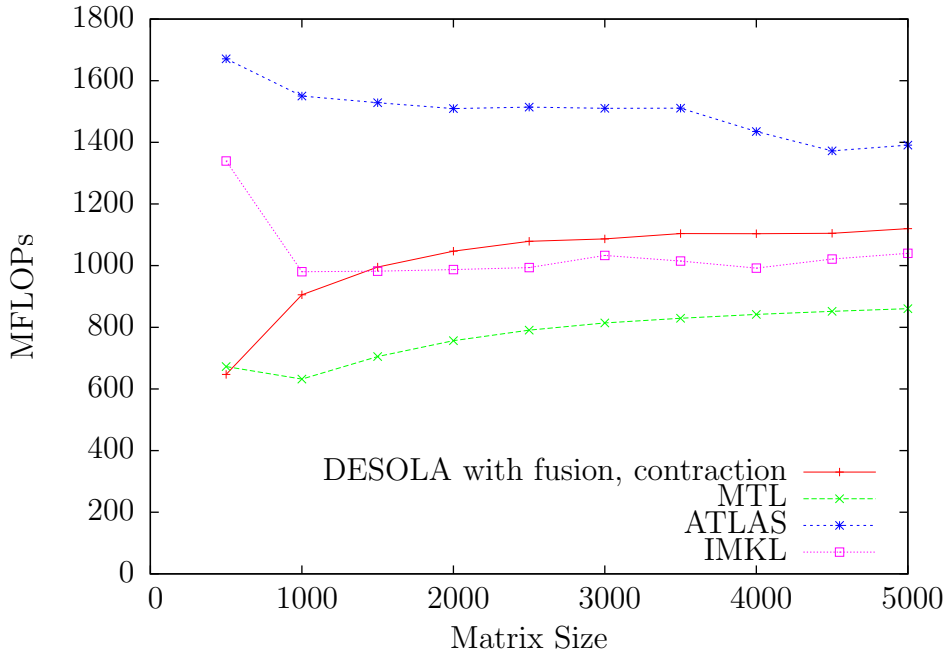


(b) architecture 2

Figure A.1: Throughput of different implementations of the BiConjugate Gradient solver on our test architectures.

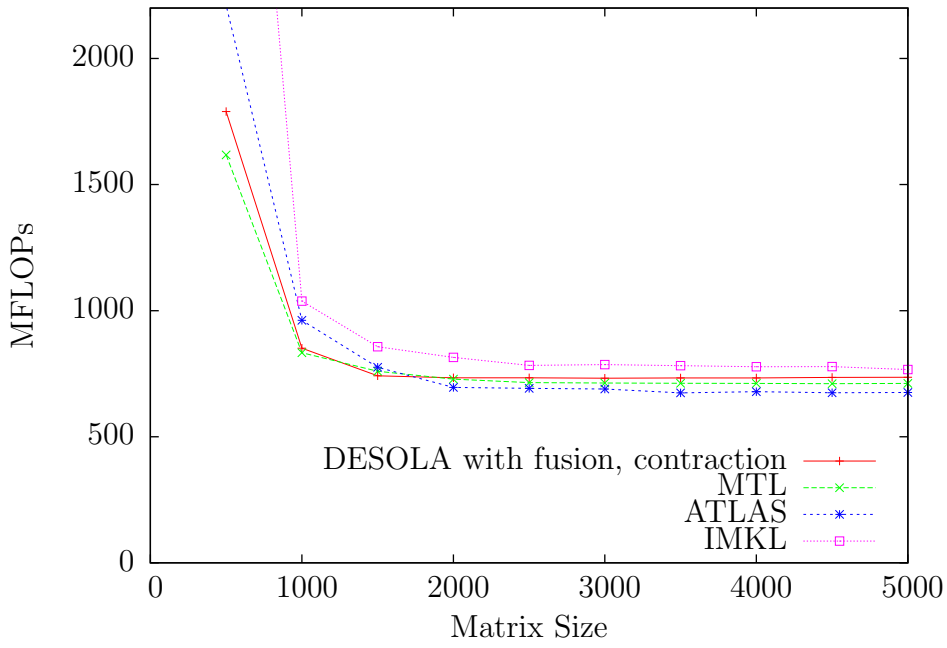


(a) architecture 1

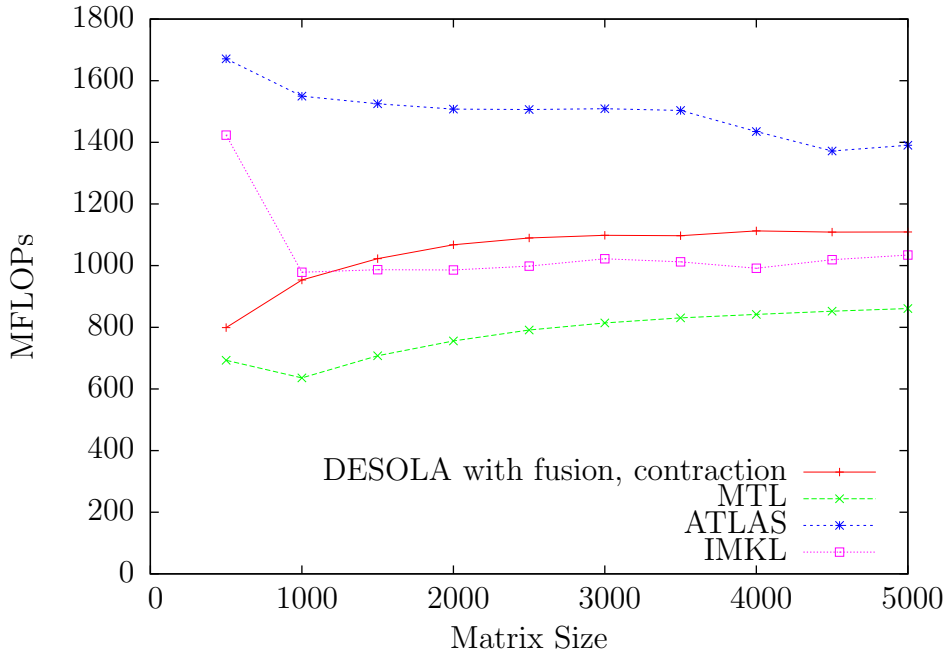


(b) architecture 2

Figure A.2: Throughput of different implementations of the BiConjugate Gradient Stabilised solver on our test architectures.

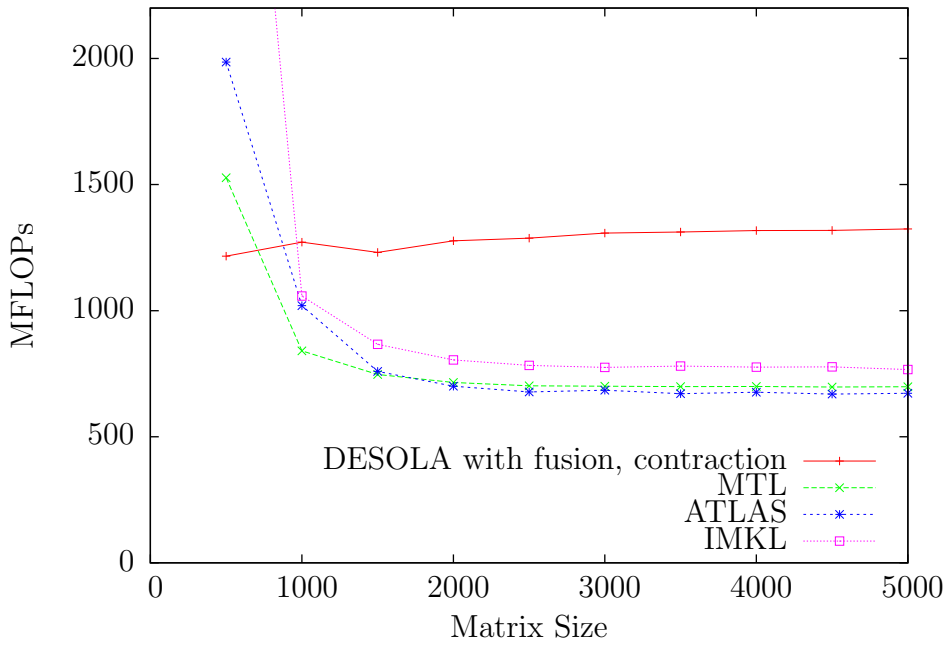


(a) architecture 1

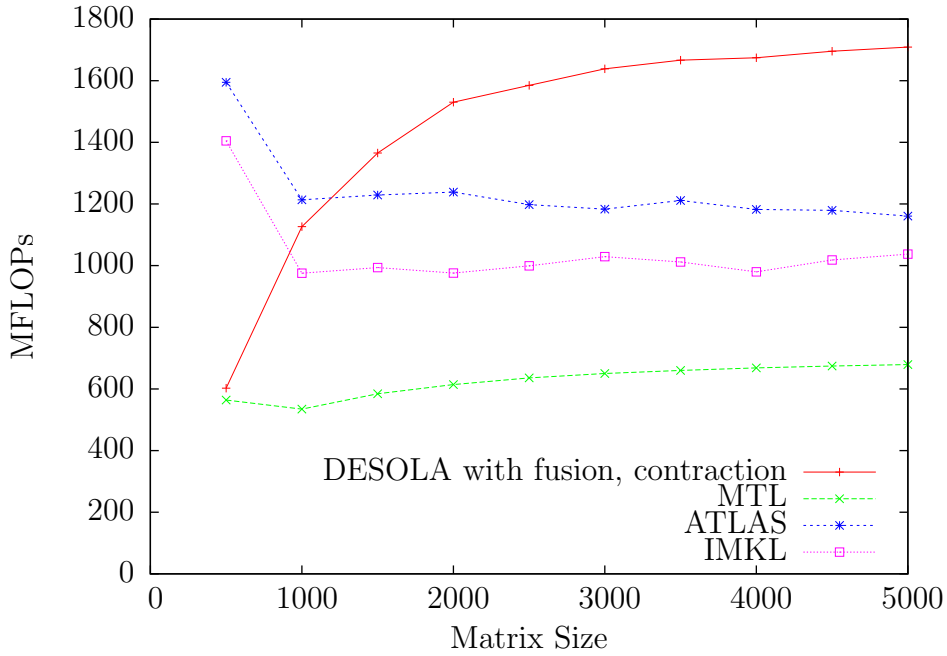


(b) architecture 2

Figure A.3: Throughput of different implementations of the Conjugate Gradient Squared solver on our test architectures.

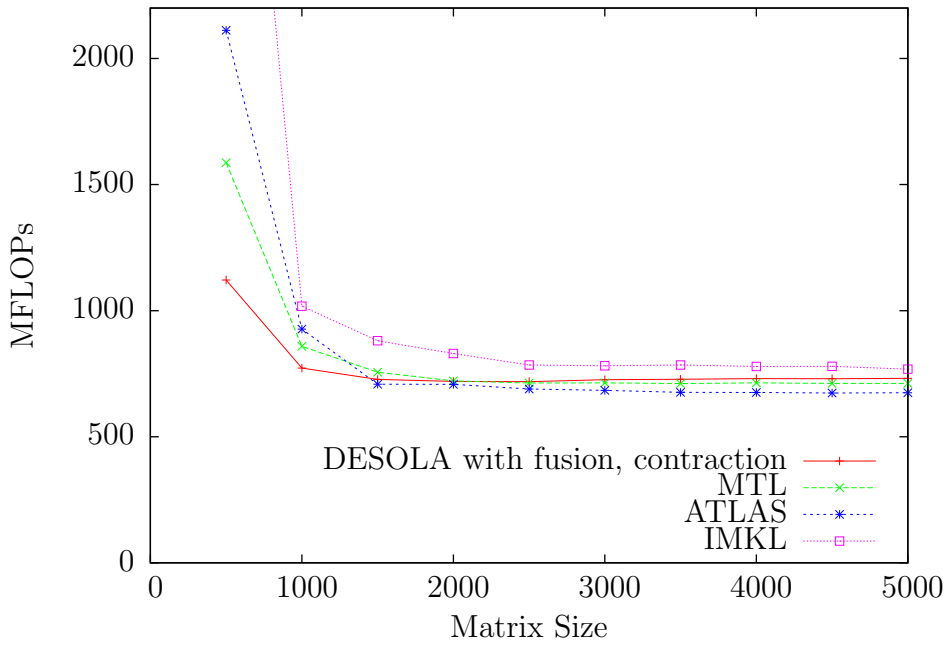


(a) architecture 1

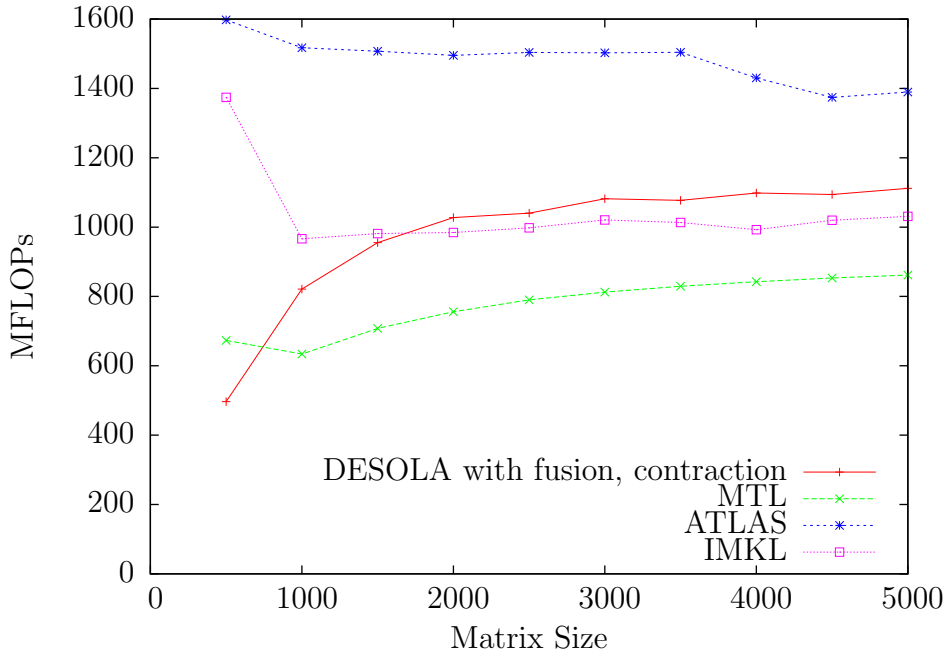


(b) architecture 2

Figure A.4: Throughput of different implementations of the Quasi-Minimal Residual solver on our test architectures.



(a) architecture 1



(b) architecture 2

Figure A.5: Throughput of different implementations of the Transpose-Free Quasi-Minimal Residual solver on our test architectures.

Appendix B

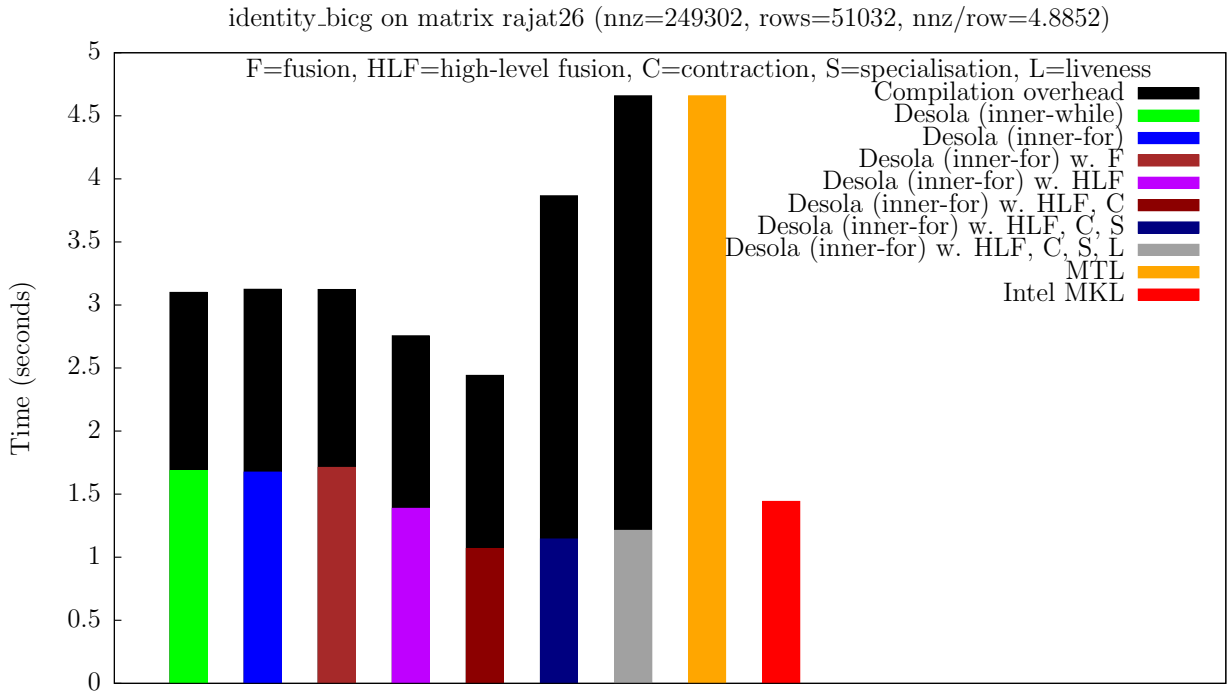
DESOLA Sparse Performance Results

The benchmark architectures were:

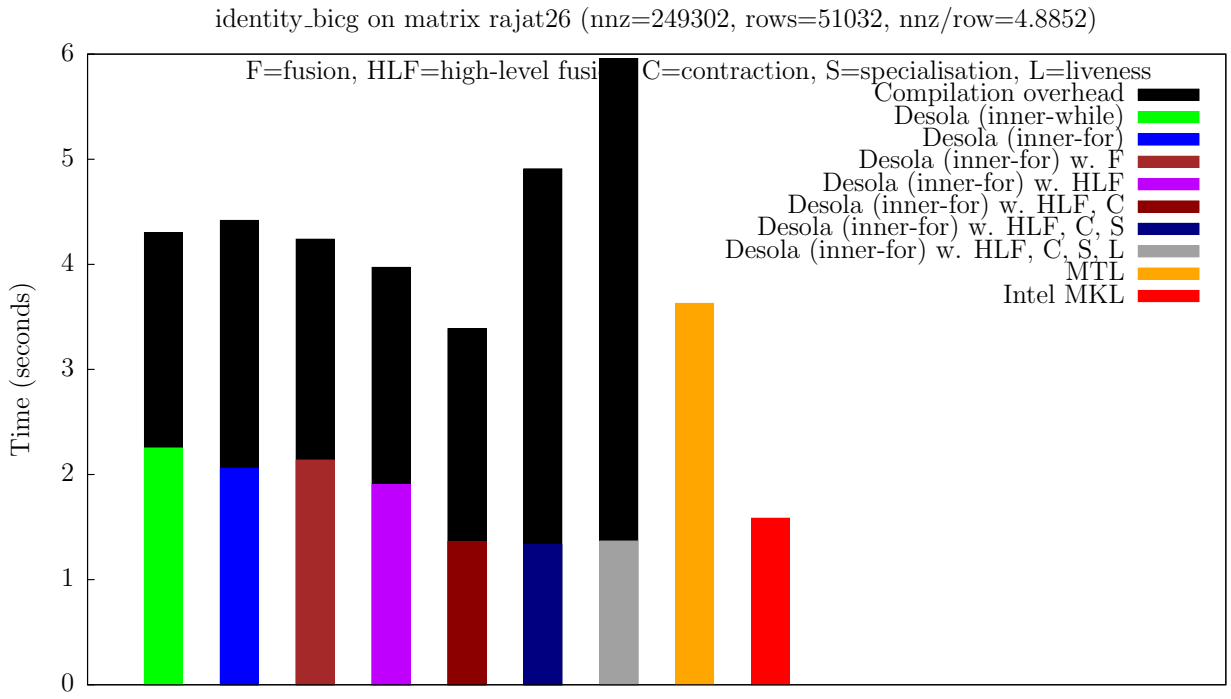
1. Intel Xeon "Clovertown" processor running at 2.66GHz, 4096 KB L2 cache with 4 GB RAM running 64-bit Ubuntu 8.10.

2. Pentium IV "Prescott" processor running at 3.2GHz with Hyper-threading disabled, 2048 KB L2 cache with 2 GB RAM running 32-bit Ubuntu 8.10.

B.1 BiConjugate Gradient Solver

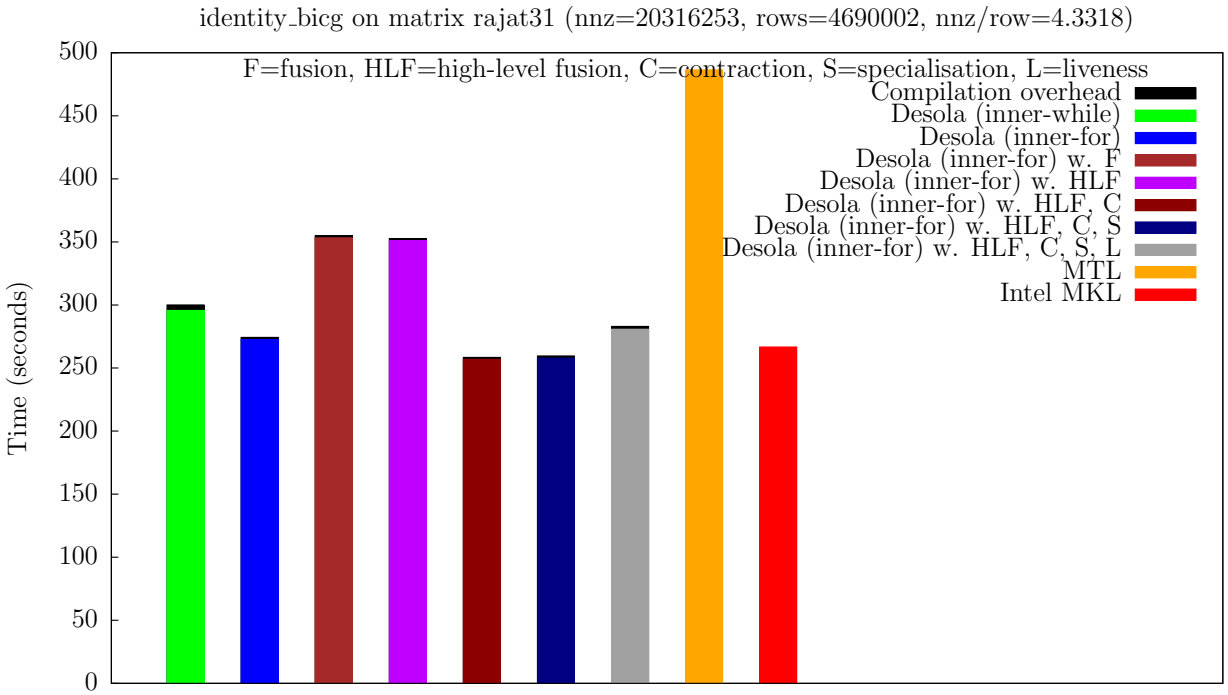


(a) architecture 1

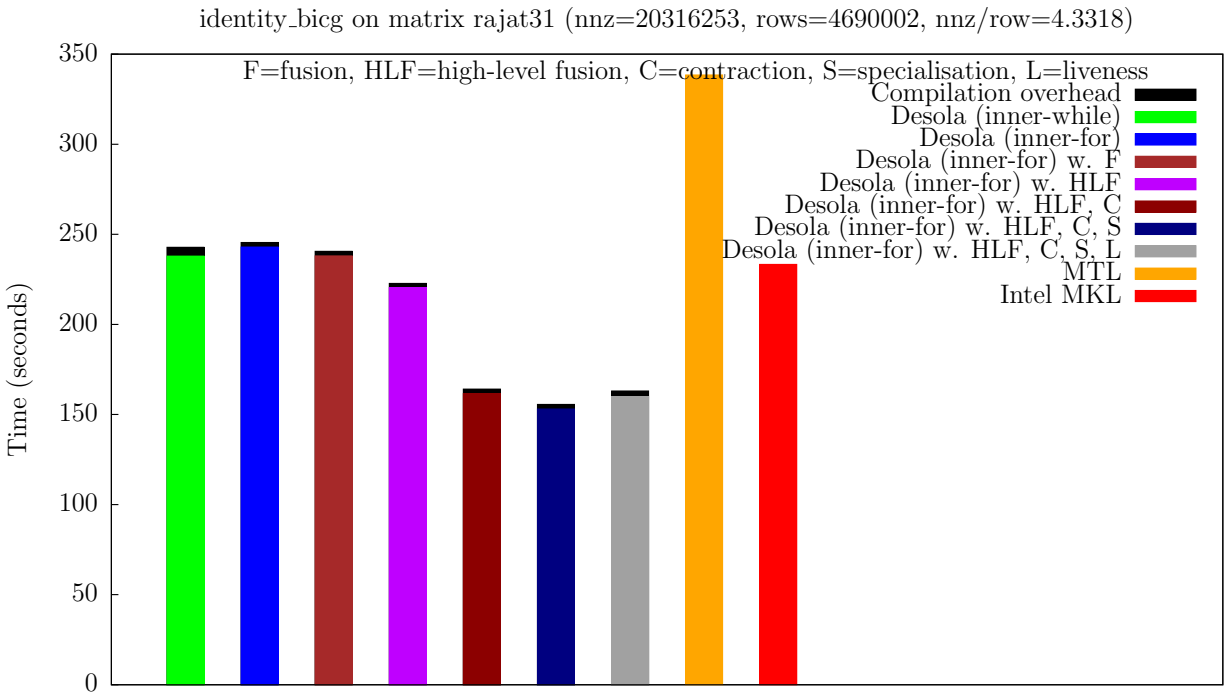


(b) architecture 2

Figure B.1: Time to execute 256 iterations of the BiConjugate Gradient solver with matrix rajat26 on our test architectures.

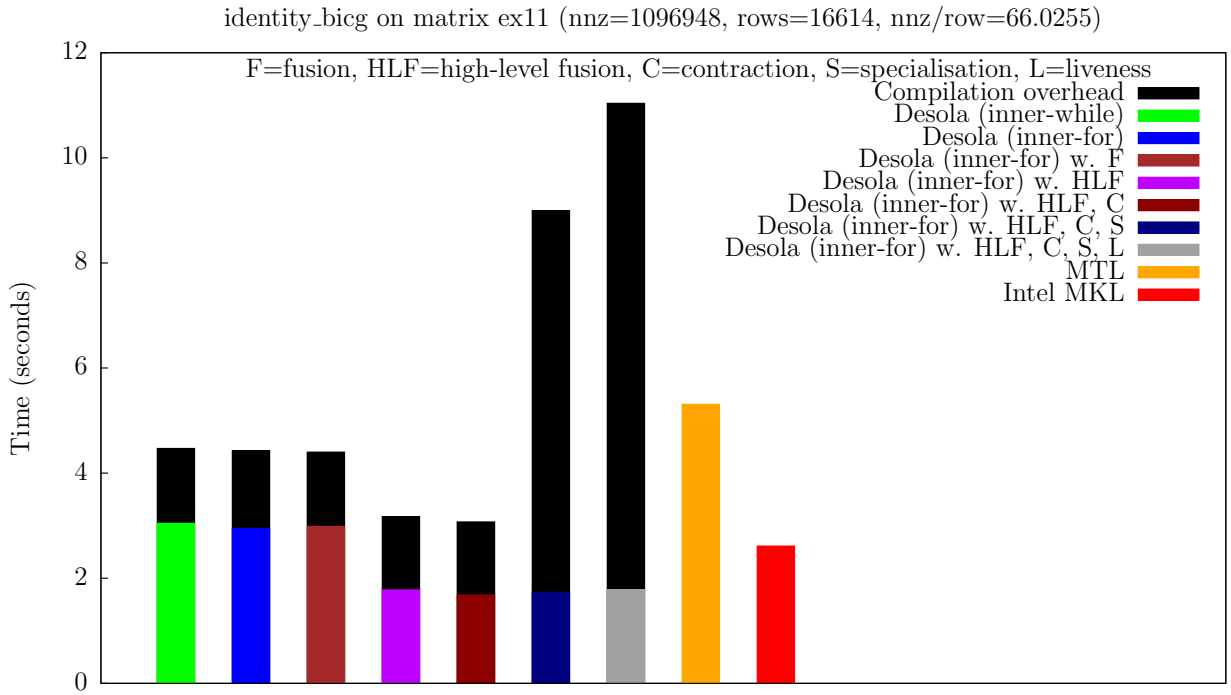


(a) architecture 1

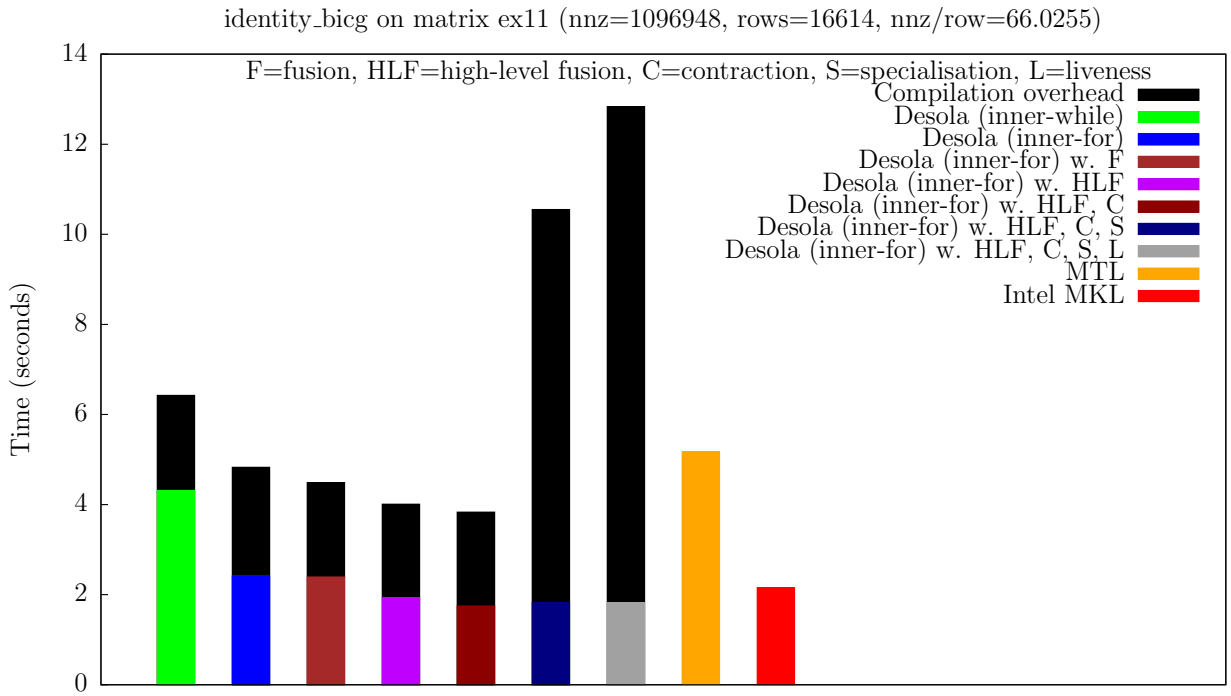


(b) architecture 2

Figure B.2: Time to execute 256 iterations of the BiConjugate Gradient solver with matrix rajat31 on our test architectures.

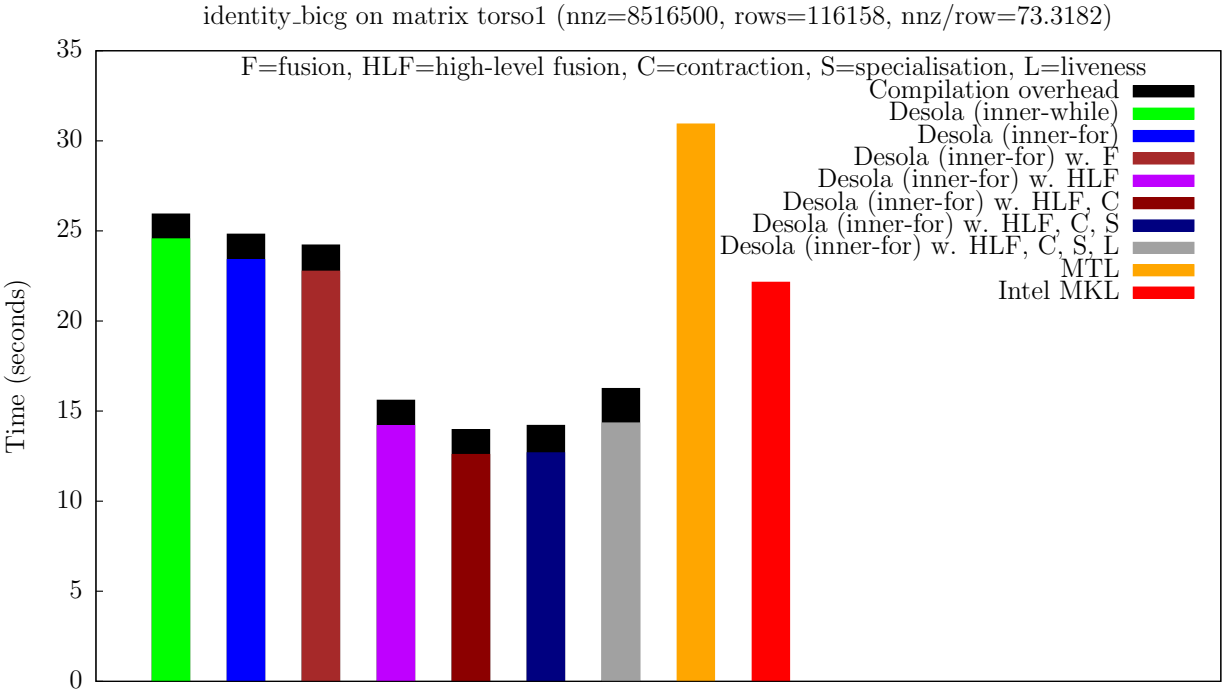


(a) architecture 1

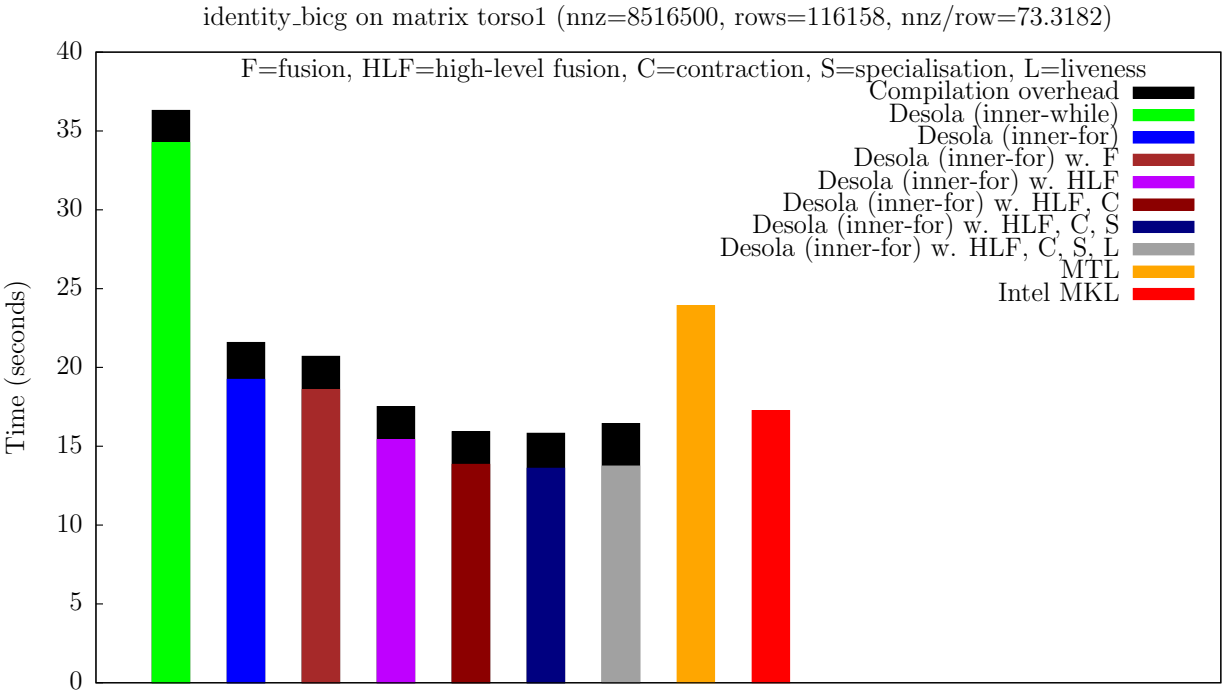


(b) architecture 2

Figure B.3: Time to execute 256 iterations of the BiConjugate Gradient solver with matrix ex11 on our test architectures.

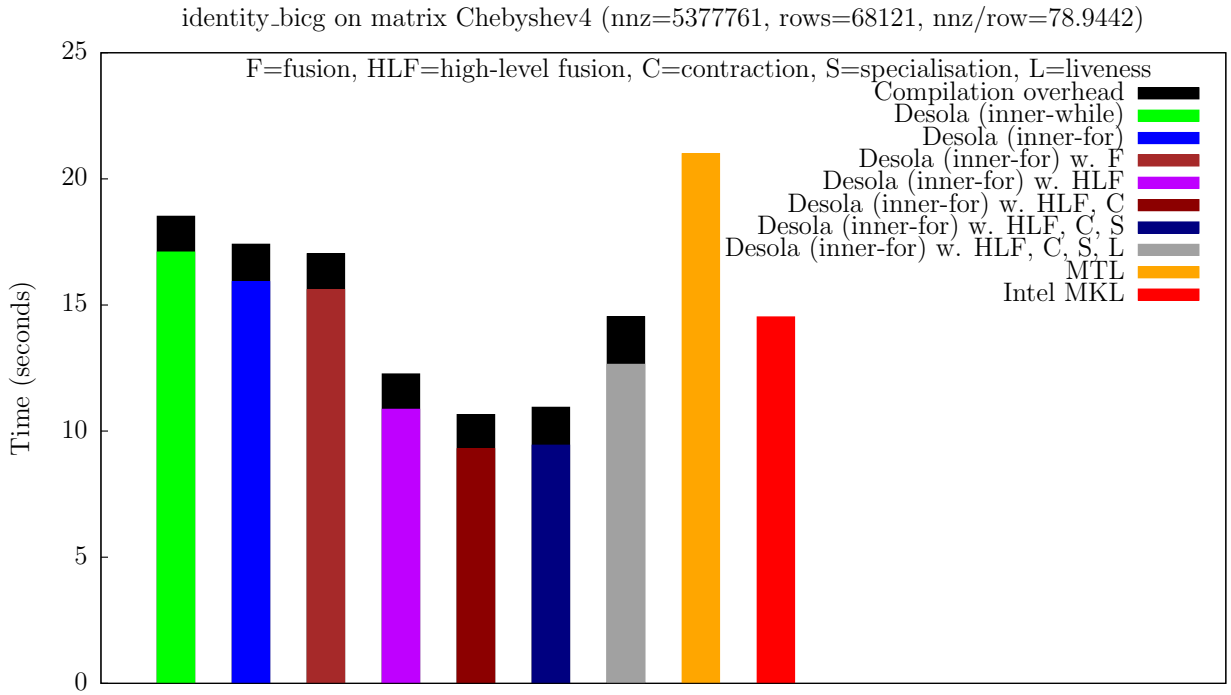


(a) architecture 1

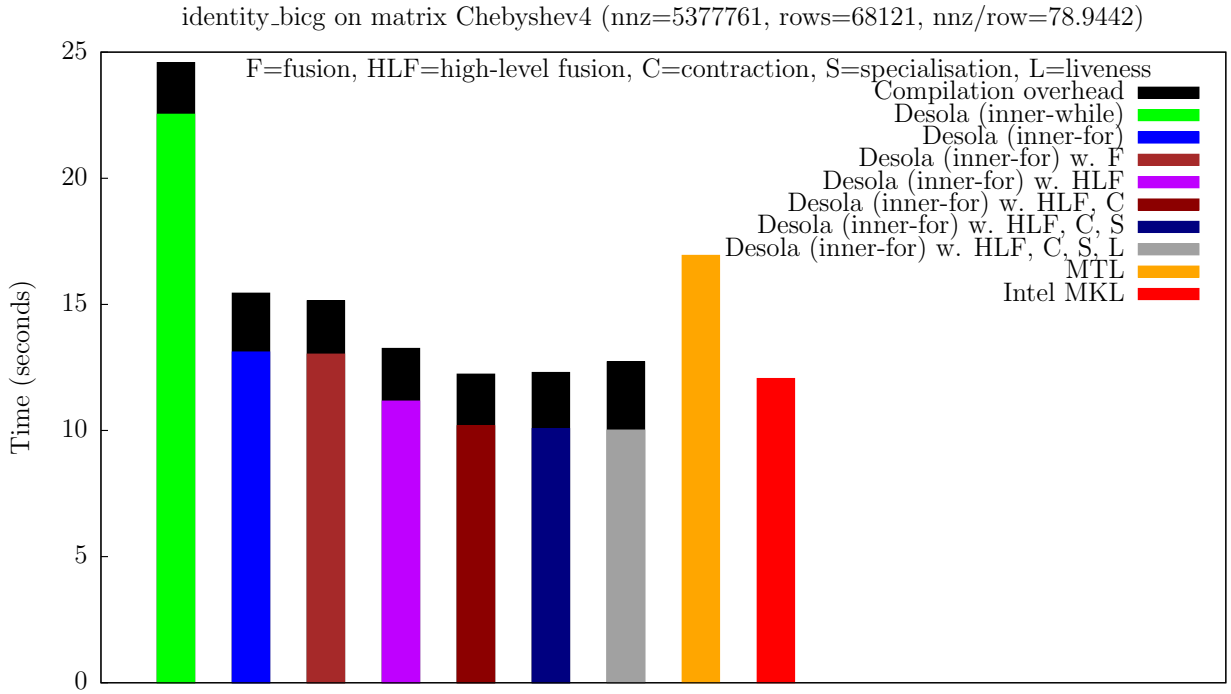


(b) architecture 2

Figure B.4: Time to execute 256 iterations of the BiConjugate Gradient solver with matrix torso1 on our test architectures.

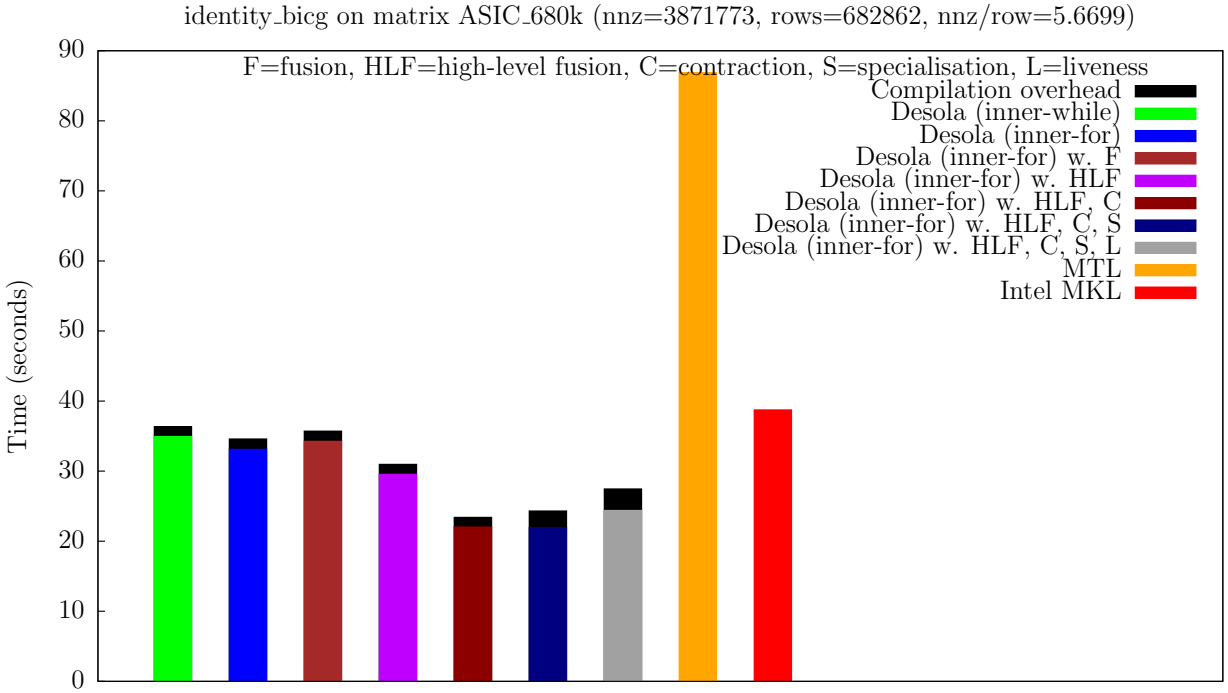


(a) architecture 1

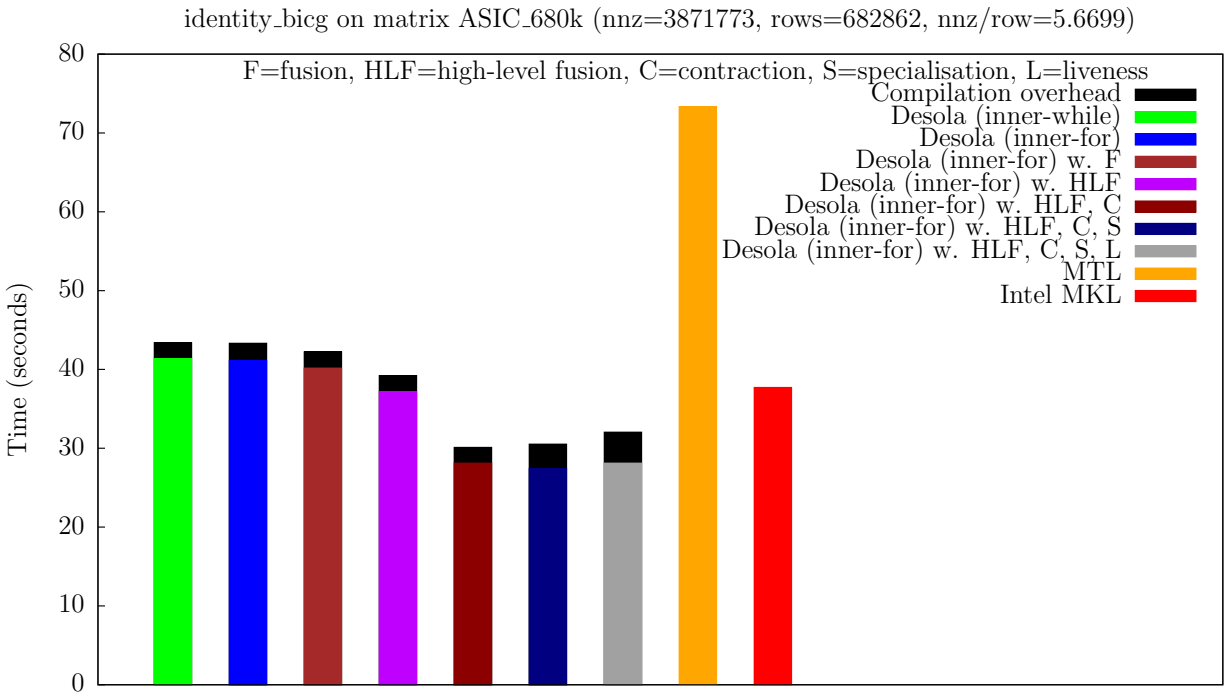


(b) architecture 2

Figure B.5: Time to execute 256 iterations of the BiConjugate Gradient solver with matrix Chebyshev4 on our test architectures.



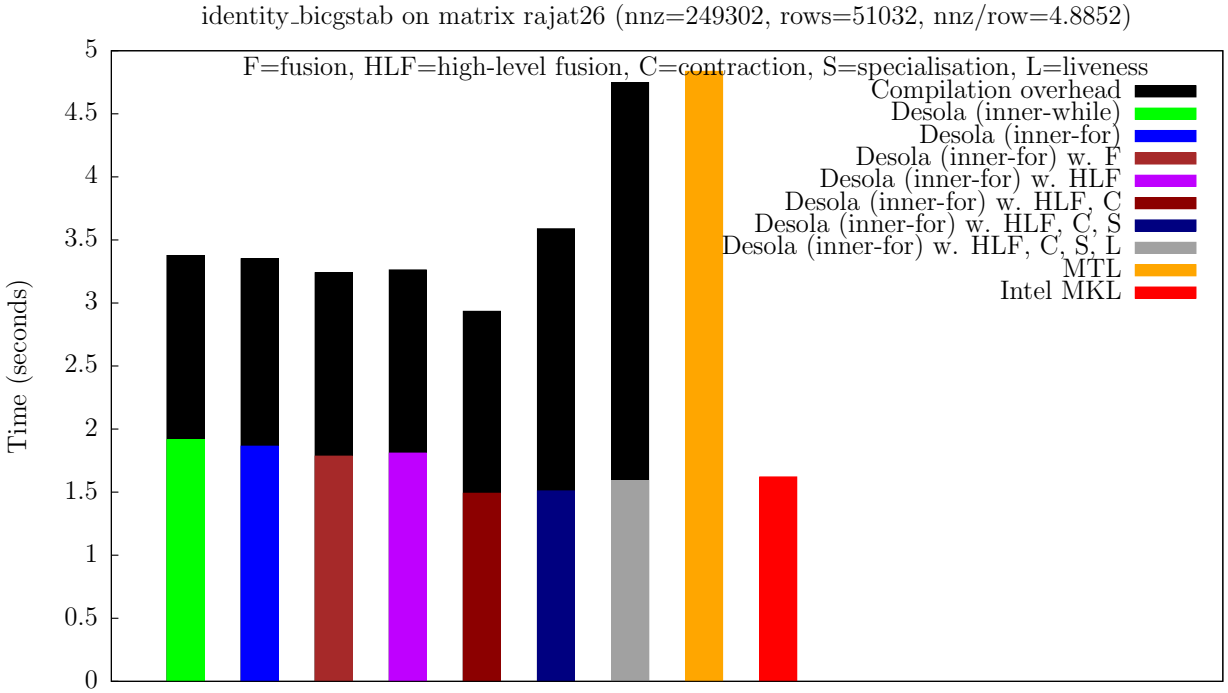
(a) architecture 1



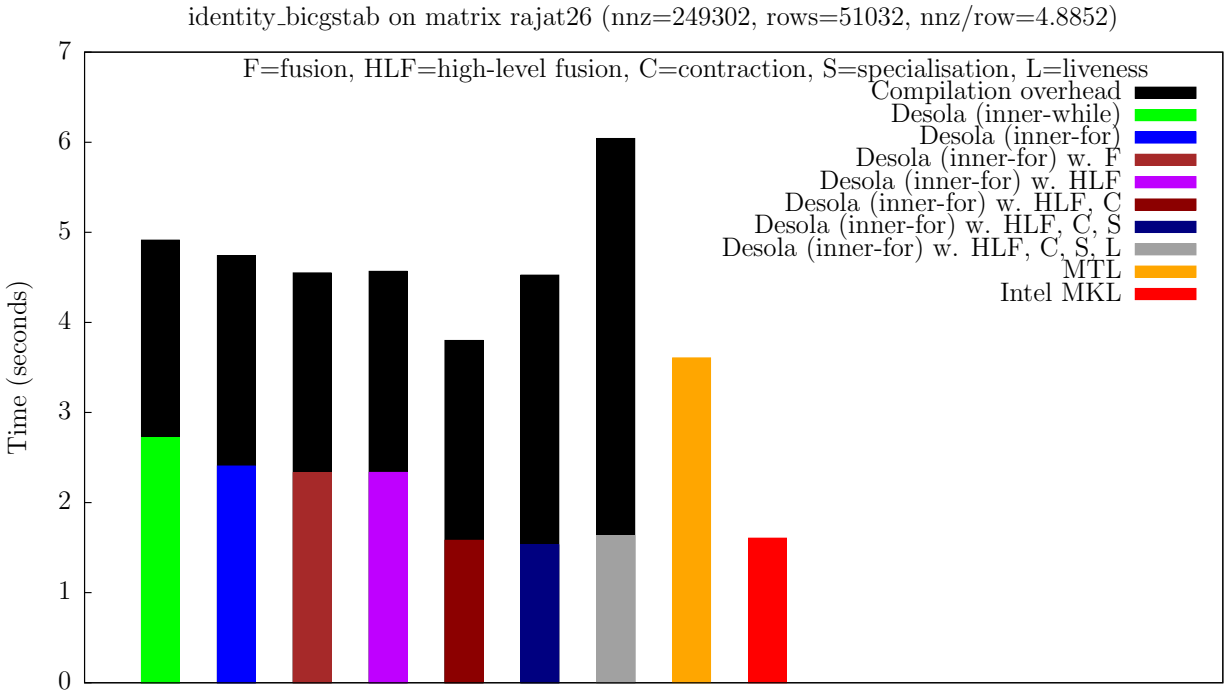
(b) architecture 2

Figure B.6: Time to execute 256 iterations of the BiConjugate Gradient solver with matrix ASIC_680k on our test architectures.

B.2 BiConjugate Gradient Stabilised Solver

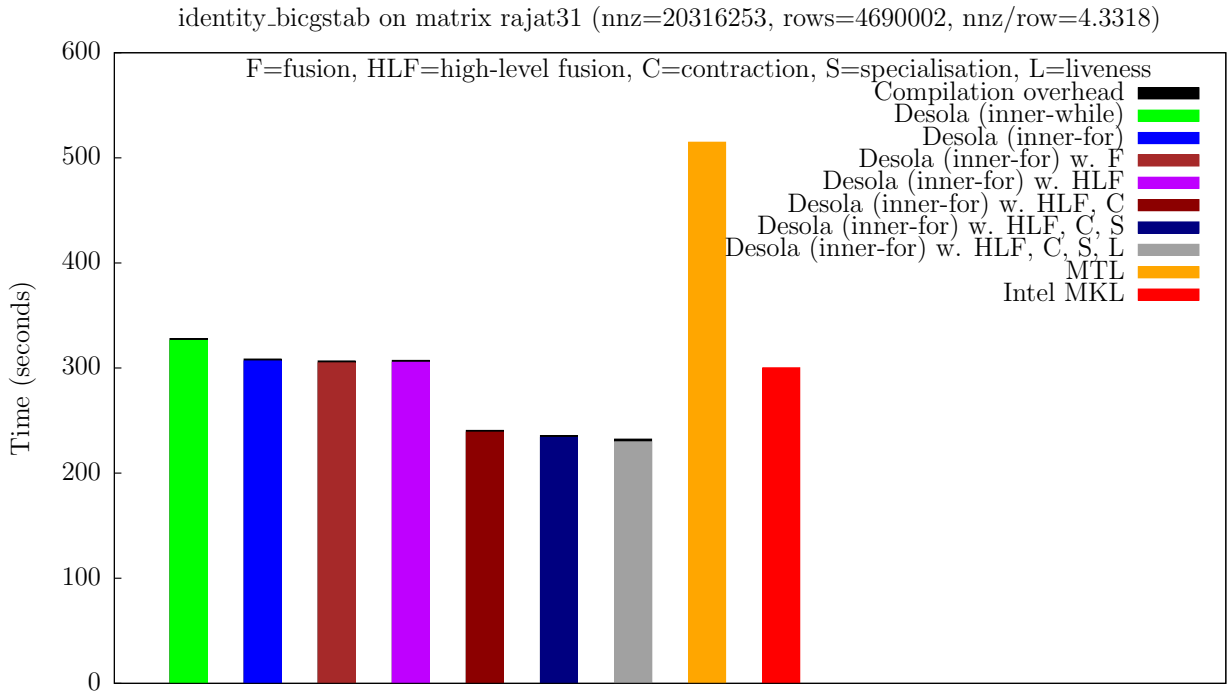


(a) architecture 1

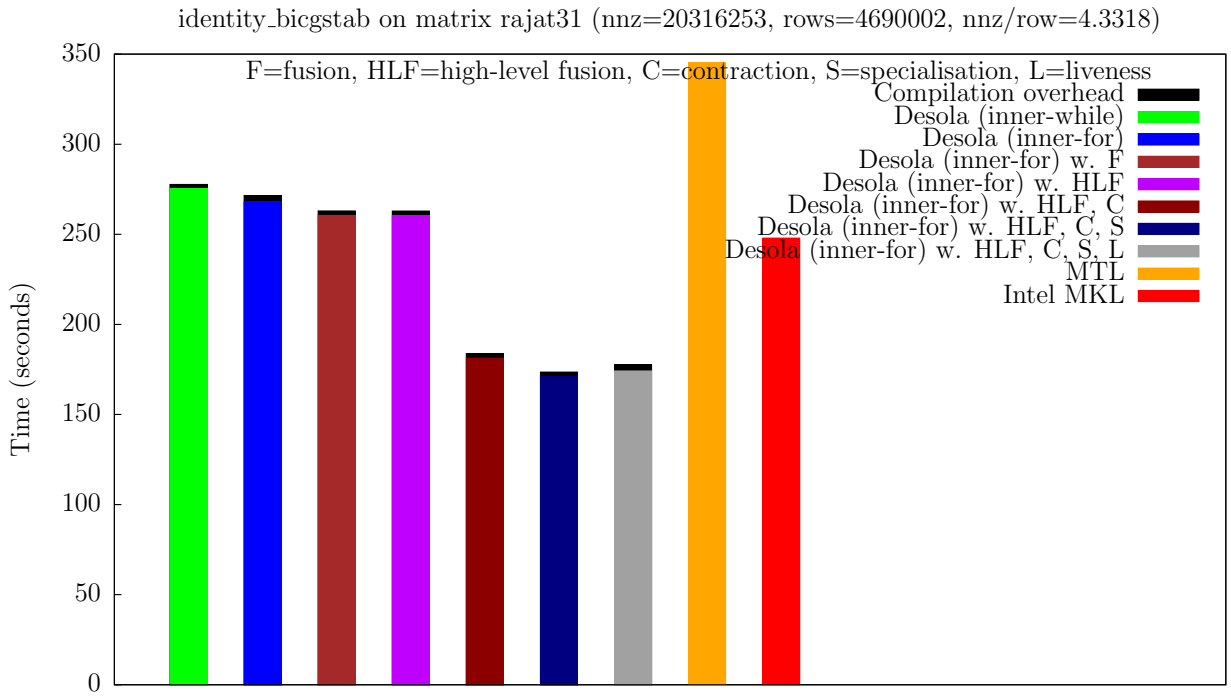


(b) architecture 2

Figure B.7: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with matrix rajat26 on our test architectures.

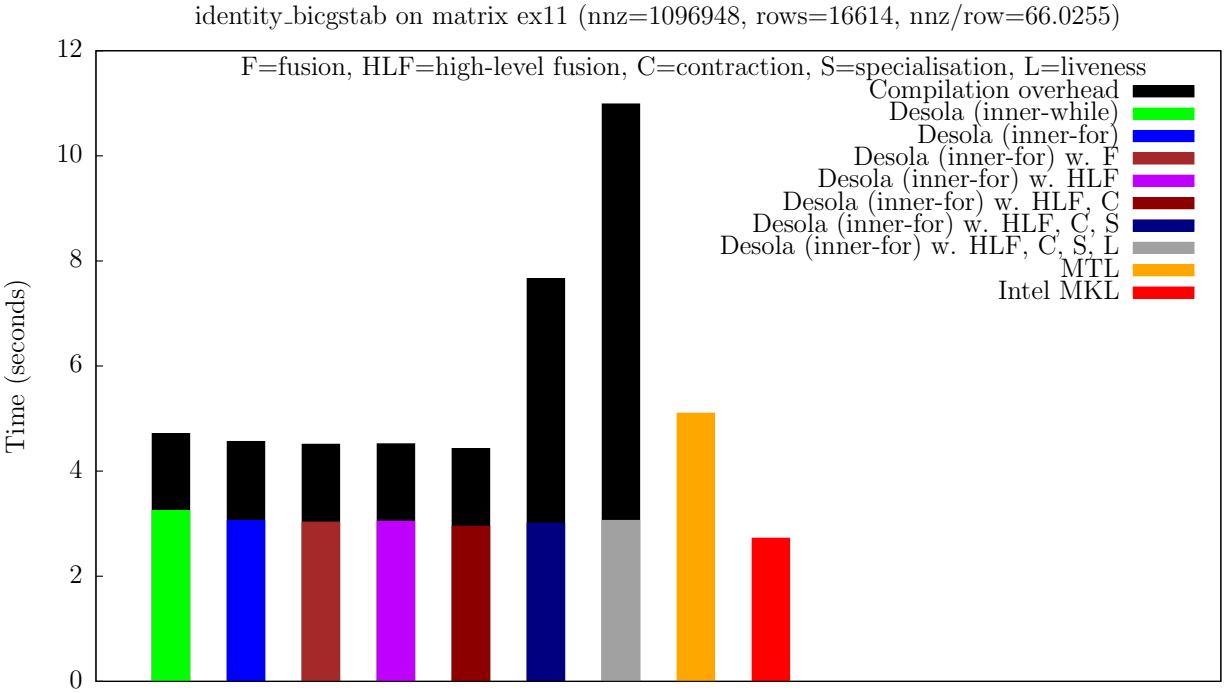


(a) architecture 1

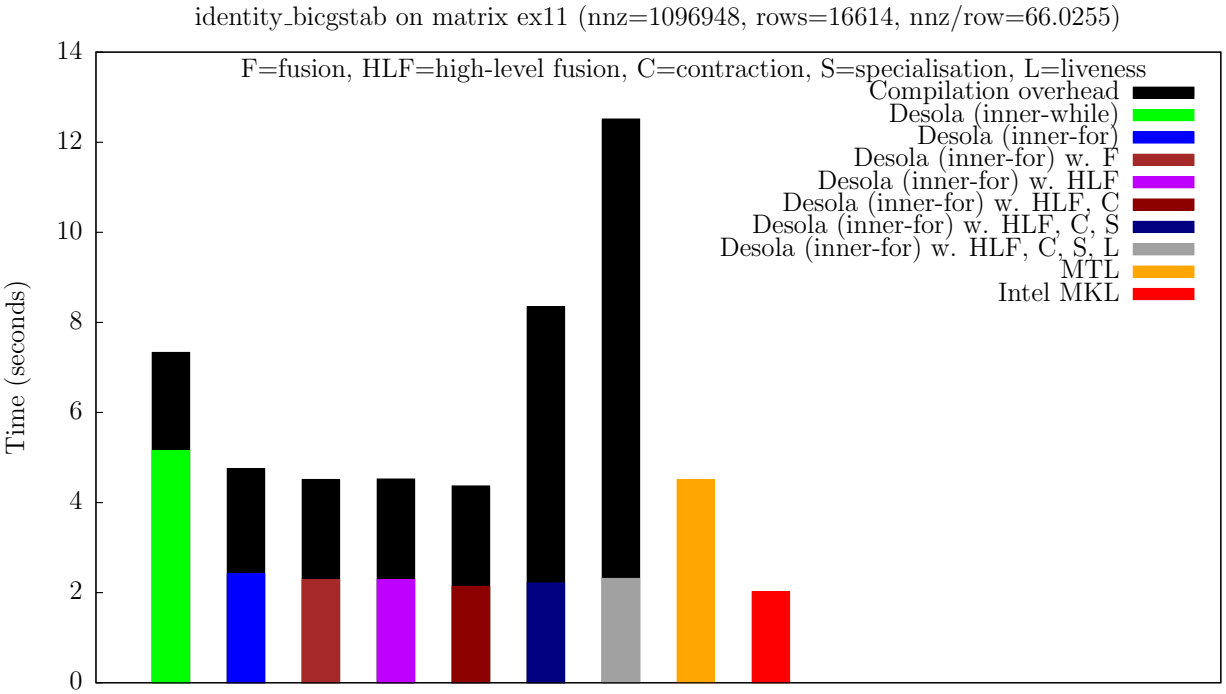


(b) architecture 2

Figure B.8: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with matrix rajat31 on our test architectures.

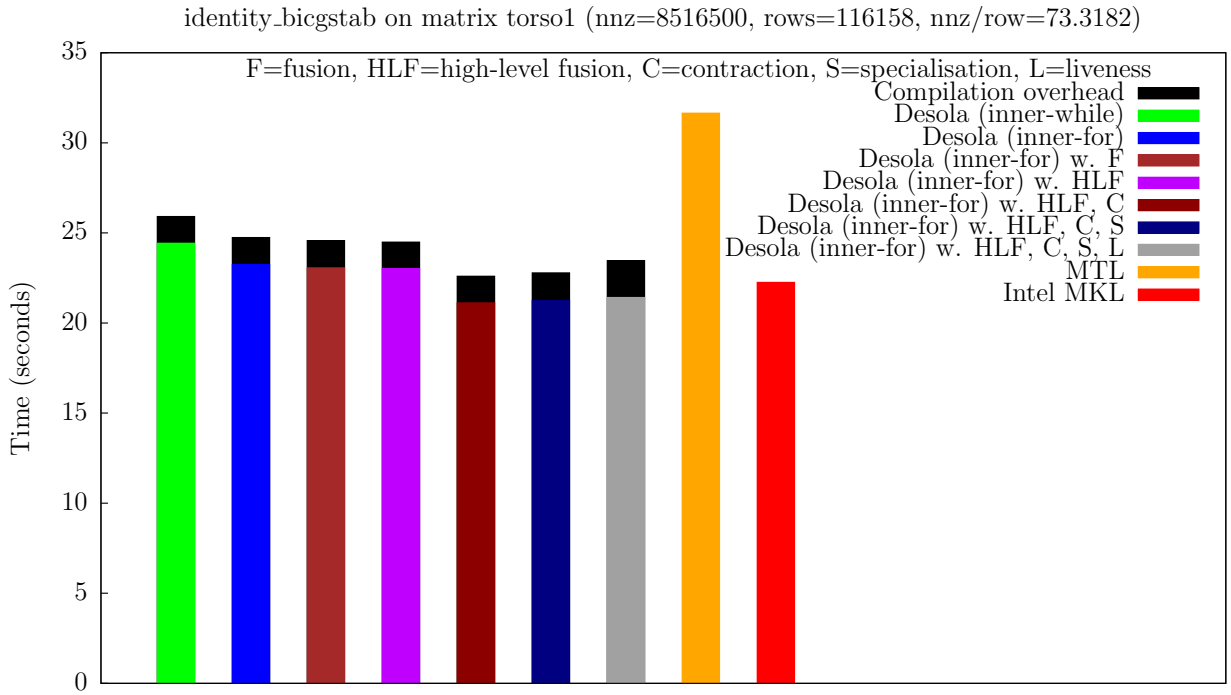


(a) architecture 1

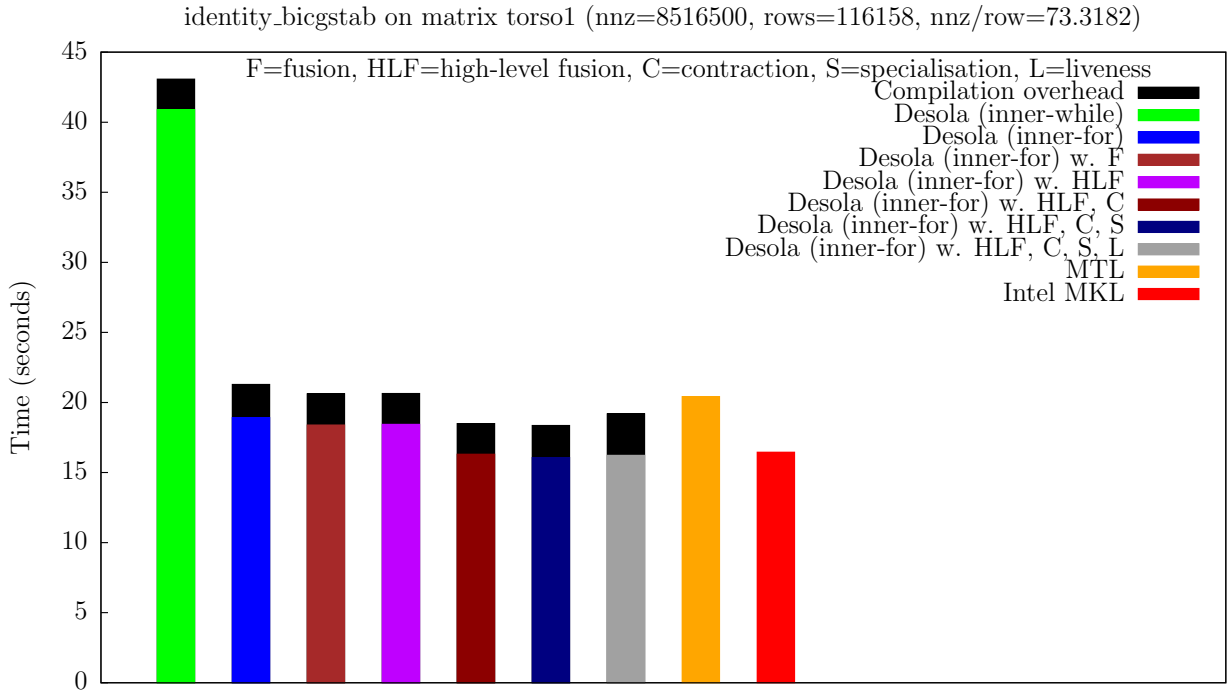


(b) architecture 2

Figure B.9: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with matrix ex11 on our test architectures.

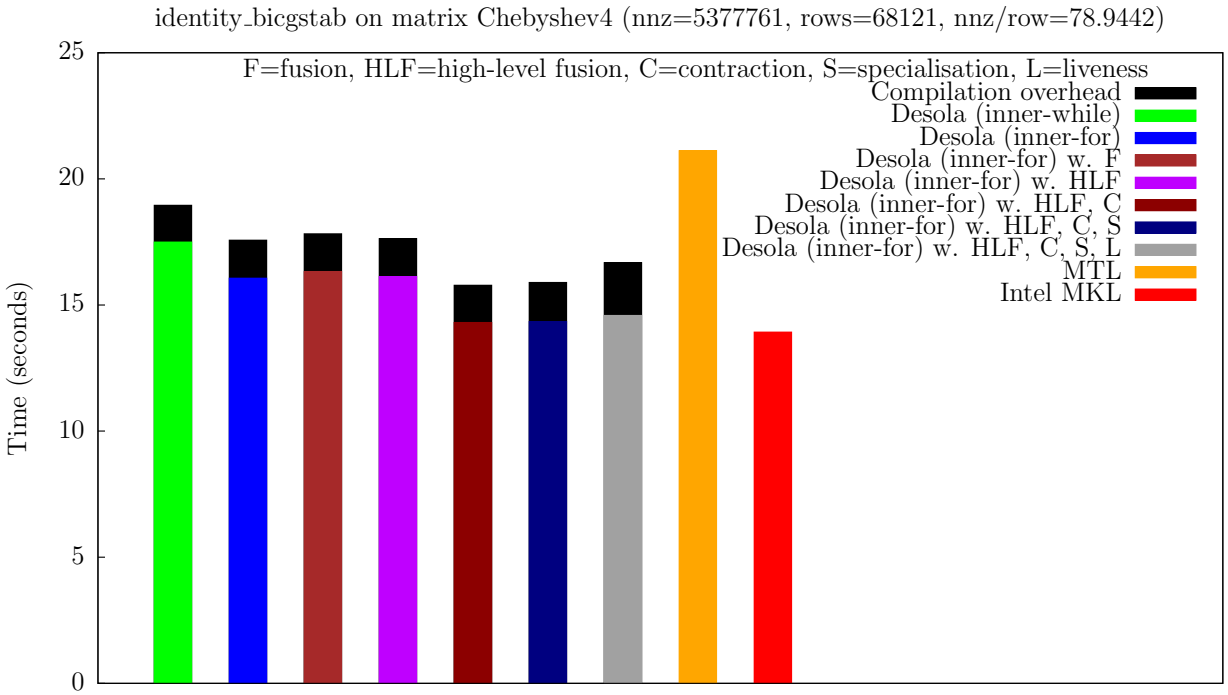


(a) architecture 1

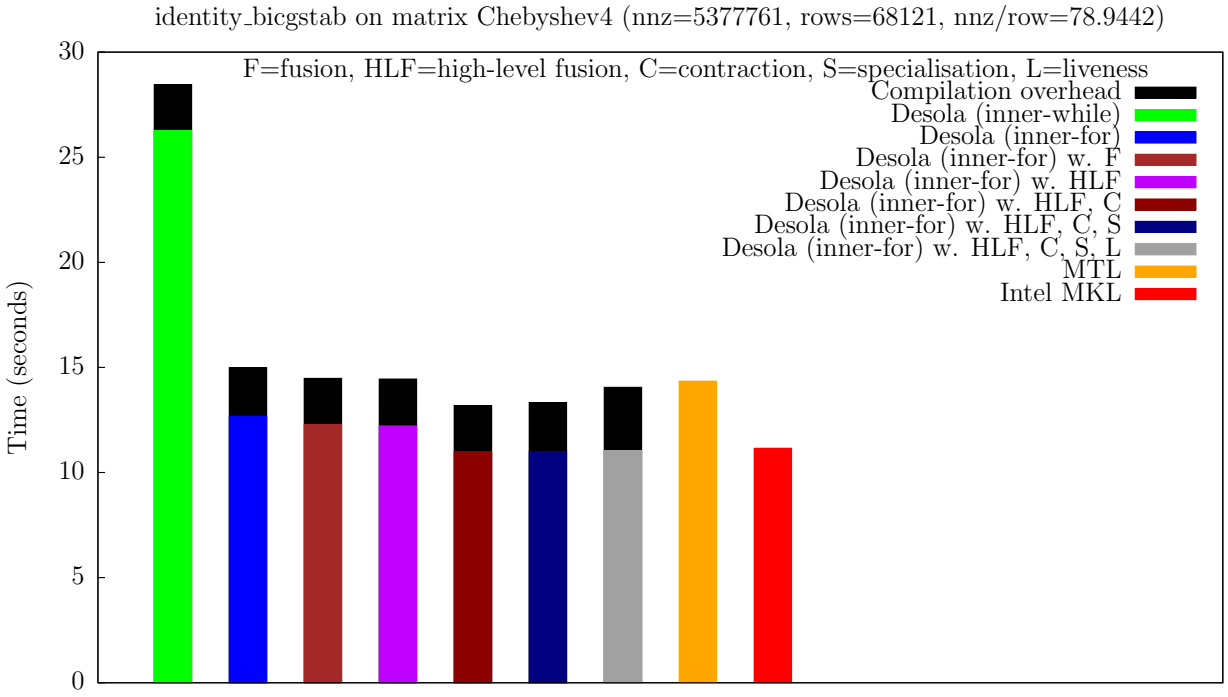


(b) architecture 2

Figure B.10: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with matrix torso1 on our test architectures.

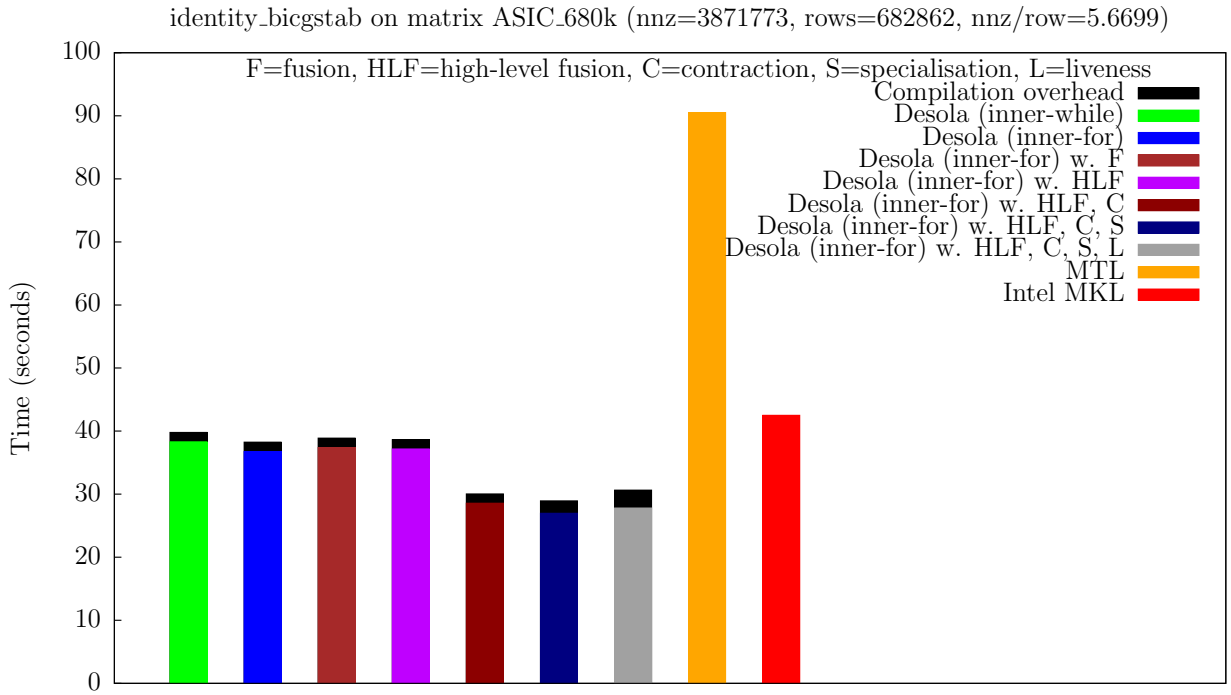


(a) architecture 1

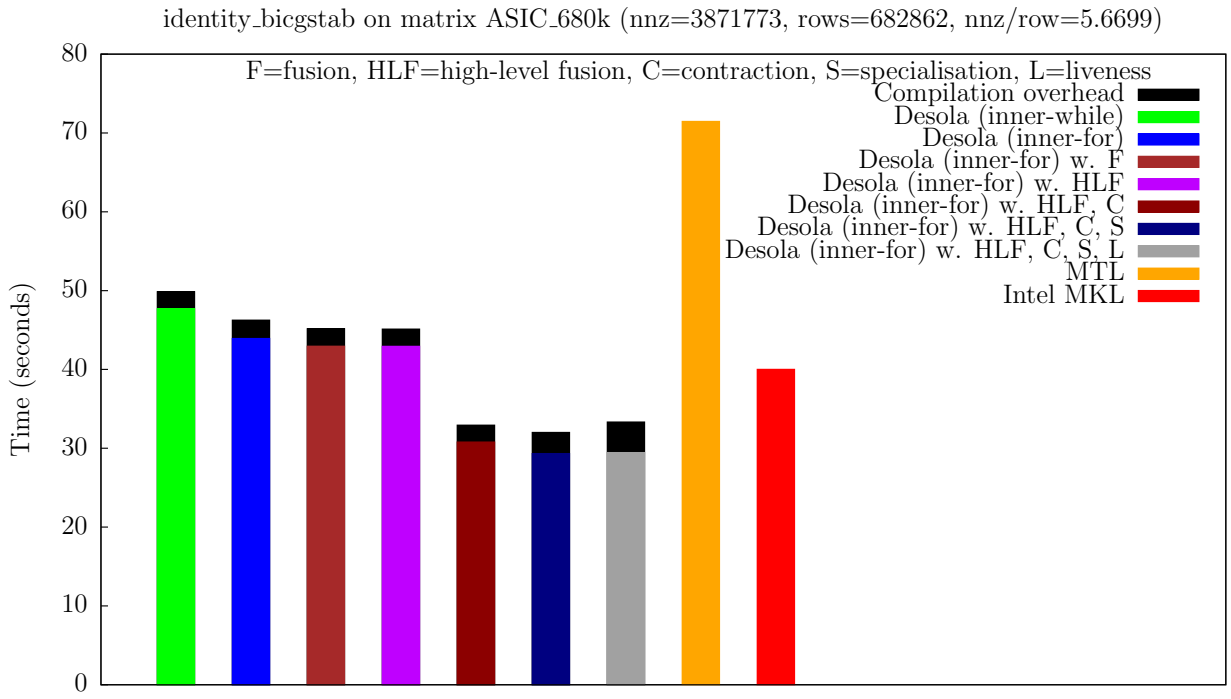


(b) architecture 2

Figure B.11: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with matrix Chebyshev4 on our test architectures.



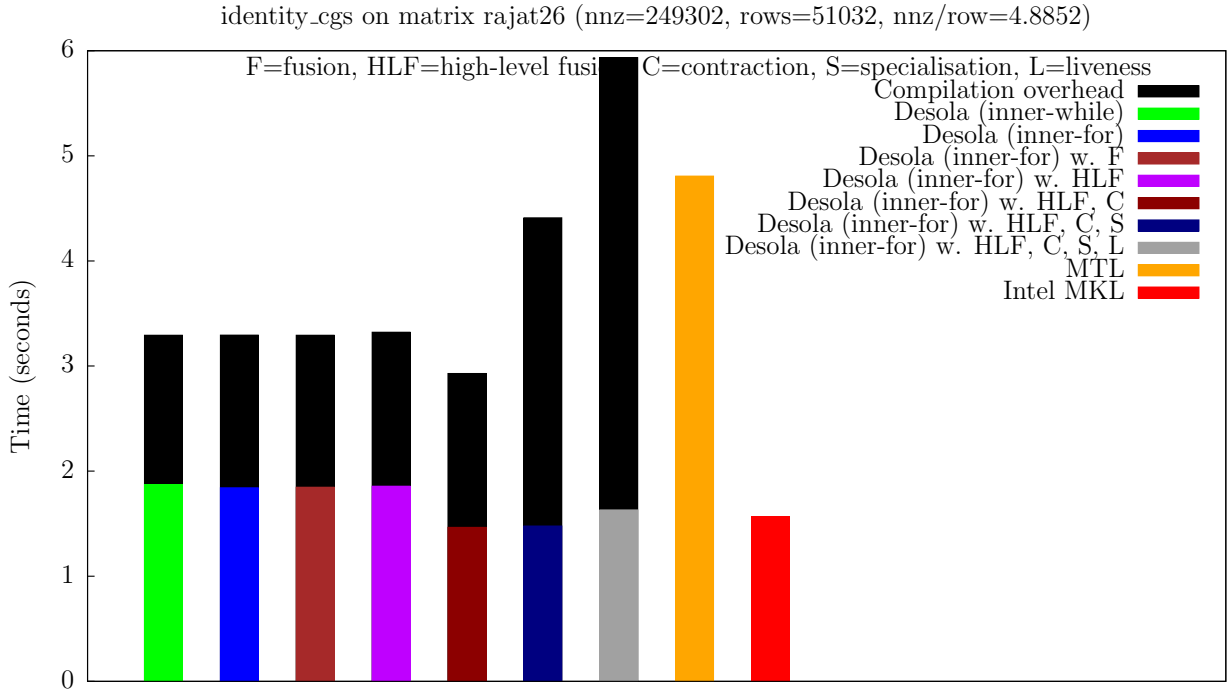
(a) architecture 1



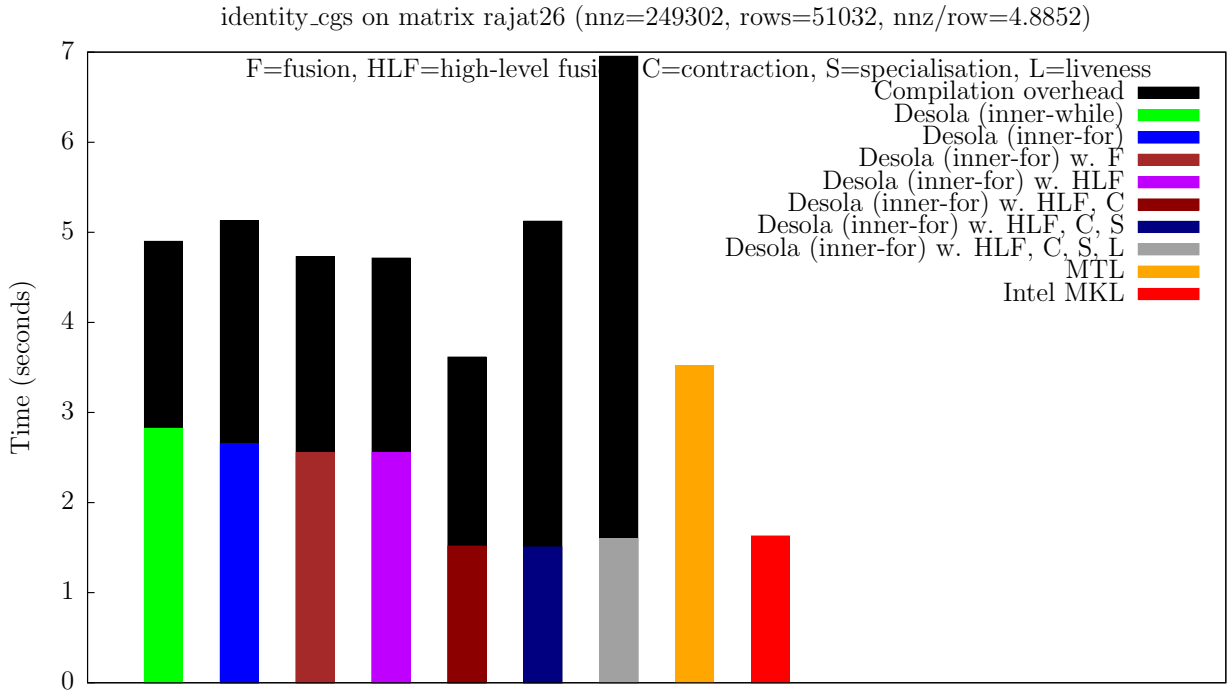
(b) architecture 2

Figure B.12: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with matrix ASIC_680k on our test architectures.

B.3 Conjugate Gradient Squared Solver

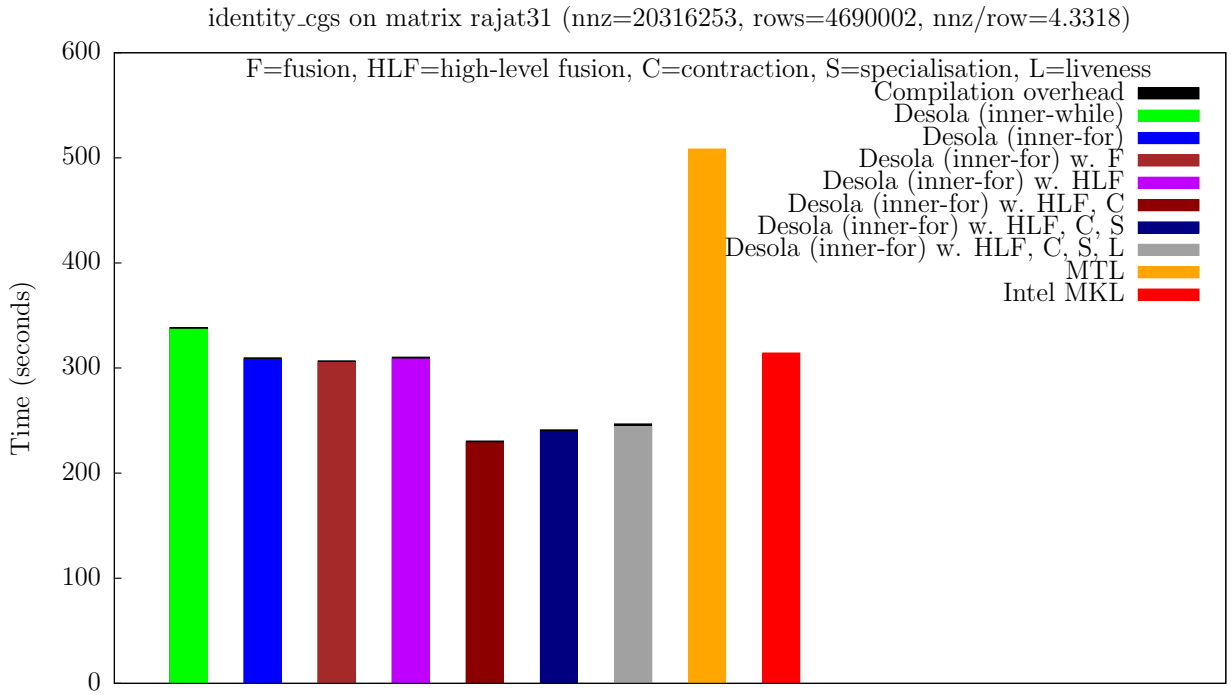


(a) architecture 1

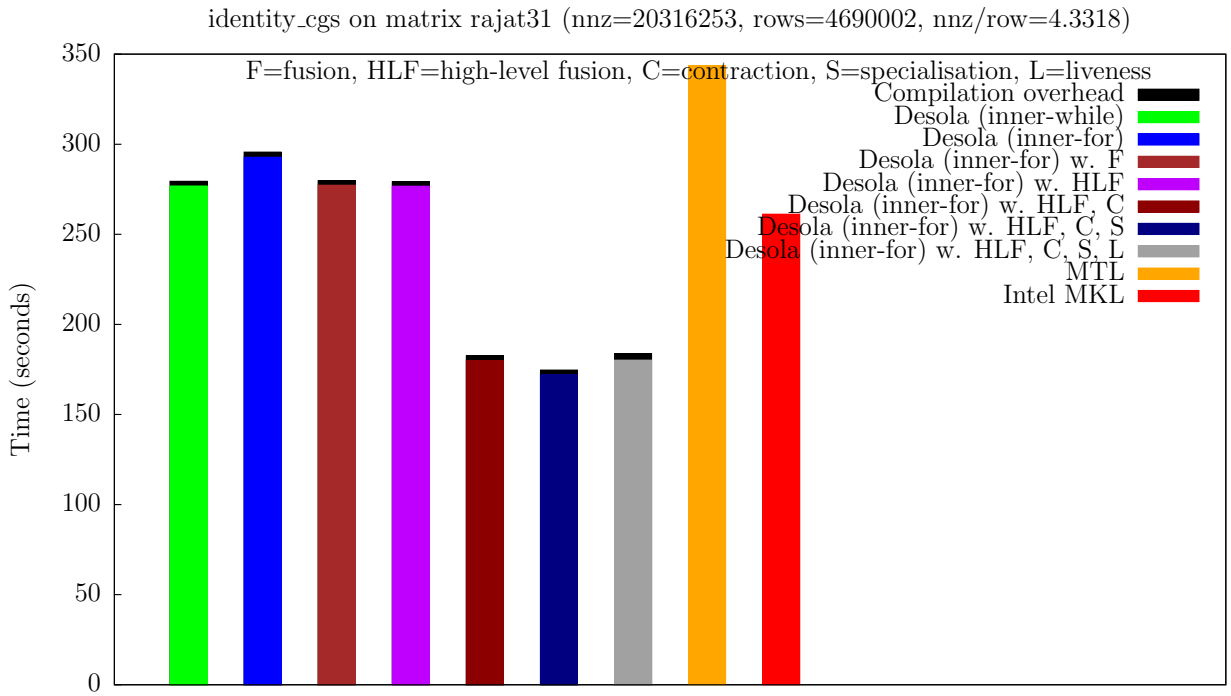


(b) architecture 2

Figure B.13: Time to execute 256 iterations of the Conjugate Gradient Squared solver with matrix rajat26 on our test architectures.

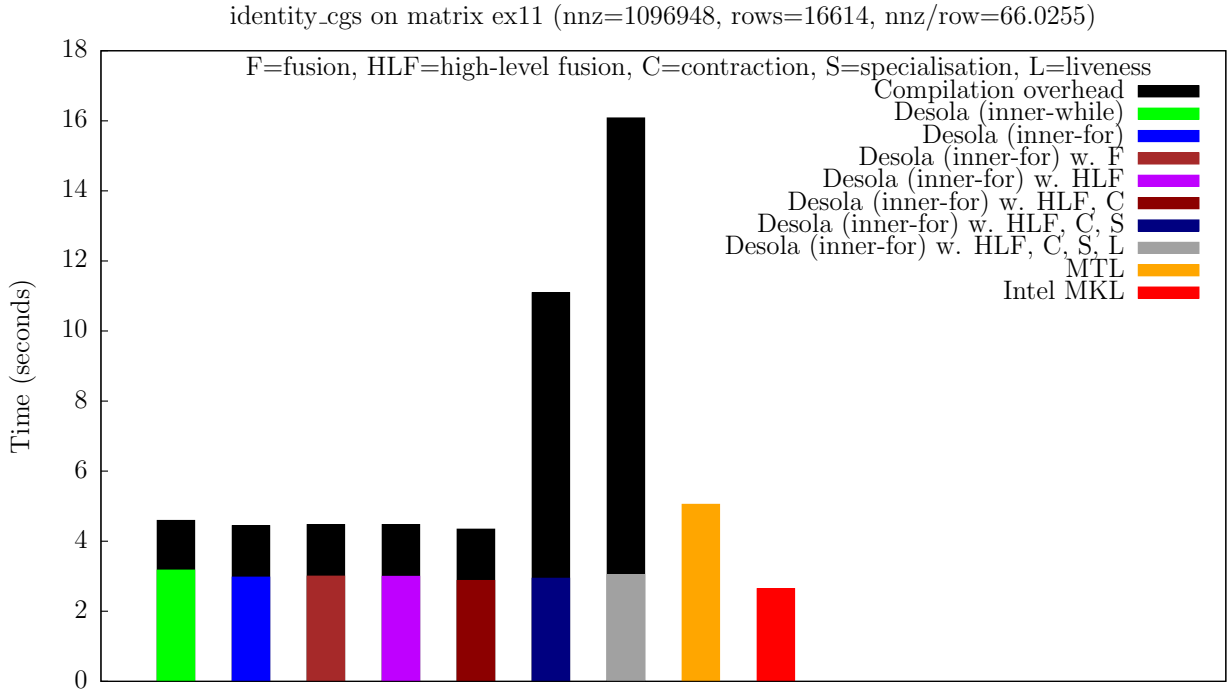


(a) architecture 1

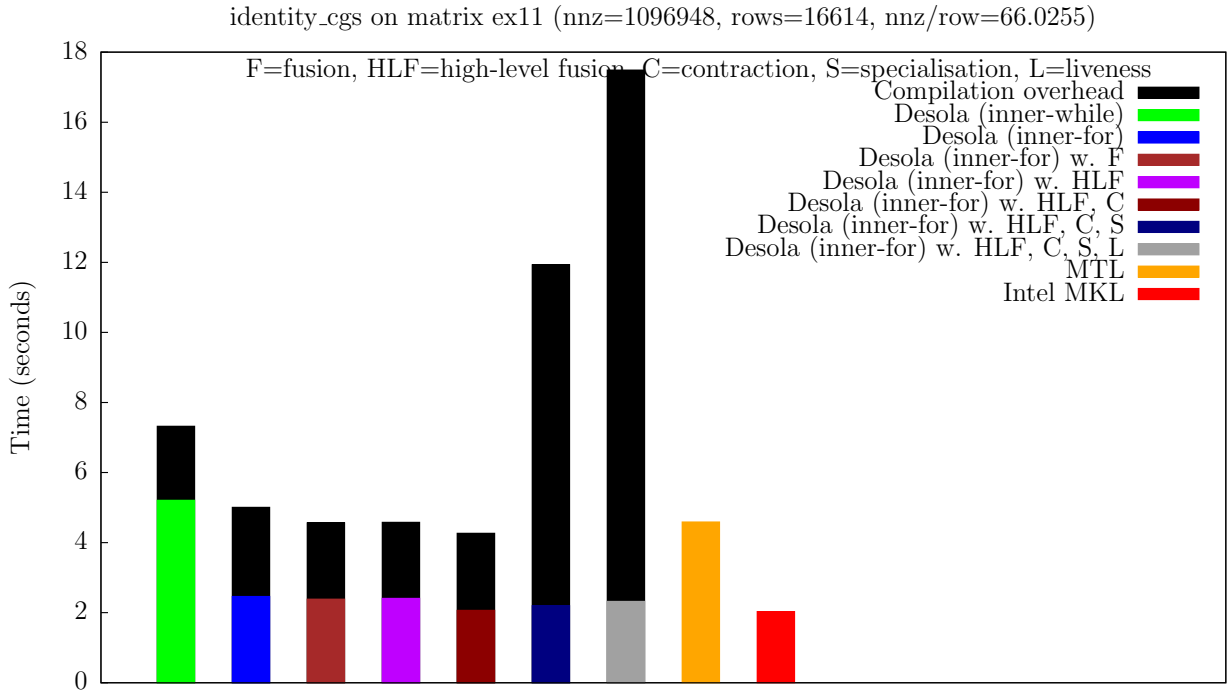


(b) architecture 2

Figure B.14: Time to execute 256 iterations of the Conjugate Gradient Squared solver with matrix rajat31 on our test architectures.

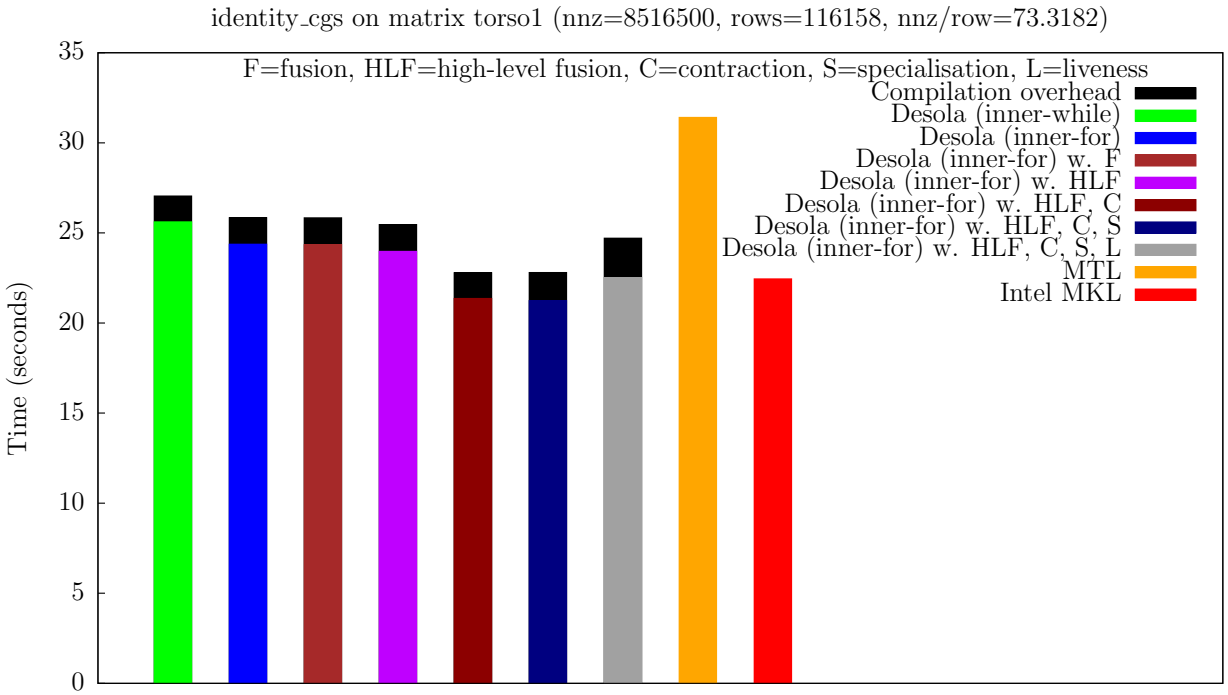


(a) architecture 1

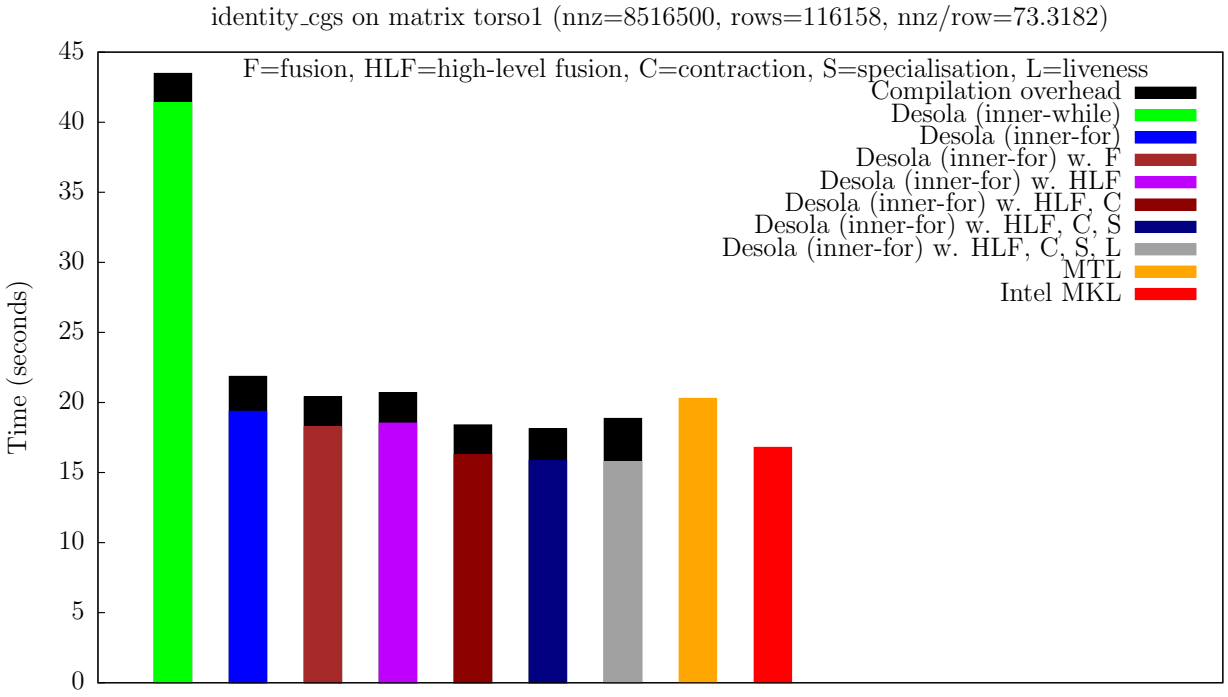


(b) architecture 2

Figure B.15: Time to execute 256 iterations of the Conjugate Gradient Squared solver with matrix ex11 on our test architectures.

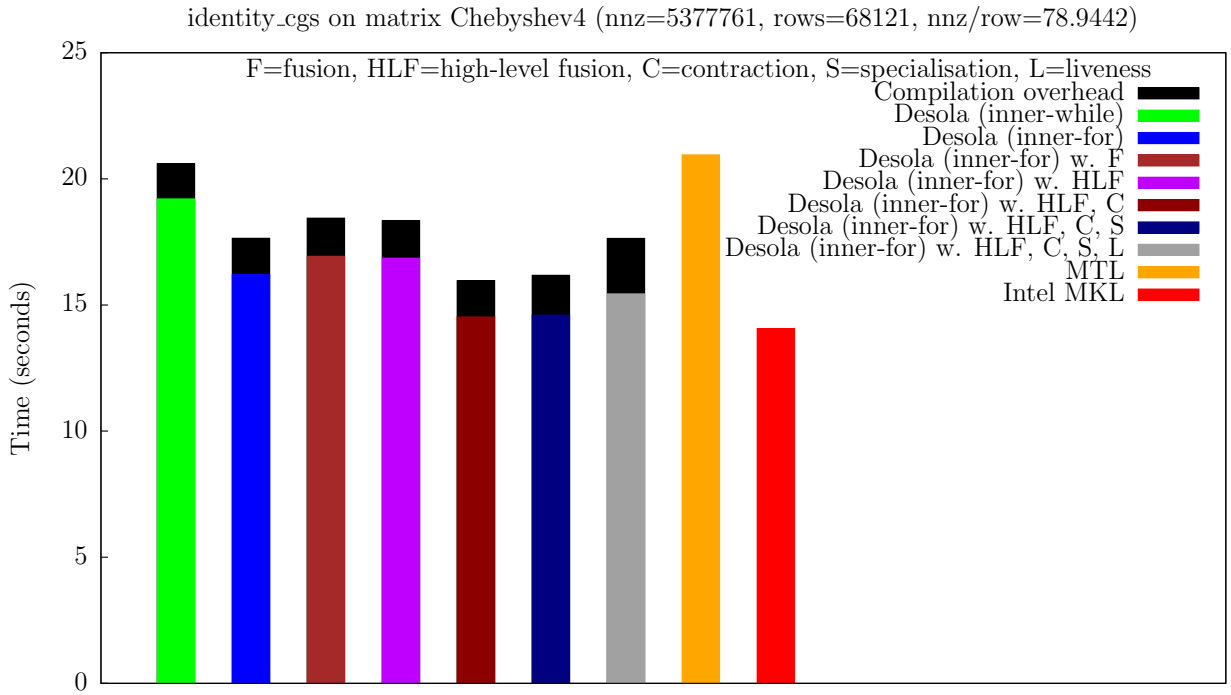


(a) architecture 1

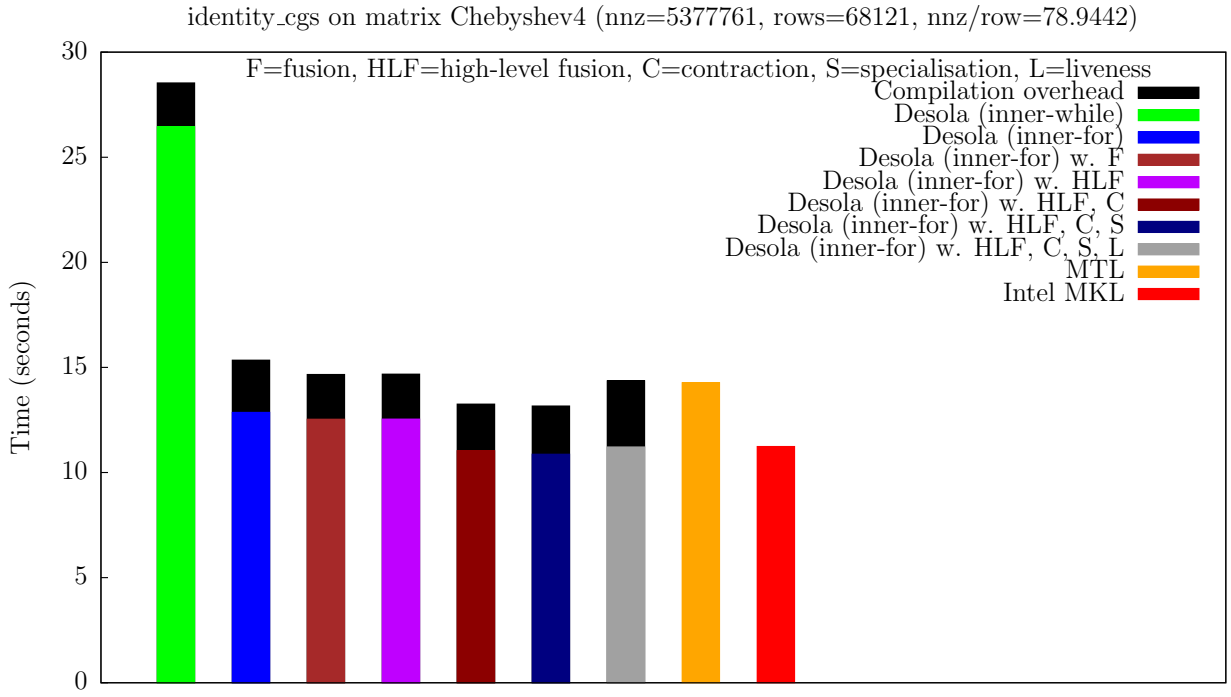


(b) architecture 2

Figure B.16: Time to execute 256 iterations of the Conjugate Gradient Squared solver with matrix torso1 on our test architectures.

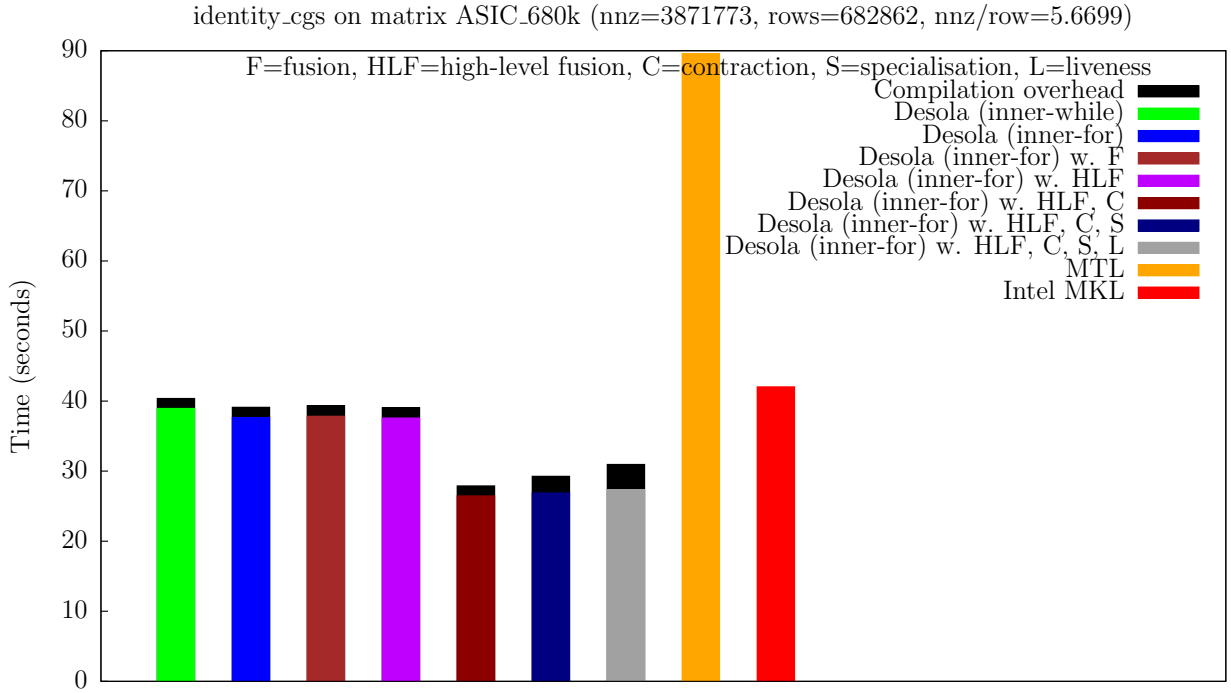


(a) architecture 1

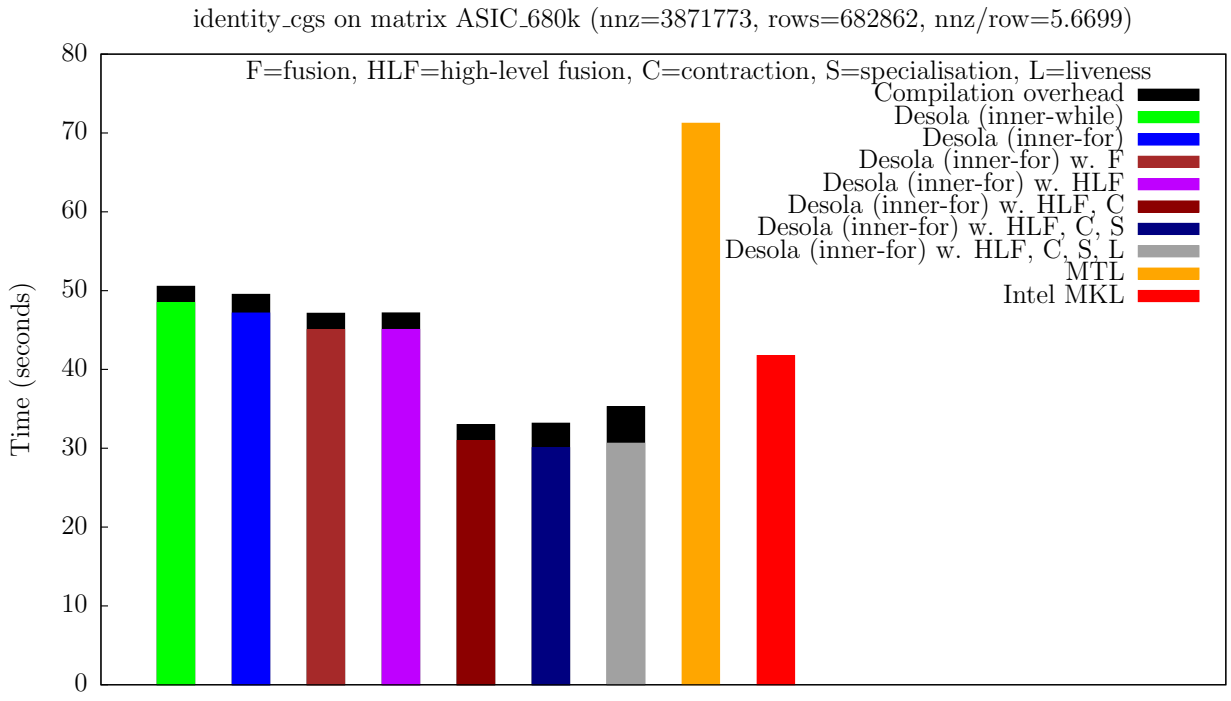


(b) architecture 2

Figure B.17: Time to execute 256 iterations of the Conjugate Gradient Squared solver with matrix Chebyshev4 on our test architectures.



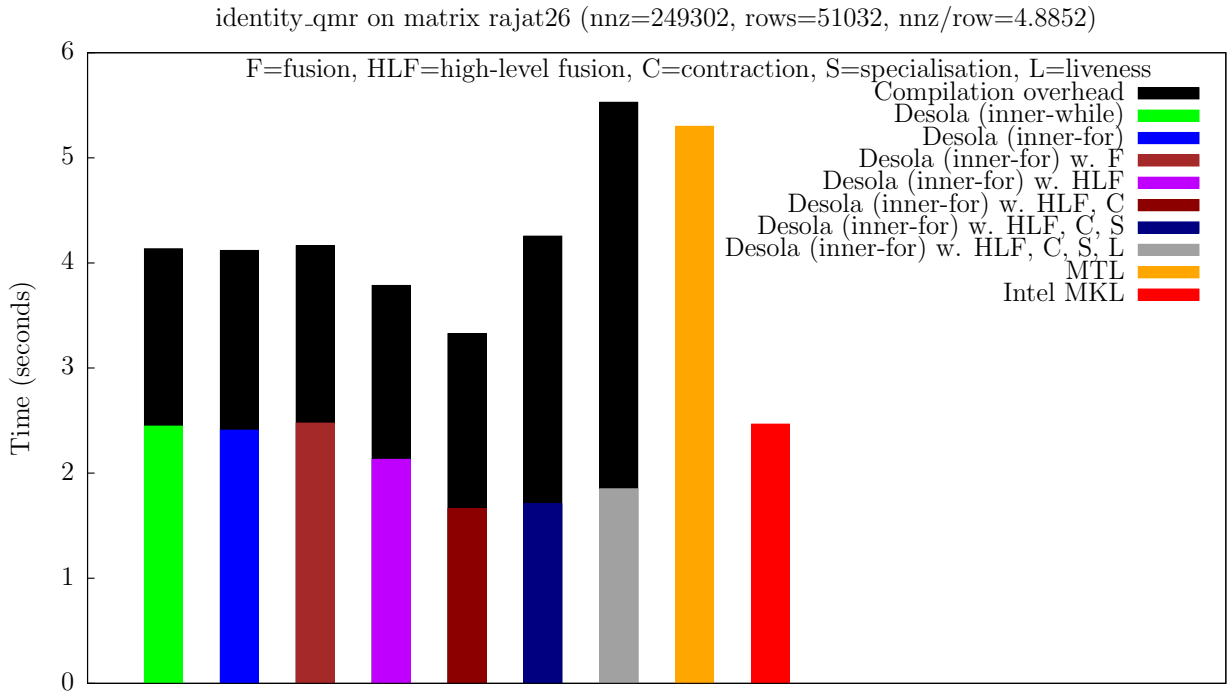
(a) architecture 1



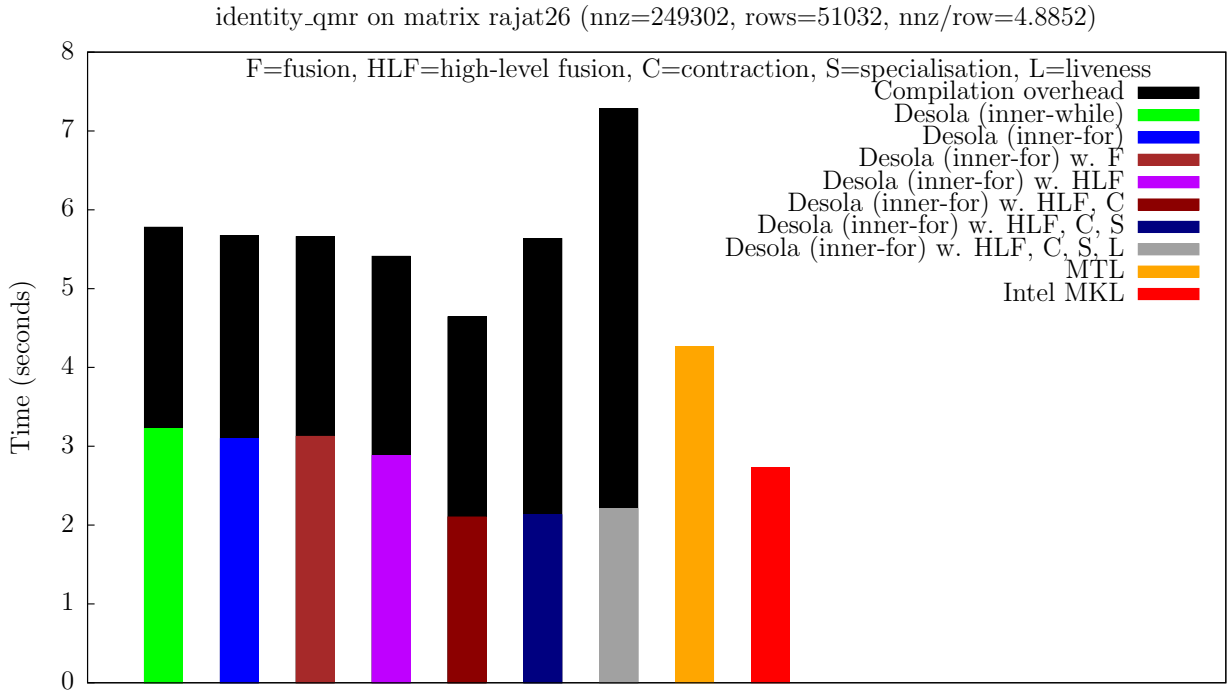
(b) architecture 2

Figure B.18: Time to execute 256 iterations of the Conjugate Gradient Squared solver with matrix ASIC_680k on our test architectures.

B.4 Quasi-Minimal Residual Solver

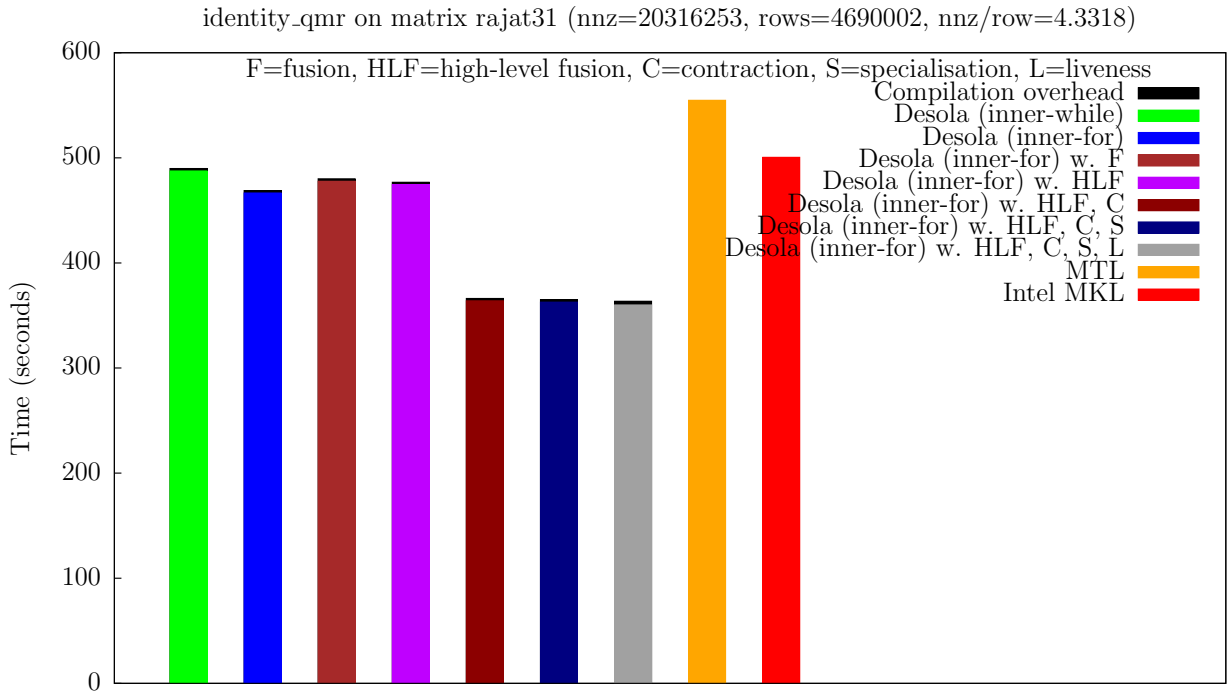


(a) architecture 1

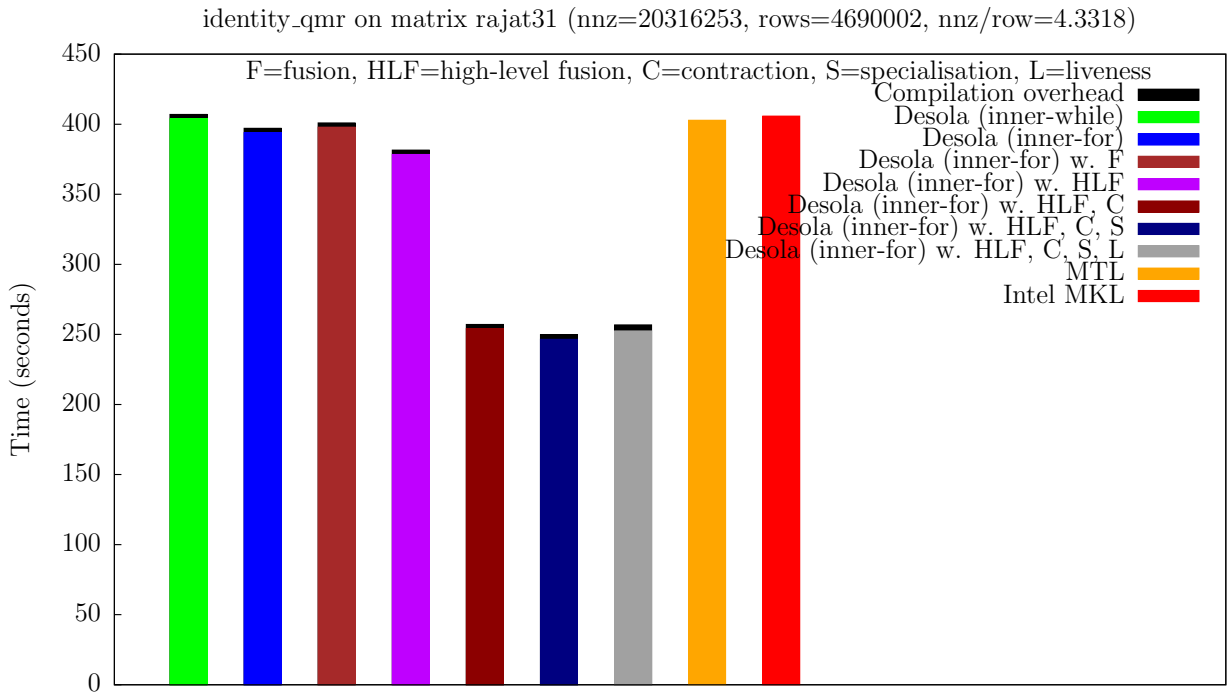


(b) architecture 2

Figure B.19: Time to execute 256 iterations of the Quasi-Minimal Residual solver with matrix rajat26 on our test architectures.

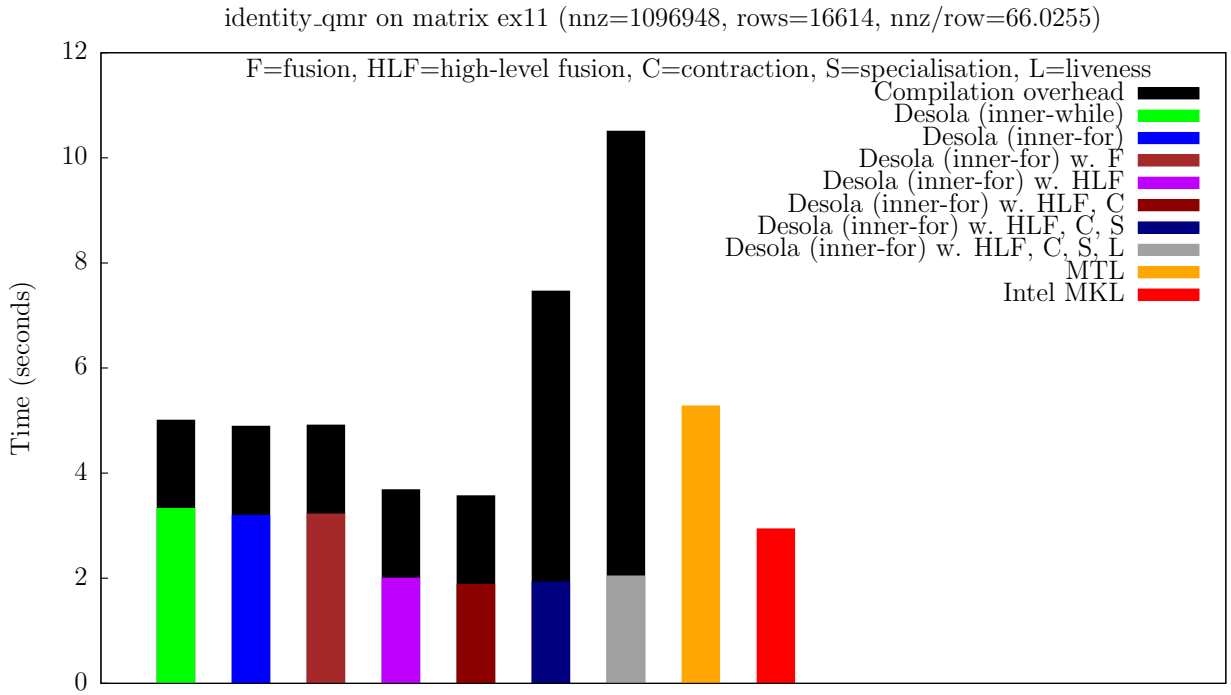


(a) architecture 1

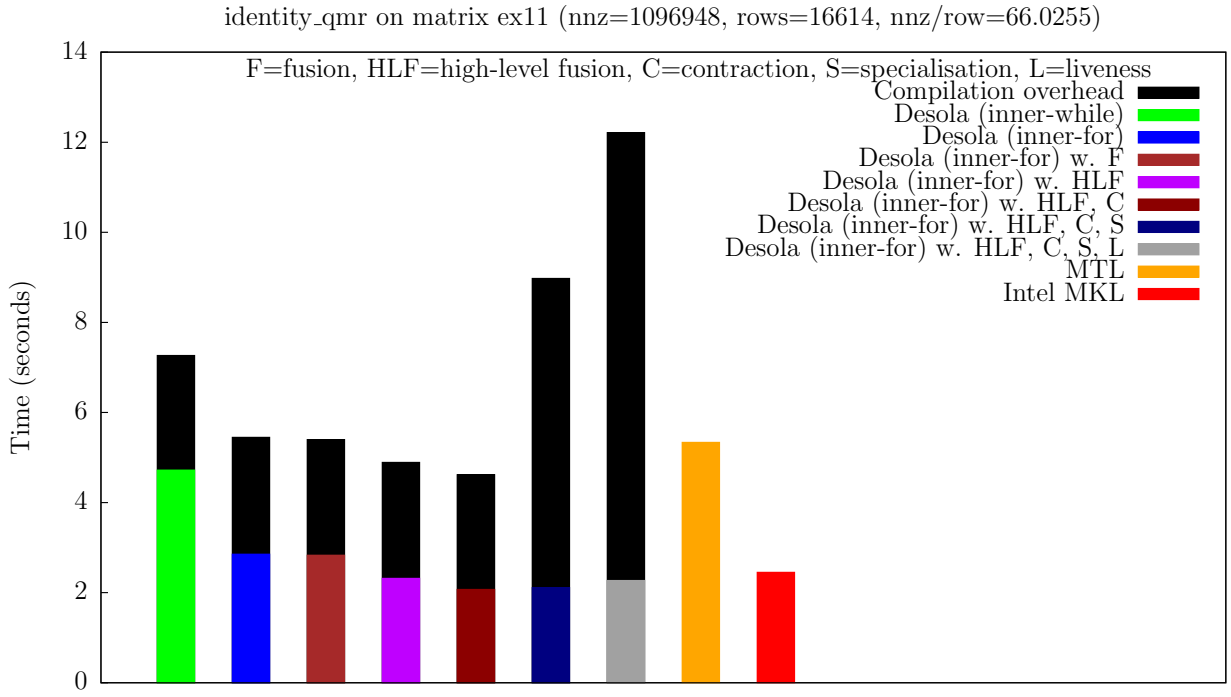


(b) architecture 2

Figure B.20: Time to execute 256 iterations of the Quasi-Minimal Residual solver with matrix rajat31 on our test architectures.

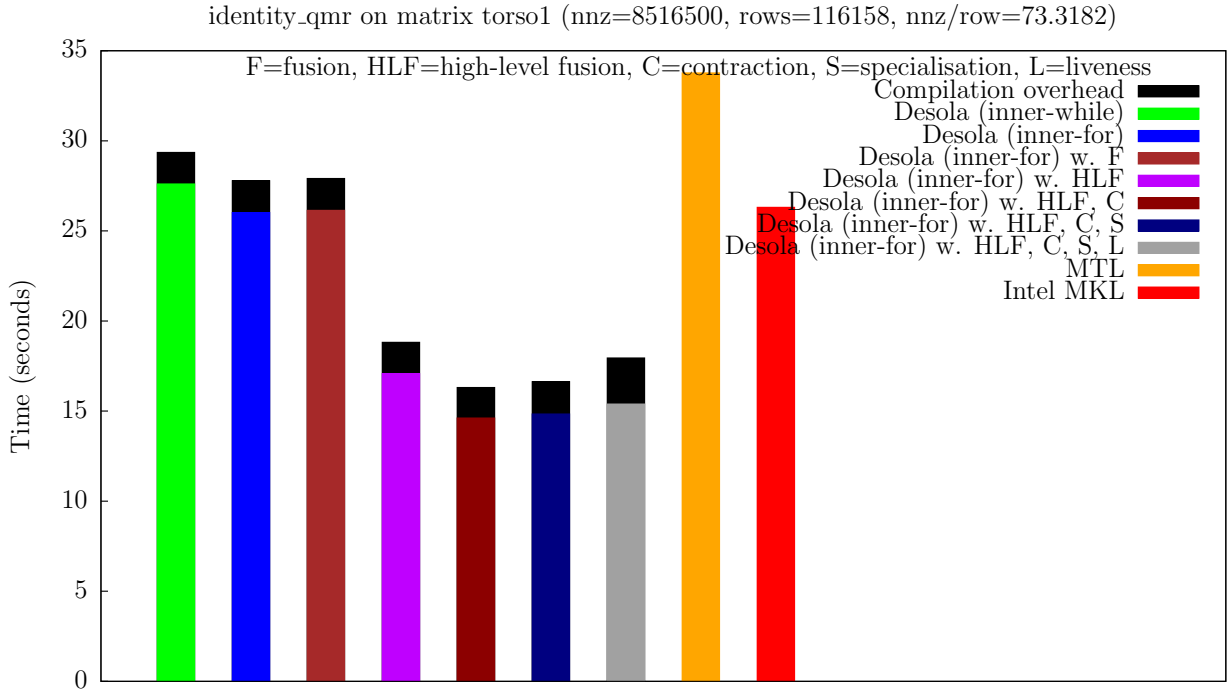


(a) architecture 1

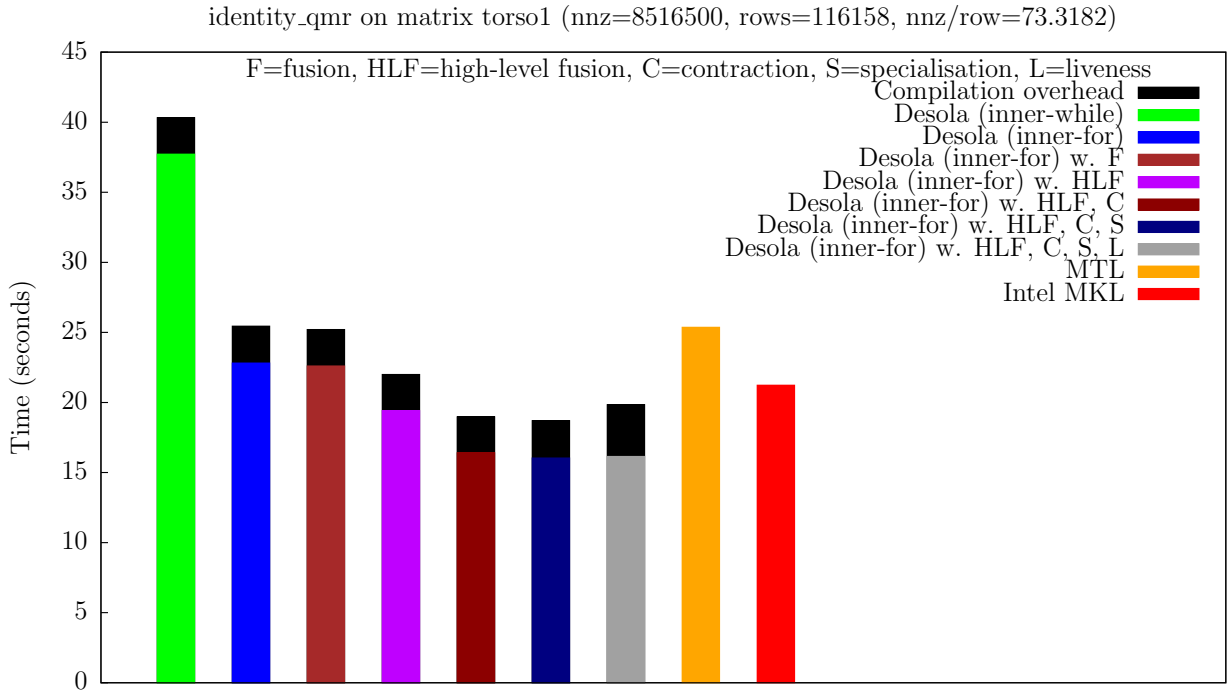


(b) architecture 2

Figure B.21: Time to execute 256 iterations of the Quasi-Minimal Residual solver with matrix ex11 on our test architectures.

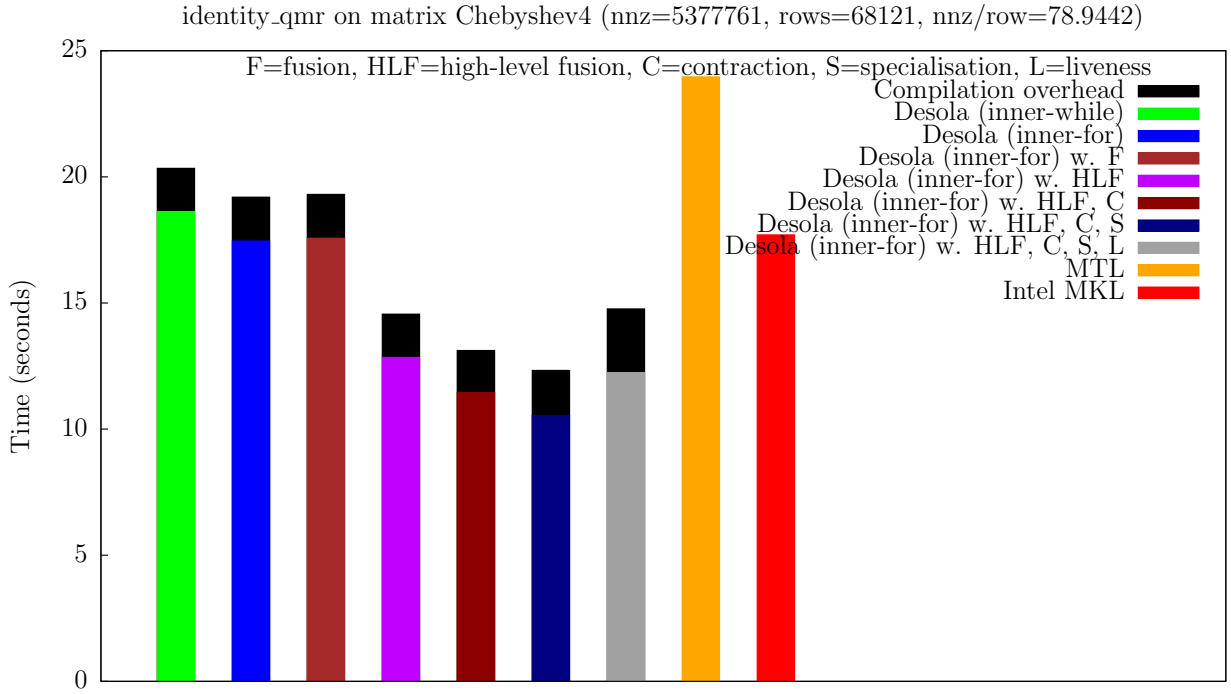


(a) architecture 1

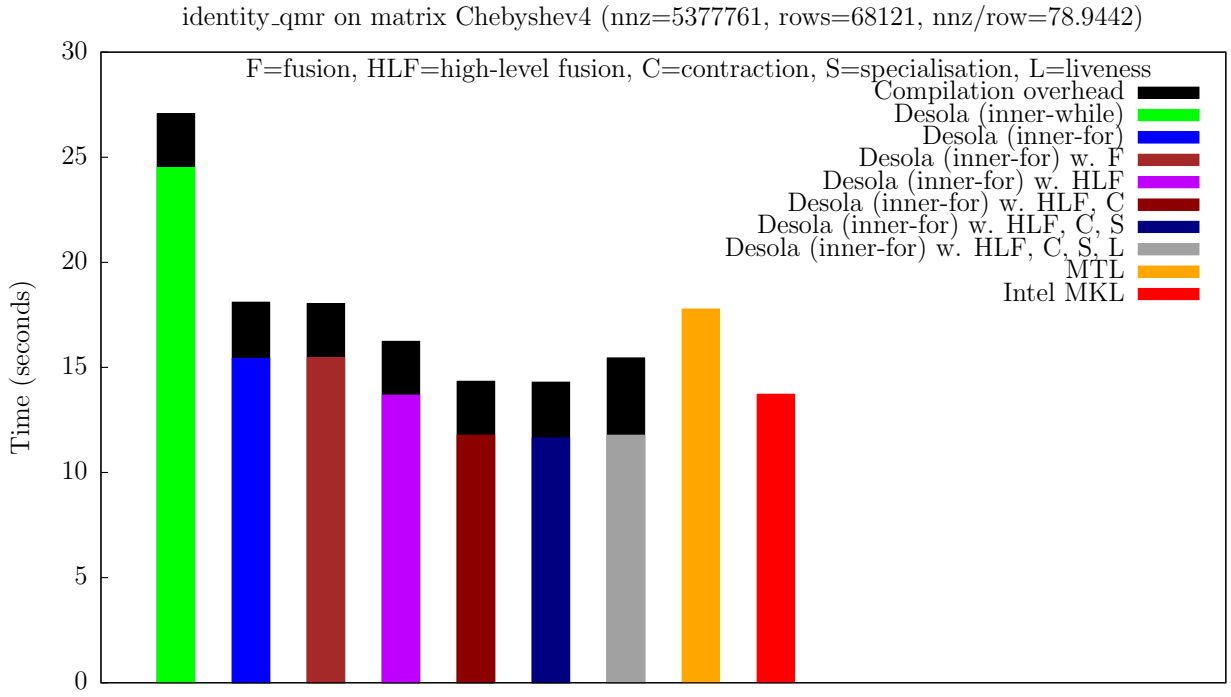


(b) architecture 2

Figure B.22: Time to execute 256 iterations of the Quasi-Minimal Residual solver with matrix torso1 on our test architectures.

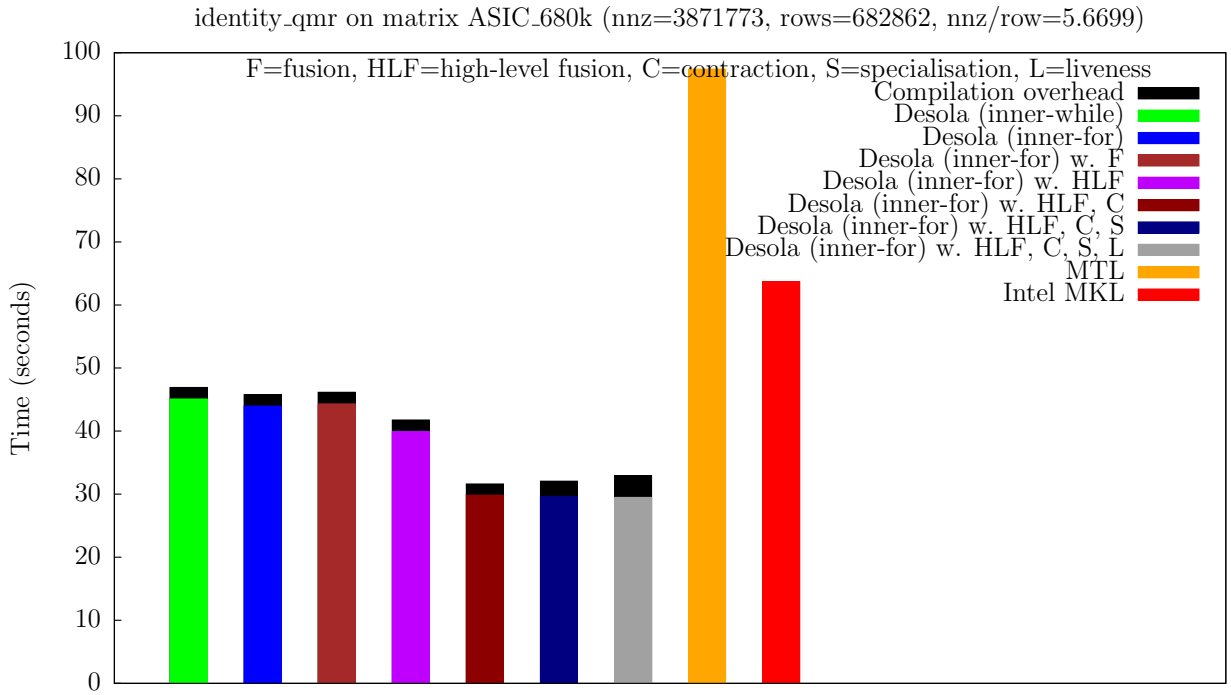


(a) architecture 1

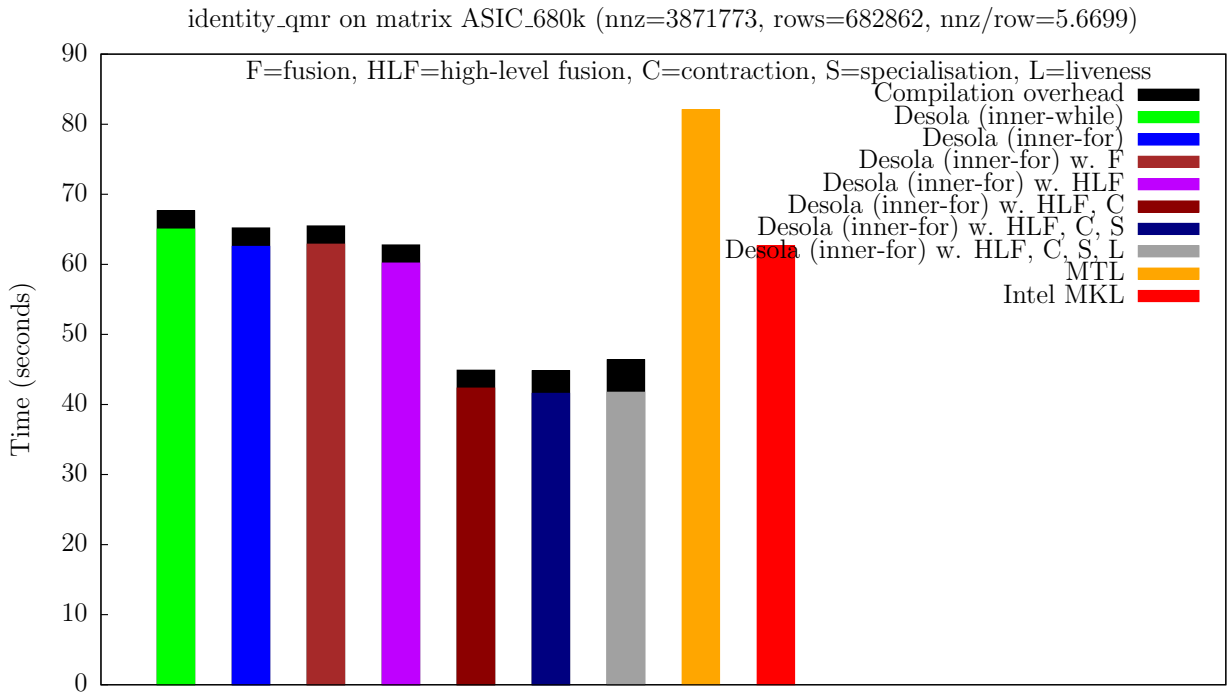


(b) architecture 2

Figure B.23: Time to execute 256 iterations of the Quasi-Minimal Residual solver with matrix Chebyshev4 on our test architectures.



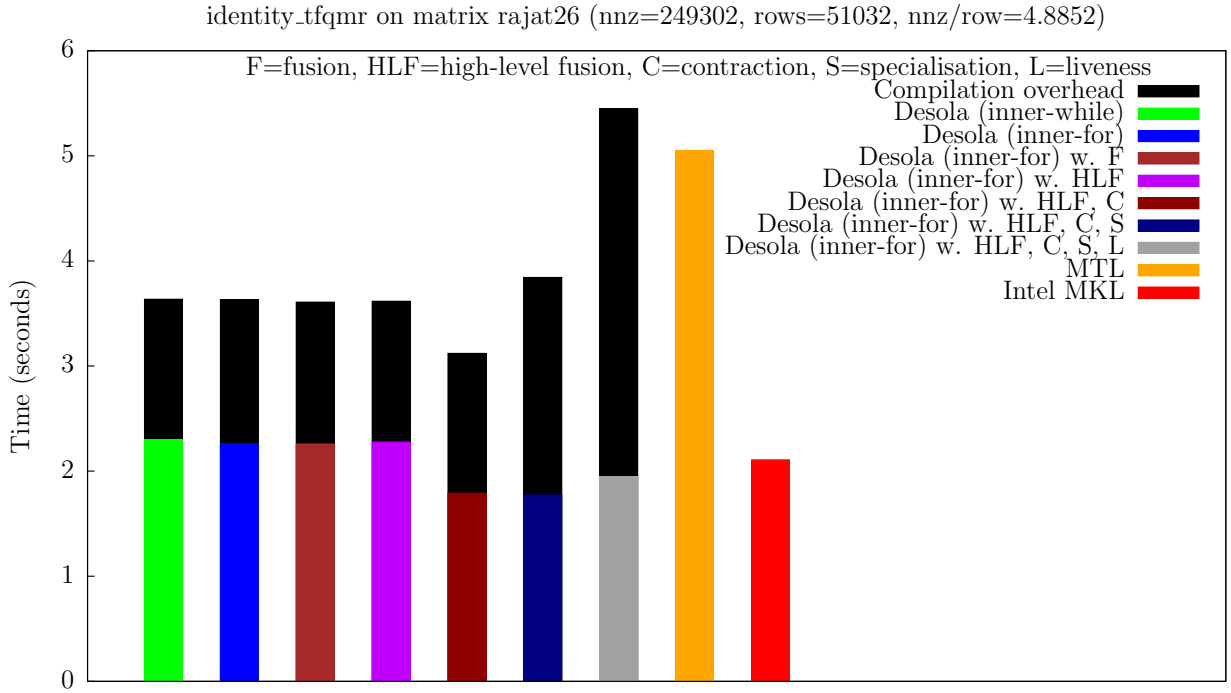
(a) architecture 1



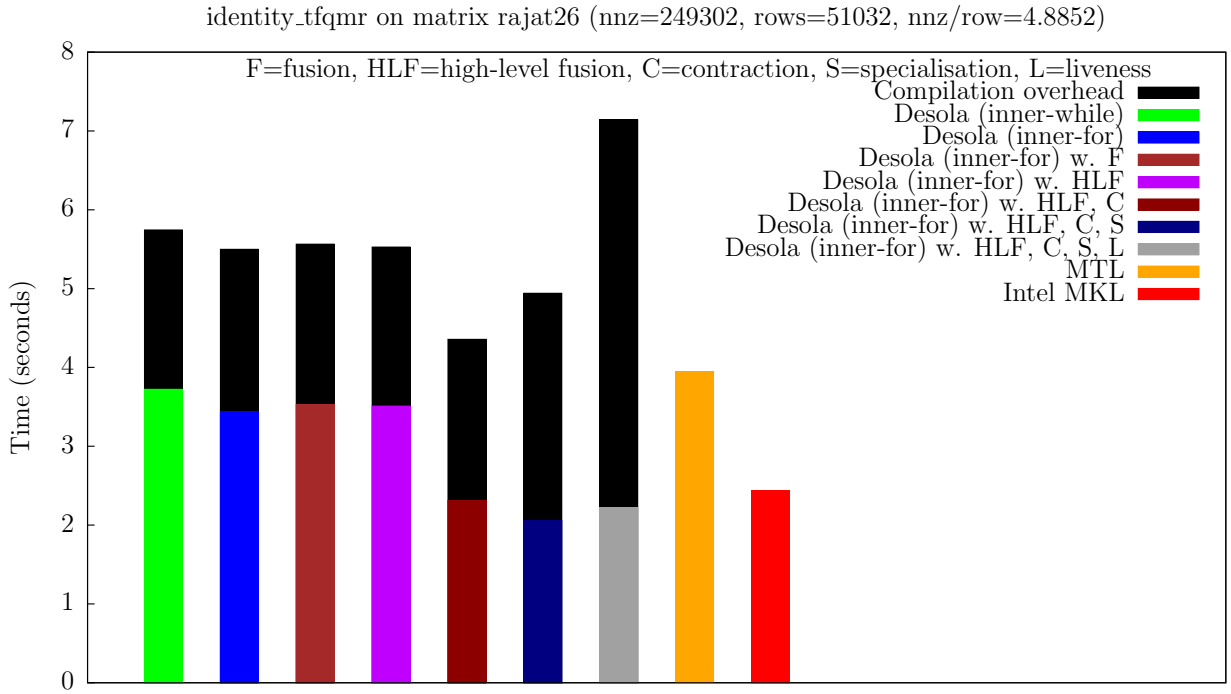
(b) architecture 2

Figure B.24: Time to execute 256 iterations of the Quasi-Minimal Residual solver with matrix ASIC_680k on our test architectures.

B.5 Transpose-Free Quasi-Minimal Residual Solver

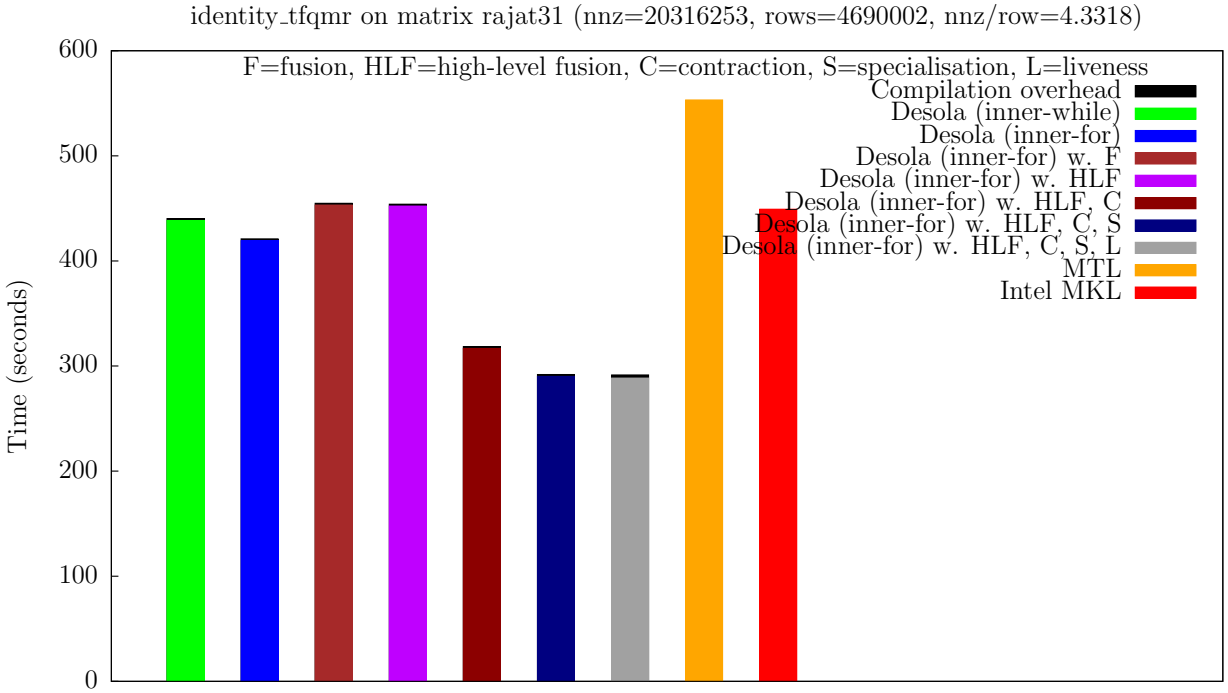


(a) architecture 1

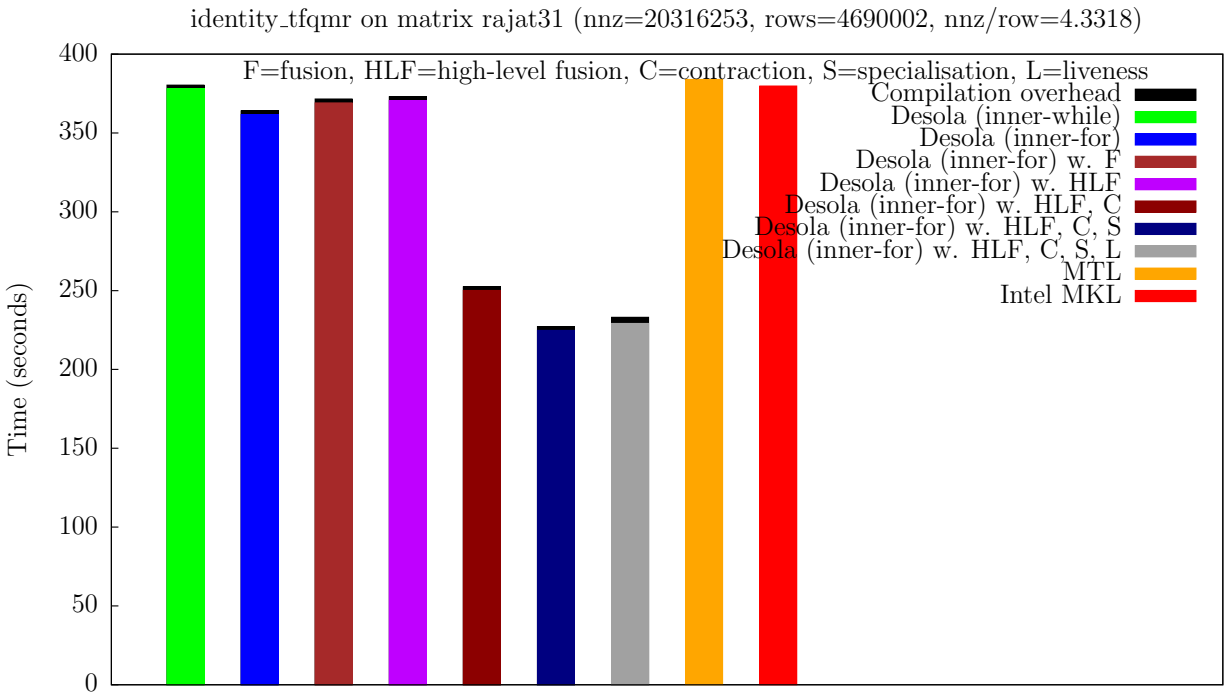


(b) architecture 2

Figure B.25: Time to execute 256 iterations of the Transpose-Free Quasi-Minimal Residual solver with matrix rajat26 on our test architectures.

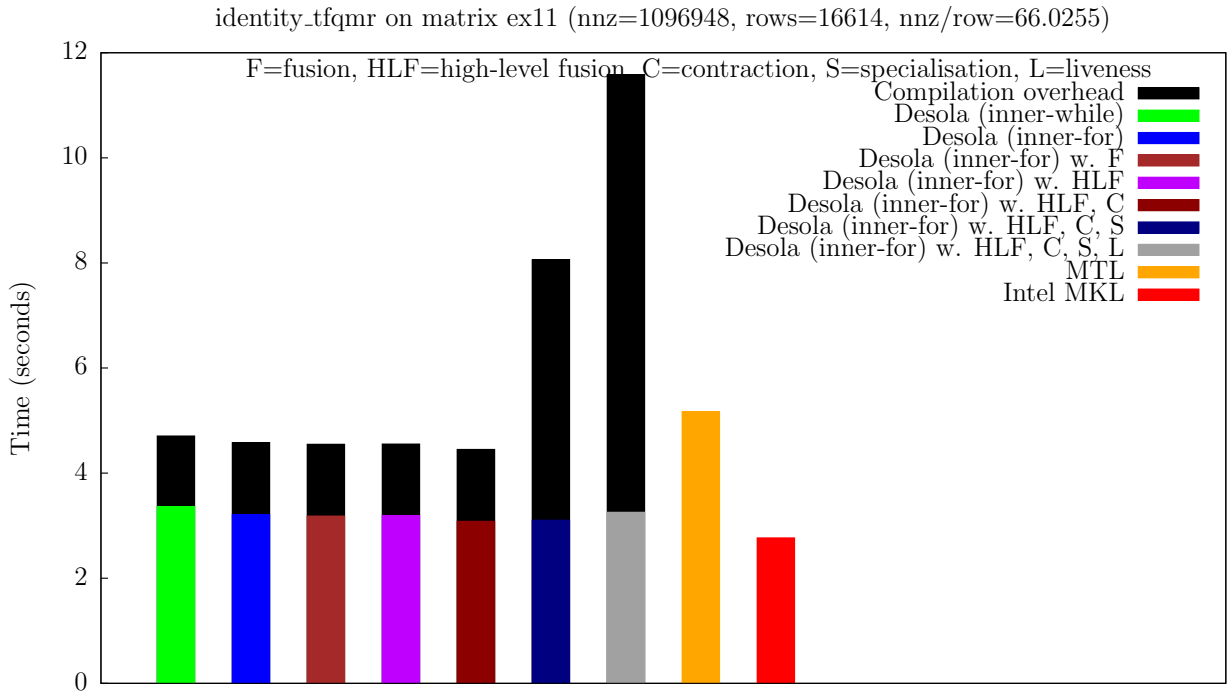


(a) architecture 1

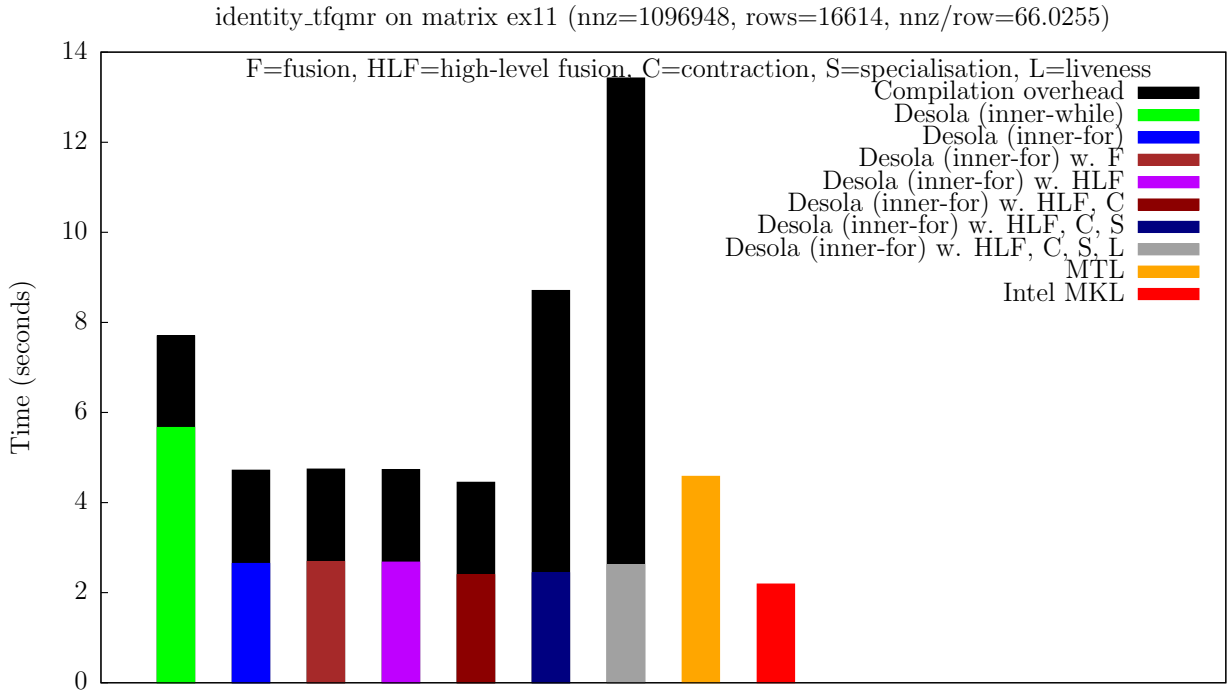


(b) architecture 2

Figure B.26: Time to execute 256 iterations of the Transpose-Free Quasi-Minimal Residual solver with matrix rajat31 on our test architectures.

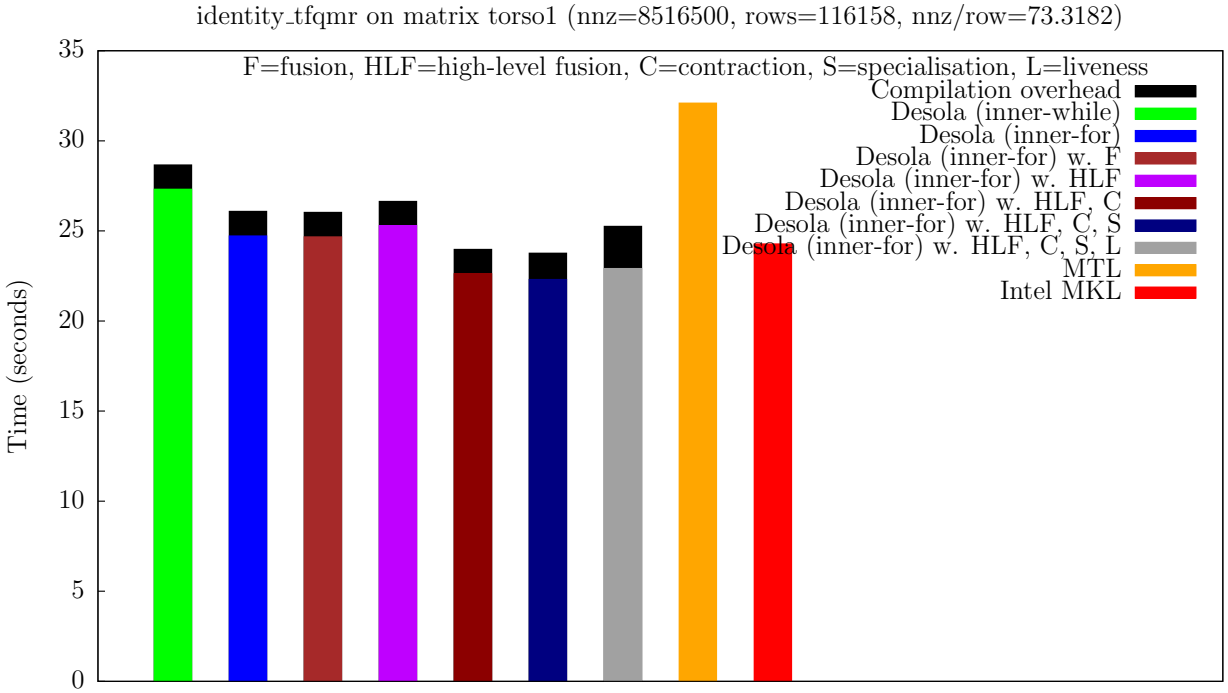


(a) architecture 1

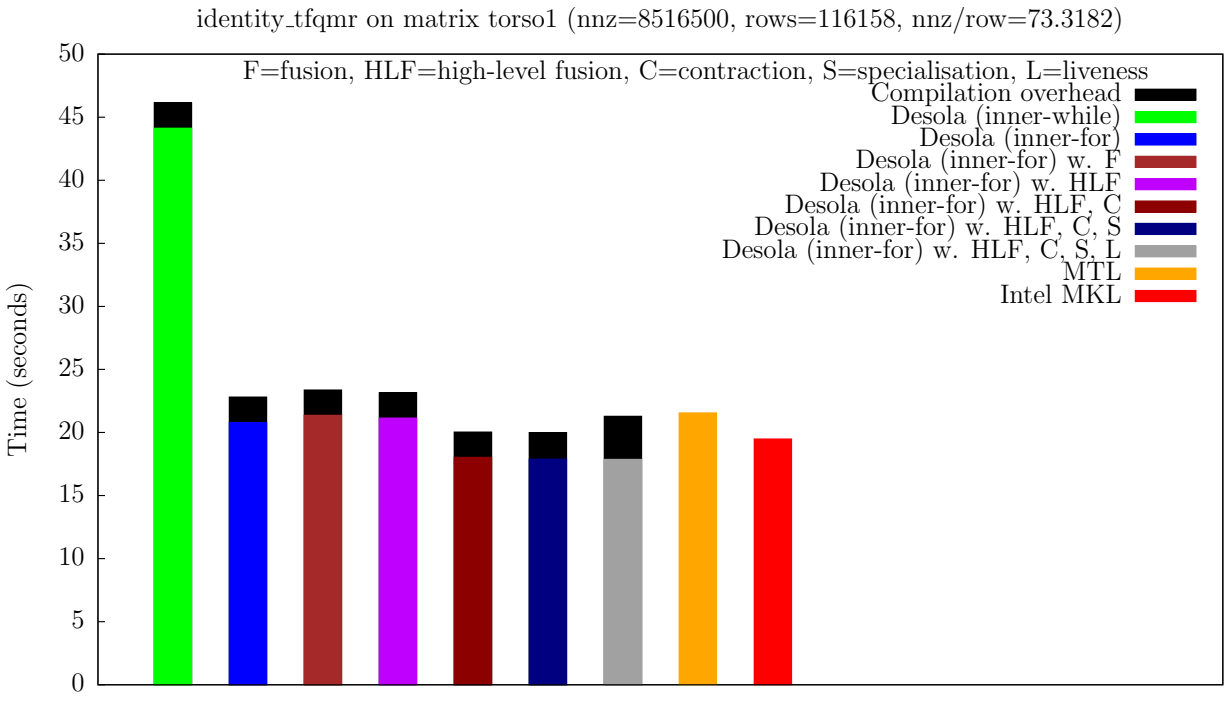


(b) architecture 2

Figure B.27: Time to execute 256 iterations of the Transpose-Free Quasi-Minimal Residual solver with matrix ex11 on our test architectures.

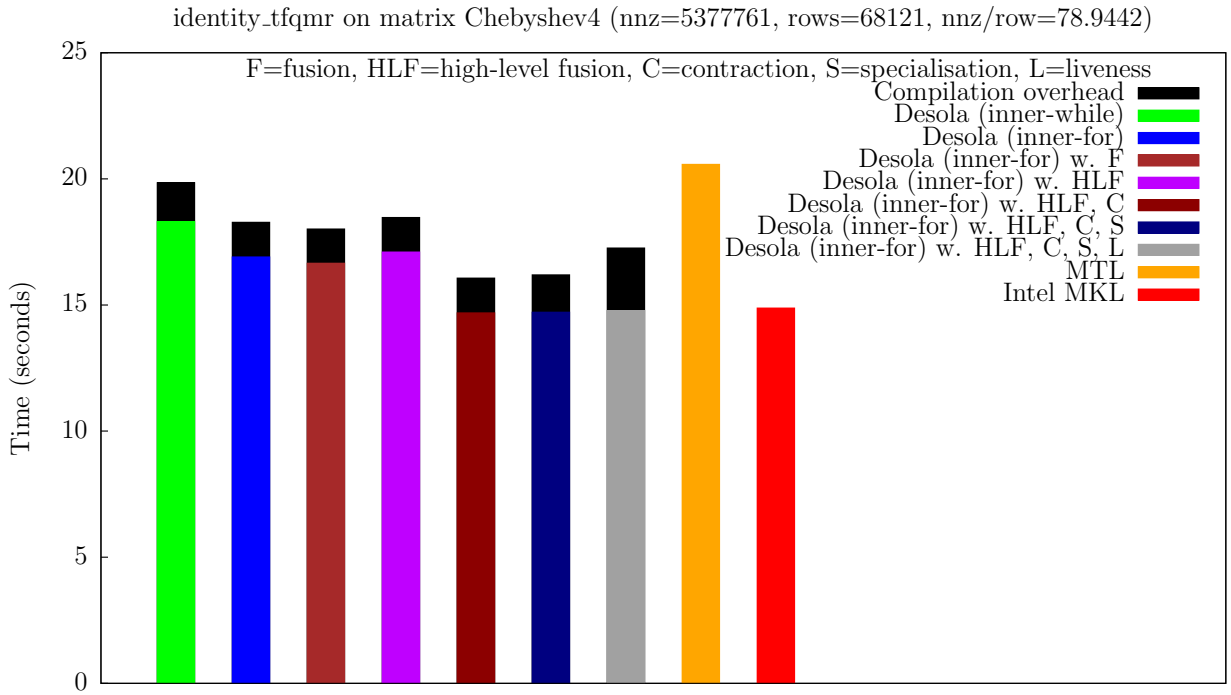


(a) architecture 1

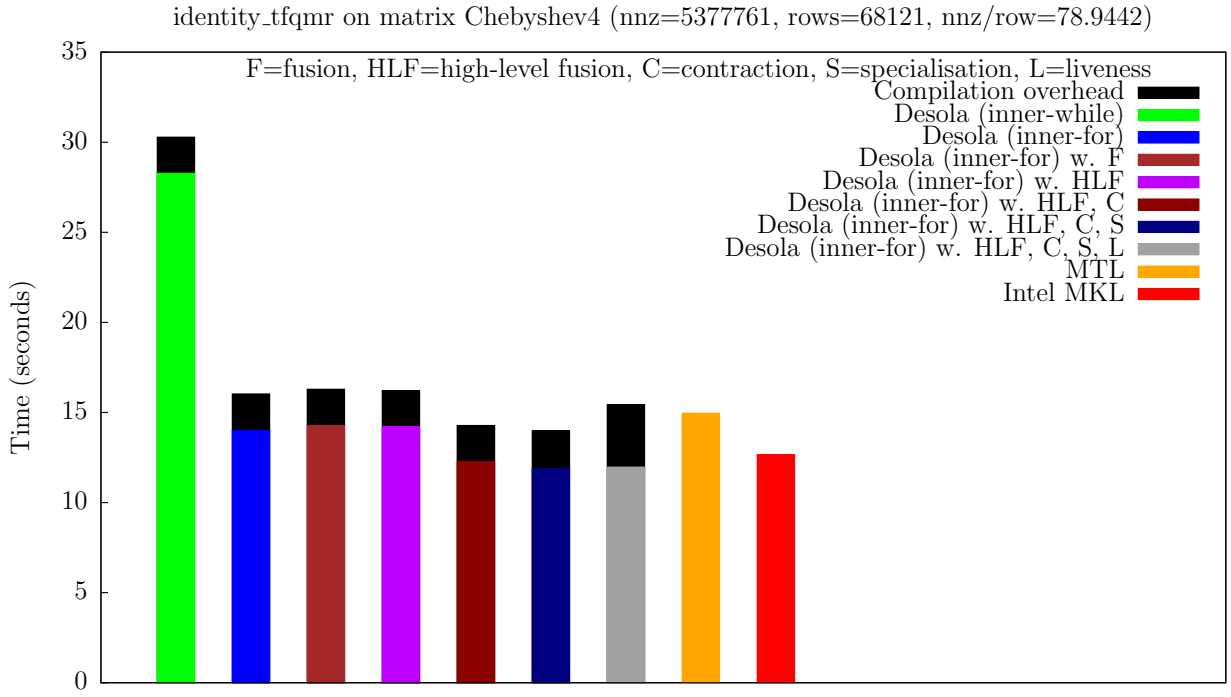


(b) architecture 2

Figure B.28: Time to execute 256 iterations of the Transpose-Free Quasi-Minimal Residual solver with matrix torso1 on our test architectures.

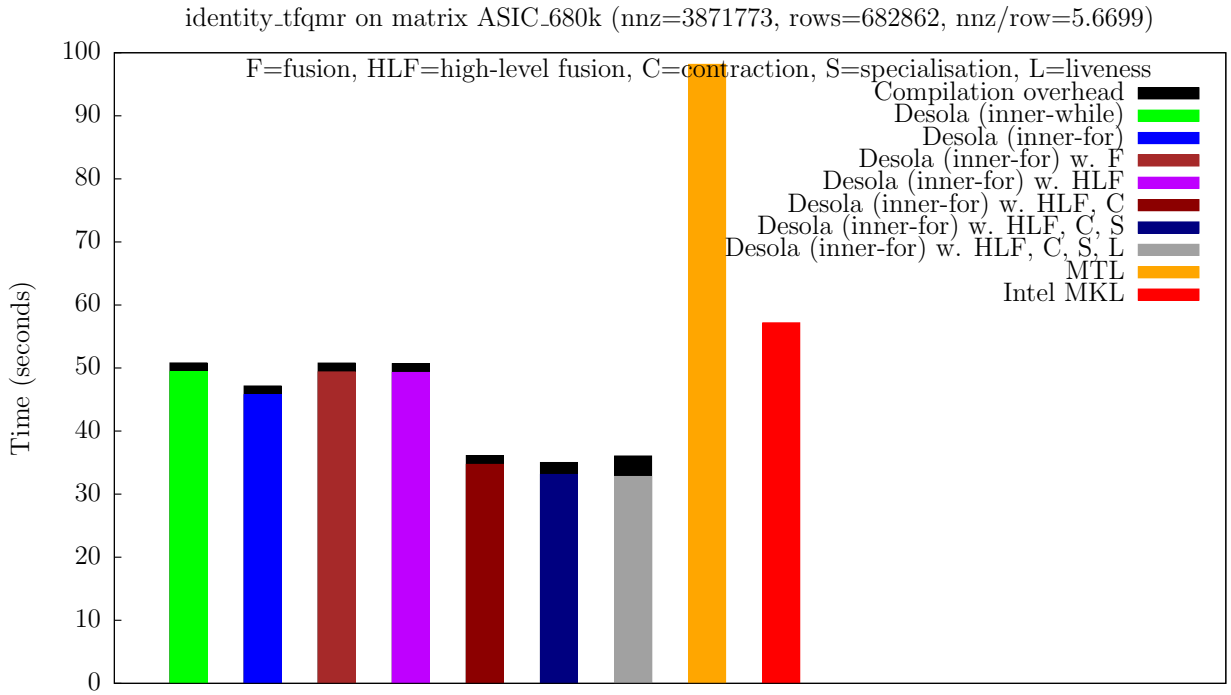


(a) architecture 1

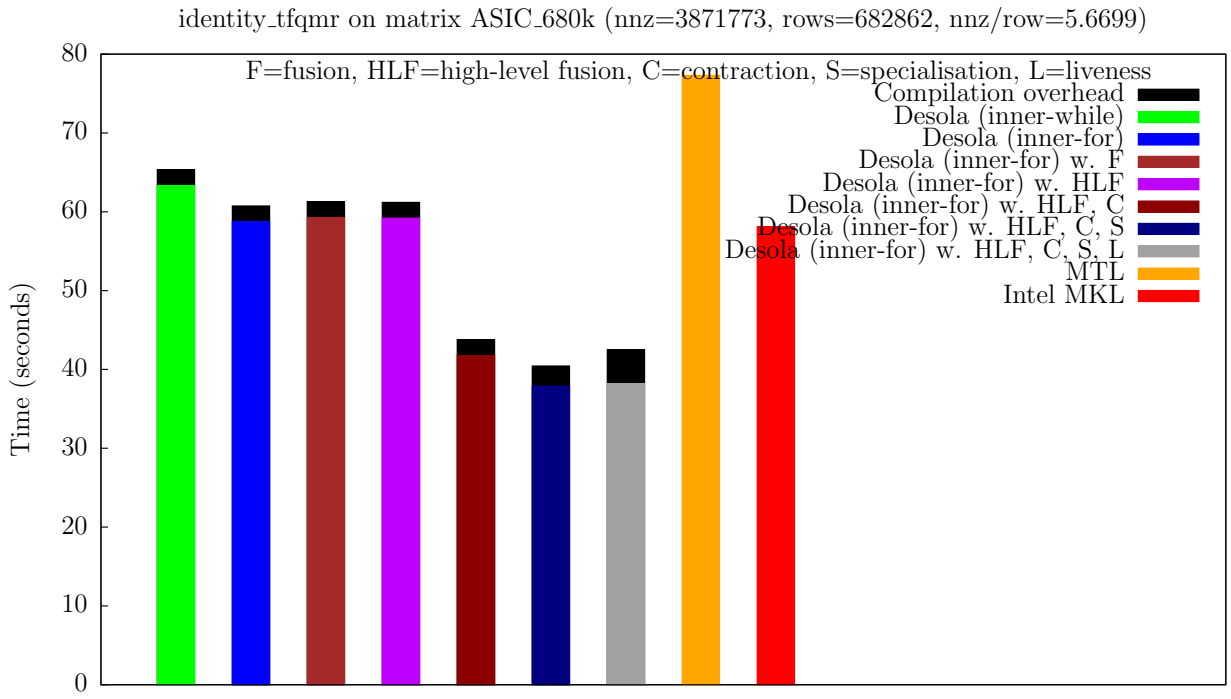


(b) architecture 2

Figure B.29: Time to execute 256 iterations of the Transpose-Free Quasi-Minimal Residual solver with matrix Chebyshev4 on our test architectures.



(a) architecture 1



(b) architecture 2

Figure B.30: Time to execute 256 iterations of the Transpose-Free Quasi-Minimal Residual solver with matrix ASIC_680k on our test architectures.

Appendix C

Excafé Library Design Notes

C.1 Introduction

One aim of my thesis is to explore the optimisation space present in the implementation of finite element solvers. In order to do this, I decided to construct a library for expressing and solving partial differential equations using the finite element method.

My work with Desola demonstrated the importance of being able to optimise components within a given context. This is especially important when providing usable abstractions to the user as these abstractions tend to expose elemental operations which cannot be optimised effectively in isolation.

My library has the following design goals:

- Capture as much information as possible about the steps involved in specifying and solving a finite element problem.
- Provide abstractions to the library user that correspond naturally to the abstractions used in specifying and solving finite-element problems in the mathematical domain.

In this context, *capture* has an extremely specific meaning. If a library provides facilities to express a computation using domain-specific abstractions, one could consider the essence of

that computation to have been captured in that code. However, the computation itself may still be opaque to that library. When we *capture* an expression, we mean that the computation has been exposed to the library in such as way that it has the ability to analyse and optimise that expression.

I will describe the various considerations weighed during the design of the library. Unsurprisingly, there is considerable overlap between these considerations and those involved in the development of other finite element libraries. However, the fact we want to capture certain operations rather than simply execute them leads to differing requirements. For a discussion of the requirements for data structures for hp adaptive finite element methods we refer the reader to Bangerth et al. [60]. For my investigation, I do not consider adaptivity, but other aspects of the analysis are relevant.

C.2 Problem Context

For our captured representation, it is important that we have a data structure that stores or contains a reference to all information about the finite element problem being solved. This structure should provide mechanisms to access all aspects of the finite element problem being modelled, and provides an appropriate place to store results of analyses and optimisations.

Our problem context should contain:

A mesh This is the mesh on which all our partial differential equations will be spatially discretised.

Finite element function spaces Our discretised finite element function spaces consist of a bounded subset of our mesh and a choice of basis functions.

Discretised tensor fields Discretised representations of tensor fields we use (e.g. a scalar pressure field). We also wish to be able to represent composite fields (i.e. fields composed from sub-fields). These are useful, for example, when solving for pressure and velocity simultaneously in the coupled Navier-Stokes equations.

Position field The position field defines mapping from a co-ordinate specified in terms of a local position on a cell to a global co-ordinate. This transform itself can be described as a vector-field defined over our mesh and we can use our finite element basis functions to define it.

Boundary conditions For our problem to be well-posed, it is often necessary to provide boundary conditions. These are typically described as a subset of the boundary of our domain and a function specifying the value a field should take on the boundary.

Field update operations These refer to any operations that update the value of our discretised fields. In a time-dependent simulation, these would typically be the operations required to calculate the value of our fields at a new time-step. However, we may wish to capture operations required for setting up the initial fields, or multiple techniques for advancing our system.

Capturing field update operations will involve the capture of:

1. Linear and bilinear forms used to assemble discrete fields and operators.
2. Linear algebra between discretised fields and operators (represented as vectors and matrices).
3. Iteration required by the solution step. For example, in the Navier-Stokes equations, Picard iteration can be used to linearise the convective acceleration term.

C.3 Mesh Representation

We identify the following requirements for our mesh representation. None of these are particular to our expression capture. In this context, the term *mesh entity* refers to elements of our mesh such as cells, facets, vertices or edges.

Global mesh entity iteration We frequently need to iterate over all entities of a particular type in our mesh. For example, iteration over all cells or facets in a mesh to evaluate cell

and facet integrals for global assembly. When we render a mesh, we require the ability to iterate over every vertex in our mesh.

Local mesh entity iteration We also require the ability to iterate over all mesh entities adjacent to a given mesh entity. For example, when we define a continuous field, we need to determine which cells share degrees-of-freedom. If we associate degrees-of-freedom with particular mesh entities, we can work out which cells share them by iterating over all cells adjacent to that entity.

Logg describes a representation for meshes that supports efficient implementation of these abstractions [61].

C.4 Finite Element

The finite element representation is the most important part of our design. All fields used in our solver will be discretised as coefficients of basis functions defined by the finite element.

We identify the following requirements for our finite element class.

True tensor-valued elements We observe that in many instances, vector and tensor-valued basis functions have been built by duplicating scalar basis functions for each tensor element. Restricting basis functions to this form enable certain sparsity related optimisations when evaluating local assembly matrices.

However, we wish to be able to support arbitrary vector and tensor valued elements such as the Raviart-Thomas [48] element.

Expression-capture of basis functions We wish to be able to perform expression capture of basis functions. In a conventional finite element library, this is impossible. Typically, the library will only support evaluating the basis function and derivatives at specified points.

As we wish to optimise local assembly and evaluation of these functions, it is important that we can analyse the underlying formulae used to compute the basis function values.

Shared degree-of-freedom resolution If our basis functions are used to define a continuous field, then the degrees-of-freedom that are co-efficients for our basis functions will be shared between cells. Given local degrees-of-freedom specified on adjacent cells, our finite element class must be able to determine which degrees of freedom are shared.

Degree-of-freedom location querying Boundary conditions constrain the degrees of freedom of a field on our mesh. As we only wish to constrain values at the edge of our domain, we require a mechanism to determine which degrees-of-freedom on a cell affect values of the field at that edge and not constrain all degrees-of-freedom on that cell.

C.5 Reference Cell

Many aspects of the finite element can be defined in terms of operations over a reference cell, and then generalised to the global case. For example, basis functions are typically defined in terms of co-ordinates on the reference cell.

Our reference cell consists of:

1. A list of vertices, with co-ordinate values usually defined in the range 0 to 1 or -1 to 1 so that defining basis function over the reference cell is as simple as possible.

2. A local numbering scheme for all mesh entities in the cell.

We identify the following requirements for our reference cell.

Global-to-local entity lookup We require a mechanism to map between global mesh entity numberings and our local one. For example, to determine the degrees-of-freedom on a particular facet, we need to:

1. Determine the cell(s) that contain the particular facet.
2. Use the reference cell to resolve the global facet identifier to a local facet identifier.
3. Query the finite element for the local numbering of the degrees-of-freedom located on that facet.

Geometry queries We need to be able to query the reference cell about the co-ordinates of its vertices. For example, to evaluate a field at a given mesh vertex, we need to:

1. Determine a cell that contains that vertex.
2. Determine the local identifier i of the vertex on that cell.
3. Query the reference cell for the local co-ordinate of vertex i.
4. Evaluate the field using the containing cell, and the local co-ordinate on that cell.

Topology queries We need a mechanism to query the reference cell about the number of mesh entities it has (e.g. number of vertices, edges) and the vertices contained in these entities. Using this information, it is possible for the mesh class to identify all mesh entities on a cell in a general manner. Hence, to add a cell to a mesh we need only supply the cell vertices and the mesh class will use our reference cell abstraction to identify the edges, facets and so on.

Co-ordinate field As mentioned before, the global co-ordinates of a position in our mesh can itself be defined as a vector field over our mesh. Unless we are using curvilinear cells, this can be represented as a linear interpolation between the cell vertices. Hence, our reference cell can provide a set of linear scalar basis functions that are multiplied by the cell vertices to form a position vector-field over the element.

Quadrature rules We need to be able to evaluate integrals over our reference element. Integrals over our reference element can then be transformed to integrals over an arbitrary element using the Jacobian of the position field. Given the polynomial order of the function we wish to integrate, our reference cell must be able to provide a quadrature rule to evaluate that integral. A quadrature rule consists of a number of points within, or on the

boundary of, the reference element to evaluate the function to be integrated and scaling factors used to compute a weighted sum.

Reference region transform Instead of using quadrature rules specific to a particular cell, it is possible to define quadrature only over the reference line, square or cube. In order to integrate over cells with different topologies (e.g. triangles) our reference cell can provide a transform from the reference region to local cell co-ordinates. Hence, to evaluate our cell integrals we have to handle two co-ordinate transformations, one from the reference region to our reference cell, and one from our reference cell to the arbitrary cell on our mesh.

Facet descriptions When handling boundary conditions, we need evaluate integrals defined over the facets of our cells. The description of cell facets is represented as a variable substitution.

For example, on a triangular mesh, our position field defines a function to calculate the global position (gx, gy) from a local co-ordinate (x, y). Given a local edge identifier and a new variable r, we can define a variable substitution such that both x and y are defined in terms of r. The value of r describes a position along that edge.

C.6 Local-to-global mapping

The local-to-global mapping is an important part of our discretisation. Each cell of our mesh has degrees-of-freedom associated with it. When modelling a continuous field, cells will share certain degrees-of-freedom. The local-to-global map provides a mapping from the degrees-of-freedom as defined locally to a cell to a global degree-of-freedom identifier.

C.7 Discretised fields and operators

Our solution steps involve performing operations between discretised fields and operators. Discretised fields are typically represented by vectors, and discretised operators by sparse matrices. In general, most of the operations we want to perform can be represented by linear algebra operations, however, we want to impose additional requirements so that those operations are valid. We do this by associating our matrices and vectors with the local-to-global mapping used for their column and row entries.

We require the following operations involving fields and operators:

Operator assembly The assembly of the discrete operator using a global matrix approach requires that integrals are evaluated over every cell in the domain and added to our matrix. This corresponds to a standard sub-matrix addition, but we require our local-to-global mapping to determine which rows and columns each cell contribution needs to be added to.

Field addition Field addition is equivalent to a vector addition provided that the fields have the same local-to-global mapping.

Operator application Operators transform a field defined in one function space to one in another space. In the finite element method, this is usually a transformation from the trial space to the test space. Implementation is usually equivalent to a matrix-vector multiply. It is valid so long as the function space for the field being transformed is the same function space the operator transforms.

Operator solve Given an operator and a field defined in function space the operator transforms to, we wish to find the source field. Implementation usually corresponds to the solution of a sparse linear system. However, we also require a way to apply Dirichlet boundary conditions during the solution process.

Sub-field extraction and assignment We require the ability to extract and assign subfields. For example, after solving the coupled Navier-Stokes equations, we often want

to extract the velocity and pressure components of the resulting composite field. To do this requires use of the local-to-global mapping to determine which entries to extract and how to arrange them.

C.8 Discretised operation description and implicit-loop syntax

As operations between discretised fields and operators resemble linear algebra, it makes sense for the library client to describe them in a similar fashion. However, one major barrier to this is the requirement for iteration.

We take the example of the convective acceleration term in the Navier-Stokes equations. This term is non-linear, and must be linearised if we are to use a linear solver to solve the resulting system. Picard iteration is one such technique for doing so.

Picard iteration requires that we iterate until our linearised term meets some convergence criteria. Rather than have the library client explicitly declare a loop, it is preferable that the library client can specify this iteration in a manner similar to subscript notation.

C.9 Form description

The assembly of discrete operators and fields requires that they be expressed using bilinear and linear forms, respectively. These are integrals over our cells, expressed in terms of our basis functions.

These forms typically have an extremely succinct mathematical expression and we wish to diverge from it as little as possible. The Unified Form Language describes a syntax for expressing variational forms that can be embedded in Python. We base our form description on UFL.

C.10 Expression representation

We wish to analyse and optimise the representation of local matrix (and vector) assembly. A complete description of the local assembly matrix combines a large number of expressions and values.

- Descriptions of the basis functions and derivatives used in the bilinear form, position field, and any discretised fields referenced from the bilinear form.
- The inverse of the Jacobian of the position field, used to transform gradients from the reference cell to a general cell.
- The determinant of the Jacobian of the position field, used in the calculation of the inverse Jacobian and as a scaling factor in the cell integral.
- Coefficients for the basis functions of any referenced discretised fields.
- Quadrature points, at which the basis functions are evaluated. These may be unnecessary if the local assembly matrix can be integrated symbolically.
- Tensor products, which are used to combine almost all operands.

For our investigation, we assume that all basis functions can be represented by polynomials. Our captured expressions will involve unknown cell vertices and discretised field coefficients so our expressions will be multivariate. Also, due to the presence of the inverse Jacobian in the local assembly matrix description, our expression representation will need to be able to handle rational functions, rather than plain polynomials.

Our representation should also preserve the tensor product structure of the original computation if possible.

Appendix D

Excafé Architecture

In this appendix, we provide additional information about Excafé's data structures. We cover the mesh related types in Section D.1 followed by the types Excafé uses to represent discretised values in Section D.2. We then describe the types related to Excafé's expression capture of discretised expressions, variational forms and scalar expressions in Sections D.3, D.4 and D.5, respectively.

D.1 Geometry-related types

We present a UML diagram of Excafé's mesh related classes in Figure D.1. Our mesh representation is based on the data structures described by Logg [61].

D.1.1 *MeshGeometry* class

The `MeshGeometry` class is responsible for storing the physical locations of vertices in the mesh. As our vertices are contiguously numbered from 0, the mapping from vertex IDs to vertices can be implemented using single array of vertices. The `MeshGeometry` class is templated by the dimension of the mesh.

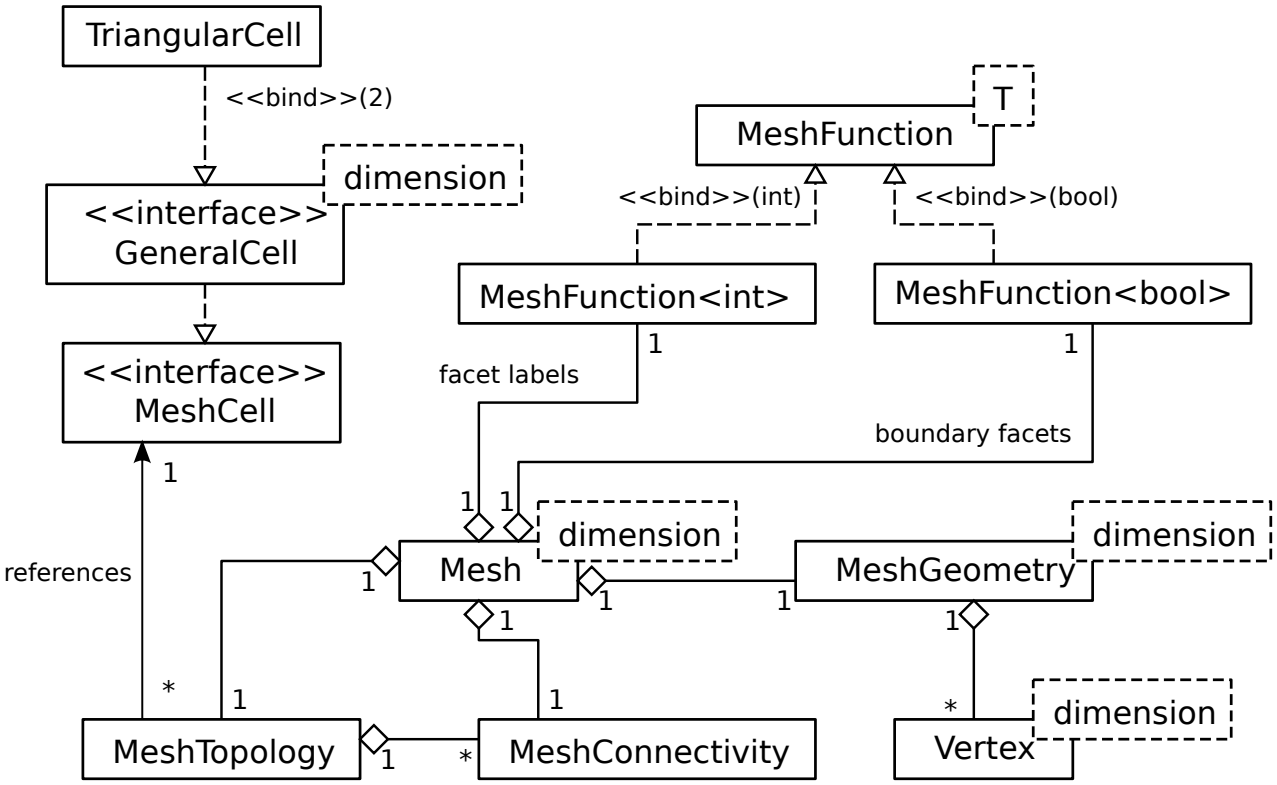


Figure D.1: A UML class diagram of EXCAFÉ's geometry-related classes.

D.1.2 *MeshConnectivity* class

The `MeshConnectivity` class stores the adjacency relationship between all topological entities of the mesh of two specified dimensions (e.g. edges and cells). Its implementation closely resembles a compressed sparse row (CSR) matrix representation.

D.1.3 *MeshTopology* class

The `MeshTopology` class permits retrieval of any topological adjacency relationship of the mesh. For a mesh of dimension n, there are $(n+1)^2$ possible adjacency relationships. When an adjacency relationship is requested, it may either return a reference to a precomputed `MeshConnectivity` object or build it on-demand. In order to compute an arbitrary adjacency relationship, the `MeshTopology` class must be supplied with an initial cell-to-vertex adjacency relationship, and a `MeshCell` instance describing the local topology of the chosen cell type.

D.1.4 *MeshFunction* class

The `MeshFunction` class associates values with entities of the mesh of a given dimension. It is templated by the type of the value it associates. A `MeshFunction<bool>` is used to store whether or not a facet is on a boundary. Also, a `MeshFunction<int>` is used to store facet labellings for applying boundary conditions.

D.1.5 *MeshCell* interface

The `MeshCell` interface provides a mechanism used by the `MeshTopology` class to query a topological description of the reference cell. This information enables the `MeshTopology` class to compute topological adjacency relationships without being hard coded to a specific cell-type (e.g. rectangular, tetrahedral). For further details, consult Logg [61].

D.1.6 *GeneralCell* interface

This is the complete interface for all types that represent a reference cell. It is templated by the dimension of the cell. It provides methods for accessing local cell vertex co-ordinates, generating quadrature rules and retrieving a symbolic description of local-to-global cell co-ordinate transformation.

D.1.7 *Mesh* class

The `Mesh` class contains both geometric and topological information, held by instances of the `MeshGeometry` and `MeshTopology` classes, respectively. The mesh also contains a `MeshConnectivity` instance which describes which vertices are located within each cell. This relationship, combined with the information provided through the `MeshCell` interface, provides sufficient information to derive all other topological adjacency relationships.

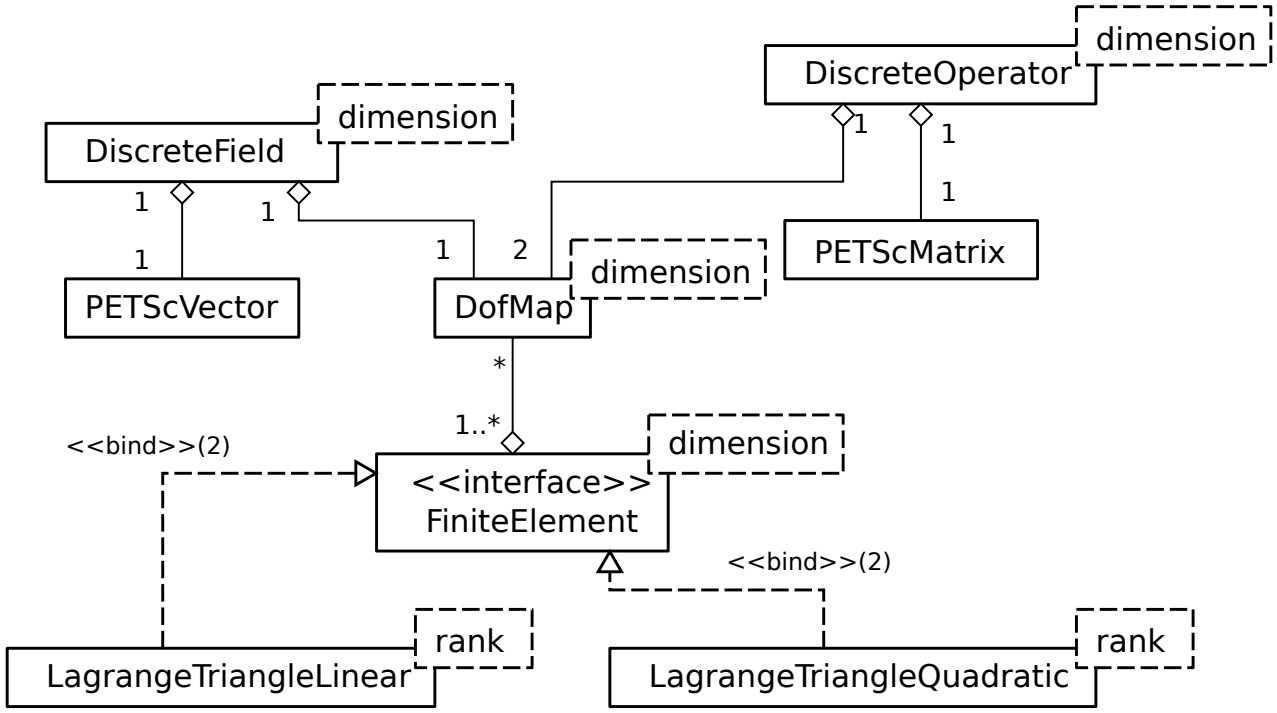


Figure D.2: A UML class diagram of EXCAFÉ's classes used to store discretised fields, operators and associated information.

The `Mesh` class also contains two `MeshFunction` instances that associate values with the facets of the mesh. One is a boolean function that is true only for facets that are located on the mesh boundary. The other is an integer valued function that returns the labels assigned to the mesh facets by the mesh generator.

D.2 Discretisation-related types

We present a UML diagram of the classes used by EXCAFÉ to handle finite element discretisation in Figure D.2.

D.2.1 *FiniteElement* interface

The `FiniteElement` interface is implemented by classes that represent a particular finite element discretisation. It provides methods to retrieve symbolic representations of the basis functions, information about the degrees of freedom located on the reference cell and deter-

mine which degrees of freedom are shared between neighbouring cells. Two implementations of this interface exist in Excafé, `LagrangeTriangleLinear` and `LagrangeTriangleQuadratic` which provide continuous linear and quadratic bases, respectively, over the reference triangle.

D.2.2 *DofMap* class

The `DofMap` class represents the global numbering of a discretised field's degrees of freedom. A degree of freedom local to a cell is defined as a tuple of the cell ID, `FiniteElement` and a local index. This tuple is mapped to a single number which is used to index rows and columns in discretised fields and vectors. This also expresses which degrees of freedom are shared between neighbouring cells. The `DofMap` can reference multiple `FiniteElement` instances, permitting it to represent composite fields (e.g. a combined velocity and pressure field).

D.2.3 *PETScVector* and *PETScMatrix* classes

These classes wrap the C PETSc [47] vector and matrix application programming interface (API) and provide a more object-oriented interface.

D.2.4 *DiscreteField* and *DiscreteOperator* classes

The `DiscreteField` and `DiscreteOperator` classes are used to hold discretised representations of fields and operators, respectively. The `DiscreteField` class holds a vector containing the discrete field coefficients and a `DofMap` which describes the discretisation. Similarly, the `DiscreteOperator` class holds a sparse matrix and references to two `DofMap` instances, which describe the discretisation of the operand and result of the operator.

Since the `DiscreteField` and `DiscreteOperator` classes hold information about their discretisations, they can detect when they are used inappropriately. The `DiscreteOperator` class also contains functionality that enables it to perform local and global assembly when provided with a bilinear form and a mesh.

D.3 Discrete Expression Capture

We show a UML class diagram of the types used to perform expression capture of discretised expressions in Figure D.3.

D.3.1 *Field*, *Operator* and *Scalar* handle classes

The `Field`, `Operator` and `Scalar` classes provide handles to the DAG built by the EXCAFÉ client representing the result of a sequence of operations involving discretised fields, discretised operators and scalars.

D.3.2 *IndexedField*, *IndexedOperator* and *IndexedScalar* handle classes

The `IndexedField`, `IndexedOperator` and `IndexedScalar` classes provide handles to indexed fields, operators and scalars, respectively, used by the EXCAFÉ client. All three handle classes are implemented by the `IndexedHolder` templated class, instantiated with different template parameters.

Each handle holds a reference to a `IndexableValue` instance, again instantiated with different template parameters, dependent on the type of the value. The `IndexableValue` is responsible for maintaining references to all expressions assigned to the indexed value, and also for maintaining a reference to the `TemporalIndexValue` corresponding to the variable it is indexed by.

Within the discrete expression DAG, references to an indexed field, operator or scalar are represented by the `DiscreteIndexedField`, `DiscreteIndexedOperator` and `DiscreteIndexedScalar` classes respectively.

D.3.3 Function space related classes

EXCAFÉ has a very basic syntax for the manipulation of function spaces. The handle type

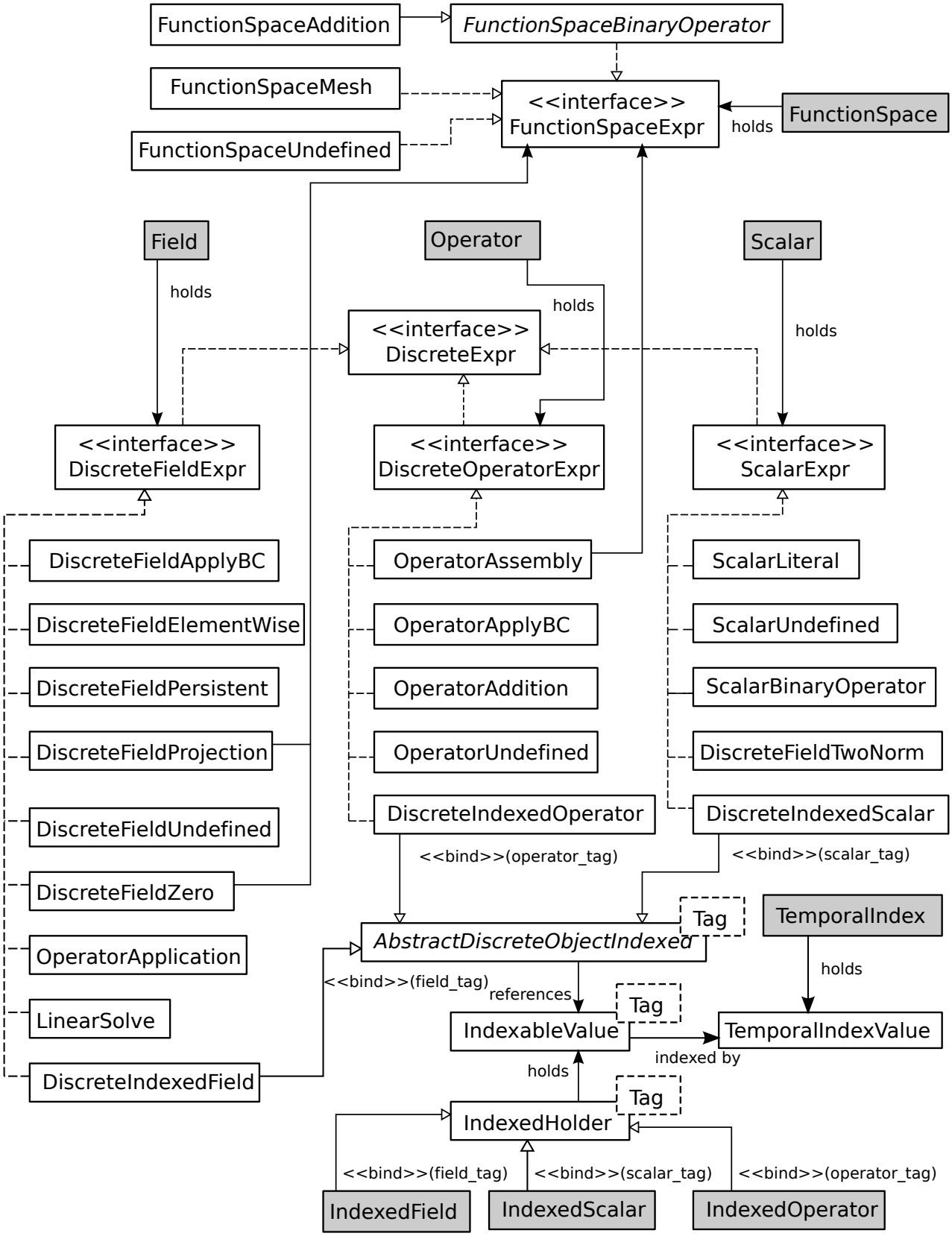


Figure D.3: A UML class diagram of the Excafé types related to capturing a description of operations performs between discretised fields, discretised operators and scalars. The handle types held by the Excafé client have been shaded.

`FunctionSpace` is exposed to the EXCAFÉ client. A function space DAG can currently represent the construction of of function space for a single `FiniteElement` over the entire mesh, and the addition of these spaces. This is sufficient for constructing a function space to represent a composite field.

The `FunctionSpaceUndefined` type is used to detect incorrect use of the handle. Further development of this syntax will allow subtraction and construction over mesh subsets (such as the boundaries) to provide better support for boundary conditions.

D.3.4 The Discrete Expression DAG Classes

All classes that represent nodes in the discrete expression DAG implement the `DiscreteExpr` interface. The extended interfaces `DiscreteFieldExpr`, `DiscreteOperatorExpr` and `ScalarExpr` are implemented by nodes representing fields, operators and scalars, respectively.

The `DiscreteExpr` contains methods to enable visitor based traversal and determination of which index variables are associated with a node. The extended interfaces allow determination of properties such which function space the node's value is in.

We briefly describe the purpose of each node type. The `ScalarUndefined`, `DiscreteFieldUndefined` and `DiscreteOperatorUndefined` nodes are used to detect incorrect handle usage. The other discrete field valued nodes have the following purpose:

DiscreteFieldApplyBC Represents the application of Dirichlet boundary conditions to a field.

DiscreteFieldElementWise Represents addition and subtraction of identically discretised fields.

DiscreteFieldPersistent Represents a reference to a discretised field that persists across different solution steps of the problem being solved (e.g. the velocity in a fluid simulation).

DiscreteFieldProjection Used to project fields onto a superset or subset function space.

DiscreteFieldZero Used to construct a zero-valued field in a specific function space.

OperatorApplication Represents the result of the application of a discrete operator to some field.

LinearSolve Represents the solution field from a linear solve.

DiscreteIndexedField Represents a reference to an indexed field.

The operator valued nodes have the following purpose:

OperatorAssembly Represents the construction of an operator from a set of integrals of bilinear forms.

OperatorApplyBC Represents the application of Dirichlet boundary conditions to an operator.

OperatorAddition Represents the addition of two identically discretised operators. This is not currently used in our implementation and was provided to enable experimentation with reusing previously assembled terms.

DiscreteIndexedOperator Represents a reference to an indexed operator.

The scalar valued nodes have the following purpose:

ScalarLiteral A constant scalar value.

DiscreteFieldTwoNorm Represents the l^2 norm of a discretised field.

ScalarBinaryOperator Represents a binary operator between two scalars such as addition or comparison.

DiscreteIndexedScalar Represents a reference to an indexed scalar.

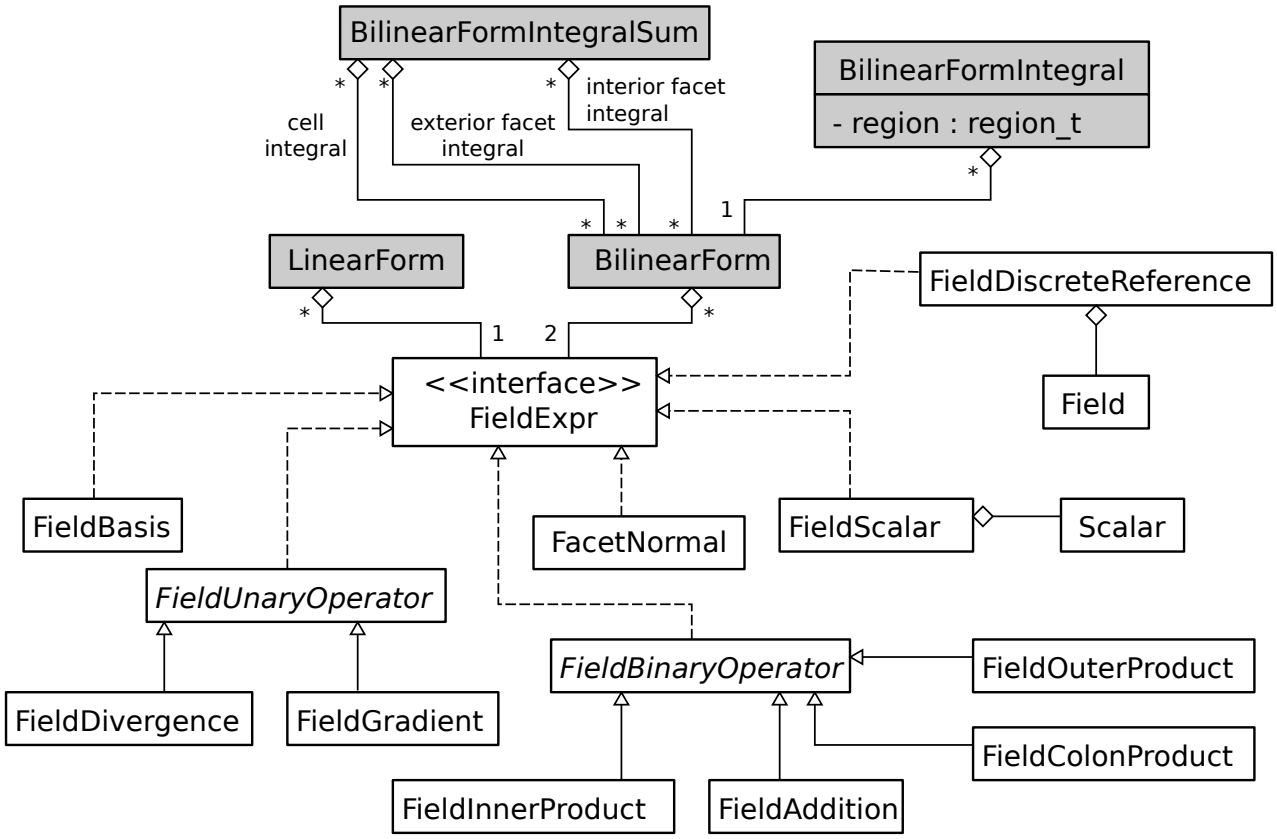


Figure D.4: A UML class diagram of the EXCAFÉ types related to capturing a description of variational forms used to perform assembly. The handle types held by the EXCAFÉ client have been shaded.

D.4 Variational Form Capture

We show a UML class diagram of the types used to perform expression capture of variational forms in Figure D.4.

D.4.1 Handle Types

We provide the handle types `LinearForm` and `BilinearForm` to represent expressions that compute scalar valued expressions (given a particular trial and/or test function). The `BilinearFormIntegral` type represents bilinear form integrated over a specific mesh region. The `BilinearFormIntegralSum` is used to represent a sum of bilinear forms, which may be integrated over the entire mesh, internal mesh facets or boundary facets.

D.4.2 Form Expression DAG Types

All nodes types that form the variational form expression DAG represent tensor-valued expressions and implement the `FieldExpr` interface. We briefly describe the purpose of each node type:

FieldDiscreteReference Hold a reference to a field defined somewhere in the discrete expression DAG.

FieldScalar Holds a reference to a scalar value defined somewhere in the discrete expression DAG. Can be considered a scalar field that takes a constant value over the entire mesh.

FacetNormal Represents the outward facing normal of a mesh facet. This node can only be used in forms integrated over interior or exterior mesh facets.

FieldOuterProduct The outer product of two tensor fields.

FieldInnerProduct The inner product of two tensor fields.

FieldColonProduct The colon product of two tensor fields.

FieldAddition The addition of two identical-rank tensor fields.

FieldGradient The gradient of a tensor field.

FieldDivergence The divergence of a tensor field.

FieldBasis A place-holder that refers to a basis function defined over the local cell. The same form will be evaluated with different choices of basis function during local assembly.

D.5 Scalar Expression Representation

Excafé currently has three mechanisms for representing scalar expressions. Each expression class is templated by the type of unknown that will be used in the expressions. Their relationships are shown in Figure D.5.

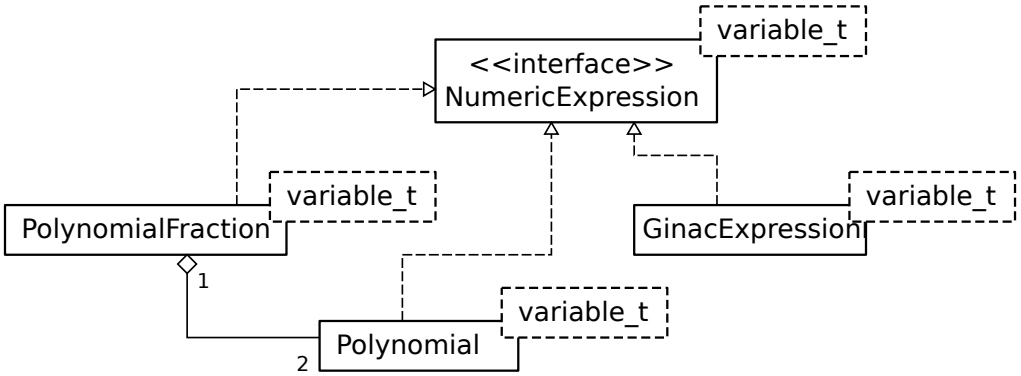


Figure D.5: A class diagram of Excafé's different scalar expression representations.

We briefly describe each one:

Polynomial This type represents a polynomial expression stored in expanded polynomial form (i.e. a sum of monomials). This format is helpful for representing expressions before performing our common subexpression elimination pass.

PolynomialFraction This type represents a rational function as a quotient between two `Polynomial` instances. It is currently not ideal for our purposes since addition and multiplication between expressions in this form may cause them to grow in complexity and obscure simplifications.

GinacExpression This type is a wrapper for expressions handled using the GiNaC [45] symbolic manipulation library. GiNaC maintains a tree of expressions that can be evaluated (or partially evaluated) on demand as well as providing algorithms such as greatest common divisor (GCD) that can be used to simplify rational expressions.

When provided with a mapping of unknowns to scalar values, all expression classes can evaluate their values. All expression classes also implement the `NumericExpression<V>` interface. The interfaces provides a representation-independent visitation mechanism that enables conversion between expression types.

We also use our expression classes to represent the elements of our local assembly matrix during construction. To do this, we instantiate our chosen expression class with a variable

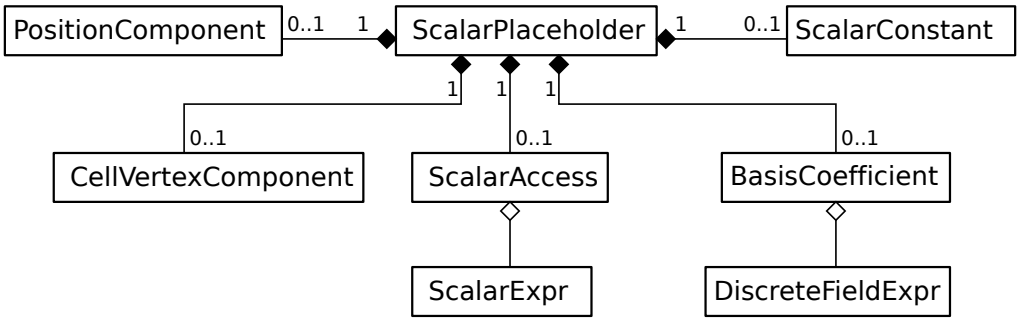


Figure D.6: A class diagram of the variable type Excafé uses to represent unknowns in its symbolic representation of a local assembly matrix.

type, `ScalarPlaceholder`, that represents the possible unknowns in a local assembly matrix. We show the class diagram of this type in Figure D.6.

The value describing the unknown is stored inside the `ScalarPlaceholder` class so that `ScalarPlaceholder` can be treated as a standard value type. We describe each of the types that can be held inside a `ScalarPlaceholder`

PositionComponent Represents the components of the position variable on the local cell. It occurs during construction of our bilinear forms, but is removed when we symbolically integrate (as these are the variables we integrate over).

CellVertexComponent This represents a component of the global position of one of the vertices of the local cell. It may be removed from Excafé when the vertex locations are instead represented by a vector field.

ScalarAccess This represents a scalar value referenced from the discrete expression DAG.

BasisCoefficient This represents a basis function coefficient that is defined on the local cell. It is this type that allows us to incorporate fields from previous time-steps by referencing fields from the discrete expression DAG.

ScalarConstant A scalar constant. Variables of this type are opaque to the expression classes and therefore do not get multiplied out. This may be useful in order to maintain more information about the structure of expressions.

Appendix E

Excafé Heat Solver Source

```
#include <cstddef>
#include <sstream>
#include <boost/format.hpp>
#include <simple_cfd/capture/scenario.hpp>
#include <simple_cfd/capture/solve_operation.hpp>
#include <simple_cfd/petsc_manager.hpp>
#include <simple_cfd/triangular_mesh_builder.hpp>
#include <simple_cfd/lagrange_triangle_linear.hpp>
#include <simple_cfd/lagrange_triangle_quadratic.hpp>
#include <simple_cfd/capture/scenario.hpp>
#include <simple_cfd/capture/fields/fields.hpp>
#include <simple_cfd/capture/forms/forms.hpp>
#include <simple_cfd/mesh.hpp>
#include <simple_cfd/exception.hpp>
#include <simple_cfd/boundary_condition_list.hpp>
#include <simple_cfd/boundary_condition_trivial.hpp>

using namespace cfd;
```

```
template<std::size_t D>
class HeatSolver
{
private:
  static const std::size_t dimension = D;
  Mesh<dimension> mesh;
  Scenario<dimension> scenario;

  Element temperature;
  FunctionSpace temperatureSpace;
  NamedField temperatureField;

  BoundaryCondition boundaryConditions;

  SolveOperation solve;

  BoundaryCondition buildBoundaryConditions()
  {
    Tensor<dimension> zero(0);
    zero  = 0;

    Tensor<dimension> source(0);
    source = 1.0;

    BoundaryConditionList<dimension> boundaryConditionList(0);
    boundaryConditionList.add(BoundaryConditionTrivial<dimension>(1, source));
    boundaryConditionList.add(BoundaryConditionTrivial<dimension>(2, zero));
    boundaryConditionList.add(BoundaryConditionTrivial<dimension>(3, zero));
```

```
    boundaryConditionList.add(BoundaryConditionTrivial<dimension>(4, zero));
    boundaryConditionList.add(BoundaryConditionTrivial<dimension>(5, source));

    return scenario.addBoundaryCondition(temperatureSpace, boundaryConditionList);
  }

public:
  HeatSolver(Mesh<dimension>& _mesh) : mesh(_mesh), scenario(mesh)
  {
    temperature = scenario.addElement(new LagrangeTriangleLinear<0>());
    temperatureSpace = scenario.defineFunctionSpace(temperature, mesh);
    temperatureField = scenario.defineNamedField("temperature", temperatureSpace);

    BoundaryConditionList<dimension> boundaryConditionList(0);
    boundaryConditions = buildBoundaryConditions();

    solve = constructSolve();
  }

  SolveOperation constructSolve()
  {
    using namespace forms;

    SolveOperation s = scenario.newSolveOperation();

    Scalar c = 1e-4;
    Scalar k  = 10.0;

    Operator massMatrix(temperatureSpace, temperatureSpace);
```

```
  massMatrix = B(temperature, temperature)*dx;

  const forms::BilinearFormIntegralSum lhsForm =
    B(temperature, temperature)*dx +
    B(k*c*grad(temperature), grad(temperature))*dx;

  Field rhs = massMatrix * temperatureField;

  LinearSystem system = assembleGalerkinSystem(temperatureSpace,
                                               lhsForm,
                                               rhs,
                                               boundaryConditions,
                                               temperatureField);

  s.setNewValue(temperatureField, system.getSolution());

  s.finish();
  return s;
}

void step()
{
  solve.execute();
}

void outputFieldsToFile(const std::string& filename)
{
  scenario.outputFieldsToFile(filename);
}
```

```
};

int main(int argc, char** argv)
{
  try
  {
    PETScManager::instance().init(argc, argv);

    static const std::size_t dimension = TriangularMeshBuilder::cell_dimension;
    static const double maxCellArea = 1.0/5000;
    static const double polySize = 0.1;
    static const std::size_t polyEdges = 4;
    static const double polyLabel = 5;
    static const double polyRotation = 0.0;

    TriangularMeshBuilder meshBuilder(1.0, 1.0, maxCellArea);
    const Polygon poly(vertex<2>(0.5, 0.5), polyEdges, polySize, polyRotation);
    meshBuilder.addPolygon(poly, polyLabel);
    Mesh<dimension> mesh(meshBuilder.buildMesh());

    HeatSolver<dimension> solver(mesh);

    for(int i=0; i<200; ++i)
    {
      std::cout << "Starting timestep " << i << "..." << std::endl;
      solver.step();
      std::ostringstream filename;
      filename << "./heat_" << boost::format("%|04|") % i << ".vtk";
      solver.outputFieldsToFile(filename.str());
```

```
    }
  }
  catch(const CFDException& e)
  {
    std::cerr << "A simple_cfd specific exception was generated: " << std::endl;
    std::cerr << e.what() << std::endl;
  }
}
```

Appendix F

Excafé Incompressible Navier-Stokes Solver Source

```
#include <cstddef>
#include <sstream>
#include <boost/format.hpp>
#include <simple_cfd/capture/scenario.hpp>
#include <simple_cfd/capture/solve_operation.hpp>
#include <simple_cfd/petsc_manager.hpp>
#include <simple_cfd/triangular_mesh_builder.hpp>
#include <simple_cfd/lagrange_triangle_linear.hpp>
#include <simple_cfd/lagrange_triangle_quadratic.hpp>
#include <simple_cfd/capture/scenario.hpp>
#include <simple_cfd/capture/fields/fields.hpp>
#include <simple_cfd/capture/forms/forms.hpp>
#include <simple_cfd/mesh.hpp>
#include <simple_cfd/exception.hpp>
#include <simple_cfd/boundary_condition_list.hpp>
#include <simple_cfd/boundary_condition_trivial.hpp>
```

```
using namespace cfd;

template<std::size_t D>
class NavierStokesSolver
{
private:
  static const std::size_t dimension = D;
  Mesh<dimension> mesh;
  Scenario<dimension> scenario;

  Element velocity;
  Element pressure;

  FunctionSpace velocitySpace;
  FunctionSpace pressureSpace;
  FunctionSpace coupledSpace;

  NamedField velocityField;
  NamedField pressureField;

  BoundaryCondition velocityConditions;

  SolveOperation coupledSolve;

  BoundaryCondition buildBoundaryConditions()
  {
    const Tensor<dimension> zero(1);
```

```
    Tensor<dimension> inflow(1);
    inflow(0) = 5.0;

    BoundaryConditionList<dimension> velocityConditionList(1);
    velocityConditionList.add(BoundaryConditionTrivial<dimension>(1, zero));
    velocityConditionList.add(BoundaryConditionTrivial<dimension>(3, zero));
    velocityConditionList.add(BoundaryConditionTrivial<dimension>(4, inflow));
    velocityConditionList.add(BoundaryConditionTrivial<dimension>(5, zero));

    return scenario.addBoundaryCondition(velocitySpace, velocityConditionList);
  }

public:
  NavierStokesSolver(Mesh<dimension>& _mesh) : mesh(_mesh), scenario(mesh)
  {
    velocity = scenario.addElement(new LagrangeTriangleQuadratic<1>());
    pressure = scenario.addElement(new LagrangeTriangleLinear<0>());

    velocitySpace = scenario.defineFunctionSpace(velocity, mesh);
    pressureSpace = scenario.defineFunctionSpace(pressure, mesh);
    coupledSpace = velocitySpace + pressureSpace;

    velocityField = scenario.defineNamedField("velocity", velocitySpace);
    pressureField = scenario.defineNamedField("pressure", pressureSpace);

    BoundaryConditionList<dimension> velocityConditionList(1);
    velocityConditions = buildBoundaryConditions();

    coupledSolve = constructCoupledSolver();
```

```
}

SolveOperation constructCoupledSolver()
{
  using namespace forms;

  SolveOperation s = scenario.newSolveOperation();

  Scalar theta = 0.5;
  Scalar k  = 0.01;
  Scalar kinematic_viscosity = 1.0/250;

  Operator nonLinearRhs(velocitySpace, velocitySpace);
  nonLinearRhs =
   B(velocity, velocity)*dx +
   B(-(1.0-theta)*k*kinematic_viscosity * grad(velocity), grad(velocity))*dx +
   B(-(1.0-theta)*k*inner(velocityField, grad(velocity)), velocity)*dx;

  Field velocityRhs = nonLinearRhs * velocityField;
  Field load(project(velocityRhs, coupledSpace));

  TemporalIndex i;
  IndexedField unknownGuess(i);

  unknownGuess[-1] = project(velocityField, coupledSpace) +
                     project(pressureField, coupledSpace);
  const forms::BilinearFormIntegralSum lhsForm =
   B(velocity, velocity)*dx +
   B(theta*k * kinematic_viscosity * grad(velocity), grad(velocity))*dx +
```

```
    B(-1.0 * k * pressure, div(velocity))*dx +
    B(div(velocity), pressure)*dx +
    B(theta*k * inner(project(unknownGuess[i-1], velocitySpace),
      grad(velocity)), velocity)*dx;

  LinearSystem system = assembleGalerkinSystem(coupledSpace,
                                               lhsForm,
                                               load,
                                               velocityConditions,
                                               unknownGuess[i-1]);
  Operator linearisedSystem = system.getConstrainedSystem();
  unknownGuess[i] = system.getSolution();

  Scalar residual = ((linearisedSystem * unknownGuess[i-1]) -
                     system.getConstrainedLoad()).two_norm();

  i.setTermination(residual < 1e-3);

  s.setNewValue(velocityField, project(unknownGuess[final-1], velocitySpace));
  s.setNewValue(pressureField, project(unknownGuess[final-1], pressureSpace));

  s.finish();
  return s;
}

void step()
{
  coupledSolve.execute();
}
```

```
  void outputFieldsToFile(const std::string& filename)
  {
    scenario.outputFieldsToFile(filename);
  }
};

int main(int argc, char** argv)
{
  try
  {
    PETScManager::instance().init(argc, argv);

    static const std::size_t dimension = TriangularMeshBuilder::cell_dimension;

    TriangularMeshBuilder meshBuilder(3.0, 1.0, 1.0/900.0);
    meshBuilder.addPolygon(Polygon(vertex<2>(0.5, 0.5), 16, 0.15, 0), 5);
    Mesh<dimension> mesh(meshBuilder.buildMesh());

    NavierStokesSolver<dimension> solver(mesh);

    for(int i=0; i<6000; ++i)
    {
      std::cout << "Starting timestep " << i << "..." << std::endl;
      solver.step();
      std::ostringstream filename;
      filename << "./navier_stokes_" << boost::format("%|04|") % i << ".vtk";
      solver.outputFieldsToFile(filename.str());
    }
```

```
  }
  catch(const CFDException& e)
  {
    std::cerr << "A simple_cfd specific exception was generated: " << std::endl;
    std::cerr << e.what() << std::endl;
  }
}
```