University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Delayed Evaluation and Runtime Code Generation as a means to Producing High Performance Numerical Software

Project Report

Francis Russell

http://www.doc.ic.ac.uk/~fpr02/final-year-project/

Supervisor: Paul Kelly

Second Marker: Tony Field

Abstract

Attaining both performance and abstraction is a challenge often faced by software engineers. This is especially the case with mathematical software, where despite the existence of languages such as C++ which enable the usage of numerical abstractions, Fortran remains a popular language due to the high effective of available compilers. The pursuit for high performance numerical code with C++ abstractions has led to the development of linear algebra libraries that use the C++ template system to control aspects of compilation.

This report will detail the investigation into an alternative approach. By delaying evaluation of linear algebra operations until their result is required, it is possible to collect information about these operations that would not have been available at compile time. When the result is required, code is generated and compiled at runtime to perform these operations. This code is more specialised than the code that could be generated at compile time and more amenable to optimisation.

The design and implementation of a prototype library using these techniques will be discussed along with a number of optimisations investigated for improving its performance. A comparison will be carried out with the Matrix Template Library, a state of the art C++ linear algebra library using a benchmark set of linear iterative solvers.

Acknowledgements

I would like to acknowledge for their help during this project:

- Paul Kelly, my supervisor for his expert knowledge, advice, and boundless enthusiasm.
- Olav Beckmann, for our discussions about delayed evaluation and report suggestions.
- Michael Mellor, for his continued patience with my TaskGraph questions.

Contents

1 Introduction 7
- 1.1 Background . . . 7
- 1.2 Motivation . . . 8
- 1.3 Approach . . . 10
- 1.4 Goals . . . 10
- 1.5 Contributions . . . 10
- 1.6 Structure of Report . . . 11

2 Background 13
- 2.1 BLAS . . . 14
- 2.2 Vendor Optimised BLAS . . . 15
- 2.3 ATLAS . . . 15
- 2.4 Blitz++ . . . 16
- 2.5 MTL . . . 17
- 2.6 Delayed Evaluation, Self-Optimising Software Components . . . 18
- 2.7 Cross Component Optimisation . . . 18
- 2.8 Fabius . . . 19
- 2.9 Tick C . . . 20
- 2.10 TaskGraph . . . 21
- 2.11 Dependence Analysis . . . 22
- 2.12 Loop Reordering . . . 23
  - 2.12.1 Loop Interchange . . . 23
  - 2.12.2 Blocking . . . 24
  - 2.12.3 Loop Fusion . . . 25
- 2.13 Array Contraction . . . 25

3 Design 27
- 3.1 The Interface . . . 27
- 3.2 Delaying Evaluation . . . 28
- 3.3 Expression Node Hierarchy . . . 31
- 3.4 Expression DAG Creation, Evaluation and Deletion . . . 31
- 3.5 Matrix, Vector and Scalar Representation . . . 35

4 Implementing Delayed Evaluation and Runtime Code Generation 37
- 4.1 The Interface . . . 37
- 4.2 Expression DAG . . . 38
- 4.3 Expression DAG Evaluation . . . 40

4.3.1 Expression DAG Exploration . . . 40
4.3.2 Evaluation Strategy Creation . . . 42
4.3.3 Strategy Execution and Expression DAG Rewriting . . . 43
4.4 The TaskGraph Evaluator . . . 43
4.4.1 Initial Problems . . . 43
4.4.2 Basic Operation . . . 45
4.4.3 TaskGraph Imposed Limitations . . . 47
4.4.4 Towards Storage Format Independent Code Generation . . . 47

5 Runtime Optimisations 51
5.1 Code Caching . . . 51
5.1.1 The Problem . . . 51
5.1.2 The Solution . . . 52
5.2 Loop Fusion . . . 53
5.3 Runtime Liveness Estimation . . . 55
5.4 Array Contraction . . . 59

6 Evaluation 61
6.1 Code Caching . . . 62
6.2 Loop Fusion . . . 64
6.3 Array Contraction . . . 68
6.4 Runtime Liveness Estimation . . . 70
6.5 Comparison Against State of the Art . . . 73
6.6 Analysis of BiConjugate Gradient Speedup . . . 80
6.7 Summary . . . 81

7 Conclusions and Future Work 83
7.1 Conclusions . . . 83
7.2 Future Work . . . 84
7.2.1 Improved Optimisations . . . 84
7.2.2 Improved Liveness Analysis . . . 84
7.2.3 A Cache Locality Model . . . 85
7.2.4 Speculative Evaluation . . . 85
7.2.5 Persistent Code Caching . . . 85
7.2.6 Alternate Methods of Delayed Expression Evaluation . . . 86
7.2.7 Fortran Code Generation . . . 86
7.2.8 Sparse Matrices . . . 86
7.2.9 User-Level Algorithms . . . 86

A Unfused BiConjugate Gradient Solver Code 91

B Fused BiConjugate Gradient Solver Code 95

C Graphs of Collected Results 97
C.1 Loop Fusion . . . 97
C.2 Array Contraction . . . 103
C.3 Runtime Liveness Analysis . . . 109

Chapter 1

Introduction

This report presents an investigation of runtime code generation techniques for linear algebra in C++ which on one benchmark application managed to achieve a performance increase of over 50% over the best available statically optimised code. The quest for performance has led many numerical applications to be written in languages or use libraries that sacrifice clarity for this goal. This project will investigate techniques that I hope will show that this no longer need be the case.

Unlike many other approaches to high performance numerical code, the one taken by this project does not require that abstractions be sacrificed to attain performance. I will compare this approach against the state of the art and evaluate its benefits and disadvantages against it.

I will continue this chapter with a background, motivation for this project, the goals I hope to achieve, the approach taken and the contributions of this report. I will conclude with an overview of the structure of the rest of the report.

1.1 Background

BLAS (Basic Linear Algebra Subprograms), is a set of routines that standard building blocks for linear algebra operations. BLAS has thee levels.

1. Scalar, vector and vector-vector operations.

2. Matrix-vector operations.

3. Matrix-matrix operations.

Each BLAS level could be implemented in terms of the previous level, but provides superior performance than would be possible with such an implementation. Unfortunately, with the increase in performance comes an increase in complexity. More than one BLAS level three method takes eleven parameters.

Much effort has been expended on improving the performance of BLAS. Many processor vendors have also developed BLAS implementations optimised for their line of processors. Whilst this provides a way to achieve high application performance for a given platform, it is dependent on the processor vendor or other company taking the time and effort to create and maintain that implementation.

Another notable effort towards improving BLAS is the ATLAS project. The ATLAS project is part of research effort into providing a portable, efficient implementation of BLAS. It does this via code generation and search heuristics in order to create binaries optimised for a given platform. Whilst the ATLAS project has been successful, it still requires that library clients use the BLAS interface.

In comparison, the development of numerical libraries for C++ incorporating the functionality of BLAS has been slow. C++ allows the user to represent matrices and vectors with objects which can be manipulated in an intuitive manner using language features such as operator overloading. Unfortunately, the obvious implementation using pairwise evaluation of operators produces low performance code. This is due to the number of temporaries matrices and vectors allocated and deallocated.

A technique called Expression Templates using the C++ template system has allowed libraries to be developed that control the way expressions using matrices and vectors are parsed. Templates can also be used to control code generation to perform transformations such as loop unrolling and blocking. Blitz++ and the Matrix Template Library use these techniques. Of these libraries MTL is the most developed

A technique called Expression Templates using C++ Template Metaprogramming has allowed libraries to be developed that control the way expressions using matrices and vectors are parsed. Templates can also be used to control code generation to perform transformations such as loop unrolling and blocking. Blitz++ and the Matrix Template Library use these techniques. Of these libraries, MTL is the most developed and can be considered a state of the art C++ numerical library.

1.2 Motivation

Template metaprogramming techniques are one approach to addressing the imbalance between abstraction and performance in C++ numerical software. The motivation for this project is the investigation of another, namely Delayed Evaluation Self Optimising software components.

DESO[15] (Delayed Evaluation Self Optimising) software components function by delaying the evaluation of an operation until the result is required. This allows them to capture runtime context information and use it to optimise their performance. One useful feature this provides is the ability to equip a performance library with semantics that are most useful to the library's client. An iterative solver that uses such an interface is given in Figure 1.1. A DESO library can delay calls to made to it, and when a result it required, execute the delayed operations in such as way to to improve the their performance.

Beckmann and Kelly have done work showing the effectiveness of DESO software components with the implementation of a DESO parallel linear algebra library[5]. Liniker et al. have shown that DESO components can work effectively with C++ abstractions[15]. To do this, they implemented a C++ interface on top of the C interface to the DESO parallel linear algebra library. They then showed that a set of templated iterative solvers could work effectively and provide performance increases using the interface.

The promise of the DESO approach begs the question whether it can be

```
template < class Matrix, class VectorX, class VectorB,
           class Preconditioner, class Iteration >
int cg(const Matrix& A, VectorX& x, const VectorB& b,
       const Preconditioner& M, Iteration& iter)
{
  typedef VectorX TmpVec;
  typename itl_traits<VectorX>::value_type rho(0), rho_1(0), alpha(0), beta(0);
  TmpVec p(size(x)), q(size(x)), r(size(x)), z(size(x));

  itl::mult(A, itl::scaled(x, -1.0), b, r);

  while (! iter.finished(r)) {
    itl::solve(M, r, z);
    rho = itl::dot_conj(r, z);
    if (iter.first())
      itl::copy(z, p);
    else {
      beta = rho / rho_1;
      itl::add(z, itl::scaled(p, beta), p);
    }

    itl::mult(A, p, q);
    alpha = rho / itl::dot_conj(p, q);
    itl::add(x, itl::scaled(p, alpha), x);
    itl::add(r, itl::scaled(q, -alpha), r);
    rho_1 = rho;

    ++iter;
  }
  return iter.error_code();
}
```

Figure 1.1: A templated method for the Conjugate Gradient iterative solver from the Iterative Template Library suite. ITL performs the algorithm using a non-performance oriented interface. Any library used will need to be able to achieve high performance using the abstracted interface. ITL is primarily designed to work with MTL, a state of the art C++ numerical metaprogramming library.

used effectively to improve the performance of a conventional non-parallelised numerical library.

1.3 Approach

To investigate this approach I intend to develop a prototype library which will allow the effectiveness of the DESO methodology to be tested. Unlike the parallel DESO library, which optimised data placement, the prototype library will need to optimise the execution of the delayed operations themselves. I intend to do this using the TaskGraph Library.

The TaskGraph[4] library is a C++ library that provides features for runtime code generation. The library provides facilities for the construction of TaskGraphs, which are fragments of code constructed at runtime. TaskGraphs are constructed using a simplified C-like sub-language implemented using macros. Once a TaskGraph is constructed, it may have optimisations applied to it. Lastly, it is converted to C, compiled, and executed.

The TaskGraph library provides a useful tool for runtime code generation. Most importantly, it provides the ability to specialise and optimise the runtime generated code by using the information gained by the delaying evaluation.

1.4 Goals

This project aims to provide a greater understanding of the benefits and disadvantages of the DESO methodology. Whilst it is clear that DESO components can be used to discover runtime information and use it to improve performance, the information discovered and the way it is used is highly domain specific. I hope this project will clarify these issues in the context of a non-parallel linear algebra system.

The speedups obtained though TaskGraph have often been achieved with relatively long executing segments of code that have been highly specialised. With numerical code, apart from loop boundaries, there is little information that can be used to specialise the code more than its compile time equivalent. Instead, other methods must be found to optimise the runtime generated code. One of this project's goals is to discover and investigate these methods and analyse their effectiveness in improving the performance of the generated code.

Lastly, it is hoped that the project will produce a prototype library, demonstrating the effectiveness and applicability of the techniques analysed. This could provide a framework for future research into these techniques. Furthermore, an analysis of the design decisions made could assist in the development of fully fledged library if the techniques turn out to be effective.

1.5 Contributions

This project has contributed:

High Performance Numerical Code A system has been developed which on one benchmark managed to outperform the best available statically compiled code by over 50%.

Implementation and Analysis of Runtime Optimisation Techniques A number of techniques have been implemented to utilise the information collected through delayed evaluation. These include:

1. Runtime code generation.
2. Code caching and reuse.
3. Loop fusion.
4. Array contraction.
5. Runtime liveness estimation.

A Comparison Against the State of the Art A comparison has been performed between the approach adopted by this project against the state of the art MTL library. This has allowed the benefits and disadvantages of this approach to be compared against template metaprogramming techniques.

Further Development of the TaskGraph System Modifications have been created to the TaskGraph system that could be of later use. These include:

1. Modifications to the loop fuser that allow the fusion of loops in more arbitrary areas of the program.
2. An array contraction pass capable of reducing array dimensionality and reduction of some arrays to a scalar value.

A Framework for Future Research A prototype library has been produced which could be used for further research. It provides an abstract interface compatible with numerical linear solvers. It enables transformations of runtime generated code, and the analysis of numerical operations executed by the client application on multiple levels.

1.6 Structure of Report

This later sections of this report comprise:

Background This will describe previous work related to performing high performance numerical applications, work relevant to the approach taken by this project, and the state of the art C++ template metaprogramming numerical libraries.

Design This section will describe the design decisions made before the implementation of the prototype library and their rationale.

Implementation This section will provide an overview of the implementation of the first functioning implementation of the prototype library.

Further Development This section will detail the later changes made to the prototype library to further the investigation. These include the modifications to the TaskGraph loop fuser, the runtime liveness estimation system and the SUIF array contraction pass.

Evaluation This will present an analysis of the relative effectiveness of the different runtime optimisation techniques used. It will also provide a performance comparison using these techniques against the state of the art. It will discuss the situations in which the this approach provides benefits and the situations in which it decreases performance. It will conclude with an analysis of the reasons for the performance increase of one of more successfully improved iterative solvers.

Conclusion This will discuss the knowledge gained in the undertaking of this project and what it can tell us about the investigated approach. It will describe the benefits and disadvantages of this approach and suggest ways to remedy the latter. It will conclude with the questions this project has raised or been unable to answer and suggest further work that could tell us more.

I have detailed this project's background including available Fortran, C and C++ numerical libraries. I have discussed this project's motivation for high performance abstract linear algebra libraries and the approach it will investigate. I have stated what this project hopes to achieve and what it has contributed. I will now continue with a detailed background covering related and relevant work.

Chapter 2

Background

I will first cover the approaches made in obtaining numerical performance though performance oriented interfaces, namely BLAS, vendor optimised BLAS, and ATLAS. All these approaches achieve high performance, but have the disadvantage of using performance oriented interfaces that complicate their usage.

I will then cover later approaches to achieving abstraction and performance though C++ template techniques. These are used by libraries such as Blitz++ and MTL which aim to provide both performance and abstraction for numerical operations.

Lastly, I will cover work related to the approach taken by this project, including:

Delayed Evaluation, Self-Optimising Software Components By delaying evaluation, these allow the capture of contextual information only available at runtime. This facilitates optimisations that could not have been performed at compile-time.

Cross Component Optimisations These are optimisations that alter components to work together more effectively. In this context, components can refer to any sort of modules that are composed together in order to perform some sort of operation, for example, BLAS routines. In the case of Level 1 BLAS routines, loop fusion between operations is one optimisation that might be performed. Cross component optimisation is a useful technique for DESO components as the extra contextual information they have allows cross component optimisations to be performed that could not have been done at compile-time.

Runtime Code Generation This is the generation, compilation, and execution of code at runtime. By generating code at runtime, cross component and other types of optimisation can be carried out using the contextual information made available through the usage of DESO software components.

Dependence Analysis, Loop Transformations and Other Optimisations Dependence analysis is used to determine the legality of certain compiler transformations. Loop transformations are a class of optimisation that attempt to restructure loops to be more efficient. As linear algebra operations tend to involve loops, it is expected that most of the useful optimisations applicable at runtime will be loop transformations. In order to perform these optimisations and determine their correctness, both an understanding of loop transformations and dependence analysis is required. Another optimisation discussed is array contraction, which allows the reduction of the dimensionality of arrays and is expected to be useful in removing temporary variables. Array contraction is often facilitated by loop fusion.

I will then describe conclusions that can be made from the presented work, and their relevance to the approach taken by this project.

2.1 BLAS

The Basic Linear Algebra Subprograms[11] are routines written in Fortran which provide basic building blocks for vector and matrix operations. They are classified into three levels:

Level 1 These are the original set of BLAS routines. They perform vector, scalar and vector-vector operations.

Level 2 These consist of matrix-vector operations.

Level 3 These consist of matrix-matrix operations.

BLAS functions usually accumulate their result into one of the operands to aid memory reuse. The Level 2 and Level 3 BLAS also contain routines optimised for symmetric, triangular and Hermitian matrices, as well as banded and packed matrix storage formats, although the burden is on the programmer to ensure that they use the correct routines to do this.

In their paper on BLAS level 2[8], Dongarra et al. note that Level 1 BLAS is not the most effective way to improve the efficiency of higher level code on modern architectures. This was because the BLAS level 1 interface inhibits optimisations that could be performed for matrix-vector operations on vector machines as the full nature of the operation is not apparent to the compiler. In their paper on Level 3 BLAS[7], Dongarra et al. note that Level 2 BLAS does not translate well to a computers with a memory hierarchy as data is not reused effectively. Level 3 BLAS allows higher performance by using blocking algorithms, which exploit the memory hierarchy, allow operation on blocks to be performed in parallel, and operations within each block to be performed in parallel. The evolution of BLAS clearly shows that by optimising across a series of smaller operations, benefits can be achieved that were not previously unavailable.

The BLAS interfaces were designed for performance. Each function, especially those at the higher levels, take a large number of parameters, and perform a large number of more fundamental operations simultaneously. Whilst the higher level BLAS functions have been chosen to be those most useful to the scientific computing community, it is still clear that the higher levels of performance one wishes to achieve with BLAS, the more specific routines one must use to do this, and the more complicated the development of high performance mathematical software.

2.2 Vendor Optimised BLAS

A number of processor vendors have created BLAS implementations for different processors. These typically contain routines optimised to use specialist instructions available on each processor such as the SIMD instructions available on the PowerPC G4 & G5. There exist versions for a number of architectures including but not limited to:

- AMD Opteron
- PowerPC G4 & G5
- Alpha
- Intel IA32
- Intel Itanium

Whilst these optimisations may significantly improve the performance of BLAS, they are typically extremely platform specific, and expensive to create and maintain. There is clearly a need for more portable performance.

2.3 ATLAS

The Automatically Tuned Linear Algebra Software[21] is a research effort using empirical techniques in order to provide portable performance. ATLAS uses a methodology named "Automated Empirical Optimisation of Software". Code generation is used to search for possible implementations of performance critical regions and timing used to select the best one. The requirements of libraries using AEOS are:

Isolation of performance critical regions Performance critical regions are isolated in order that their behaviour can be analysed.

Methods of adapting the software to different environments AEOS relies on iteratively trying performance critical regions whilst varying the code either though parameters or by having it produced by a parameterised code generator.

Robust, context sensitive timers Timing is required in order to able select the best code. The timers used need to be accurate, and robust enough to produce correct results even if the machine being used is under load from unrelated processes. The timing must be context sensitive in the sense that the timing conditions must replicate the way the code is likely to be used. This is necessary to account for factors such as the contents of the cache when the routine is called. For example, if a routine is likely to be called with an empty cache, the cache must be flushed before beginning timing.

Appropriate search heuristic If the search space for different implementations of a performance critical region is large, as in the case of complex code generation systems, heuristics are required in order to allow the search to complete within an acceptable time.

Through these techniques ATLAS can adapt to a given architecture without needing to be hand-tuned to it. It also allows ATLAS to leverage the latest compiler technology as it adapts the optimisations it provides. ATLAS has few restrictions in order to be able to adapt to a given platform, namely:

Adequate C compiler As ATLAS performs its own code generation, it does not require the compiler to be able to perform many optimisations. In fact, too heavy optimisation by the compiler may reduce the efficiency of the generated code. However, the compiler should be effectively use the underlying Instruction Set Architecture in order for ATLAS to perform well.

Hierarchical Memory ATLAS assumes a memory hierarchy is present and will produce best results when the system being targeted has registers and a L1 cache. Without hierarchical memory, ATLAS's tuning for blocking and register usage become overheads. Even if this is the case, performance may be acceptable.

ATLAS shows that it is feasible to achieve portable high performance across a number of architectures. The success of ATLAS's approach, achieved though yet more effective optimisation of BLAS might seem to indicate that abstraction is cannot be maintained if high performance is required. However, this is not the case, as demonstrated by Blitz++ and MTL.

2.4 Blitz++

Blitz++ is a numerical library written with the intention of obtaining performance rivalling FORTRAN whilst preserving a C++ object oriented interface.

```
Array<float,3> A, B, C, D;
A = B + C + D;
```

Using Blitz++[19], the expression summing three 3-dimensional arrays as above has its parse tree represented as a template type. The generated code does not possess extra loops for each term in the addition and uses no temporaries, resulting in code structured as below.

```
for (int i=0; i<N1; ++i)
  for(int j=0; j<N2; ++j)
    for(int k=0; k<N3; ++k)
      A(i,j,k) = B(i,j,k) + C(i,j,k) + D(i,j,k)
```

This has been achieved through a technique called "Expression Templates"[18]. Expression Templates allow the passing of expressions as function arguments. Upon compilation, the compiler produces an instance of the function that contains the expression inline. This technique also allows code to be written as above.

The ability Blitz++ to manipulate the parse tree allows it to perform other optimisations including:

- Loop Interchange
- Collapsing Inner Loops
- Partial unrolling of inner loops
- Exploit common strides and hoist invariant stride computations
- Detect stencils, and perform tiling to optimise cache use.

Blitz++ clearly demonstrates that is is possible to achieve high performance numerical computing in C++ whilst maintaining object oriented abstractions.

2.5 MTL

The Matrix Template Library[16] is written in C++ and aims to attain both appropriate abstractions and performance though the use of generic programming.

Algorithms are expressed independently of data storage formats. using iterators to traverse the data stored in containers. In this way, algorithms are unaware of the indexing in the object they are operating on. MTL relies on the optimising abilities of the compiler to remove this level of abstraction.

MTL is built on top of BLAIS[17], the Basic Linear Algebra Instruction Set, which is layered on top of FAST, the Fixed Algorithm Size Template library. BLAIS provides functionality similar to Level 1, 2, & 3 BLAS. FAST is basically an implementation of the Standard Template Library but for computations whose size is known at compile time.

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

// 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<());
```

Both implementations iterate though the arrays x and y, summing the values at each index, and storing each result at the index in y. The primary difference between the two calls is that the number of operations to be done has been supplied as a template parameter. The FAST implementation of the transform function is recursive, resulting in inlined code on compilation, and no loops.

Through the use of templates as a compile time code generation mechanism and generic programming as an abstraction mechanism, MTL has been able to attain high levels of performance for many mathematical operations whilst maintaining abstractions. Thus, MTL demonstrates that high performance numerical code with C++ abstractions is possible.

2.6 Delayed Evaluation, Self-Optimising Software Components

Beckmann and Kelly describe a delayed-evaluation self-optimising linear algebra library[5] for a distributed memory multicomputer. Through delayed evaluation, a directed acyclic graph is built which represents the computation to be performed.

The point where execution can be delayed no further is known as a *force point*. In the library described, the encounter of a force point triggers the construction of an optimised execution plan. The plan stores data redistributions which are defined as affine functions mapping array index vectors onto virtual processor indices. Building an execution plan entails minimising the cost of the different data redistributions.

A strategy is also devised for reusing execution plans. As the optimisation problem characterised by the DAG of operations is complex and traversal to check for cache hits expensive, a hash value is calculated for each node which encodes the placement or placement constraints of that node. For each execution plan, additional information is stored with regards to whether it is believed the plan can be optimised more and whether its last usage was sub-optimal or not.

This library effectively demonstrates that delayed-evaluation self-optimising software components can be an effective method for obtaining context information only determinable at runtime. It also shows that runtime collected information can be used to improve performance.

Further work by Liniker et al.[15] demonstrates that this technique is effective in separating the interface to a linear algebra library from the concerns of performance by presenting a C++ interface to the library that provides the functionality required by IML++[6], a set of templates for iterative solving methods in C++.

2.7 Cross Component Optimisation

We have already discussed the evolution of BLAS, and the optimisations possible by optimising across different operations. Work by Ashby et al.[1] mentions the importance of modularity and encapsulation in software engineering. They discuss how modularity of components imposes limitations on the optimisations that can be performed program wide. They present a case study in which the performance of a numerical benchmark, an iterative solver, is analysed when implemented with ATLAS, Fortran, and Aldor.

The Aldor compiler generates an intermediate representation called FOAM, which during linking allows the compiler to perform extensive levels of cross component optimisation. After optimisation, the FOAM representation is translated to C and linked against a small runtime library.

The Aldor algorithm implementation of the iterative solver uses level 1 BLAS routines, also implemented in Aldor. A comparison is made between the solvers using ATLAS, Fortran, the unoptimised Aldor implementation and the Aldor implementation at three levels of optimisation. Analysis of the various Aldor optimised solvers show that the compiler has fused many of the function calls together, with more fusions and increasingly aggressive code rearrangement occurring at the higher optimisation levels.

Results show that for larger problems sizes (those incapable of being effectively cached by the processor) that a significant speed up is possible over both the ATLAS and Fortran implementations. It should be noted that these optimisations were compared against ATLAS's Level 1 BLAS. The performance against higher BLAS levels is unknown.

2.8 Fabius

Leone and Lee discuss a prototype compiler they have created, Fabius[13], to investigate the notion of deferred compilation. The characteristics of deferred compilation in Fabius are as follows:

Lightweight Each part of of a compiled program that performs run-time code generation is "hard wired" to optimise and generate code for a small portion of the input program, avoiding the need to process any intermediate representation at run time.

Automatic Manual construction of code templates or run-time code generators are not required. Instead syntactic cues and programmer hints are used to determine what code should be subjected to run-time compilation.

General Many standard optimisations such as function inlining can be efficiently employed at run-time.

Fabius compiles a rudimentary, strict, first order functional language, generating native code for the MIPS R2000. The three stages of compilation are:

- Staging analysis to determine which computation stages at which it may be profitable to perform run-time code generation.
- Register allocation
- Code generation, compiling "early" computations in the normal way, and "late" computations as machine code that generates optimised instruction sequences at run-time.

Fabius allows optimisations such as instruction selection, inlining, loop unrolling and specialisation to occur at run-time.

Benchmarks comparing matrix multiplies using Fabius against statically optimised code showed code showed performance improvements for dense matrices larger than 20x20 and sparse matrices larger than 2x2.

Fabius demonstrates that run-time code generation can be effective tool for gaining performance improvements through code specialisation and other runtime optimisations, and that this can be an effective technique for even small operations.

It should be noted that whilst Fabius's code generation is "lightweight" avoiding intermediate representations at runtime, the approach taken by this project, which will involve the generation and optimisation of an intermediate form at runtime, is not.

2.9 Tick C

Dawson et al. describe 'C[9] (Tick C), a superset of ANSI C that allows machine independent specification of dynamically generated code. 'C provides support for dynamically generated code through two type constructors and three unary operators. The goals of 'C are described as follows:

- To be a clean extension to ANSI C, not affecting the syntax or semantics of ANSI C.
- Allow flexible calling of dynamically generated code, to the extent that functions can be constructed for which the type and number of parameters is unknown at compile time.
- Allow efficient implementation. To do this, the majority of the code generation costs need to be paid at compile time.

The extensions used are described at follows:

The ' Operator The back quote operator, which cannot be nested, proceeds a statement or compound expression to specify that it is dynamic code. Dynamic code is lexically scoped so that variables in enclosing static code can be captured by free variables in the dynamic code. The limitations on the dynamic code are that usage of "break", "continue", "case" and "goto" statements cannot transfer control outside the dynamic code. An example call to printf, where j must be in the enclosing scope:

```
'printf("%d", j)
```

cspec Types These refer to types of the dynamically generated code. This is necessary in order to perform static type checking. The type for a piece of dynamic code is *type cspec* where *type* is the type of the dynamic value of the code. For example:

```
int cspec num = '10 * 10;
```

The @ Operator The @ operator is used to combine dynamic code specifications. The operands must either be vspecs (described later) or cspecs.

```
int cspec c1 = '4;
int cspec c2 = '5;
int cspec sum = '(@c1 + @c2);
```

vspec Types A vspec (variable specification) type represents a dynamically generated lvalue. These can be initialised through the 'C library functions "param" and "local". The function "param" is used to create parameters for functions being constructed, and "local" to reserve space for a variable local to the dynamic code.

The $ Operator The $ operator allows program values to be incorporated as constants into the dynamic code. $ may be applied to any expression within dynamic code that is not a cspec or vspec. An example $ usage:

```
int cspec c1, cspec c2;
void cspec c;
int x = 1;

c1 = '$x;
c2 = 'x;
c = '{ printf("$x = %d, x = %d\n", @c1, @c2); };
x = 14;
compile(c, TC_V)();
```

The first value of x is incorporated as a constant into the dynamic code, but the second reference to x is not so this example will print: ”$x = 1, x = 14”.

The prototype 'C compiler demonstrates that significant gains can be achieved in performance from generating code at runtime as well as the fact that this can be achieved comprehensibly with only a few change to the language. However, from the perspective of portable high performance linear algebra 'C has the disadvantage that it relies on non-standard language extensions, and therefore less available compilers, in order function on a given platform, and is strongly based on C, which provides little help in providing a clean interface to linear algebra operations.

2.10 TaskGraph

The TaskGraph library[4] is a C++ library for dynamic code generation. A TaskGraph represents a fragment of code which can be constructed and manipulated at run-time, compiled, dynamically linked back into the host application and executed. TaskGraph enables optimisation with respect to:

Runtime Parameters This enables code to 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.

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

An example function in C++ for generating a matrix multiply in the TaskGraph sub-language resembles a C implementation:

```
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];
  }
}
```

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.

2.11 Dependence Analysis

A dependence is a relationship between two computations that constrains their execution order. Dependence analysis[2] identifies these constraints in order to determine if a given transformation can be applied without changing the computation's semantics.

There are two types of dependence. Control dependence refers to the case where one statement determines if the other will be executed. Data dependence refers to the case where two statements cannot be executed simultaneously because of two conflicting uses of the same variable. There are three types of data dependence.

Flow Dependence A statement S has a flow dependence on a statement *S'* if *S'* must be executed first because it writes a value that is later read by S.

Anti-Dependence A statement S has an anti-dependence on *S'* when S overwrites a variable that *S'* must read.

Output Dependence A statement S has an output dependence on *S'* if S overwrites a value that *S'* previously wrote.

In order to determine the dependence information for a piece of code, a compiler will usually create a *dependence graph*. Each node in the graph usually represents a statement and each edge a dependence between the two nodes.

Loop dependence analysis is more complex because there may be relationships between each iteration in the loop. A dependence that occurs across different iterations in a loop is called a *loop-carried* dependence. Whilst scalar expression are relatively simple to analyse, array expression are more complex as they require analysis of the subscripts used in the array references.

For perfectly nested loops we can uniquely define an iteration with a tuple of d elements, $I = (i_1, ..., i_d)$ where each i is the value of an index in its corresponding loop. The leftmost index refers to the index variable of the outermost loop.

A dependence exists between two references in iterations I and J, when at least one of the references is a write and the subscript values are the same. We can find the *dependence distance* by subtracting the dependence distances $I - J = (i_1 - j_1, ..., i_d - j_d)$. When a dependence distance for a pair of references is the same across all iterations, it can be referred to as a *distance vector*. It should be stressed that the distance vectors represent distances between iterations not between array elements. The first non-zero element of a distance vector is always positive, as a negative value would indicate a dependence on a future iteration, which is impossible. Sometimes, it is not possible to determine the exact value of a distance vector at compile time, but it can be partially characterised as a *direction vector*. Together, these can be referred to these as *dependence vectors*.

When determining dependence, a compiler usually tries to prove independence using various tests on subscript expressions, typically with the requirement that they are linear. If dependencies are too complex to analyse, dependence is assumed. There also exist tests to prove independence. Searching for dependencies is an NP-Complete problem although approximate tests exist.

2.12 Loop Reordering

Loop reordering transformations change the relative order of execution of the iterations of loop nests. This can help improve memory locality and expose parallelism. Only the transformations expected to be the most useful in this investigation have been discussed here, for a detailed treatment consult Bacon et al.[2].

2.12.1 Loop Interchange

Loop interchange[2] exchanges the position of two loops in a perfect loop nest (the only statement inside the outer loop is the inner loop containing the statements iterated over by both loops). Loop interchange may be performed to:

- Enable vectorisation by exchanging an inner, dependent loop with an outer independent loop.

- Improve vectorisation by moving the independent loop with the largest range into the innermost position.

- Improve parallel performance by moving an independent loop outwards in a loop nest. This increases the granularity of each iteration, reducing the number of barrier synchronisations.

- Reduce loop stride.

- Increase the number of loop-invariant expressions in the inner loop.

Consider the following code for a matrix multiply of 2 NxN matrices, *A* and *B* where the elements are stored in row-major order[10]:

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
      for(k=0; k<N; k++)
            C[i][j] += A[i][k]*B[k][j];
```

A is traversed in row-major order, but *B* is traversed in column-major order. This will lead to poor cache locality when accessing the elements of *B*. After exchanging the loops:

```
for(i=0; i<N; i++)
  for(k=0; k<N; k++)
    for(j=0; j<N; j++)
      C[i][j]+=A[i][k]*B[k][j];
```

Now, both *C* and *B* are traversed in row-major order resulting in improved cache usage.

2.12.2 Blocking

Blocking works by reordering the execution of loop nesting so data is not evicted from the cache before it is needed again. Taking an example matrix multiply loop[10]:

```
for(i=0; i<N; i++)
  for(k=0; k<N, k++)
  {
      r = A[i][k];
      for(j=0; 5<N; j++)
        C[i][j] += r*B[k][j];
  }
```

”Strip Mining” is applied in which the loops are fragmented into smaller segments or strips.

```
for(i=0; i<N; i++)
  for(kk=0; kk<N; kk+=BLKSZ)
    for(k=kk; k<min(kk+BLKSZ, N); k++)
    {
      r = A[i][k];
      for(jj=0; jj<N; jj+=BLKSZ)
        for(j=jj; j<min(jj+BLKSZ, N); j++)
          C[i][j] += r*B[k][j];
    }
```

The inner loops have each been transformed into two nested loops but the execution order remains the same.

```
for(kk=0; kk<N; kk+=BLKSZ)
  for(jj=0; jj<N; jj+=BLKSZ)
    for(i=0; i<N; i++)
      for(k=kk; k<min(kk+BLKSZ, N); k++)
      {
        r = A[i][k];
        for(j=jj; j<min(jj+BLKSZ, N); j++)
          C[i][j] += r*B[k][j];
      }
```

After loop interchange, the operation becomes a number of multiplications of pairs of partial matrices of size BLKSZ x BLKSZ. If BLKSZ is chosen correctly, then a whole BLKSZ x BLKSZ sub-matrix of B and BLKSZ elements of a row of C will fit into the cache.

2.12.3 Loop Fusion

Loop fusion[2] can improve performance by:

- Reducing loop overhead
- Increasing instruction parallelism
- Improving register, vector, data cache, 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 may be fused so long as there are no statements S_1 in the first loop and S_2 in the second loop such that S_1 has a dependence on S_2.

Of course, it is also possible for loop fusion to decrease performance. This could 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.

2.13 Array Contraction

Array contraction[2] is one of a number of memory access transformations designed to optimise the memory access of a program. It allows the dimensionality of arrays to be reduced, decreasing 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.[2].

If the iteration variable of the p^{th} loop in 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

- x is not used subsequent to the loop

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

Chapter 3

Design

In order to investigate the techniques previously discussed, I decided that a prototype library using them should be written. It was important that the effectiveness of these techniques could be evaluated with realistic use cases. Discussions with my supervisor suggested that the solution of linear systems of equations would reflect common usage of a linear algebra libraries, and allow for simple performance testing.

In this investigation, as decisions were made to investigate certain techniques further, later choices about design were made. These will be covered in future chapters. The design described here details and discusses the decisions made for the first working version of the prototype library.

The objectives for the first working version of the library were:

- An interface suitable for use by an application making use of linear algebra.
- Implementations of matrices, vectors and scalars and working implementations of the most common linear algebra operations.
- Delayed evaluation of these operations transparent to the client of the system.
- Runtime code generation and execution in order to perform these operations.

Amongst the factors influencing the design decisions were the need to produce code that would be amenable to loop fusion and code for which that loop fusion could help improve cache usage.

3.1 The Interface

Previous work done by Beckmann[5] on DESO techniques for data placement in a parallel library used IML++[6], a set of C++ iterative methods for solving linear systems of equations. IML++ provides templated code for each iterative solver, but does not provide the linear algebra objects or the implementation of the operations between them. Hence, the prototype library would be free to provide a DESO implementation of vectors, and matrices and the operations involving them.

The methods required by IML++ are as follows[1]:

```
Vector()                                    // Null vector construction
Vector(unsigned int n)                      // Vector of size n construction
Vector  Matrix::operator*(Vector)           // Matrix-vector multiply
Vector  Matrix::trans_mult(Vector)          // Transposed matrix-vector multiply
Vector& Vector::operator=(Vector)           // Vector assignment
Vector& Vector::operator=(Scalar)           // Assignment of scalar
                                            // to all elements in vector
Vector  Vector::operator+(Vector)           // Vector addition
Vector  Vector::operator-(Vector)           // Vector subtraction
Vector  Vector::operator*(Vector)           // Vector cross product
Scalar& Vector::operator()(int)             // Vector element access
Scalar  dot(Vector, Vector)                 // Vector dot product
Real    norm(Vector)                        // Vector norm
Vector  Preconditioner::solve(Vector)
Vector  Preconditioner::trans_solve(Vector)
```

Preconditioners are matrices that are used to transform the spectral properties of a coefficient matrix in a linear system. Take a linear system: $Ax = b$ where A represents a matrix of coefficients in a set of linear equations and we want to find x. A preconditioner matrix M transforms the linear system such that $M^{-1}Ax = M^{-1}b$. If M approximates A in some way, then this transformed system has the same solution as the original system but the system a coefficient matrix of $M^{-1}A$ may have more favourable properties with respect to convergence rate. For a more detailed description, consult Barrett et al.[3].

The IML++ preconditioner object represents some preconditioned system such that when *solve* and *trans_solve* are given b, they will try to find x where $Ax = b$ and $A^T x = b$, respectively. The method *solve* and less often *trans_solve* are used by the iterative solvers.

3.2 Delaying Evaluation

Liniker describes the design decisions made in creating an IML++ compatible interface, DESO++, to the parallel linear algebra library developed by Beckmann[15]. In this interface, the linear algebra objects manipulated by the client of the library are merely handles to a Directed Acyclic Graph (DAG) of the delayed expression. It is also stated that these handles are actually integer indices into a data structure storing the DAG.

I decided to to follow a similar approach in the prototype library, with clients holding references to delayed expressions represented as a DAG. Each node in the DAG holds references to the handles pointing to it, as well as other nodes in the DAG that use its value. When there are no longer any references to the node, it will delete itself.

There are a couple of different ways in which a DAG could be used to represent the delayed evaluation of expressions.

[1] The IML++ manual does not describe the methods required in standard syntax nor does it fully state their semantics, so certain assumptions have been made in the description here

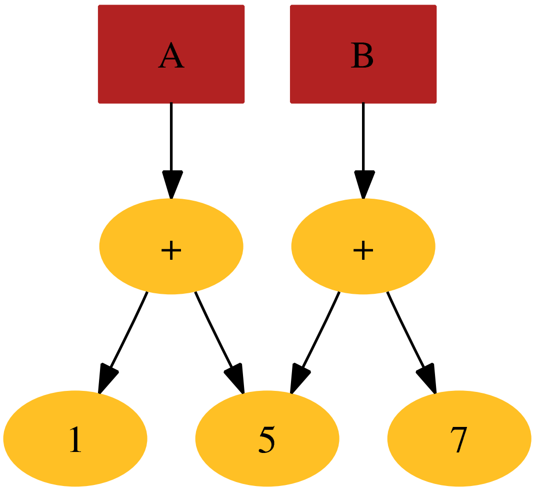


Figure 3.1: A dependency graph of the expression nodes with immutable operations. A and B represent handles to delayed expressions.

1. The DAG represents an expression that must be evaluated in order to calculate the value for a handle. None of the operators in this representation mutate the operands. The copying and assignment of handles create or modify handles to point to the same delayed expression node.

2. The DAG represents a sequence of delayed operations. Copying and assignment of handles result in delayed copies of the expressions they point to. Mutating operations such as += are represented explicitly in the DAG.

Consider the code:

```
A = 5
B = A + 7
A += 1
```

Figure 3.1 shows how first representation captures dependencies relating from the values required to evaluate each expression. Figure 3.2 shows that the second representation captures the same dependencies but also ordering dependencies resulting from the += operation, which constrains the evaluation of the expression "A + 7" to occur before incrementing A which is equal to 5.

From a delayed evaluation perspective, the first representation is more useful in the sense that it allows for greater freedom in the rescheduling of operations. Also, as the all nodes in the DAG represent immutable expressions, copying and assignment of handles can be implemented such that they do not cause data to be copied, which could be useful if vector and matrix values are frequently passed to functions. Liniker notes in his work on DESO++ that his initial implementation of copy constructors resulted in a large number of delayed copy operations so they were redefined as creating aliases whilst only assignment operations actually copy the data. Presumably altering these aliases also cause the original value to be modified. By having immutable expressions, it is possible to avoid data copying for both assignment operators and copy constructors, and

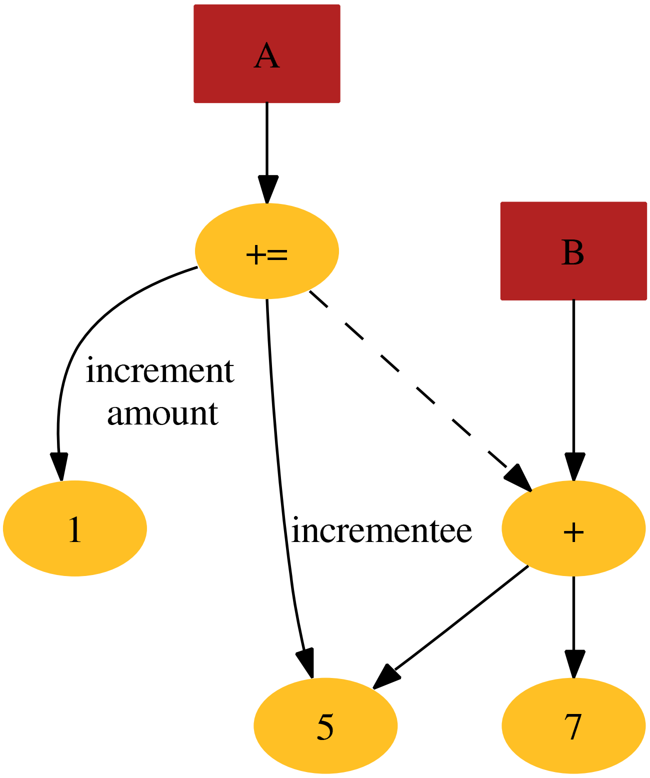


Figure 3.2: A dependency graph of the expression nodes with mutable operations. A and B represent handles to delayed expressions. The dashed line represents the constraint that the += operation can only be performed on the value ”5” after it has been added to ”7”

still preserve the semantics of these operations that a client of the library would expect.

Liniker also describes the use of Expression Templates to parse expressions using the objects from the library. Using these allow the generation of inlined code for generating the DAG which is executed by the assignment operator "=". Expression Templates are used in Blitz++ and MTL for controlling the way an expression is parsed to avoid the inefficiency of evaluating linear algebra expressions in a pairwise manner. However, their role in the DESO++ interface is unclear. As expression evaluation in DESO++ only results in DAG generation, it would seem that the use of Expression Templates is an unnecessary attempt at optimisation. As Expression Templates are complex to implement and debug, and their use in DESO++ is not well justified, I decided not to use them in the implementation of the prototype library.

3.3 Expression Node Hierarchy

Figure 3.3 shows the relationship between the types in the expression node hierarchy. Pairwise nodes represent operations that are composed of operations between corresponding elements in matrices or vectors. Addition and subtraction are examples of this class of operation. Scalar piecewise nodes represent operations that use a scalar value to transform every element in a matrix or vector. Multiplication of a vector or matrix by a scalar is an example of this class of operation.

What Figure 3.3 does not show is what restrictions there are on the type of operands each of these operations can take, what the result type is, or how this might be determined. This information can be encoded though the use of template parameters which represent the type of the operands and the result. For example, a binary operation would have template parameters determining the types of the two operands, and the result type. The matrix-vector multiply operation would not need any template parameters, as the operand and result types are well defined. By using template parameters, a single class template can be used to instantiate classes representing operations between different types. For example a Pairwise class template can be used to instantiate classes represent an addition operation between both vectors and matrices.

3.4 Expression DAG Creation, Evaluation and Deletion

The references held by expression nodes, handles and the operations that occur during evaluation need to be well defined. We have every handle hold a reference to a single expression node. This represents the current value of the handle. Characteristics of expression nodes are:

- They can either be literals, representing an actual value of a scalar, vector or matrix, or they can represent operations between them such as a dot product, cross product, addition etc. These operations never mutate the value of the operands and have already been described.

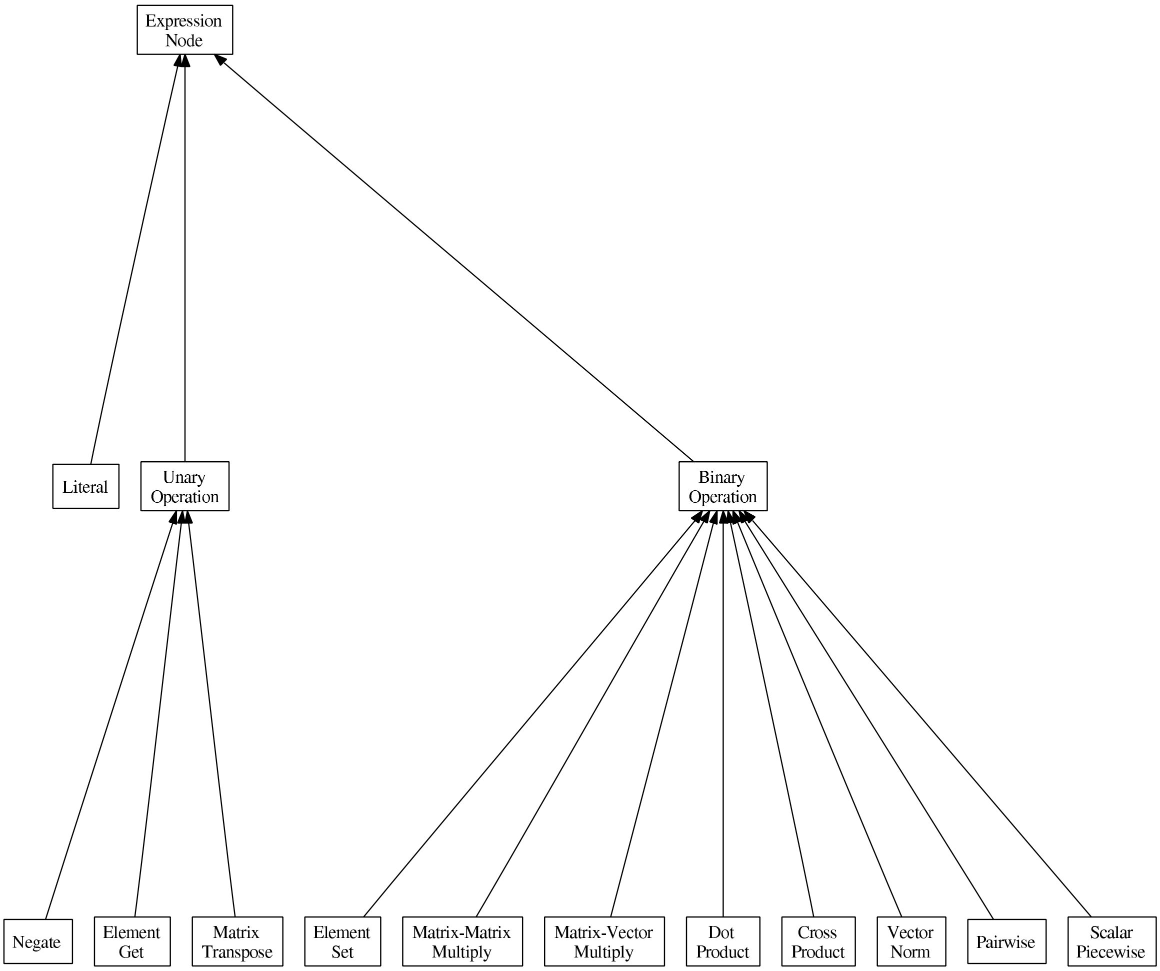


Figure 3.3: The expression node hierarchy.

- Literals contain references to the data in a vector, matrix or scalar, as well as its storage format. The expression nodes representing operations do not contain storage format information. This allows for the possibility of an appropriate storage representation being chosen for a node when it is evaluated.
- Expression nodes register with the expression nodes that they depend on.
- When an expression node is no longer required by any other expression nodes and not referenced by any handles, it deletes itself.
- When an expression node deletes itself, it will unregister from all the expression nodes it depends on.
- Expression nodes hold references to the handles that reference them, the expression nodes they depend on and the expression nodes that depend on them. Having a node hold references to the nodes that depend on it makes it simpler to work out which nodes to update when it is replaced in the expression DAG, and also makes it possible to use DAG exploration, discussed later, to find related expressions.
- Evaluating an expression node replaces it with a literal.
- When an expression node is evaluated, the handles and expression nodes referencing it modify their references to point to the literal representing the evaluated expression, and the original node is deleted.
- Assignment and copying of handles only create more references to the same expression node.
- Expression nodes are evaluated when their result is required. This can occur when their value is requested by the client though the handle's interface, or as part of the evaluation of some node that depends on the expression node.
- When an expression node is evaluated, the nodes it depends on will be deleted if there are no other nodes or handles depending on them. Otherwise, they will also become literals.

In order to evaluate a delayed expression, it is necessary to have references to the nodes in the expression being evaluated. Given some handle, it is always possible to do this as expression nodes always hold references to the nodes they depend on.

However, given that we wish to maximise the opportunities for loop fusion and array contraction during the evaluation of some expression, it might be more beneficial to evaluate more nodes that just those required for expression being evaluated. Consider Figure 3.4. It is highly likely that the loops representing the vector additions required for A and B could be fused, and that this could improve performance though the presence and reuse of Vec2 in the cache. It seems plausible that if related expressions are evaluated at the same time, the reuse of cached data could increase, resulting in better performance. However, evaluating C at the same time would most likely increase contention in the cache without providing any benefit.

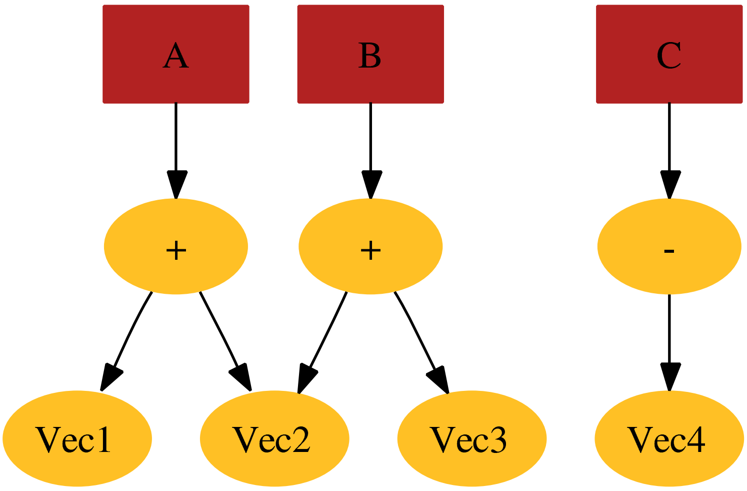


Figure 3.4: It could improve performance to evaluate the values of A and B at the same time as they share the operand Vec2.

Thus, we arrive at some heuristics which hopefully improve performance, but could possibly degrade it, depending on what we are evaluating:

1. Delay evaluation as much as possible so that as many opportunities for loop fusion can be found.

2. When evaluation is forced, evaluate as many related expressions as possible to try to increase chances of improved cache use.

3. Avoid evaluating unrelated expressions together as they will not assist reuse of cached data.

Given that we want to find related expressions to the one being evaluated, we have a few possibilities:

Factories Force handles to be created through factory objects which then hold references to the expression nodes created though the interactions between the handles. The result is that factories will contain references to expressions generated in the same part of the program and if evaluated together, could improve performance.

The main problem with this method is that software using this technique must now handle the issues surrounding the creation and passing around of the factory objects. IML++, for example, creates large numbers of local scalars and vectors and would need to be rewritten to use factories. Many linear algebra libraries do not require factories to be used and modifying any application that uses one of those libraries to use the prototype library would require more effort to enable it use the factories correctly. Also, if factories were used to produce nodes for unrelated expressions, it could lead to degraded performance as unrelated expressions become associated and evaluated with each other.

Global Register Have a global set of all unevaluated nodes in the program. This may work for single threaded applications, but with multi-threaded applications, it is no longer simple to determine which nodes are related to each other. Furthermore, synchronisation could become an overhead.

DAG exploration As expression nodes hold references to both the nodes they depend on, and the nodes that depend on them, it is possible to perform a complete exploration of all connected nodes in the DAG. These most likely represent related expressions. This does not affect the library's interface, and avoids detecting relationships between unrelated expressions.

I chose DAG exploration as the technique to detect related nodes as it seemed to provide the most effective way to locate related expression nodes without providing any burden on the client of the library.

3.5 Matrix, Vector and Scalar Representation

In deciding what storage formats to support, BLAS was the obvious choice to examine. BLAS supports:

- Single and double precision numbers
- Real and complex numbers
- Specialised routines for general, symmetric and triangular matrices
- Routines handling matrices in conventional, packed and banded storage formats

Although BLAS does not support sparse matrices, an extension to BLAS called Sparse BLAS has been proposed.

It was decided for simplicity to initially target real, general matrices in conventional format, and only to target other formats if they became of further interest to the investigation. Furthermore, these formats can be easily represented by 1D and 2D arrays, leading to loops that are more likely to be fusible.

As stated before, literals in the expression tree contain references to the actual values of scalars, vectors and matrices. For the purposes of the prototype library, I decided to make these values immutable. I did this because nodes in the expression DAG already represented immutable values and operations. Attempting to map these back onto mutable operations could help improve memory reuse but I considered it beyond the scope of this project. Furthermore, it is my hope that the TaskGraph representation will enable array contraction to occur, which would allow the arrays representing temporary vectors and matrices to be reduced to scalar values thus removing the copying overhead.

Chapter 4

Implementing Delayed Evaluation and Runtime Code Generation

4.1 The Interface

Originally, it was intended that the prototype library satisfy the requirements for the IML++[6] set of iterative solvers. Research showed that an updated version of IML++ had been developed, called the Iterative Template Library[12](ITL). ITL extends the ideas of IML++ to make the library more flexible.

As with IML++, ITL defines templated methods that implement a number of different iterative solvers. It also requires that the client library provides matrix and vector implementations, and optionally preconditioner objects.

Unlike IML++, ITL does not define member methods of the matrices and vectors used by the library. Instead, it defines a number of methods that can be called with these matrices and vectors as parameters. This allows ITL to work with different linear algebra libraries without modification. IML++ would require that the libraries either be modified or use a wrapper to provide the appropriate interface.

ITL benchmarks also indicate that it achieves a 2X speedup over IML++. It is worth noting that one of the ITL interface methods include a delayed vector scaling operation, something the prototype library supports implicitly. Most importantly, ITL includes an interface for the Matrix Template Library allowing for benchmarking of the prototype library against state of the art C++ numerical template metaprogramming.

The majority of the methods used by the iterative solvers are listed in Table 4.1. In addition to various methods, ITL also requires certain typedefs defined in Traits[1] classes. A couple of iterative methods required more specialised features such as the *upper_tri_solve* method described in Table 4.1 and the ability to retrieve column vectors from a matrix. Even without satisfying some of the more specialised interface requirements, the prototype library was able to

[1] A Traits class is a commonly used tool in C++ template programming. It allows compile types or constants to be mapped to other types or constants.

Method	Description
mult(A, x, y, z);	x = y + A * x
mult(A, x, y);	y = A * x
trans_mult(A, x, y);	y = A^T * x
scaled(x, alpha);	Delayed scaling of vector x by alpha
size(x)	The dimension of vector x
solve(M, x, y);	Solve preconditioner system
trans_solve(M, x, y);	Solve transpose preconditioner system
dot(x, y);	Inner product: x * y
dot_conj(x, y);	Conjugate inner product: x * y^T
add(x, y, z);	z = x + y
copy(x, y);	Copy elements of vector x to y
two_norm(x);	The two norm of x
add(x, y);	y += x
add(x, y, z);	z = x + y
upper_tri_solve(A, x, i);	Backward substitution for upper triangular part of Hessenberg matrix

Table 4.1: The ITL interface. All methods are templated. Uppercase parameters denote matrix objects, lowercase denote vector objects, and alpha donates a scalar value.

provide delayed matrix, vector and scalar implementations for the following 8 iterative solvers:

- Conjugate Gradient
- Conjugate Gradient Squared
- BiConjugate Gradient
- BiConjugate Gradient Stabilised
- Quasi-Minimal Residual
- Transpose Free Quasi-Minimal Residual
- Chebyshev Iteration
- Preconditioned Richardson

Another important part of the ITL was its set of preconditioners. Unfortunately, their implementation was specific to MTL and proved complex to understand. For this reason, I decided not to implement the preconditioners in the prototype library, and instead test the solvers using the identity preconditioner.

4.2 Expression DAG

An expression DAG similar to that discussed in the design was implemented. Each class used in the DAG has at least one template parameter, the type of the elements or element held by the matrix, vector or scalar. All other template parameters have the type *ExprType* which represents the different types of linear algebra object.

```
enum ExprType
{
  scalar,
  vector,
  matrix
};
```

For example, take the declaration of BinOp:

```
template<ExprType resultType, ExprType leftType, ExprType rightType,
         typename T_element>
class BinOp;
```

The BinOp class is templated by three parameters which represent the types taken by the binary operation and the type of the result. It also has the parameter *T_element* which is the type of elements of both the operands and the result.

A consequence of BinOp only having one *T_element* parameter is that both the operands must have the same element types. Hence, it is impossible to represent the type promotion that could occur with the addition of a vector of doubles and a vector of floats.

However, the number of type parameters of the expression DAG nodes is not the main reason why it is not possible to represent type promotion. In order to be able to process the DAG of delayed evaluations, it was important that a *Visitor* could be defined capable of being accepted by the nodes in the DAG.

Interfaces in C++ are typically defined using pure virtual methods. Ideally, it would be possible to define an interface in this way:

```
class DAGVisitor
{
public:

  template<typename T_element>
  virtual void accept(MatrixMult<T_element>& m) = 0;

  template<ExprType exprType, typename T_element>
  virtual void accept(Pairwise<exprType, T_element>& p) = 0;

  ...
};
```

However, C++ does not support templated virtual methods. We can cope with the element type parameters by making them a type parameter of the visitor class itself. This is why all nodes in the expression DAG must have the same *T_element* parameter. As for the *ExprType* parameters, we define methods for all possible values. The interface described above becomes:

```
template<typename T_element>
class DAGVisitor
{
public:
  virtual void accept(MatrixMult<T_element>& m) = 0;

  virtual void accept(Pairwise<scalar, T_element>& p) = 0;
  virtual void accept(Pairwise<vector, T_element>& p) = 0;
  virtual void accept(Pairwise<matrix, T_element>& p) = 0;

  ...
};
```

A class of DAGVisitor<double> can only be used to visit an expression DAG where all the nodes represent expressions involving scalars, vectors and matrices of doubles.

4.3 Expression DAG Evaluation

An expression DAG evaluation occurs when the value of an expression node is requested by a handle pointing to it. It can be separated into three main phases:

Expression DAG Exploration This involves performing a search to find all nodes that need to be evaluated in order to evaluate the node pointed to by the handle, as well as other nodes that might benefit from being evaluated at the same time.

Evaluation Strategy Creation This involves deciding how each node in the expression DAG should be evaluated. Issues that need to be considered include deciding the storage layouts of the scalar, vector and matrix literals created and determining which nodes to be evaluated do or do not require memory to be allocated for them. Also, whilst we are only considering evaluation of nodes using TaskGraph generated code, it could be possible to evaluate the nodes using other methods, such as calls to BLAS.

Evaluation Strategy Execution and Expression DAG Rewriting Memory is allocated for the results of the expression node evaluations, the previously generated code is executed, and the expression DAG is rewritten with Literals replacing the nodes that have been evaluated.

4.3.1 Expression DAG Exploration

First all nodes to be evaluated need to be found. This proceeds as follows:

- Perform a depth-first search from the node to be evaluated. The depth-first search follows the DAG dependency arcs in both directions so evaluation of the node pointed to by handle A in Figure 4.1 will find all reachable nodes in the DAG and evaluate the value of handle B as well.

- Of all the nodes found in the first step, only those that have no expression nodes depending on them are retained. These are the roots of the expression DAG. In Figure 4.1 these are the nodes pointed to by handles A and B. The root nodes in a DAG will always have handles pointing to them because otherwise, they would have no handles or nodes depending on them at all and would have deleted themselves.

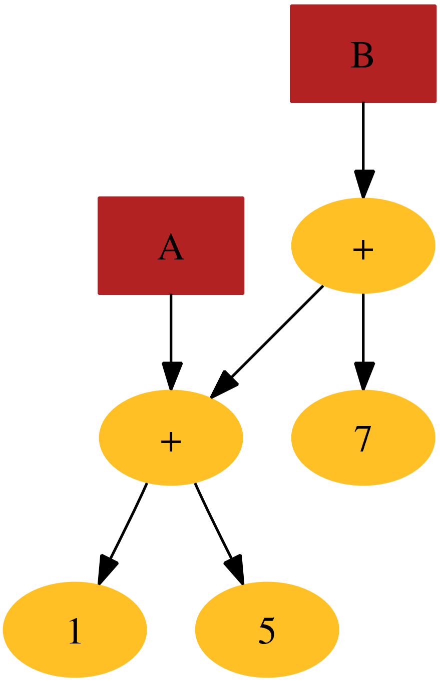


Figure 4.1: A possible DAG. Evaluation of expression node pointed to by handle A will result in the evaluation of both the expression nodes pointed to by handles A and B.

- A depth-first search of the dependencies of the root nodes is performed and the nodes are recorded in the order they are discovered.

The result of this algorithm is a list of all nodes reachable from the node to be evaluated, in topological order. Reversing this list gives us a possible order for the evaluation of the nodes in the graph. It should be noted that the topological sort and hence the order of evaluation is non-unique. This list is used to construct an *ExpressionGraph* object which is basically a list of expression nodes in a legal evaluation order.

4.3.2 Evaluation Strategy Creation

To allow for the possibility of evaluation of expression nodes by other methods than TaskGraph generated code, I decided to invent an interface, *Evaluator*. *Evaluator*s are objects capable of evaluating nodes in an expression DAG. To allow for the possibility of having more than one way to evaluate an expression DAG, and to be able to choose which one to use, I decided to create a class *EvaluationStrategy*. An *EvaluationStrategy* represents a particular way to evaluate an expression DAG.

Once the *ExpressionGraph* has been created, finding a way to evaluate it proceeds as follows:

1. An *EvaluationStrategy* is created. It takes an *ExpressionGraph* as a parameter, the graph for which the *EvaluationStrategy* will provide a possible way to evaluate.

2. The *EvaluationStrategy* is passed factories which it can use to create *Evaluator*s.

3. The *EvaluationStrategy* uses the factories to create an *Evaluator* and offer it nodes from the expression DAG. The nodes are offered as a list in a legal execution order to simplify work for the *Evaluator*. The evaluator returns the nodes that it agrees to evaluate.

4. The *EvaluationStrategy* gets the *Evaluator* to create the Literals that represent the evaluated nodes. The Literals contain information about the storage representation of the data they will contain, but do not yet reserve memory for it. The *EvaluationStrategy* maintains a map of expression DAG nodes to the Literals they will evaluate to. This will be used later for rewriting the expression DAG. An *Evaluator* only needs to return Literals for nodes that have handles with references to them, or that have nodes depending on them that are not being evaluated by the same *Evaluator*. In this way, *Evaluator*s claim sections of the expression DAG, and are free to optimise away any node that does not have references to it outside the subgraph claimed by the *Evaluator*.

5. The *EvaluationStrategy* must find an evaluator for every node in the expression DAG before it can be used to evaluate it.

As the *EvaluationStrategy* builds up a mapping of *Evaluators* to expression nodes, each new *Evaluator* may need to reference the value of an expression node that has not yet been evaluated. The *EvaluationStrategy* uses its map of expression nodes to Literals to provide this.

Once an *EvaluationStrategy* has been created, and has decided how to evaluate all the nodes for an *ExpressionGraph*, it can be executed. Alternatively, it can be deleted, and all created *Evaluator*s and Literals will be deleted.

In the current implementation, there are only two evaluators:

- A *LiteralEvaluator* which agrees to evaluate all literals, and does not need to do anything.
- A *TGEvaluator* which uses TaskGraph to generate code to evaluate the nodes it claims.

4.3.3 Strategy Execution and Expression DAG Rewriting

Execution of an *EvaluationStrategy* has three parts:

- Allocate memory for the Literals storing the results of each *Evaluator*.
- Execute each *Evaluator* in the order it was added to the *EvaluationStrategy*.
- Rewrite the expression DAG, replacing each node in the DAG with its corresponding Literal.

It is important that the DAG is rewritten in a specific order, otherwise, a node in the DAG being replaced may have already deleted itself. This could occur if all nodes depending on a node have already been replaced with Literals.

4.4 The TaskGraph Evaluator

The TaskGraph evaluator, implemented in the class *TGEvaluator* is responsible for generating the TaskGraph code to evaluate an expression DAG. The first problem faced when writing the TaskGraph evaluator was the latest version of TaskGraph's mechanism for passing parameters to the generated code.

4.4.1 Initial Problems

The types of the parameters and return value were specified using template parameters. This meant that the type of parameters, the type of the return value, and the number of parameters needed to be known at compile time. Figure 4.2 shows example code using this mechanism. This was unacceptable for the prototype library as it needed to be able to generate code taking arbitrary numbers of parameters depending on runtime context.

Research of previous TaskGraph papers revealed that this had not always been the case. It was previously possible to define the number of parameters of the TaskGraph at runtime, but without the benefits of type safety. I was able to get Michael Mellor, who was doing research involving TaskGraph, to supply me with a version of the version TaskGraph system with the old type-unsafe parameter passing mechanism restored. It was now possible to write the program from Figure 4.2 as in Figure 4.3.

```
typedef TaskGraph< Par<int>, Ret<int> > addc_TaskGraph;

int addOne(int a)
{
  addc_TaskGraph T;           // Defines a TaskGraph T

  // Adds code to TaskGraph
  taskgraph(addc_TaskGraph, T, tuple1(number))
  {
    tReturn(number + 1);
  }

  T.compile(tg::GCC, true);   // Compiles TaskGraph
  return T.execute(value);    // Executes code and returns value
}
```

Figure 4.2: A simple procedure that uses a type-safe TaskGraph to generate and execute code to add 1 to a value. Parameter and return types are specified using templates.

```
int addOne(int a)
{
  tuTaskGraph T;             // Defines a TaskGraph T

  // Adds code to TaskGraph
  tuTaskGraph(T)
  {
    tParameter(int, num);
    num = num + 1;
  }

  T.compile(tg::GCC, true);   // Compiles TaskGraph
  T.execute(&a);              // Executes code
  return a;                   // Return modified a
}
```

Figure 4.3: A simple procedure that uses a type-unsafe TaskGraph to generate and execute code to add 1 to a value. The address of a is passed to the generated code, and the actual value of a is modified. This is different to the code in Figure 4.2 in which a is passed to the TaskGraph by value, and the result returned by the *execute* statement of the TaskGraph.

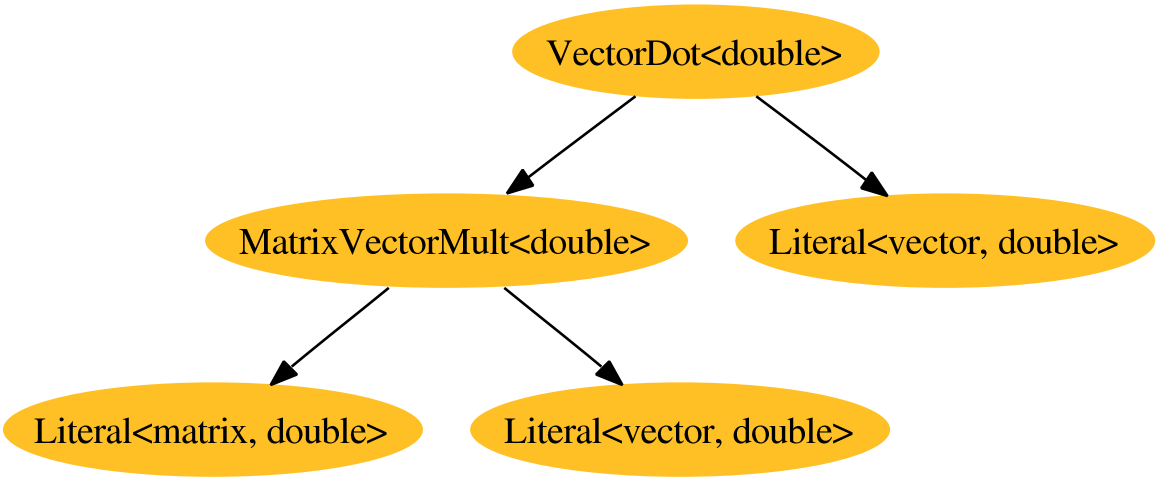


Figure 4.4: The TaskGraph evaluator will assume responsibility for the evaluation of all expression DAG nodes passed to it except Literals. Given this expression DAG, it will construct the TaskGraph DAG in Figure 4.5.

4.4.2 Basic Operation

With a version of the TaskGraph system that allowed parameter types and numbers to be determined at run-time, it was possible to write the TaskGraph evaluator.

The TaskGraph evaluator claims all expression DAG nodes it is offered. This is because it is the only *Evaluator* in the current system (apart from the trivial *LiteralEvaluator*). As the nodes offered to the *Evaluator* are already in a legal execution order, it does not need to worry about reordering their evaluation and stores them in a list.

The TaskGraph evaluator then visits all the nodes it claimed and builds another DAG. This DAG will have an identical structure to the nodes in the original DAG, but uses a different hierarchy of nodes. I will call this a TaskGraph DAG. Consider the expression DAG in Figure 4.4.

The nodes in the TaskGraph DAG store the following information:

- The other nodes in the TaskGraph DAG that they depend on.
- The storage format of the object represented by the node. This is different to the primary expression DAG, which only stores storage format representation information for Literals. Hence, the TaskGraph Evaluator has the flexibility to choose appropriate storage formats for the matrices and vectors allocated inside the TaskGraph, and those it returns.

The storage format representation information in the TaskGraph DAG includes:

- The name of the scalars or arrays that represent the data inside the TaskGraph. Conventional storage of matrices and vectors only require one array to hold them, but further extensions such as sparse matrices could require more.

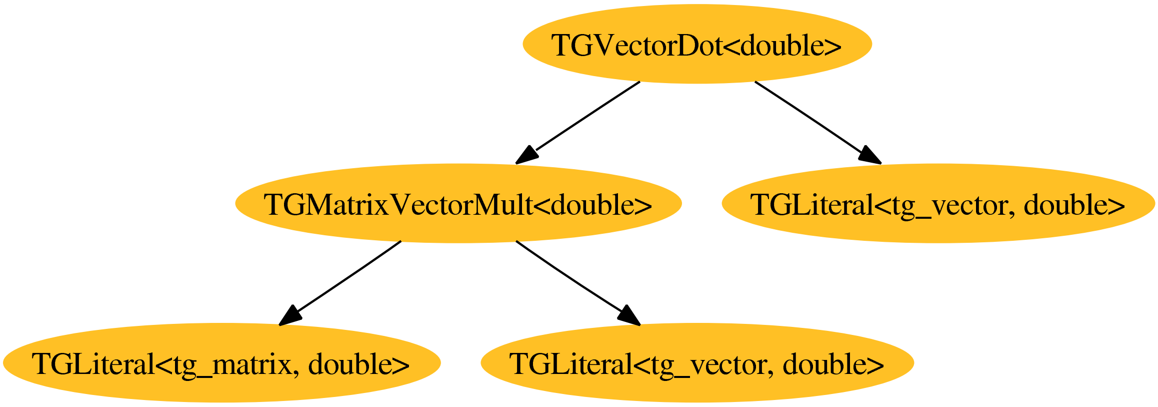

TGVectorDot<double>
TGMatrixVectorMult<double>
TGLiteral<tg_vector, double>
TGLiteral<tg_matrix, double>
TGLiteral<tg_vector, double>

Figure 4.5: The TaskGraph DAG created from the expression DAG in Figure 4.4. The TaskGraph evaluator does not assume responsibility for evaluating Literals. The *TGLiteral* class is used to represent any Literal evaluated by a different *Evaluator*.

- The TaskGraph storage declarations for the data. Conventional storage stores vectors and matrices as 1D and 2D arrays respectively.

- The dimensions of the object represented.

- Whether the object is allocated inside the TaskGraph, or passed in as a parameter.

Once the TaskGraph DAG has been created, it is then passed a visitor which performs the code generation. Visiting each node in the TaskGraph DAG generates code for that particular operation. As each node in the TaskGraph DAG contains information about the storage format of the data represented by that node, the generated code can be specialised with constant loop bounds which makes optimisations like loop fusion easier to perform.

The result is a TaskGraph capable of evaluating the expression DAG, and a TaskGraph DAG which contains both a high-level representation of the operation performed by the TaskGraph, and the information required to create a mapping between TaskGraph parameter names and the memory locations of the data to be passed into the TaskGraph.

Once the TaskGraph has been created, there is one last thing that must be done before it can be executed. All the memory to be passed into the TaskGraph as parameters to the TaskGraph needs to be mapped to TaskGraph parameter names.

As there is a one-to-one mapping between expression DAG nodes and TaskGraph DAG nodes, each expression DAG node is passed to a function *createParameterMapping* called on the corresponding TaskGraph DAG node. A map of parameter names to pointers to memory locations is built up, and the TaskGraph is ready to be executed.

4.4.3 TaskGraph Imposed Limitations

C++ provides many features that simplify the creation of mathematical software. One template available in the Standard Template Library is the *complex* template, which simplifies the passage and usage of complex numbers. Another is the *vector* template which allows sequences of elements to be stored in contiguous memory and provides automatic memory management. As well as providing a simple way to store arrays and matrices in conventional layout, their ability to be resized also allows them to be used to represent sparse matrices and vectors. The Matrix Template Library makes extensive use of similarly designed container classes to represent both its sparse and dense objects.

Unfortunately, it is not possible to replicate this approach in the prototype library. The TaskGraph library is designed to work with statically sized objects and does not support any form of dynamic memory allocation. Creation of sparse matrices would require another *Evaluator* capable of performing dynamic memory allocation. However, it would not restrict the usage of the TaskGraph Evaluator for evaluating a sparse matrix-vector multiply, as the memory for the sparse matrix would have already been allocated.

Clearly the prototype library requires that any storage format used can be represented in the TaskGraph system. The version of TaskGraph used supports scalars, and multidimensional arrays. For this reason, storage of data outside the TaskGraph has been done with scalars, and statically sized *vector* objects. It is simple to map STL *vector* objects to TaskGraph arrays as they allocate their memory contiguously and provide a simple way to locate the starting address. As for representing complex numbers, using arrays or *vector*s of the *complex* template could lead to alignment and padding issues when passed to the TaskGraph. A simpler solution would be to follow each number's real part by its imaginary part in any array, doubling the array's final dimension. Complex numbers have not yet been implemented in the prototype library.

4.4.4 Towards Storage Format Independent Code Generation

One desirable feature of the Matrix Template Library is that the code for all algorithms, such as a matrix-vector multiply, has been written in a way that is independent of storage format. This means that the same algorithm can be used for the multiplication of a sparse matrix or a dense matrix by a vector. Consider the code in Figure 4.6 for a matrix-vector multiply in MTL. Traversal of the matrix and vectors is abstracted away using iterators, and an *index* method is used to obtain index values when they are required. The result is a matrix-vector multiply that will perform effectively on both dense and sparse matrices, as the iterators over the matrix can traverse only the non-zero elements.

If sparse matrices, or other matrix and vector storage layouts were added to the prototype library, it would also be helpful if the code generation could adapt itself to the new storage representations.

Consider the TaskGraph code generated in Figure 4.7. The generated code is specific to dense matrices arranged in Row-major order. *TGVector* and *TGMatrix* are classes representing the memory blocks involved in the matrix-vector multiply. The *addExpression* and *setExpression* methods generate the code required to add and set elements in the vectors and matrices involved. The expres-

```
template<class Matrix, class IterX, class IterY>
void matvec::mult(Matrix A, IterX x, IterY y)
{
  typename Matrix::row_2Diterator i;
  typename Matrix::RowVector::iterator j;

  for(i = A.begin_rows(); not_at(i, A.end_rows(); ++i)
  {
    typename Matrix::PR tmp = y[i.index()];
    for(j=i->begin(); not_at(i, i->end()); ++j)
    {
      tmp += *j * x[j.index()];
    }
    y[i.index()] = tmp;
  }
}
```

Figure 4.6: A generic matrix-vector multiply for the Matrix Template Library. Parameter A is the matrix, x is an iterator to the vector being multiplied and y is an iterator to the result vector.

```
virtual void visit(TGMatrixVectorMult<T_element>& e)
{
  using namespace tg;

  // Get references to the matrix, vector and result vector
  TGVector<T_element>& result(e.getInternal());
  TGMatrix<T_element>& matrix(e.getLeft().getInternal());
  TGVector<T_element>& vector(e.getRight().getInternal());

  tVarNamed(int, i, getIndexName());
  tVarNamed(int, j, getIndexName());
  tFor(i, 0, matrix.getRows()-1)
  {
    result.setExpression(i, TGScalarExpr<T_element>());
    tFor(j, 0, matrix.getCols()-1)
    {
      result.addExpression(i, matrix.getExpression(i, j).
                              mul(vector.getExpression(j)));
    }
  }
}
```

Figure 4.7: A implementation of a method in the prototype library to generate TaskGraph code for a matrix-vector multiply.

sions passed around are not TaskGraph expressions, but rather *TGScalarExpr* objects, which act as a simple wrapper around TaskGraph expressions, but allow for future expansion to complex numbers, which TaskGraph cannot represent directly.

Clearly, this approach will generalise to other storage representations, as access to the matrices and vectors is abstracted, but will generate inefficient code, as all elements of a sparse matrix or vector will be traversed, non-zero or otherwise, and it will always be done in Row-Major order.

We need an approach that will allow the matrix or vector being iterated over to control the way in which code is generated for it to provide efficient access. I will now discuss a suggested approach to this problem. It has not been implemented in the prototype library due to time constraints and the fact that the only storage representations supported are dense and perform well with Row-major traversal.

Instead of having the operation using the matrices and vectors generate the traversal code, have the matrix or vector generate it itself. In an operation involving multiple matrices or vectors, a decision needs to be made with regards to which operand should control the data traversal. In the case of a matrix-vector multiply, it makes sense for the matrix to control traversal, as an inefficient matrix traversal would most likely cause poorer performance than poor vector traversal.

The suggested way of doing this is shown in Figure 4.8. In this implementation the matrix-vector multiply is represented as function object which is passed to the matrix. The matrix is responsible for generating the TaskGraph code to traverse all the elements in the matrix, and inside this, it calls the function object which generates code for the multiply itself. When the function is called, it is provided with the row and column of the element in the matrix, and the value of the element itself. The function then uses these parameters to generate the appropriate code.

In this chapter, I have detailed the implementation of the delayed expression DAG. I have described the framework for enabling different parts of expression DAGs to be evaluated using different techniques. I have explained the workings of the TaskGraph *Evaluator* and the way it stores information. I have suggested ways in which it could be enhanced to provide code generation for different storage formats without modification. In the next chapter, I will now discuss the optimisations applicable at runtime that were used to enhance the prototype library's performance.

```
virtual void visit(TGMatrixVectorMult<T_element>& e)
{
  using namespace tg;

  // Get references to the matrix, vector and result vector
  TGVector<T_element>& result(e.getInternal());
  TGMatrix<T_element>& matrix(e.getLeft().getInternal());
  TGVector<T_element>& vector(e.getRight().getInternal());

  MatrixVectorMultiplyFuction func(vector, result);
  matrix.acceptIteratorFunction(func);
}

template<T_element>
class MatrixVectorMultiplyFunction
{
private:
  TGVector<T_element>& operand;
  TGVector<T_element>& result;

public:
  MatrixTraversalFunction(TGVector<T_element>& op, TGVector<T_element> res) :
                          operand(o), result(res)
  {
  }

  void operator() (tg::TaskExpression row, tg::TaskExpression col,
                   TGScalarExpr<T_element> value)
  {
    result.addExpression(row, value * vector.getExpression(col))
  }
};
```

Figure 4.8: A possible way to generate TaskGraph code for a matrix-vector multiply with efficient matrix traversal.

Chapter 5

Runtime Optimisations

I will now describe some of the further developments made in the prototype library after the initial working version was completed.

5.1 Code Caching

5.1.1 The Problem

Comparing the performance of the iterative solvers using MTL to the iterative solvers using the prototype library showed that the solvers using MTL performed significantly faster.

I ran the Conjugate Gradient solver using MTL and the prototype library. GCC was used at its highest optimisation setting for both compilation and runtime code generation. Performing 256 iterations with a 3312x3312 matrix on a Mobile 2GHz Athlon64, the MTL solver took 13 seconds. The prototype library solver took 55 seconds. The other iterative solvers performed similarly poorly.

Looking at the source files generated by the prototype library, it was possible to determine that GCC had been invoked 516 times during the execution of the iterative solver. This corresponded to two GCC invocations per iteration. A similar test with the BiConjugate Gradient Stabilised solver, indicated that GCC had been invoked 5 times each iteration. In fact, it was not even possible to perform 256 iterations with this solver because TaskGraph had opened too many dynamically generated libraries[1].

Analysis of the generated code showed that almost identical code files were being repeatedly created and compiled during the execution of the iterative solver. This was clearly as problem as multiple redundant compiler invocations were being performed when existing code could be reused.

In order to overcome this, it was decided that code caching should be implemented, in which the creation of identical code could be detected, and previously compiled code reused. Beckmann and Kelly used a similar approach when they cached execution plans for a DESO distributed linear algebra library[5].

It might seem obvious that code caching would be required, especially given the repetitive nature of the operations performed by the iterative solver, and

[1] This was found to be due to a bug in SUIF, the TaskGraph back-end

the fact that an iterative solver will always have at least one force point each iteration as it checks for convergence. However, experiments using the TaskGraph library such as the image filtering analysis performed by Beckmann et al.[4], have shown reduced overall execution time when using TaskGraph, even with the overhead of compiler invocation.

An important feature of the image filtering example is that the generated code has been heavily specialised. The values of the convolution matrix being used are incorporated as compile time constants into the TaskGraph generated code, allowing loop unrolling to occur. In comparison, the code generated by the prototype library is only specialised by loop bounds, so the compiler overhead must be reclaimed though other techniques, such as code caching, and the generated code optimised though other means, such as memory access transformations.

The increased specialisation of code also has disadvantages. The more specialised the code, the less likely it is that it can be reused. This could result in greater numbers of compiler invocations for similar code. Less specialised code would allow greater reuse but possibly at the cost of performance.

5.1.2 The Solution

In order to perform compiled code reuse, it was necessary to be able to discover isomorphisms between the TaskGraph DAG being evaluated and previously generated and compiled TaskGraph DAGs. Furthermore, I wanted to avoid performing multiple isomorphism checks between the TaskGraph DAG being evaluated, and previous TaskGraph DAGs.

DAG isomorphism is equivalent to graph isomorphism is general. Graph isomorphism is a well studied problem, although not known to be solvable in polynomial time nor known to be NP-Complete. Although, given restrictions on the graph such as bounds on its degree or that it is planar, polynomial time algorithms exist.

As it was essential that an isomorphism could be found as quickly as possible, the following decisions were made:

1. Each TaskGraph DAG would have a hash value. Using a map of hash values to TaskGraph DAGs, it would be possible to find likely candidates for isomorphism. Each of the candidates would be checked for isomorphism against the TaskGraph DAG being evaluated. If a match was found, the already compiled and optimised code would be used to execute the operation.

2. Both the hash value and the equality checking between TaskGraph DAGs would be specific to the order nodes were stored in the TaskGraph DAG class. The TaskGraph DAG nodes are stored in the order they were created, which is the order that the corresponding nodes from the expression DAG were offered to the TaskGraph evaluator by the *EvaluationStrategy* class.

Whilst this strategy is computationally simple, it does require that when the expression DAG being evaluated is flattened to a legal execution order, it is always done the same way. For this reason, all dependency tracking in the expression DAG was was rewritten to use vectors, which preserve order unlike sets. This ensured that the various depth-first searches used to generate the evaluation order from the expression DAG would always give the same result.

Hashing was implemented as follows:

- Each node in the TaskGraph DAG is assigned a unique integer value corresponding to its position in the list it is stored in.
- A visitor that calculates the hash of a TaskGraph DAG node visits each item in the list.
- The hash value of each node is calculated using the C++ runtime type information as well as other information such as the storage representation of the value represented by the node, and the names of the TaskGraph declarations. References to other nodes are encoded by the integer vales assigned to them.
- The hash values of each node in the list are combined in order using a non-commutative operator.

Equality checking between TaskGraph DAGs is implemented in a similar way:

- A check is made that both DAGs have the same number of nodes.
- A map is created that holds the correspondence between the nodes in TaskGraph DAGs being compared.
- An equality checking visitor visits each node in one of the TaskGraph DAGs. It checks that if the corresponding node in the other DAG has the same type, that all other information in the node matches.
- The visitor also checks that the references held by the node to other nodes in the TaskGraph DAG are consistent to those held by the corresponding node in the other TaskGraph DAG, with respect to the mapping between TaskGraph DAGs.

Using these algorithms for hashing and equality checking, it is possible to find previously generated isomorphic TaskGraph DAGs without a computationally expensive algorithm. The algorithms for hashing and isomorphism are linear in their number of visits to each node, but are of polynomial complexity due to the Standard Template Library maps used, which have logarithmic lookup time. This could be amended by using hashmaps when they make it into the C++ STL. The algorithm's primary disadvantage is that it requires dependency information held in the expression DAG to be ordered. This means that dependency information in the expression DAG is now held in vectors instead of sets, which are slower to update when rewriting the expression DAG.

5.2 Loop Fusion

Loop fusion 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 loops displace each other in the cache.

The TaskGraph library already contained a loop fuser. Whilst SUIF (the TaskGraph back-end) contains quite developed loop dependence analysis tools, it doesn't have a full dependence framework, and for this reason the loop fuser was quite rudimentary. The existing TaskGraph loop fuser proceeded as follows:

1. For each loop and the next one following it in the same scope.

2. Check that the loops have constant bounds and identical tests.

3. Check neither loop nor the code in-between involves jump statements.

4. Check that there are no dependencies between the scalars read and written in the loops.

5. Check there are no scalar or array access dependencies between the second loop and the code in-between the loops.

6. Fuse the body of the second loop onto the first loop.

7. Perform an array dependence analysis of the fused loop.

8. If the dependence vectors between the array accesses from bodies of the fused loop are legal, remove the remainder of the second loop. Otherwise, move the body of the second loop back to its original location.

9. Repeat for further loops in the same scope and for sub-loops.

Analysis of the code produced by the TaskGraph library using this loop fuser showed that some loops had been fused, but a number of easily recognisable fusions that had not occurred. This was because the fuser required that there be no dependencies between the code in-between the loops and the second loop. Figure 5.1 shows the fragment of a C program where the loop fuser would fail. The assignment of 10 to c blocks the fusion, although it is perfectly legal to prepend the body of first loop onto the second.

My first change to the loop fuser consisted of allowing the body of first loop to be prepended onto a second loop as well as the original behaviour as long as the fusion was legal. This allowed greater fusions of adjacent loops to occur, and would enable the loops in Figure 5.1 to be fused.

After further developments of the prototype library, I also experimented with other changes to the loop fuser. I modified the fuser to allow the code in between the loops being considered for fusion to contain other loops which would not be fused. This increased the number of candidates for loop fusion by removing the requirement that the loops to be fused be adjacent.

Given the higher number of candidate fusions, I decided to implement the strategy of only attempting to fuse loops that accessed at least one common array, to try to avoid cache pollution. Unfortunately, given the code in Figure 5.2 this strategy cannot fuse the two nested loops as it cannot move either the assignment to x or initialisation of c to allow the fusion to occur. A similar pattern of code to that in Figure 5.2 was observed in the code generated by the prototype library. For this reason, I implemented a very aggressive loop fuser, that would attempt to fuse as many loop as possible. This fuser enabled the

```
int i;
int a[10];
int b[10];
int c;

for(i=0; i<10; ++i)
  a[i]++;

c = 10;

for(i=0; i<10; ++i)
{
  b[i] = a[i] + 10;
  c++;
}
```

Figure 5.1: A fragment of C code showing two loops the original TaskGraph loop fuser couldn't fuse.

two nested loops to be fused, first by fusing the initialisation of vector *c* onto the initialisation of *a*, allowing the body outer loop of the second matrix-vector multiply to be appended onto that of the first. The fusion of the inner loops can then be performed.

The strategies presented are far from optimal. The aggressive fusion strategy can easily fuse loops in such as way to prevent further beneficial fusions as well as allowing more beneficial fusions to occur. Much research has been done on loop transformations that improve cache locality. Yi and Kennedy[22] describe an algorithm for improving cache locality though a transformation combining loop interchange and fusion that they call *dependence hoisting*. Lim et al.[14] describe an algorithm that maximises parallelism and minimises communication using affine partitioning, a framework which unifies a number of loop transformations.

These transformations need to be implemented in TaskGraph to fully assess the effectives of runtime code transformation. Implementing them in SUIF would require significant effort due to its lack for a full dependence framework. Other possibilities include implementing other back-ends to TaskGraph in which these transformations are available or more easily implementable.

5.3 Runtime Liveness Estimation

When analysing the code generated by the prototype library, it became apparent that a large number of vectors were being passed into the code as parameters. This made it impossible to perform array contraction on them, as array contraction can only be performed on arrays allocated locally to the program. My supervisor and I realised that by creating a system designed to recover runtime information, we had lost the ability to use static liveness information.

Consider the code in Figure 5.4. The evaluation of expression can be delayed until the scaled product is printed. When this evaluation occurs there is still a handle, the variable *product*, to the result of the cross product. Even though this

```
// Initialisation of a
for(int i=0; i<1000; ++i)
  a[i] = 0;

// A matrix-vector multiply
for(i=0; i<1000; ++i)
  for(j=0; j<1000; ++j)
    a[i] = a[i] + mat[i][j] * b[j];

// Arbitrary usage of array a
x = a[0];

// Initialisation of elements of c
for(i=0; i<1000; ++i)
  c[i] = 0;

// Another matrix-vector multiply
for(i=0; i<1000; ++i)
  for(j=0; j<1000; ++j)
    c[i] = c[i] + mat[i][j] * b[j];
```

Figure 5.2: The nested loops for the matrix-vector multiplies are an ideal candidate for fusion as they use both matrix *mat* and vector *b*. Either the initialisation of c or the assignment to x must be moved from between the loops before they can be fused by the TaskGraph loop fuser.

```
// Initialisation of a
for(int i=0; i<1000; ++i)
{
  a[i] = 0;
  c[i] = 0;
}

// A matrix-vector multiply
for(i=0; i<1000; ++i)
{
  for(j=0; j<1000; ++j)
  {
    a[i] = a[i] + mat[i][j] * b[j];
    c[i] = c[i] + mat[i][j] * b[j];
  }
}

// Arbitrary usage of array a
x = a[0];

// Initialisation of elements of c

// Another matrix-vector multiply
```

Figure 5.3: The result of the run of the aggressive fuser on the code from Figure 5.2.

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

Figure 5.4: A function that takes two vectors, finds their cross product, multiplies it by a scalar, and prints the result.

value is dead, and could be allocated locally to the TaskGraph, the prototype library has no way of determining this because the handle still exists and it must assume the value is required. As a result, the memory allocated for *product* cannot be optimised away, even after the loops calculating the product and multiplication are fused.

To combat this, I adopted a similar approach to the one used to perform TaskGraph DAG caching:

- At each force point, a *Profiling DAG* is created sharing the same structure as the expression DAG being evaluated. If the same *Profiling DAG* has been created previously, the hashing and equality checking techniques described previously are used to find it, and this is used instead.
- Each node in the *Profiling DAG* places a liveness monitoring object on its corresponding node in the expression DAG.
- Upon the usage of the value of an expression DAG node, or its deletion without use, the expression DAG node informs all objects monitoring it that it is either live or dead.
- The *Profiling DAGs* builds up statistics on whether each node in the expression DAGs that match it are live or dead.
- Upon evaluation of an expression DAG, the corresponding *Profiling DAG* annotates nodes in the expression DAG as to whether it believes their values are live or dead.
- The *Evaluator* evaluating the annotated nodes is then free to optimise away values that are believed to be dead.
- Should the value of a node believed to be dead actually be needed, then its value will need to be computed again.

The best case scenario for this approach is that the liveness information is consistent, and we never optimise away values that are needed later. The worst case scenario is that we optimise away values that are live, and have to recompute them again later. If we assume that all values with handles to them are live, and only change this after significant statistical evidence to the contrary, then we can reduce the likelihood of this possibility. Hence, we can expect to be able to make decisions about the reliability of the liveness information we recover, and never cause a significant performance hit.

Of course, this approach could also recover runtime liveness information which the compiler could not infer at compile time. However, static liveness information was my primary objective. This approach only works when the same code with the same liveness is executed repeatedly, which is the case with iterative solvers, but may not be the case with other applications. Furthermore, the idea scenario would be for the compiler to be able to convey the static liveness information to the prototype library, in which case, the information would always be correct. I will discuss other possible uses for the runtime liveness information in the Conclusion.

5.4 Array Contraction

Array contraction allows the dimensionality of an array to be reduced, leading to lowered memory usage and better cache utilisation. It is often facilitated by loop fusion. In some cases, an array can be reduced to a single temporary value that is stored in a register and never read or written to memory.

The loop fusion and runtime liveness optimisations were implemented with the expectation that they would increase the number of arrays allocated locally to the runtime generated code which could be optimised away. My supervisor and I were aware that GCC could not perform array contraction but hoped that ICC would, given favourable conditions. Favourable conditions were having arrays defined inside a function scope, and only accessed inside a single (possibly nested) loop.

Unfortunately, analysis of the assembly produced by ICC showed that despite the fact the values written to the candidate arrays for contraction were never read later in the program, they were written anyway. To solve this, I wrote an array contraction pass for SUIF. This pass checked that if expression used to access the i^{th} dimension of an array was always the loop index of the same loop, then the i^{th} dimension of the array would be reduced size 1. Although this was a restricted form of the dependency conditions required for array contraction, it was sufficient to perform all possible contractions in the runtime generated code.

Unfortunately, despite having reduced a number of vectors in the runtime generated code to the size of a single element, memory reads and writes were still being made. I believe this was most likely due to that fact that the single element array was still a pointer, and ICC couldn't reason about it fully. In order to solve this, I used the Porky program from the SUIF distribution. In particular, I incorporated Porky's scalarize pass into the array contraction pass. Porky's scalarize pass converts an array of size n into n scalars, each representing an element of an array. It can only do this if all accesses to the array use constant indices. As this was the case for the contracted arrays, it successfully converted them to scalars. This was sufficient to get ICC to use registers to hold the contracted arrays and never write them back to memory.

Chapter 6

Evaluation

In this section, I will evaluate the different optimisation mechanisms implemented in the prototype library. I will analyse their effectiveness in relation to their purpose, and with respect to the performance of the ITL iterative solvers.

The MTL iterative solvers, and the prototype library's runtime generated code were compiled using the Intel C/C++ compiler[1]. We chose this compiler because it was the most advanced available for the target architecture. The invocations of the Intel C and C++ compilers were supplied the options in Table 6.1.

Results were collected for two platforms. They will be referred to by the name given to their particular machine configuration in the Imperial College Department of Computing:

Rays Pentium IV processor running at 3.2GHz with Hyperthreading. 2048 KB L2 cache and 1GB RAM.

Vertices Pentium IV processor running at 3.0GHz with Hyperthreading. 512 KB L2 cache and 1 GB RAM.

Benchmark results were collected for 2 test runs performed on 9 machines of each platform. This gave 18 results to be averaged for each graph data point. Results were collected for the following iterative solvers:

- Conjugate Gradient Squared (cgs)
- BiConjugate Gradient (bicg)
- BiConjugate Gradient Stabilised (bicgstab)
- Quasi-Minimal Residual (qmr)
- Transpose Free Quasi-Minimal Residual (tfqmr)

The Chebyshev Iteration and Preconditioned Richardson solvers were not tested because they frequently diverged and became unstable. The Chebyshev

[1] The prototype library itself was compiled with g++ with -O3 and -march=pentium4 flags. As the majority of the execution time will be spent in the runtime generated code, this should provide little performance difference from a version compiled with the Intel C++ Compiler.

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 qualifying pointers. The compiler will assume that data pointed to by a *restrict* qualified pointer will only be accessed though that pointer in that scope. As the *restrict* keyword is not used anywhere in the runtime generated code, this should have no effect.
-ansi-alias	Allows icc to perform more aggressive optimisations if the program adheres to the ISO C aliasing rules.
-xW	Generate code specialised for Intel Pentium 4 and compatible processors.

Table 6.1: The options supplied to Intel C/C++ compilers and their meanings.

Iteration solver required additional knowledge of the matrix properties in order to converge and it is possible the Preconditioner Richardson required stronger preconditioning than the identity preconditioner (the only one available). I decided not to benchmark the Conjugate Gradient solver as it was only applicable to symmetric matrices and this test data was not available.

Test data was obtained using sparse matrices from the Harwell-Boeing collection although the solvers themselves are dense. Results were collected for the following matrix sizes: 1224, 1806, 2597, 3948, 4562 and 5005.

A detailed collection of graphs is given in Appendix C. For clarity, I have shown only the graphs that demonstrate a trend or are of particular interest.

6.1 Code Caching

Figure 6.1 shows a the execution time of a number of iterative solvers. These solvers were run with and without code caching and the amount of time spent compiling runtime generated code was measured.

With code caching disabled, significantly more time is spent performing the compilation of the runtime generated code than is spent in its execution. Whilst compilation time is independent of problem size, it would require extremely large problem sizes before a performance benefit could possibly be obtained.

Table 6.2 shows the number of compiler invocations needed to run each iterative solver for 256 iterations with and without code caching enabled. Code caching has enabled the overhead of compilation to be reduced to a constant value, independent of the number of iterations.

Code caching is clearly an effective technique for improving the performance iterative solvers, which execute the same sequence of operations numerous times. It enables performance improvements to be obtained from code whose statically compiled execution time would have been less than the invocation of a compiler and the execution time of the runtime generated code. To obtain these improvements, the cached code segment needs to be executed numerous times until the increased performance of the runtime generated code mitigates the overhead of the initial compiler invocation. If the runtime compiled code has

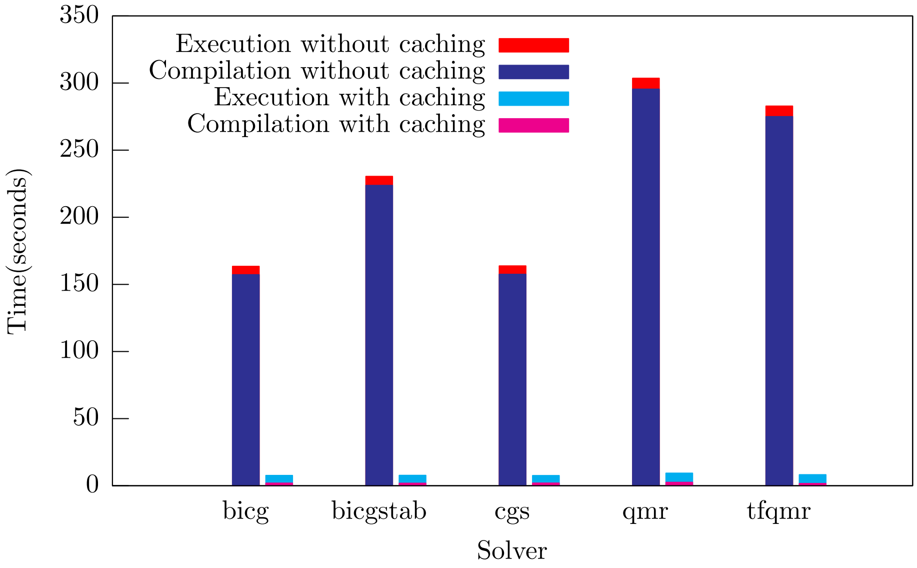


Figure 6.1: Time to execute 256 iterations for a number iterative solvers with and without code caching enabled on a 1806x1806 matrix.

Solver	Total compiler invocations without code caching	Total compiler invocations with code caching
bicg	772	9
bicgstab	1284	10
cgs	772	9
qmr	1540	12
tfqmr	1542	9

Table 6.2: Number of compiler invocations for 256 iterations of different iterative solvers with code caching enabled and disabled.

Solver	Total loop fusions	Loop fusions in repeatedly executed code
bicg	37	12
bicgstab	28	14
cgs	39	13
qmr	37	22
tfqmr	27	17

Table 6.3: Number of loop fusions occurring in each iterative solver. *Total loop fusions* refers to the number of loop fusions performed during the execution of the solver. *Loop fusions in repeatedly executed code* refers to the number of loop fusions in code executed by the solver each iteration.

that same performance as the statically compiled code, the initial compilation overhead cannot be mitigated, and we must rely on other runtime generated code to provide a speedup.

The effectiveness of this technique on other numerical applications also warrants attention. Other numerical applications may not execute repeated segments of code as frequently, making code caching less effective.

6.2 Loop Fusion

As code caching provides guaranteed gains, it makes sense to evaluate the effects of loop fusion in combination with code caching.

The tfqmr, qmr, and bicgstab solvers show no apparent performance change after enabling the loop fuser. Results for the qmr solver are shown in Figure 6.2.

The bicg solver is the exception to this trend and shows significant performance improvements for both test platforms. These results are shown in Figure 6.3.

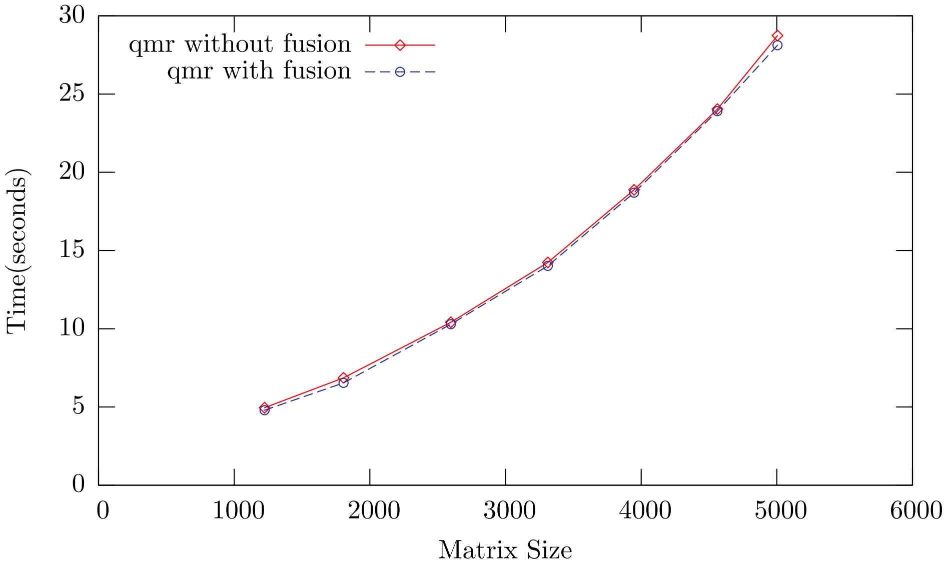
In order to assess whether the fusion pass was successfully fusing loops in all the iterative solvers I decided to record the number of loops fused for each program, as well as how many of fused loops were in code executed each iteration. The results are shown in Table 6.3.

The loop fuser successfully performs a number of loop fusions on all the iterative solvers. Despite this, a number of iterative solvers do not show any apparent speedup. There are a few possible explanations:

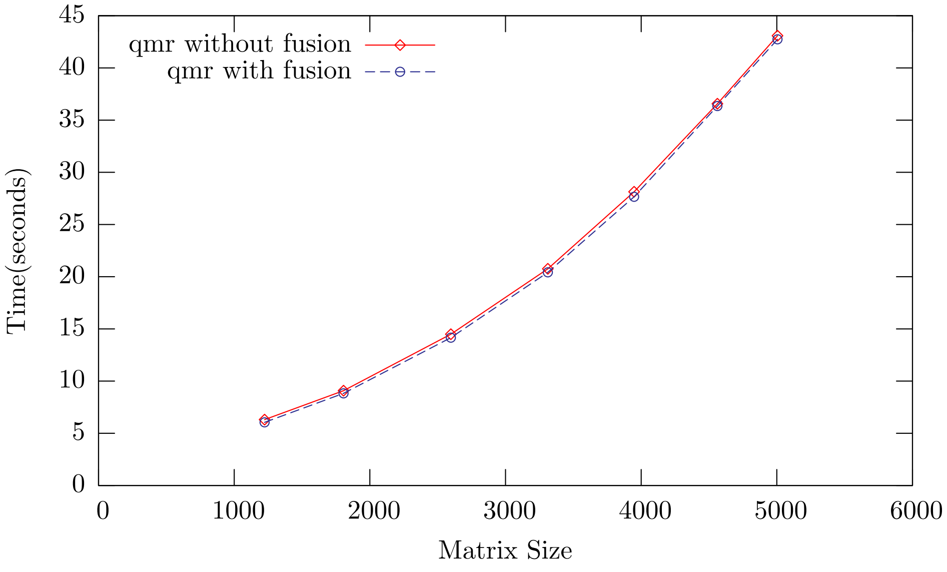
Non-Beneficial Fusions As the loop fuser does not posses a model of cache locality, it is incapable of making choices about what loop fusions to perform to provide better performance. As a result it is possible that the fusions performed do not provide a benefit to the program.

Loop Fuser Limitations As the loop fuser does not posses a full dependence model, it may be unable to recognise legal loop fusions. This could also prevent the loop fuser from performing fusions beneficial to the program.

Low Relative Cost of Improved Code Inspection of the code generated by the loop fuser shows that the fused loops typically consist of operations between vectors. As the most expensive operations in the iterative solvers

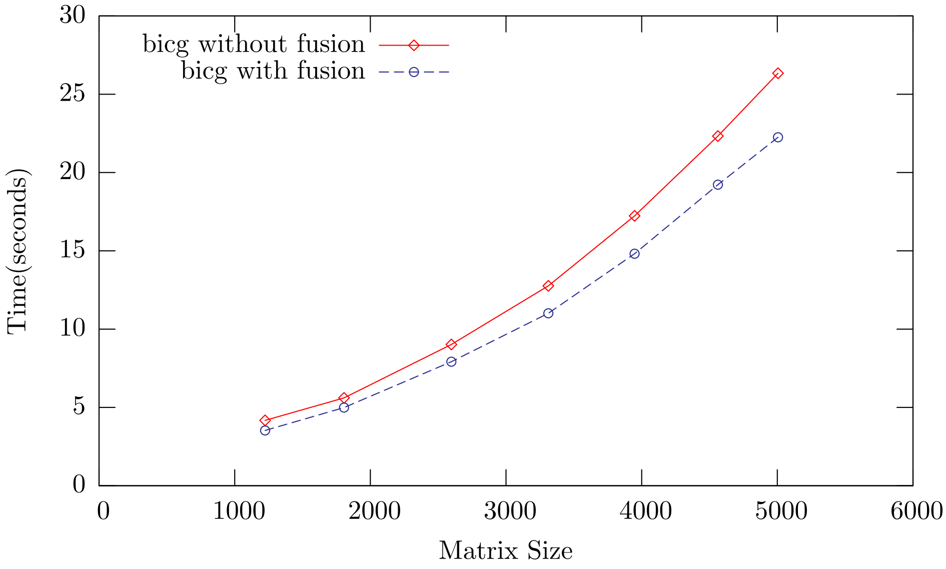


(a) qmr on rays

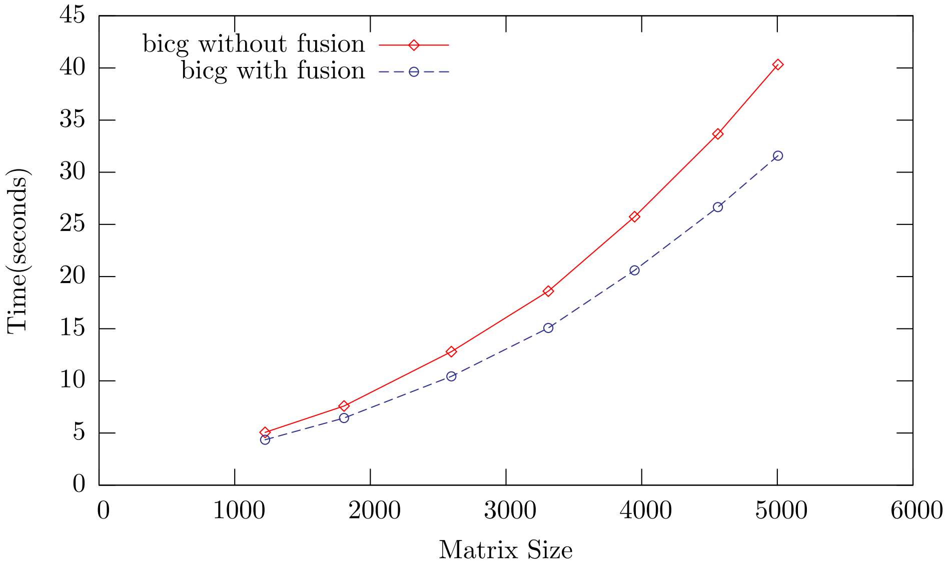


(b) qmr on vertices

Figure 6.2: Time taken to execute 256 iterations of the Quasi-Minimal Residual solver with and without the loop fuser enabled. The Conjugate Gradient Squared, BiConjugate Gradient Stabilised and Transpose Free Quasi-Minimal Residual solvers produced similar results



(a) bicg on rays



(b) bicg on vertices

Figure 6.3: Time taken to execute 256 iterations of the BiConjugate Gradient solver with and without the loop fuser enabled (lower is better). Significant speedups are present on both platforms.
.

are matrix-vector multiplies, it is possible these operations are dominating execution time, concealing the benefits of the loop fusion.

To determine why the BiConjugate Gradient solver managed to achieve such a significant speed up when loop fusion was applied, I will analyse the performance of this particular benchmark in a later section.

Solver	Total array contractions	Array contractions in repeatedly executed code
bicg	12	3
bicgstab	15	7
cgs	14	4
qmr	11	7
tfqmr	16	10

Table 6.4: Number of array contractions occurring in each iterative solver. *Total array contractions* refers to the number of array contractions performed during the execution of the solver. *Array contractions in repeatedly executed code* refers to the number of array contractions in code executed by the solver each iteration.

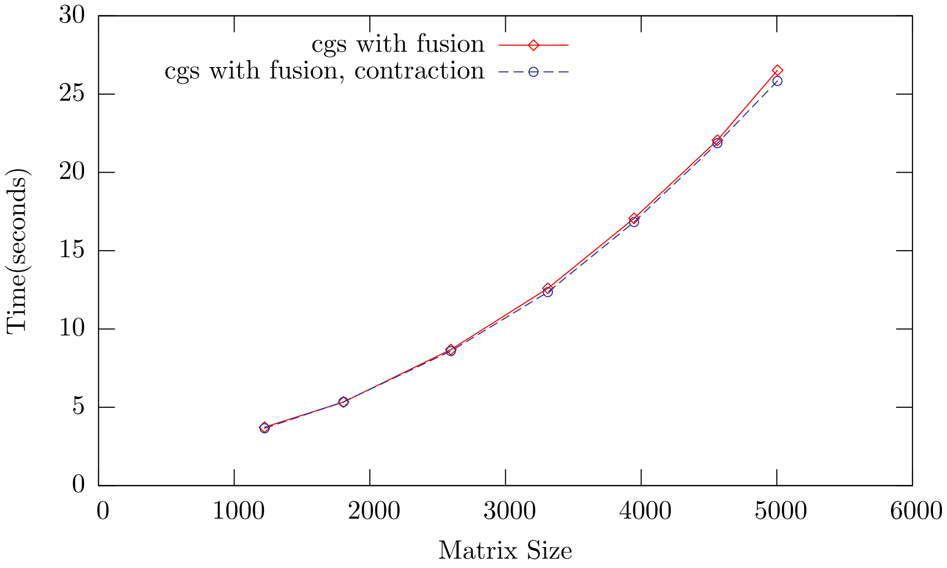
6.3 Array Contraction

Array contraction is a memory transformation that allows the dimensionality of an array to be reduced, possibly to the extent that the array becomes a scalar. This optimisation can only occur when the dependence vectors between the array accesses meet certain criteria. As these are most often facilitated by loop fusion, the array contraction results were obtained with both loop fusion and code caching enabled.

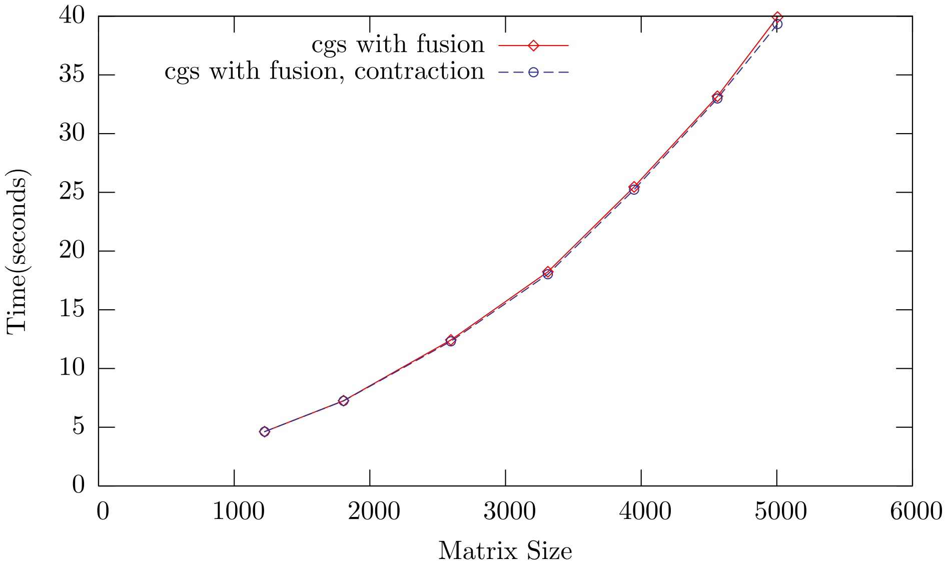
Unfortunately, none of the iterative solvers show any significant improvement with the array contraction pass enabled. Results for the Conjugate Gradient Square solver are shown in Figure 6.4. To check the number of array contractions being performed, I inspected the code generated by each solver and determined the number of arrays contracted in each execution, and how many of these occurred in the code executed each iteration. The results are shown in Table 6.4.

The results show that array contractions are occurring in all the iterative solvers, some them in code executed by the solvers each iteration. Analysis of the assembly code produced by ICC during the implementation of the loop fuser showed that the array contraction pass was successful in converting accesses of arrays allocated in main memory to accesses of a single register value.

It seems likely that the lack of speedup is due to the fact that the only arrays contracted are vectors. It is not possible to perform array contraction on any of the matrices used in the iterative solvers because the only matrix allocated (which represents the linear system) is passed as a parameter into the runtime generated code. With other numerical applications that mainly use vectors, or involve temporary matrices, it is likely that a significant speed up could be achieved.



(a) cgs on rays



(b) cgs on vertices

Figure 6.4: Time taken to execute 256 iterations of the Conjugate Gradient Squared solver with and without array contraction enabled. Both solvers have loop fusion enabled. Other iterative solvers produced similar results.

Solver	Number of locally allocated arrays with liveness analysis disabled	Number of locally allocated arrays with liveness analysis enabled
bicg	5	7
bicgstab	8	9
cgs	8	11
qmr	10	10
tfqmr	11	11

Table 6.5: Number of arrays allocated locally to the runtime generated code frequently executed by each iterative solver. Results are for the liveness analysis mechanism enabled and disabled.

6.4 Runtime Liveness Estimation

The runtime liveness estimation system attempts to recognise repeated sequences of operations and determine whether the values involved are used directly, or only indirectly as part of the evaluation of another expression. When an expression DAG is evaluated, if a node has no handles to it, it can be allocated locally to the runtime generated code and possibly optimised away. If a client holds a handle to a delayed expression, it is possible this value might be used later on, and it cannot be allocated locally to the program.

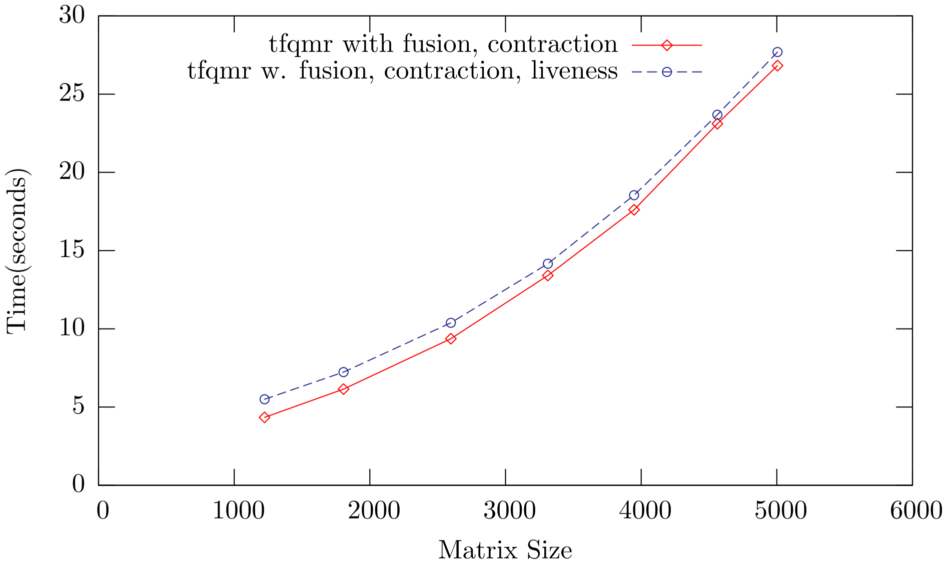
Runtime liveness estimation allows the prototype library to guess whether the value of a delayed expression will be used. Even if a client holds a handle to a delayed expression, if the library believes it will not be used, it will allow it to be allocated locally to the runtime generated code and possibly optimised away. The drawback of this approach is if the library guesses incorrectly, the library may have to evaluate the expression again to obtain its value.

For the information provided by the runtime liveness estimation system to be useful, allocating a value locally to the runtime generated code must provide a benefit. For this reason, it makes sense to evaluate the liveness estimation system in the presence of array contraction, loop fusion and code caching.

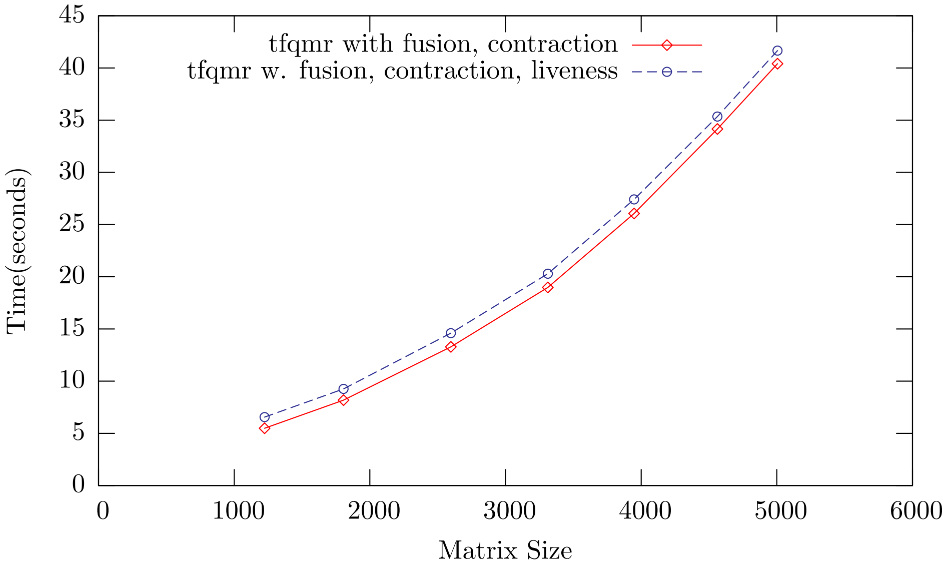
In order to asses whether the runtime liveness estimation technique was proving useful information, I decided to count the number of locally allocated arrays in each iterative solver with the liveness analysis enabled and disabled. As liveness analysis requires that a series of operations be executed multiple times before it can build up liveness information, it only makes sense to count the locally allocated arrays in the code executed during each iteration. I decided not to count locally allocated scalar values as they are less likely to have an impact on performance.

Results for the Transpose Free Quasi-Minimal Residual solver are shown in Figure 6.5. The results do not show any noticeable performance improvement from enabling the liveness analysis mechanism. In fact, many show a constant overhead. This is most likely due to extra code compilation caused by variations in the liveness analysis mechanism changing what values it thinks are live or dead as it moves towards a stable state.

The runtime liveness mechanism has managed to determine that some of the vectors used in three of the ITL iterative solvers do not have their value used directly, and has allowed them to be allocated locally to the runtime generated



(a) tfqmr on rays



(b) tfqmr on vertices

Figure 6.5: Time taken to execute 256 iterations of Transpose Free Quasi-Minimal Residual solver with and without liveness analysis enabled. Both solvers have loop fusion and array contraction enabled. The overhead caused by the runtime liveness analysis appears to be constant. The other iterative solvers produced similar results.

Solver	Total compiler invocations without liveness analysis	Total compiler invocations with liveness analysis
bicg	9	10
bicgstab	10	12
cgs	9	11
qmr	12	16
tfqmr	9	14

Table 6.6: Number of compiler invocations for 256 iterations of different iterative solvers with liveness analysis enabled and disabled.

code. To test this, I compared the number of compiler invocations with and without the liveness analysis mechanism enabled. These results are shown in Table 6.6.

These results show that the liveness analysis mechanism increases the number of compilations that occur in all iterative solvers, even the ones in which it did not provide any benefit to the code executed each iteration. The graphs also indicate the overhead caused by the runtime liveness analysis mechanism is primarily due to compilation, and not to the overhead of the system itself. With a system capable of persisting cached code between program runs which could be added at a later date, this overhead would no longer be present.

Unfortunately, no noticeable performance increases have been obtained through the runtime liveness analysis system although I have shown that it does provide enough information to some allow arrays to be allocated locally to runtime generated code that could not be before. Despite this, I believe development of the runtime analysis mechanism may provide benefits. I have discussed these changes in the Conclusion.

6.5 Comparison Against State of the Art

MTL is the most advanced C++ numerical library available for testing. It makes extensive use of C++ template metaprogramming techniques to control its own compilation. The Iterative Template Library provides an MTL interface making it possible to compare its performance to the prototype library developed.

Figures 6.6, 6.7, 6.8, 6.9 and 6.10 show the results of the comparison between the prototype library and MTL. As the runtime liveness analysis mechanism only provided a constant overhead for the benchmarks tested, I decided to compare MTL against the prototype library with the loop fusion and array contraction passes enabled but with the liveness analysis mechanism disabled. It is these results that are used throughout the rest of this section.

The results for the prototype library show the execution time including and excluding the overhead of runtime compilation. I believe that this is an appropriate result to show given that a code cache that persists between program invocations or a longer run of the iterative solver would minimise the impact of the compilation overhead. Furthermore, the result without the compilation overhead corresponds to the execution time spent in the iterations themselves.

On the Vertices, the MTL has a lower execution time for many of the benchmarks at the matrix sizes tested. On average, the prototype library executes iterations slightly faster but has to claim back the cost of the initial compilations before speedups can be obtained.

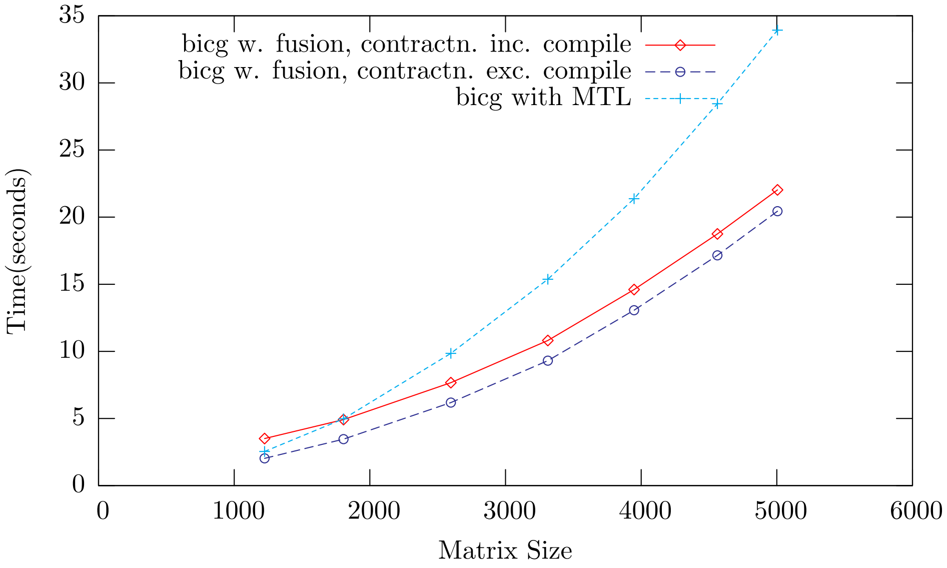
On the Rays, the prototype library consistently outperforms the MTL for higher matrix sizes, even when including the cost of compilation.

The bicg solver using the prototype library performs significantly better than its MTL counterpart. Of the improvements achieved, it is the only one that is present on both the Vertices and the Rays. On the Rays with a matrix size of 5005 (the largest) the MTL solver takes 33.9 seconds compared to 22.0 seconds for the prototype library for 256 iterations. This corresponds to an overall performance increase of 54.0%, or excluding the compilation overhead of prototype library of 1.6 seconds, an iteration execution performance increase of 66.0%. On the Vertices at the largest matrix size, the MTL benchmark executes in 40.6 seconds and the prototype library in 31.2 seconds. This gives an overall performance increase of 30.1%, or excluding the cost of compilation of the prototype library of 1.9 seconds, an iteration execution performance increase of 38.6%.

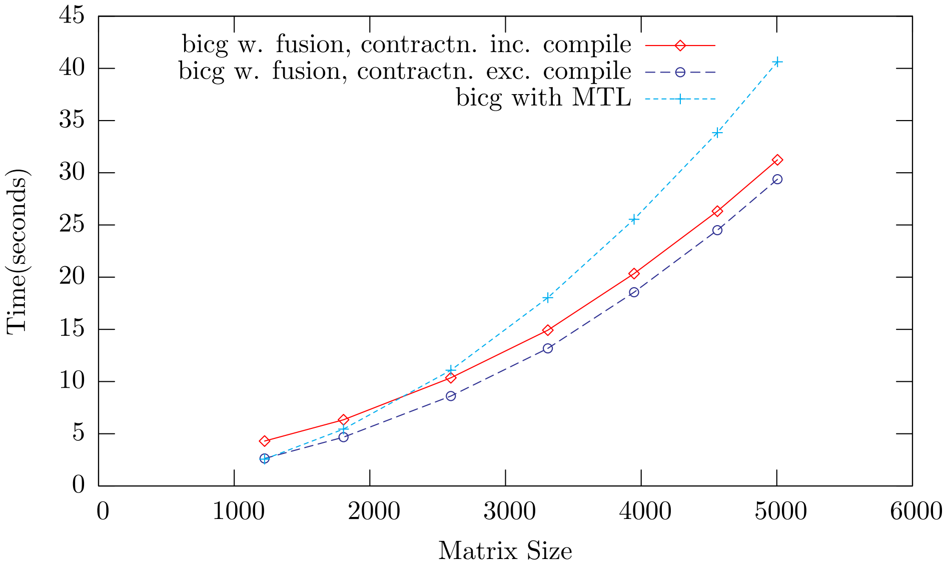
One of the most interesting aspect of these results is that for a majority of the benchmarks the performance increases present are on only one of the platforms despite the fact the same code is being executed. The Rays possess a cache size of 2048 KB compared to 512 KB on the vertices. It is possible the Intel C Compiler was able to use more effectively. The question then becomes why was the Intel C Compiler not able to provide this improvement for MTL running on the Rays as well?

Unfortunately, I do not yet have an answer to this question at this time. Analysis of the assembly produced as well as the cache usage of the appropriate runtime generated libraries is required before this can be answered. However, I can offer a couple of suggestions. One possibility is that the constant loop bounds allowed the compiler to unroll and block the loop more effectively than the loops whose bounds were unknown at compile time. Another is that the Intel C Compiler is more effective at optimising code that the Intel C++ compiler, especially considering the C supplied was generated to be easily optimisable.

Given the significant improvement in the bicg iterative solver not present in the other solvers, I will analyse the reasons for this in the next section.

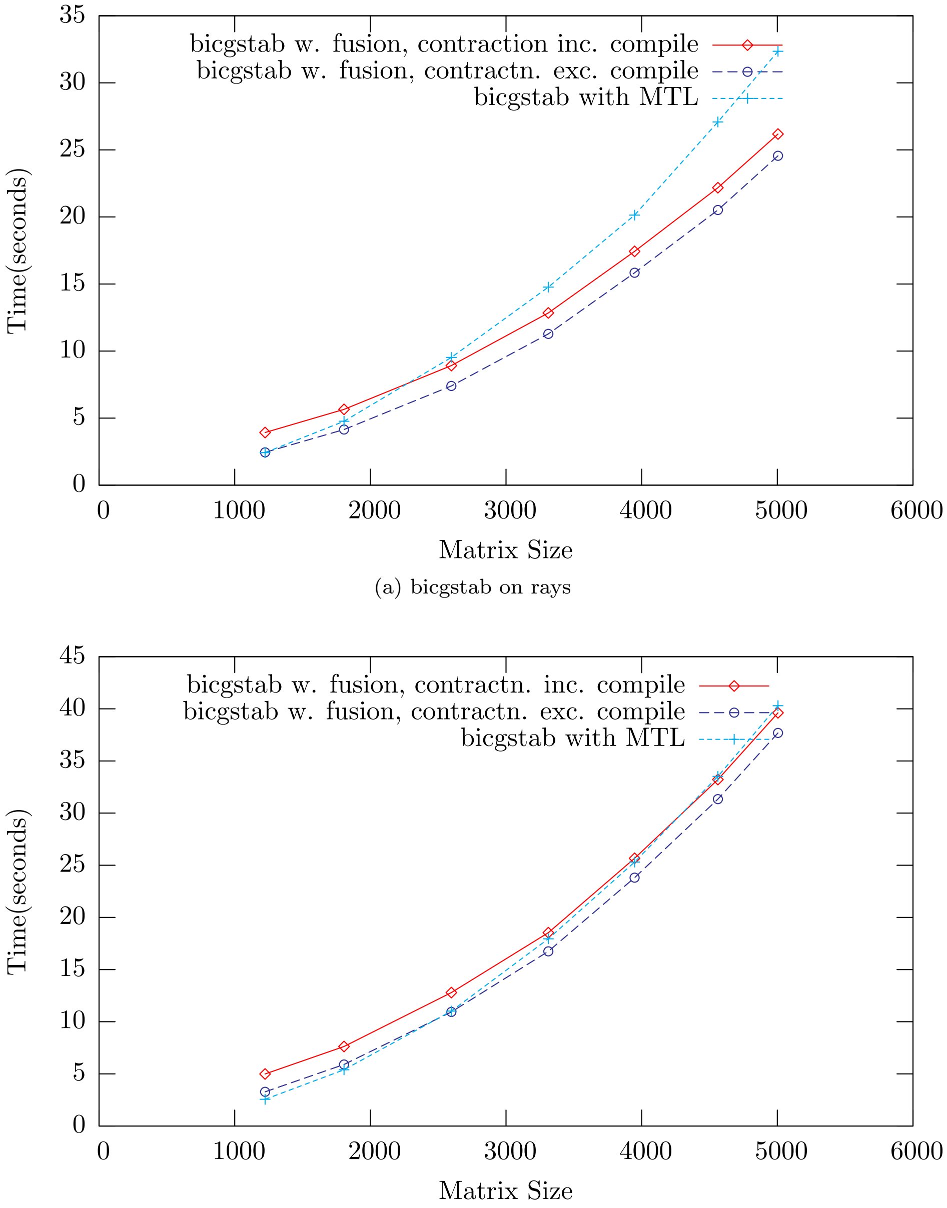


(a) bicg on rays



(b) bicg on vertices

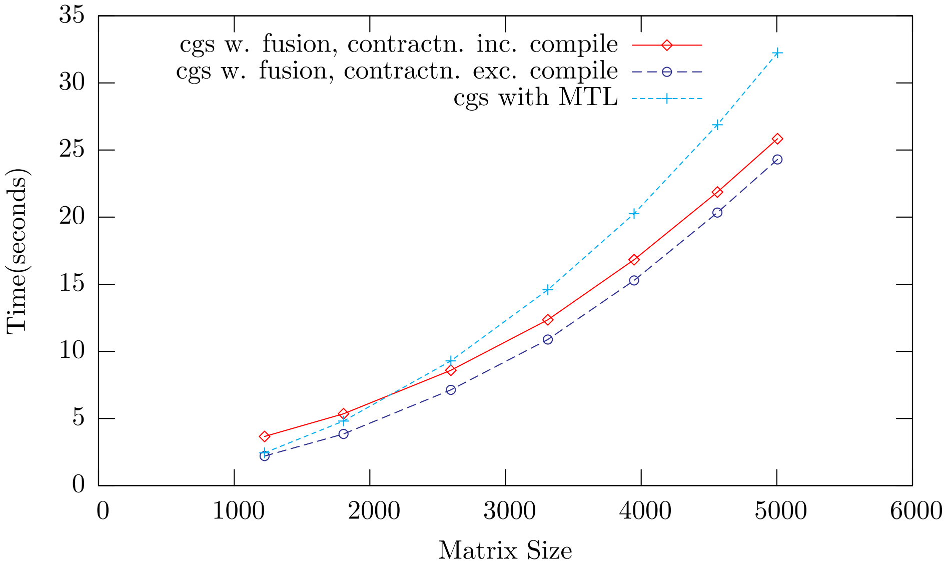
Figure 6.6: Time to execute 256 iterations of the bicg solver with the prototype library and MTL. Results for the prototype library are shown including and excluding runtime code compilation time.



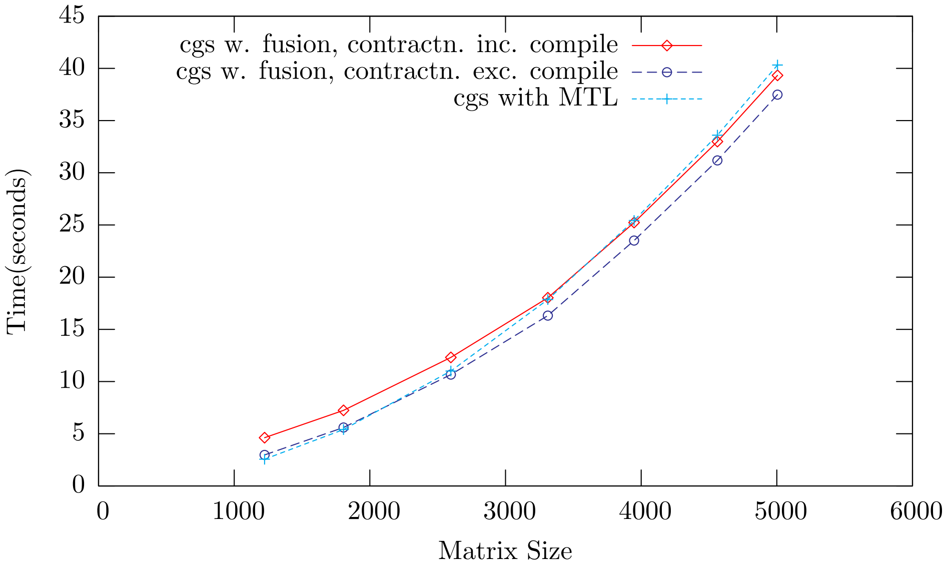
(a) bicgstab on rays

(b) bicgstab on vertices

Figure 6.7: Time to execute 256 iterations of the bicgstab solver with the prototype library and MTL. Results for the prototype library are shown including and excluding runtime code compilation time.

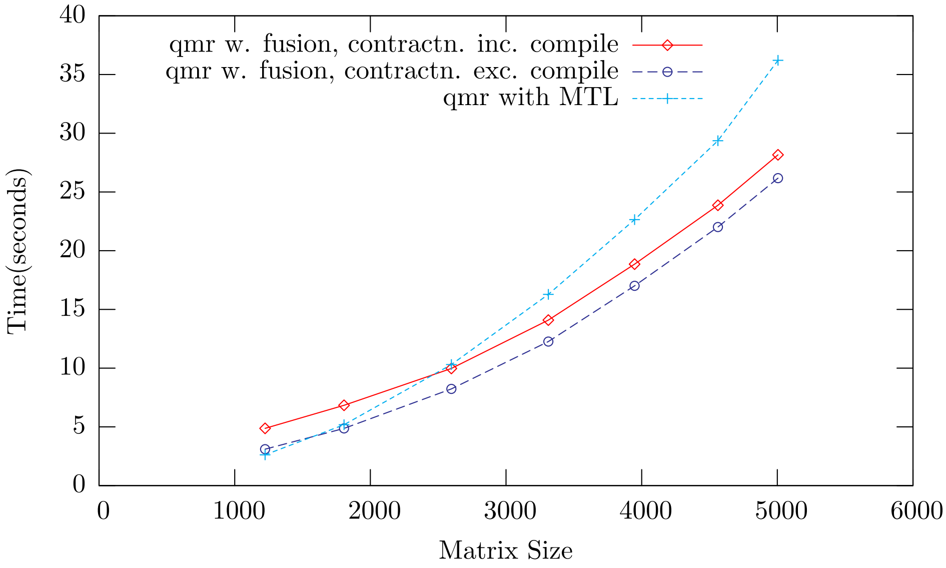


(a) cgs on rays

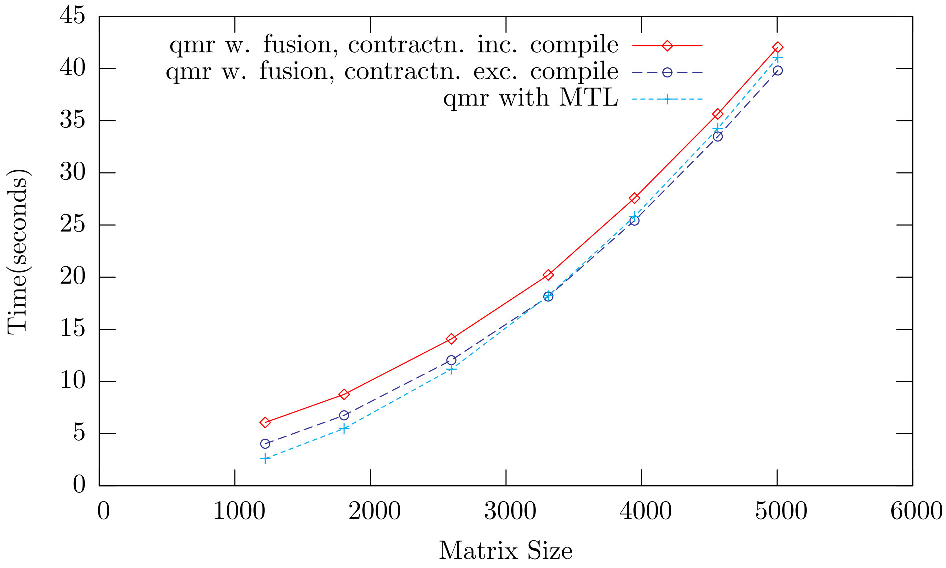


(b) cgs on vertices

Figure 6.8: Time to execute 256 iterations of the cgs solver with the prototype library and MTL. Results for the prototype library are shown including and excluding runtime code compilation time.

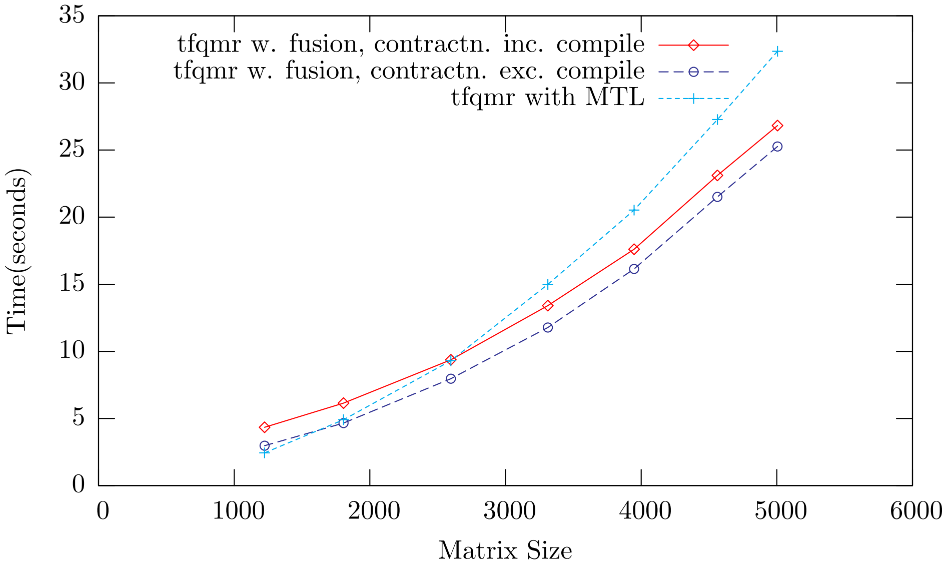


(a) qmr on rays

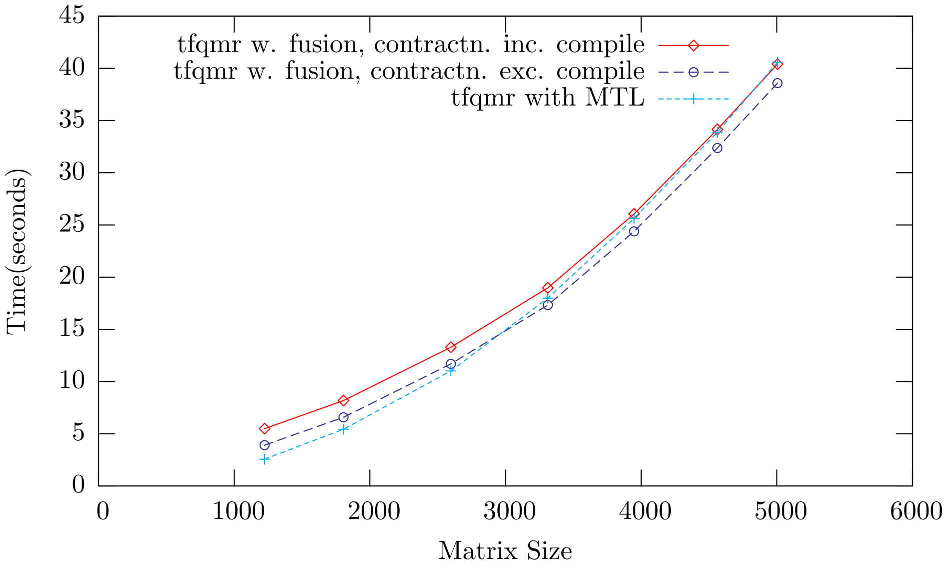


(b) qmr on vertices

Figure 6.9: Time to execute 256 iterations of the qmr solver with the prototype library and MTL. Results for the prototype library are shown including and excluding runtime code compilation time.



(a) tfqmr on rays



(b) tfqmr on vertices

Figure 6.10: Time to execute 256 iterations of the tfqmr solver with the prototype library and MTL. Results for the prototype library are shown including and excluding runtime code compilation time.

TaskGraph	Time without fusion (s)	Time with fusion (s)
taskGraph_0	0.03502	0.03750
taskGraph_1	0.00174	0.00179
taskGraph_2	0.00002	0.00002
taskGraph_3	0.06144	0.06640
taskGraph_5	0.00615	0.00610
taskGraph_6	0.60941	0.03507
taskGraph_9	15.5393	8.75058
taskGraph_770	0.00001	0.00001
taskGraph_771	0.03064	0.03021

Table 6.7: Time spent in execution of each TaskGraph for 256 iterations of the bicg solver for a matrix size of 3312.

6.6 Analysis of BiConjugate Gradient Speedup

In this section, I intend the analyse the reasons for the performance increase in BiConjugate Gradient iterative solver over MTL which were far more significant than those for any other iterative solver. The results indicate the most significant contribution to the improvement of the performance of the solver was the loop fusion pass. Firstly, I measured the amount of time spent in each TaskGraph with fusion enabled and disabled. The results are shown in Table 6.7.

Clearly the majority of the time is being spend in fragment taskGraph_9 and this is where the loop fusion benefit has also occurred. The bodies of the taskGraph_9 are given in Appendices A and B.

Analysis of the code reveals that numerous loop fusions have been performed, of these the most interesting is the one involving the nested loop. With a trip count far higher than any other code in the program it is likely that the majority of the execution time is being spent in the nested loop created from the fusion of the two matrix-vector multiply operations.

If this is the case, fusing the two nested loops would allow elements of convMatrix_0 to be loaded into the cache only once, and then used by both matrix-vector multiplies.

The performance increase obtained demonstrates the importance of cross component optimisation. Both matrix vector multiplies use different vectors for their multiplies and their results. Only systems possessing cross component optimisation can enable this to occur, with standard library interfaces unable to optimise two matrix multiplies together in this manner. Even BLAS's performance oriented interface is too restrictive to allow this to occur.

Another important observation that can be drawn is the effect force points have on the performance of delayed evaluation libraries. In the bicg solver, the most performance significant code exists within a single TaskGraph. Analysis of the code produced by other iterative solvers showed that this was not true for other iterative solvers. Most importantly, nested loops operating on the matrix were in different TaskGraphs, making them impossible to fuse. Whether this occurred because evaluation order forced them to be in different TaskGraphs, or this was merely the ordering chosen by the prototype library is difficult to determine.

Further research is required to determine how the effect of force points can be reduced in delayed evaluation libraries, and how decisions can be made to best improve the chances for loop fusions and other optimisations involving code that may not yet have been generated. Two techniques which could assist, *Speculative Evaluation* and further developments to *Runtime Liveness Analysis* will be discussed in the conclusion.

6.7 Summary

Ignoring compilation overheads, improvements in execution of over 65% have been obtained. Including compilation, these are improvements of over 50%. As the compile time overheads will always be a constant factor independent of problem size, large matrix sizes and/or large numbers of iterations are required in order to provide a performance benefit.

For 256 iterations running on the Rays, the break even point tends to occur with matrix sizes between 1500 and 2500. This is of course an iteration dependent value, more data would have to be collected for characterise the break even points for other iteration sizes. On the Vertices, the break even point between MTL and the prototype library only occurs for very large matrix sizes (except the bicg solver) and in some cases may not even occur. The extent to which this could be a problem for other applications using the library will be dependent on the number of runtime compiler invocations. For an application making extensive reuse of existing code, the constant overhead can probably be ignored. If the program constantly invokes the compiler, the compile overheads will accumulate.

Of all the optimisations implemented, code caching has been the most successful. For the iterative solvers, It has allowed the reduction of the compile costs of the runtime code generation mechanism to be reduced from an iteration dependent value to a constant overhead.

Loop fusion has also shown benefits, demonstrated in the speedup of the BiConjugate Gradient solver. Loop fusion benefits have not been demonstrated in the other iterative solvers and I have provided reasons why this could occur. These include the distribution of force points in the other iterative solvers and limitations in the loop fuser. The latter can be overcome with a better dependence analysis framework, I will discuss solutions to the first in the Conclusion.

Array contraction has not shown any significant benefit. Unlike the other optimisations implemented, successful array contraction is highly likely to reduce contention for the cache and has shown to be effective in optimising away a number of arrays. Therefore, it seems likely that the performance increase obtained from the array contraction is being shadowed by larger, more time consuming operations, probably the matrix-vector multiplies present in each iterative solver.

The liveness analysis mechanism has been successful in recovering liveness information previously invisible to the runtime system and used it to allocate arrays locally to the runtime generated code which could not have been done so before. Unfortunately, the number of these arrays are limited, and do not provide significant benefits when allocated locally. Despite this, I believe this is an important starting step for a technique which when extended could provide significantly more useful information. I will discuss further developments to this mechanism in the conclusion.

The reasons for the performance benefits against MTL have not been determined fully. Loop fusion has provided a significant benefit to the bicg benchmark, the performance improvements of the other benchmarks originate almost entirely elsewhere. I have suggested possible sources for these including the optimisability of specialised code generated, and the complexity of optimising C++ over C.

In the next chapter, I will discuss the conclusions that can be drawn from this work, and future development that could be made to this project.

Chapter 7

Conclusions and Future Work

In this chapter, I will first discuss the conclusions that can be drawn after the undertaking of this project. I will then suggest future research that could be undertaken to extend this project and future development to the prototype library produced.

7.1 Conclusions

I believe the single most important conclusion that can be drawn from this project is that cross component optimisation is essential for high performance numerical code. To provide these benefits, libraries such as BLAS have resorted to large numbers of specialised methods with numerous parameters. Both MTL and the prototype library produced are similar in that they employ unconventional techniques to work around the restrictions of a standard library.

MTL uses C++ template metaprogramming to generate specialised code, and is not a library in the conventional sense. It contains no shared code to link against. Instead, it consists solely of header files containing templates to be instantiated and generate code at application compile time. The prototype library produced is more similar to a conventional library than MTL but again uses complex techniques to work around the library interface, in this case delayed evaluation.

This project has shown that delayed evaluation and runtime code generation is a viable method for producing high performance code. Runtime information can assist in making decisions important for performance that cannot be decided statically. This project has demonstrated a runtime system transparent to the library client, and implemented without compiler extensions can use this information to improve its own performance. It has also shown that in order to do this compile time information must be sacrificed, although some of it can be recovered though techniques like runtime liveness estimation.

As the importance of information from all stages of compilation is realised, a trend towards what Veldhuizen calls *Active Libraries*[20] continues. These are libraries that use unconventional techniques to extract as much information from their environment as possible. These include C++ templates, as used by MTL, or empirical techniques, as used by ATLAS. Using this information, libraries can optimise their performance be it through decisions made at compile time or at runtime. The prototype library implemented is among this spectrum.

The effectiveness of array contraction and loop fusion have also been demonstrated, even with the restricted implementations used by this project. This is not new information, both have been heavily researched as have many other memory access transformations. The fact that array contraction was not performed by either GCC or ICC (a new SUIF pass had to be written) indicates that available compilers still have a way to progress in their quest for optimal code. It is uncertain whether compilers possessing loop fusion and array contraction features would have enough information to perform them on statically compiled code, however, it does show the effectiveness of a system such as the one developed in providing useful optimisations when a compiler implementing them does not exist.

7.2 Future Work

In this section, I will present both further developments to this investigation and the prototype library produced.

7.2.1 Improved Optimisations

Loop fusion has been shown to be an effective runtime optimisation for a particular iterative solver. Other optimisations such as array contraction have the potential for improving performance when used in the right context.

The implementation of these optimisations is far from optimal. Techniques such as affine partitioning[14] unify numerous loop transformations and allow programs to be transformed in ways that should improve cache locality.

An effective implementation of these techniques is required for a detailed understanding of the benefits they can provide to applications utilising runtime code generation.

7.2.2 Improved Liveness Analysis

The liveness analysis implemented in the prototype library had a single purpose: to recover some of the static liveness information not available to the runtime system. However, this information is available to the C++ compiler compiling the program. If this information could be conveyed to the prototype library along with the application being compiled, this would allow the prototype library to optimise away temporaries more effectively.

To enable a compiler to convey this information to an application would most likely require compiler specific modifications. For a system that aims to be compatible with multiple standard C++ compilers, this approach is unacceptable. However, a library discarding this restriction would be able to perform more effective liveness analysis.

The liveness analysis implemented was merely intended to overcome some of the limitations of the information available to the runtime system. However, liveness analysis has some other more interesting applications which do not require modifications to the compiler.

At a force point in the current system, it is possible to guess whether a value is live or dead if the same sequence of operations has been performed before. However, if a value is estimated as being live, it is not possible to determine how long it will be before it is used. If the library were modified to make this information available, it would allow the library to make decisions based on what delayed operations are evaluated and when they are evaluated. This is an improvement over the existing system, whose evaluation decisions only consider what delayed operations are evaluated, but not when.

7.2.3 A Cache Locality Model

In the previous section, I discussed how an improved an liveness analysis mechanism could help provide greater flexibility with regards to when to evaluate a delayed operation. One way this could be helpful is in improving cache locality. At a force point in the current system, decisions can be made regarding what to evaluate to try to improve cache locality. With a cache locality model of the expression DAG and estimated liveness information, it becomes possible to optimise for cache locality by choosing both what delayed expressions to evaluate and when. This is optimising for cache locality with respect to estimated future operations.

7.2.4 Speculative Evaluation

Optimising with respect to estimated future operations can be taken one step further. Performing estimated future operations in advance. I have already shown it is possible to recognise repeatedly generated sequences of operations. Using a similar technique, it would be possible to recognise repeated sequences of operations, and estimate what the program is likely to do next.

This technique could assist in reducing the overheads associated with a program with large numbers of force points. Force points make it difficult to produce optimised code as they limit the number of delayed operations that can be performed at the same time. This reduces the chances of performing effective cross component optimisation. Take an iterative solver, each iteration it must check for convergence. This forces evaluation. Speculative evaluation would allow the runtime system to assume that the solver will probably perform another iteration, and enable it to be evaluated in advance.

7.2.5 Persistent Code Caching

Reducing the overhead of runtime code generation and compilation is especially important if the generated code does not provide a significant performance improvement or, the original code would have taken less time to execute than the compiler did. Running an iterative solver with a small matrix which converges quickly will perform significantly worse when it uses runtime code generation. The ability to persist cached code between program invocations could help reduce this problem even further than the runtime code caching.

7.2.6 Alternate Methods of Delayed Expression Evaluation

So far, the only implemented method to evaluate delayed operations has been to generate code to perform them, compile it, and execute it. For large numbers of fast executing operations, this technique becomes a performance overhead. This is especially true if code caching cannot be used to reuse the compiled code. An alternative method to handle these operations would be to use a statically compiled version of these operations in the program. Another would be to use BLAS to evaluate these operations.

Even evaluation using BLAS calls provides opportunities for research. Given a DAG of delayed operations, a search problem exists in which the solution is the mapping of BLAS calls onto the DAG that will evaluate it most efficiently.

7.2.7 Fortran Code Generation

TaskGraph provides a simple sub-language in which arrays exist as first class objects. However, the code produced by the TaskGraph system is always C. In C, arrays are little more than syntactic sugar for pointers. Developments to get compilers to optimise C code using arrays effectively often involve the creation of extra compiler flags. For example, the Intel C Compiler supports the *-ansi-alias* and *-restrict* flags which allows the compiler to make certain assumptions about aliasing in the program.

The alternative is to generate code in Fortran. Fortran provides first class arrays which allows compilers to optimise it more effectively than C. The primary incompatibility is Fortran array layout, which is Column-major, as opposed to Row-major in C. However, this difficultly could easily be overcome by altering the runtime generated code to traverse the arrays correctly. The result would be runtime generated code which could be optimised even more effectively.

7.2.8 Sparse Matrices

Sparse matrices pose a number of interesting problems. As the size of a sparse matrix is usually unknown in advance, it is impossible to allocate them as local arrays to the TaskGraph system. The TaskGraph system does not support dynamic memory allocation so their size cannot be decided or altered inside TaskGraph generated code. Another important issue is how the generate code that accesses the sparse matrix efficiently. As finding a given element in a sparse matrix requires multiple lookups, it is important that these are not replicated when performing an operation such as matrix-vector multiply. The problem of generating efficient traversal code for different storage formats is a problem successfully tackled in the MTL library. I discussed a possible approach in Section 4.4.4.

7.2.9 User-Level Algorithms

In the implemented library, delayed operations are represented as nodes in an expression DAG, which each node type representing a different operation. One disadvantage of this approach is that adding a new linear algebra operation will involve:

1. Adding a new node type to the expression DAG and updating all visitors.
2. Adding a new node type to the runtime liveness analysis DAG and updating all visitors including those for hashing and equality checking.
3. Adding a new node type to the TaskGraph DAG and updating all visitors including those for hashing, equality checking and code generation.

This is a significant effort for the library maintainer, and a difficult challenge for any user. The user could implement the operation as a combination of low level matrix and vector accesses but this would result in extremely poor performance, large DAG sizes and make it difficult for the library to recognise repeated operations.

The solution is to enable to user to define their own algorithms which can then be incorporated into the various DAGs like any other node. Amongst other requirements, it is important that the approach taken be:

Simple To delay evaluation, the algorithm's semantics must be fully captured. It must be possible for the library client to specify a sequence of instructions in an intuitive manner otherwise the usefulness of this feature would be devalued. One solution could involve creating some sort of sub-language inside C++ to express linear algebra algorithms much in the same way that runtime generated code is expressed in the TaskGraph sub-language.

Abstract The algorithm should be able to be expressed in a way that is independent of the storage formats of the matrices and vectors involved. It should be the responsibility of the runtime code generation mechanism to ensure that the code produced is efficient.

Bibliography

[1] Ashby, T. J., Kennedy, A. D., and O'Boyle, M. F. P. Cross component optimisation in a high level category-based language. In *Euro-Par* (2004), pp. 654–661.

[2] Bacon, D. F., Graham, S. L., and Sharp, O. J. Compiler transformations for high-performance computing. *ACM Computing Surveys 26*, 4 (1994), 345–420.

[3] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[4] Beckmann, O., Houghton, A., Mellor, M., and Kelly, P. H. J. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation* (2003), pp. 291–306.

[5] Beckmann, O., and Kelly, P. H. J. Efficient interprocedural data placement optimisation in a parallel library. In *LCR* (1998), pp. 123–138.

[6] Dongarra, J., Lumsdaine, A., Pozo, R., and Remington, K. IML++ v. 1.2: Iterative methods library reference guide. Tech. rep., Knoxville, TN 37996, USA, 1996.

[7] Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software 16* (1990), 1–17.

[8] Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software 14*, 1 (1988), 1–17.

[9] Engler, D. R., Hsieh, W. C., and Kaashoek, M. F. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Symposium on Principles of Programming Languages* (1996), pp. 131–144.

[10] Kelly, P. H. J. Compiler issues: dependence analysis, vectorisation, automatic parallelisation, Jan. 2005. Lecture notes for Course 332, Advanced Computer Architecture, at Imperial College, London.

[11] Lawson, C., Hanson, R., Kincaid, D., and Krogh, F. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft. 5* (1979), 308–323.

[12] Lee, L.-Q., Lumsdaine, A., and Siek, J. Iterative Template Library. http://www.osl.iu.edu/download/research/itl/slides.ps.

[13] Leone, M., and Lee, P. Lightweight run-time code generation. In *PEPM* (1994), pp. 97–106.

[14] Lim, A. W., Cheong, G. I., and Lam, M. S. An affine partitioning algorithm to maximize parallelism and minimize communication. In *International Conference on Supercomputing* (1999), pp. 228–237.

[15] Liniker, P., Beckmann, O., and Kelly, P. H. J. Delayed evaluation, self-optimising software components as a programming model. In *Euro-Par* (2002), pp. 666–674.

[16] Siek, J. G., and Lumsdaine, A. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE* (1998), pp. 59–70.

[17] Siek, J. G., and Lumsdaine, A. 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* (1998), pp. 468–469.

[18] Veldhuizen, T. Expression templates. *C++ Report 7*, 5 (June 1995).

[19] Veldhuizen, T. Blitz++: The library that thinks it is a compiler. In *SciTools* (1998).

[20] Veldhuizen, T. L., and Gannon, D. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)* (1998), SIAM Press.

[21] Whaley, R. C., Petitet, A., and Dongarra, J. J. Automated empirical optimization of software and the ATLAS project. *Parallel Comput. 27*, 1–2 (2001), 3–25.

[22] Yi, Q., and Kennedy, K. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. *Int. J. High Perform. Comput. Appl. 18*, 2 (2004), 237–253.

Appendix A

Unfused BiConjugate Gradient Solver Code

The following code was produced at runtime by the BiConjugate gradient solver with loop fusion disabled. The variable declarations have been omitted for clarity.

```
*convScalar_0 = *convScalar_1 / *convScalar_2;
for (index_0 = 0; index_0 <= 3311; index_0++)
{
  convVector_0[index_0] = convVector_1[index_0] * *convScalar_0;
}

for (index_1 = 0; index_1 <= 3311; index_1++)
{
  convVector_2[index_1] = convVector_3[index_1] + convVector_0[index_1];
}

for (index_2 = 0; index_2 <= 3311; index_2++)
{
  convVector_4[index_2] = 0;
  for (index_3 = 0; index_3 <= 3311; index_3++)
  {
    convVector_4[index_2] =
      convVector_4[index_2] +
      convMatrix_0[index_2][index_3] * convVector_2[index_3];
  }
}

for (index_4 = 0; index_4 <= 3311; index_4++)
{
  convVector_5[index_4] = convVector_6[index_4] * *convScalar_0;
}

for (index_5 = 0; index_5 <= 3311; index_5++)
{
  convVector_7[index_5] = convVector_8[index_5] + convVector_5[index_5];
}

convScalar_3 = 0;
for (index_6 = 0; index_6 <= 3311; index_6++)
{
  convScalar_3 = convScalar_3 + convVector_7[index_6] * convVector_4[index_6];
}

*convScalar_4 = *convScalar_1 / convScalar_3;
convScalar_5 = -*convScalar_4;
for (index_7 = 0; index_7 <= 3311; index_7++)
{
  convVector_9[index_7] = convVector_4[index_7] * convScalar_5;
}

for (index_8 = 0; index_8 <= 3311; index_8++)
{
  convVector_10[index_8] = convVector_3[index_8] + convVector_9[index_8];
}

convScalar_6 = 0;
for (index_9 = 0; index_9 <= 3311; index_9++)
{
  convScalar_6 = convScalar_6 + convVector_10[index_9] * convVector_10[index_9];
}

convScalar_6 = sqrt(convScalar_6);
*convScalar_7 = fabs(convScalar_6);
*convScalar_8 = *convScalar_7 / *convScalar_9;
for (index_11 = 0; index_11 <= 3311; index_11++)
{
  convVector_11[index_11] = 0;
}

for (index_10 = 0; index_10 <= 3311; index_10++)
{
  for (index_11 = 0; index_11 <= 3311; index_11++)
  {
    convVector_11[index_11] =
      convVector_11[index_11] +
      convMatrix_0[index_10][index_11] * convVector_7[index_10];
  }
}

convScalar_10 = -*convScalar_4;
for (index_12 = 0; index_12 <= 3311; index_12++)
{
  convVector_12[index_12] = convVector_11[index_12] * convScalar_10;
}
```

```
for (index_13 = 0; index_13 <= 3311; index_13++)
{
  convVector_13[index_13] = convVector_8[index_13] + convVector_12[index_13];
}

for (index_14 = 0; index_14 <= 3311; index_14++)
{
  convVector_14[index_14] = convVector_2[index_14] * *convScalar_4;
}

for (index_15 = 0; index_15 <= 3311; index_15++)
{
  convVector_15[index_15] = convVector_16[index_15] + convVector_14[index_15];
}
```

Appendix B

Fused BiConjugate Gradient Solver Code

The following code was produced at runtime by the BiConjugate gradient solver with loop fusion enabled. The variable declarations have been omitted for clarity.

```
*convScalar_0 = *convScalar_1 / *convScalar_2;
for (index_1 = 0; index_1 <= 3311; index_1++)
{
  convVector_0[index_1] = convVector_1[index_1] * *convScalar_0;
  convVector_2[index_1] = convVector_3[index_1] + convVector_0[index_1];
  convVector_5[index_1] = convVector_6[index_1] * *convScalar_0;
  convVector_7[index_1] = convVector_8[index_1] + convVector_5[index_1];
  convVector_11[index_1] = 0;
}

convScalar_3 = 0;
for (index_6 = 0; index_6 <= 3311; index_6++)
{
  convVector_4[index_6] = 0;
  for (index_3 = 0; index_3 <= 3311; index_3++)
  {
    convVector_4[index_6] =
      convVector_4[index_6] +
      convMatrix_0[index_6][index_3] * convVector_2[index_3];
    convVector_11[index_3] =
      convVector_11[index_3] +
      convMatrix_0[index_6][index_3] * convVector_7[index_6];
  }
  convScalar_3 = convScalar_3 + convVector_7[index_6] * convVector_4[index_6];
}

*convScalar_4 = *convScalar_1 / convScalar_3;
convScalar_5 = -*convScalar_4;
convScalar_6 = 0;
```

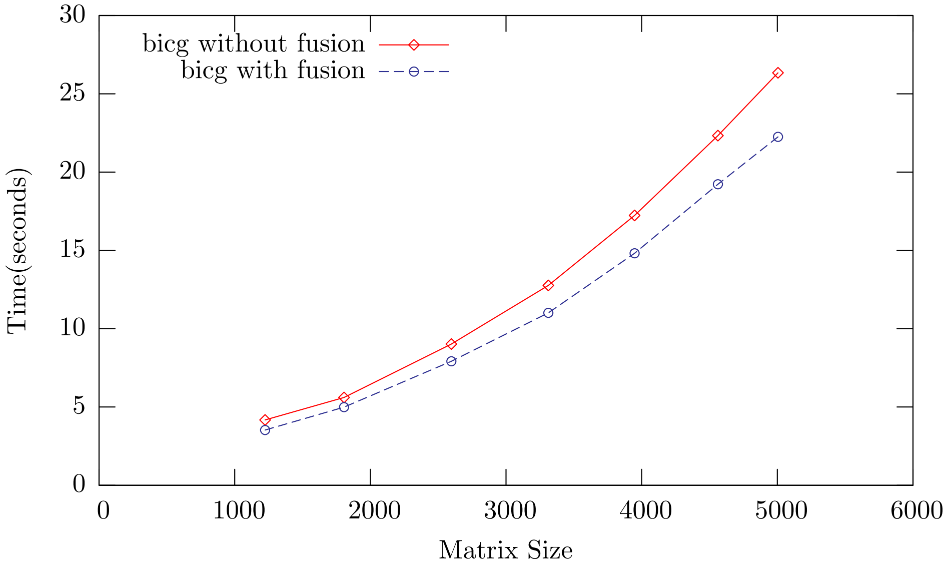
```
for (index_9 = 0; index_9 <= 3311; index_9++)
{
  convVector_9[index_9] = convVector_4[index_9] * convScalar_5;
  convVector_10[index_9] = convVector_3[index_9] + convVector_9[index_9];
  convScalar_6 = convScalar_6 + convVector_10[index_9] * convVector_10[index_9];
  convVector_14[index_9] = convVector_2[index_9] * *convScalar_4;
  convVector_15[index_9] = convVector_16[index_9] + convVector_14[index_9];
}

convScalar_6 = sqrt(convScalar_6);
*convScalar_7 = fabs(convScalar_6);
*convScalar_8 = *convScalar_7 / *convScalar_9;
convScalar_10 = -*convScalar_4;
for (index_13 = 0; index_13 <= 3311; index_13++)
{
  convVector_12[index_13] = convVector_11[index_13] * convScalar_10;
  convVector_13[index_13] = convVector_8[index_13] + convVector_12[index_13];
}
```

Appendix C

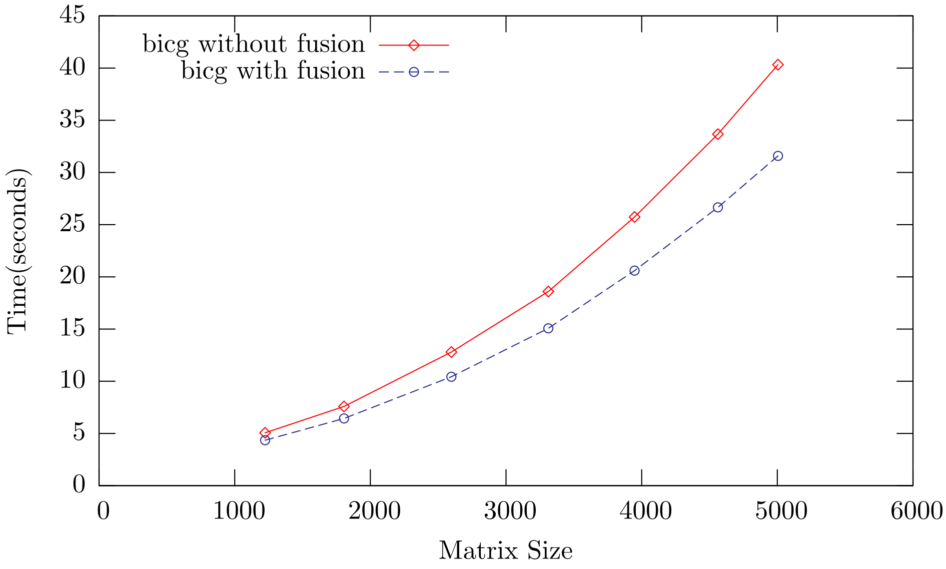
Graphs of Collected Results

In this section, I have included all graphs comparing the performance of loop fusion, array contraction and runtime liveness analysis functionality implemented in the prototype library. Details of the platforms and test data used are present in the Conclusion. Also present in the Conclusion are the benchmark results of the prototype library against MTL.

C.1 Loop Fusion

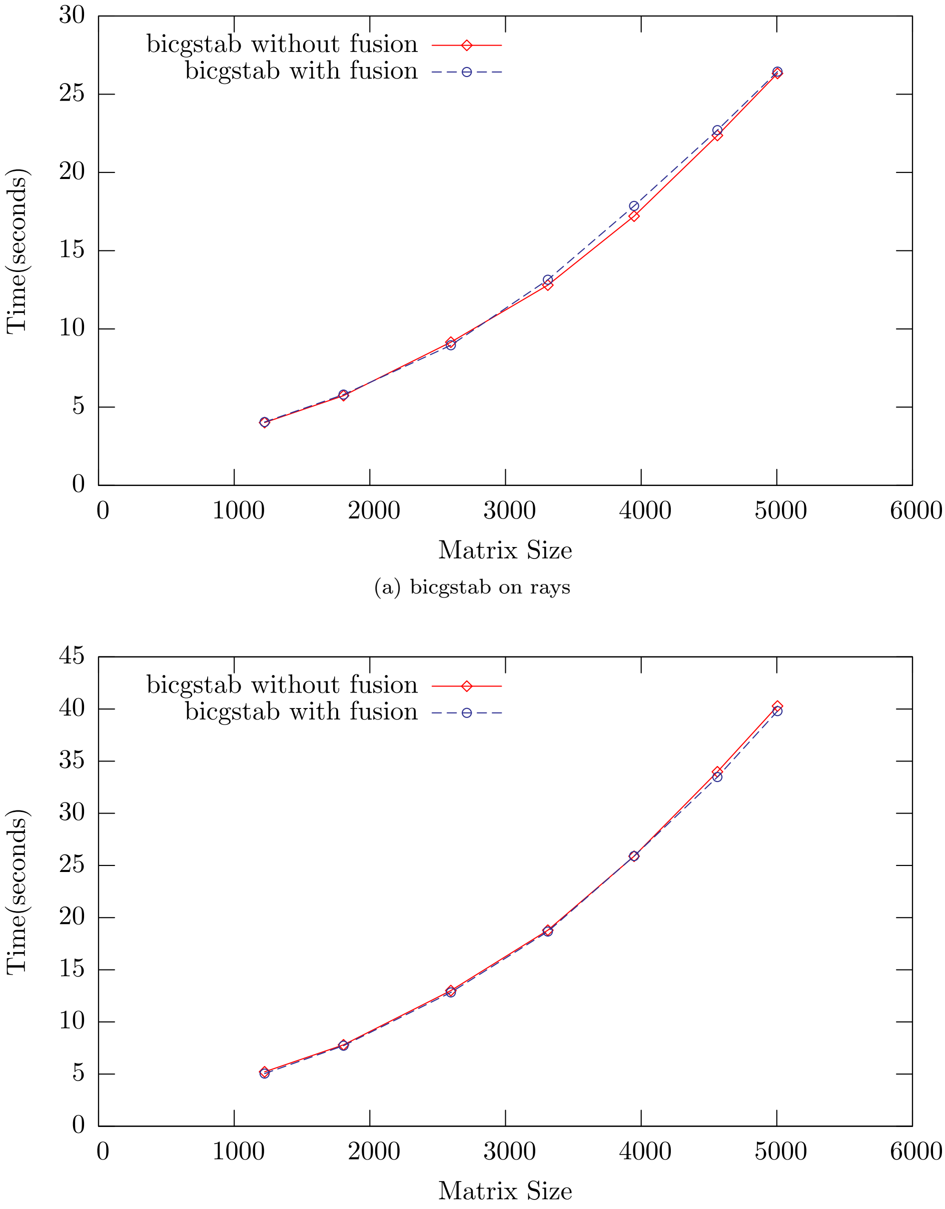


(a) bicg on rays



(b) bicg on vertices

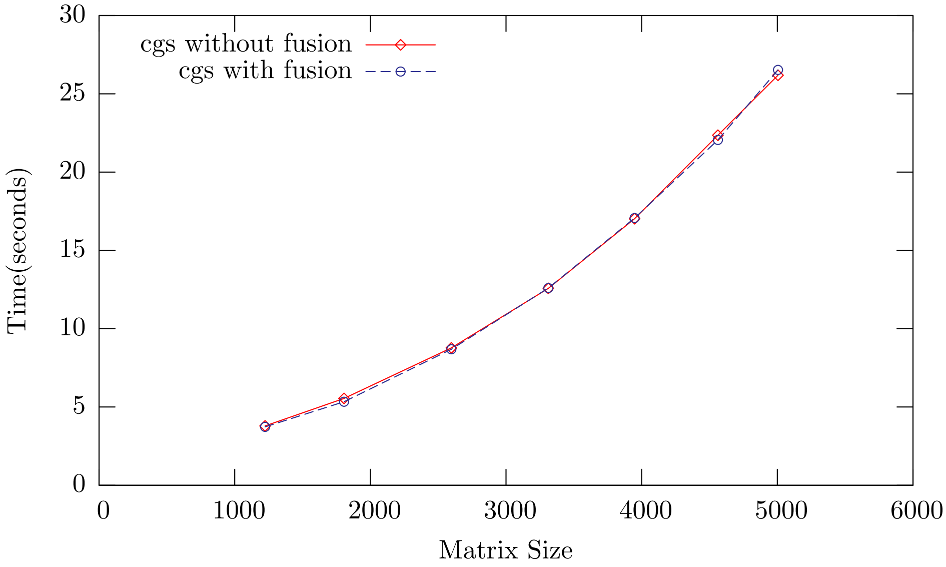
Figure C.1: Time to execute 256 iterations of the BiConjugate Gradient solver with and without loop fuser enabled (lower is better).



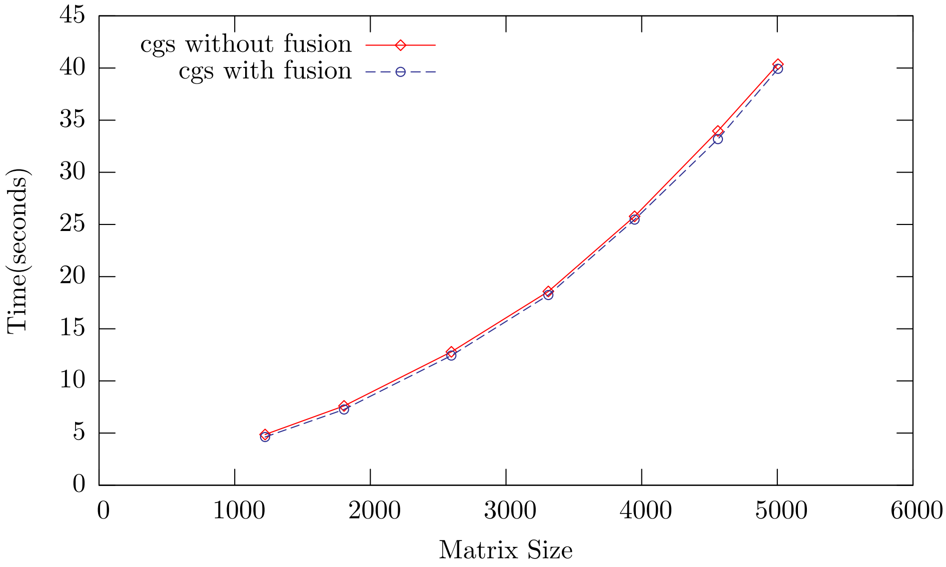
(a) bicgstab on rays

(b) bicgstab on vertices

Figure C.2: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with and without loop fuser enabled (lower is better).

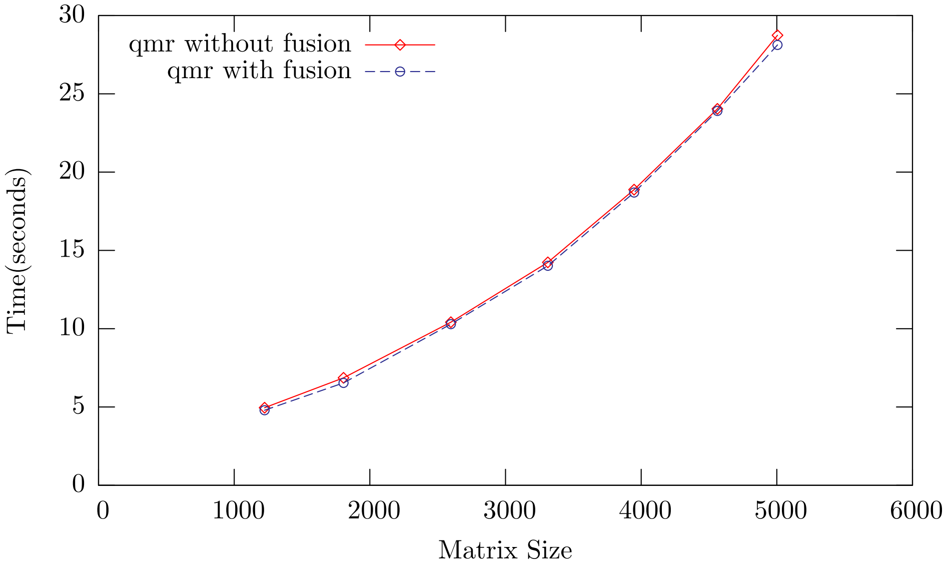


(a) cgs on rays

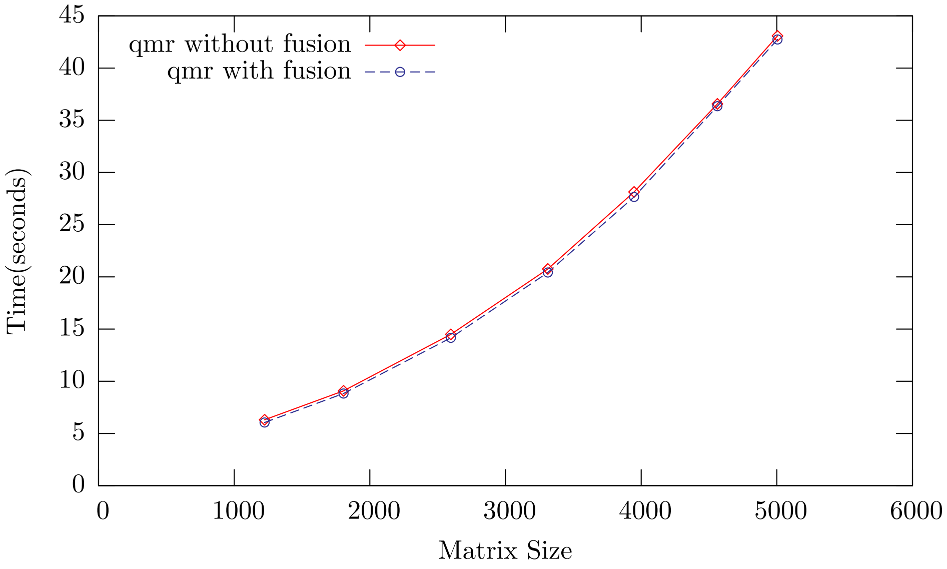


(b) cgs on vertices

Figure C.3: Time to execute 256 iterations of the Conjugate Gradient Squared solver with and without loop fuser enabled (lower is better).

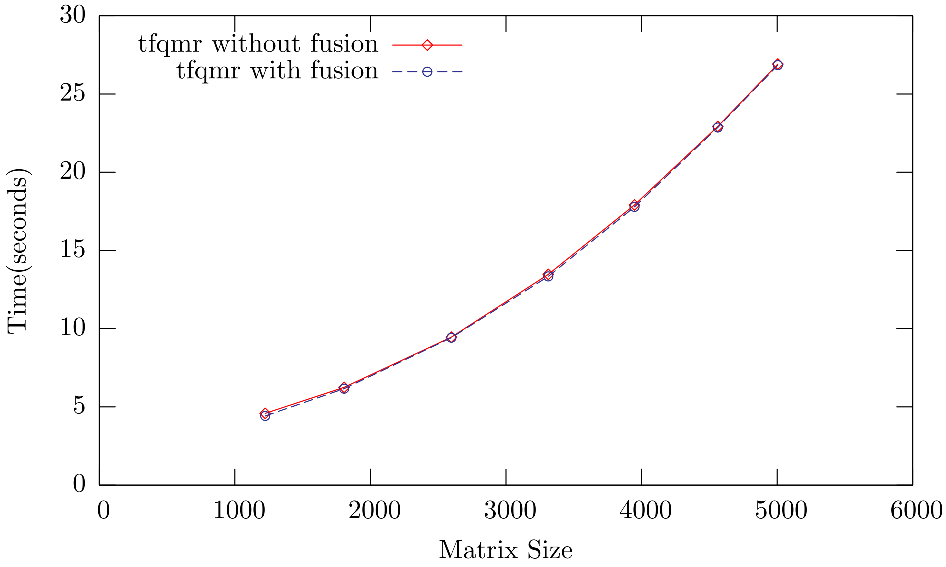


(a) qmr on rays

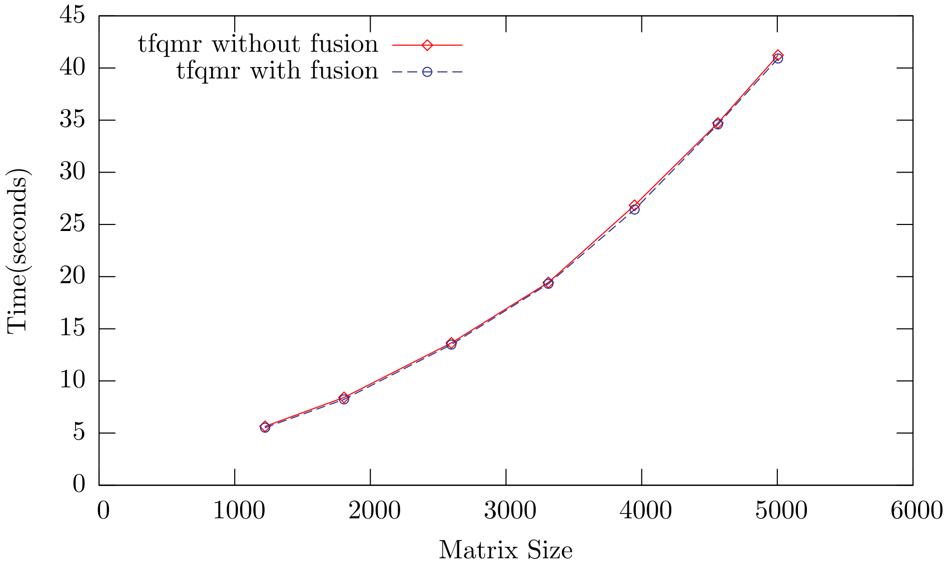


(b) qmr on vertices

Figure C.4: Time to execute 256 iterations of the Quasi-Minimal Residual solver with and without loop fuser enabled (lower is better).



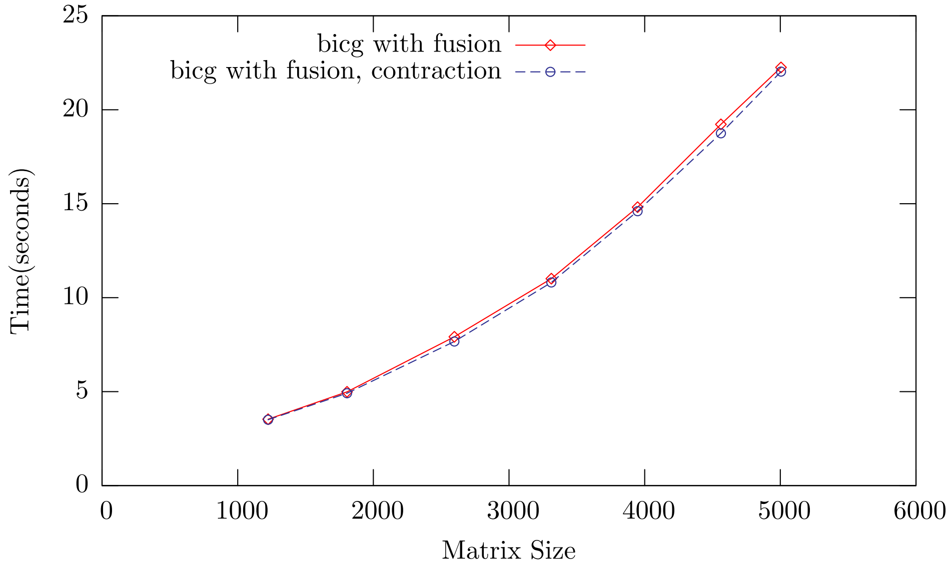
(a) tfqmr on rays



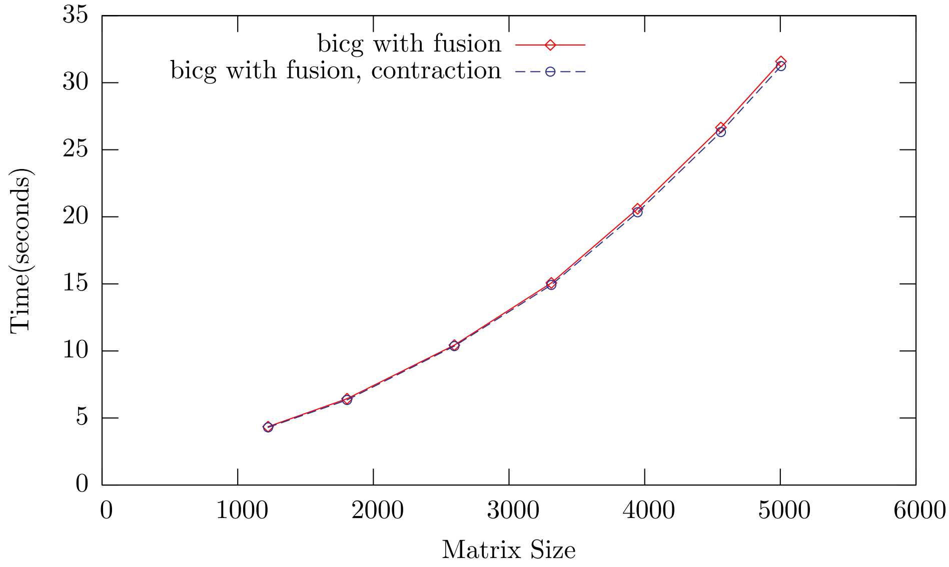
(b) tfqmr on vertices

Figure C.5: Time to execute 256 iterations of the Transpose Free Quasi-Minimal Residual solver with and without loop fuser enabled (lower is better).

C.2 Array Contraction

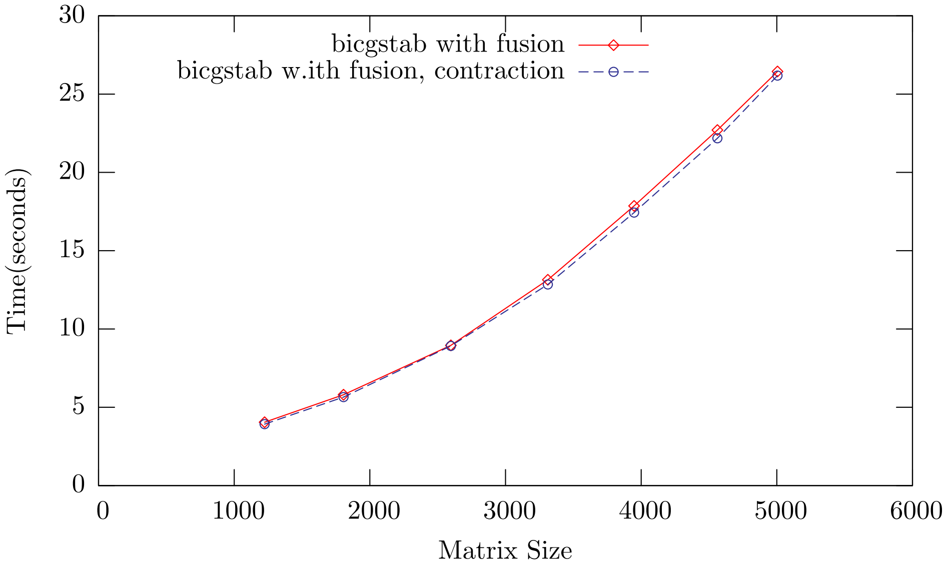


(a) bicg on rays

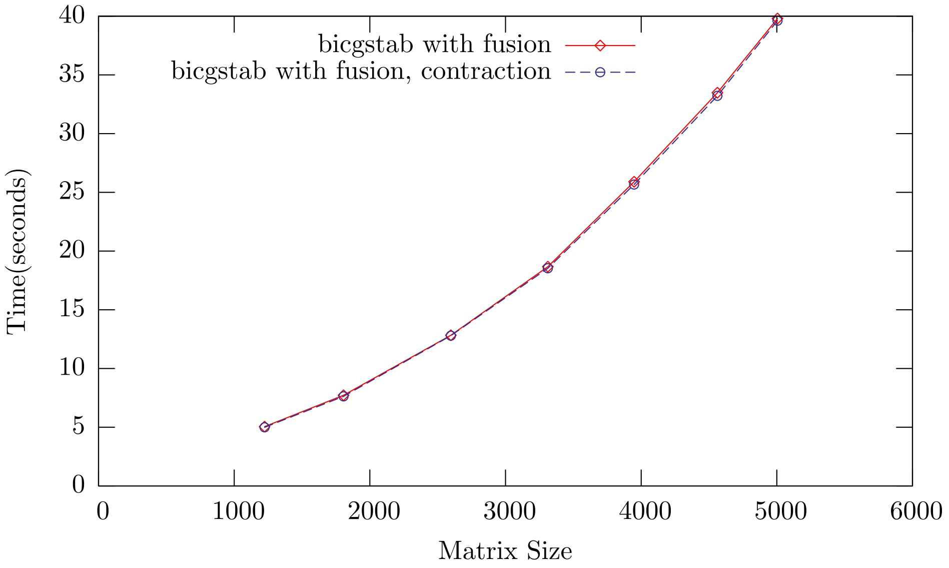


(b) bicg on vertices

Figure C.6: Time to execute 256 iterations of the BiConjugate Gradient solver with and without array contraction enabled (lower is better). Both solvers have loop fusion enabled.

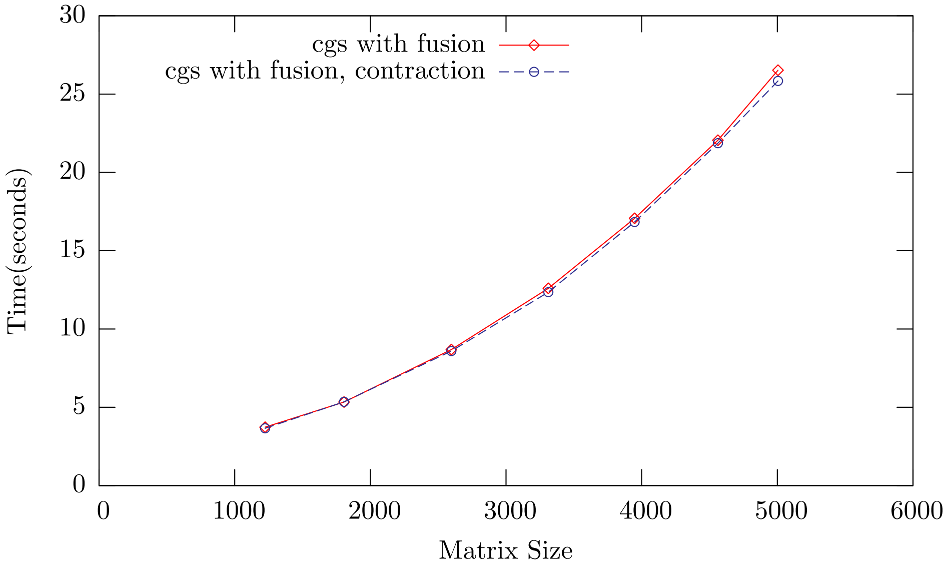


(a) bicgstab on rays

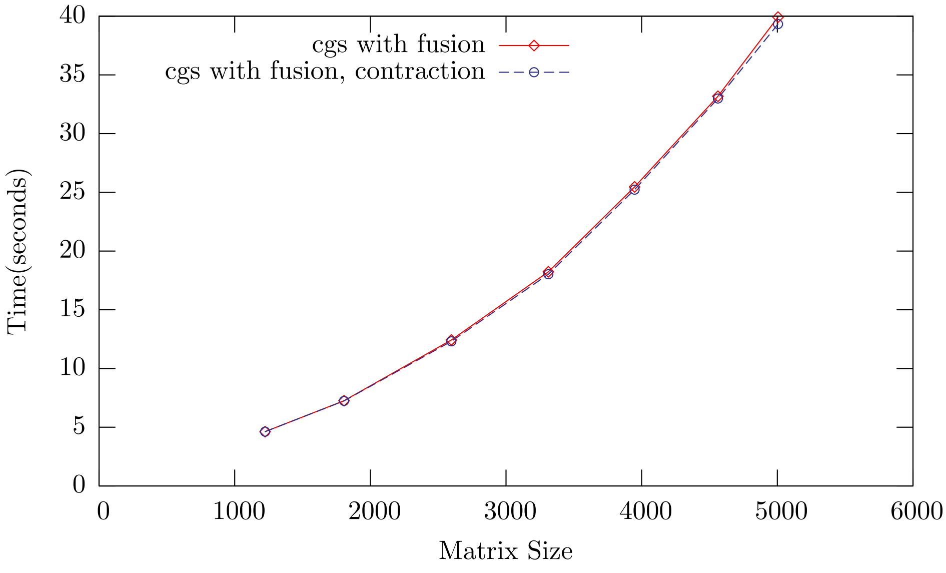


(b) bicgstab on vertices

Figure C.7: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with and without array contraction enabled (lower is better). Both solvers have loop fusion enabled.

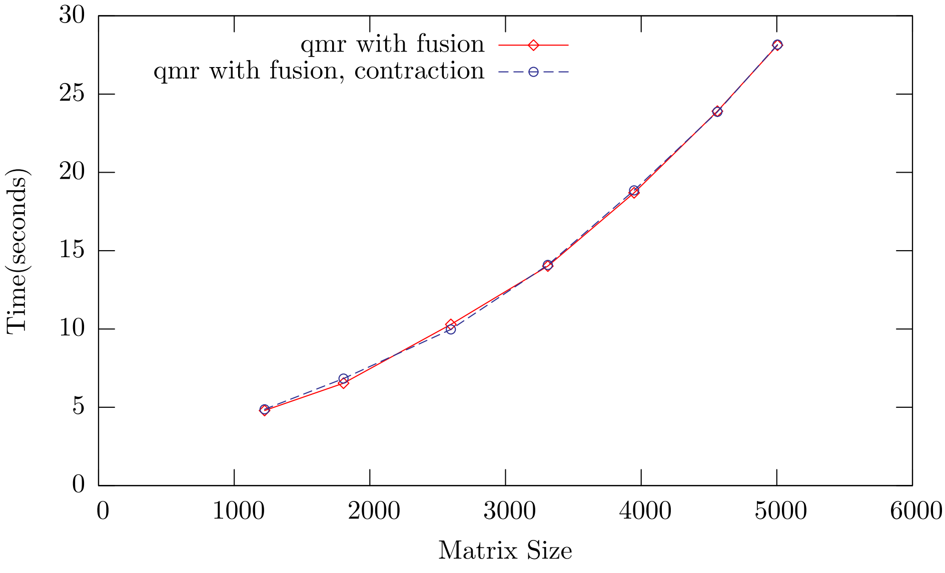


(a) cgs on rays

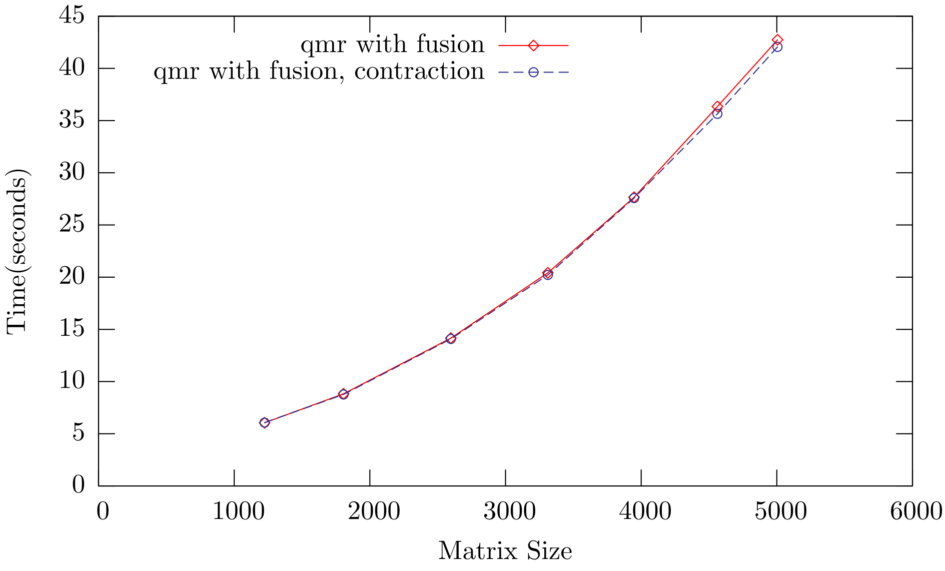


(b) cgs on vertices

Figure C.8: Time to execute 256 iterations of the Conjugate Gradient Squared solver with and without array contraction enabled (lower is better). Both solvers have loop fusion enabled.

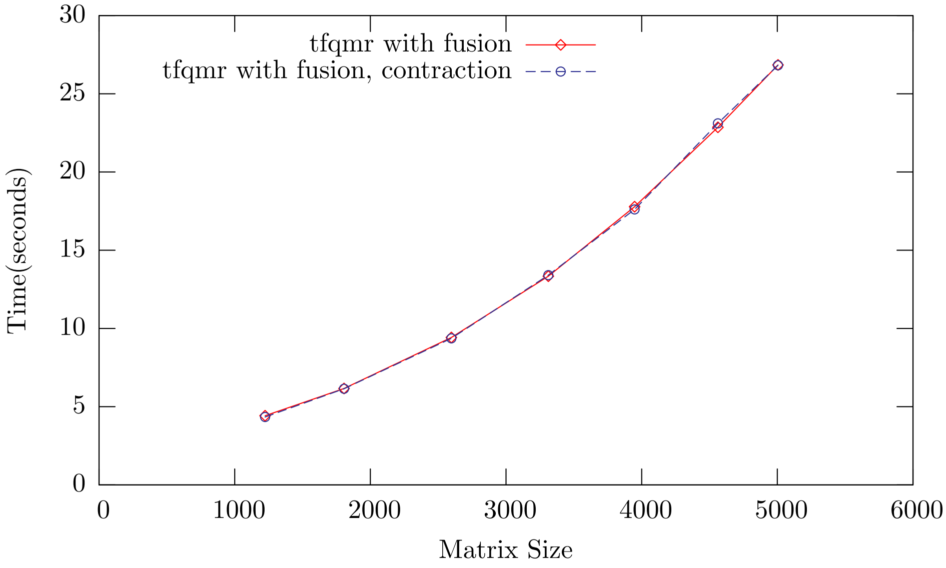


(a) qmr on rays

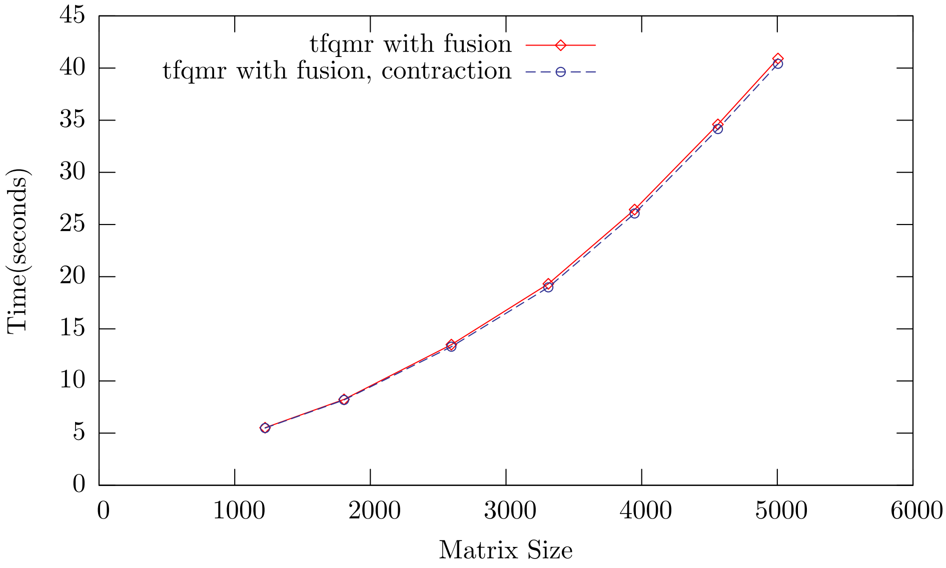


(b) qmr on vertices

Figure C.9: Time to execute 256 iterations of the Quasi-Minimal Residual solver with and without array contraction enabled (lower is better). Both solvers have loop fusion enabled.



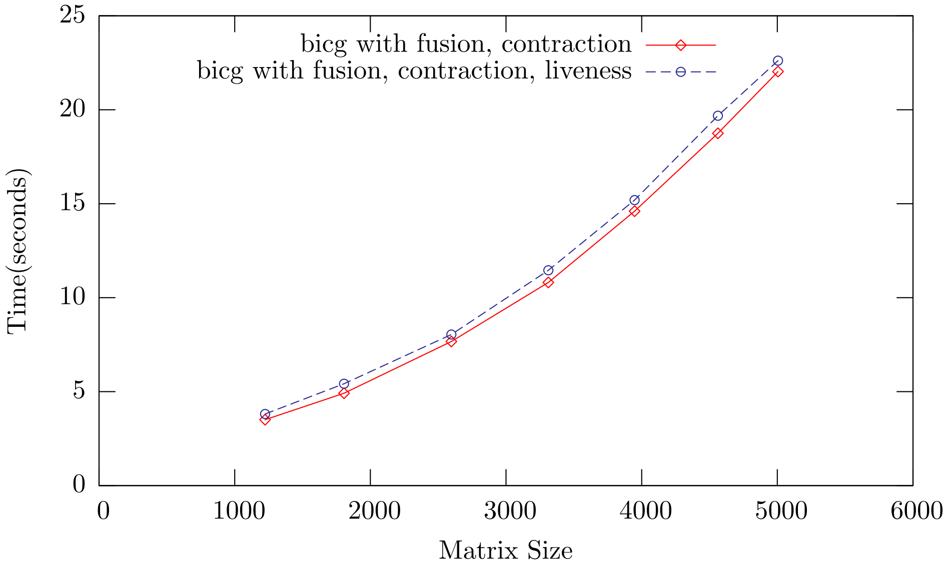
(a) tfqmr on rays



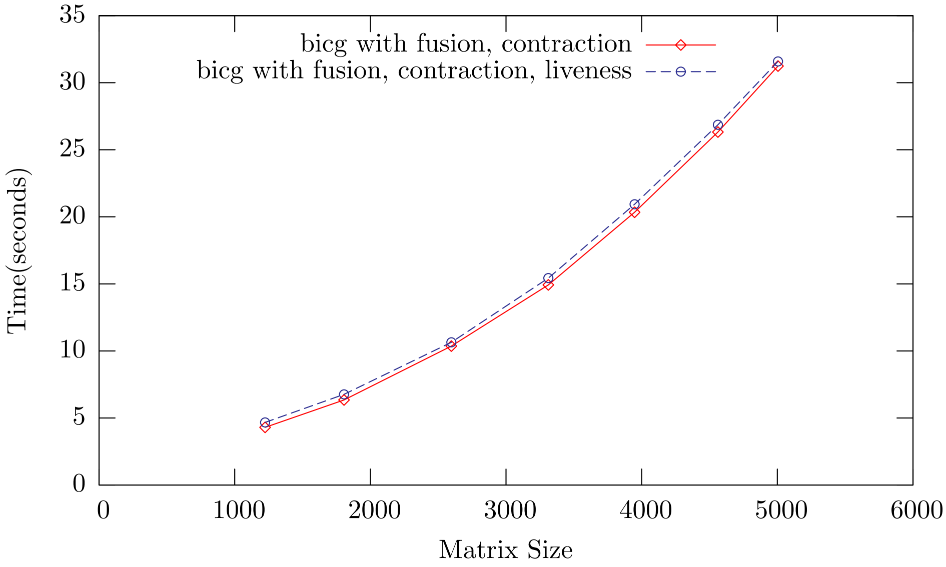
(b) tfqmr on vertices

Figure C.10: Time to execute 256 iterations of the Transpose Free Quasi-Minimal Residual solver with and without array contraction enabled (lower is better). Both solvers have loop fusion enabled.

C.3 Runtime Liveness Analysis

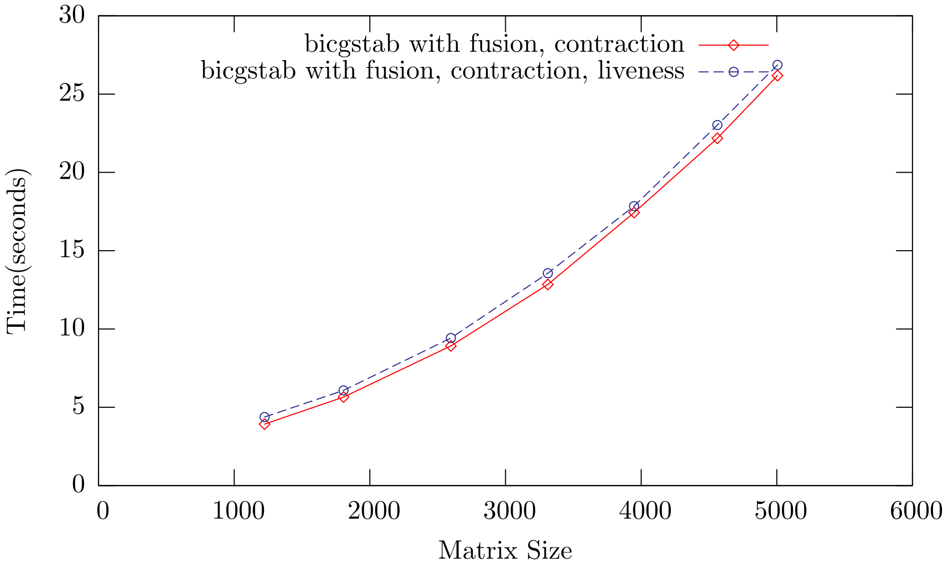


(a) bicg on rays

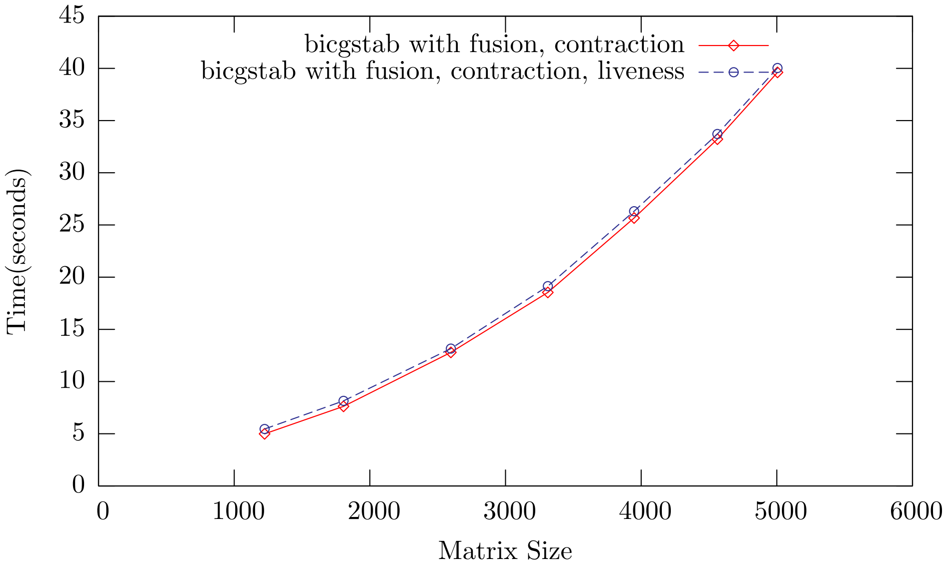


(b) bicg on vertices

Figure C.11: Time to execute 256 iterations of the BiConjugate Gradient solver with and without liveness analysis enabled (lower is better). Both solvers have loop fusion and array contraction enabled.

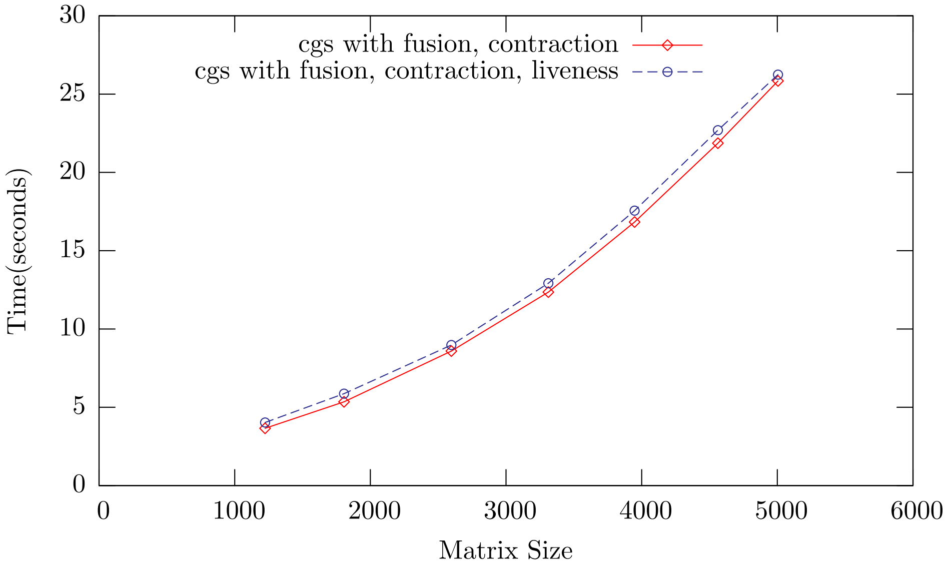


(a) bicgstab on rays

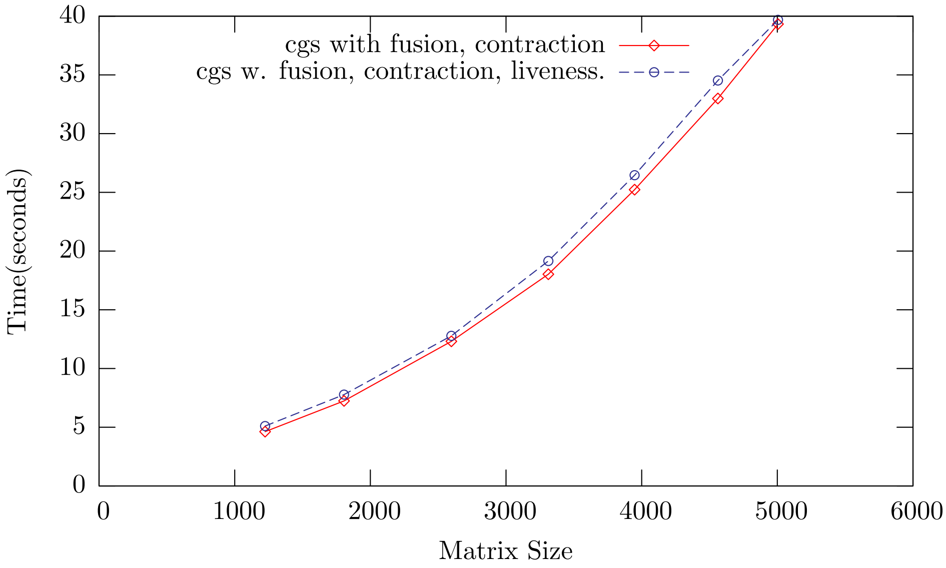


(b) bicgstab on vertices

Figure C.12: Time to execute 256 iterations of the BiConjugate Gradient Stabilised solver with and without liveness analysis enabled (lower is better). Both solvers have loop fusion and array contraction enabled.

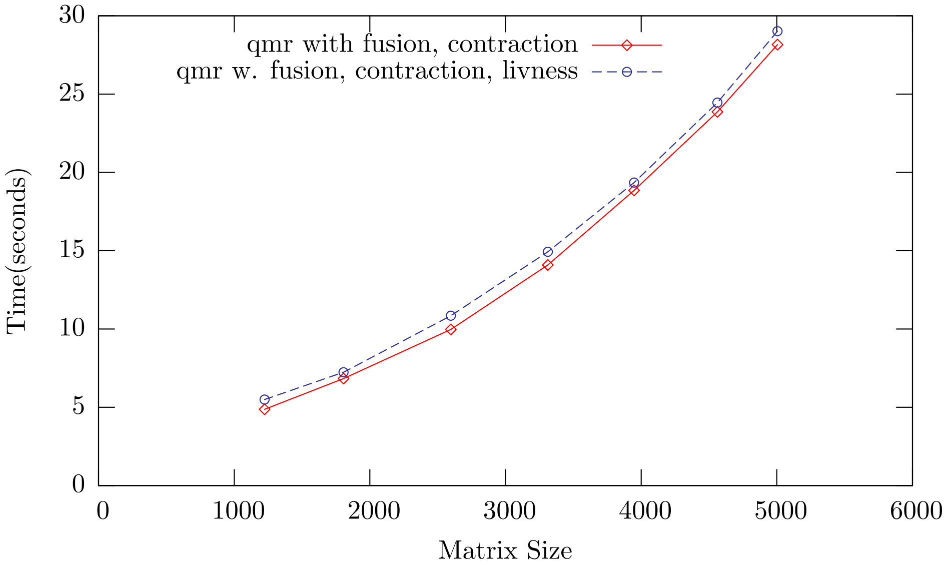


(a) cgs on rays

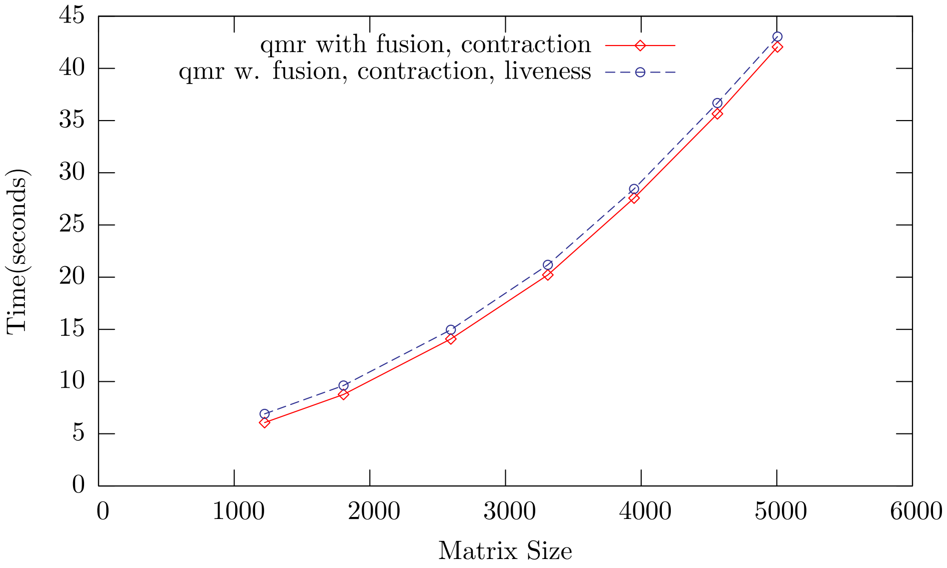


(b) cgs on vertices

Figure C.13: Time to execute 256 iterations of the Conjugate Gradient Squared solver with and without liveness analysis enabled (lower is better). Both solvers have loop fusion and array contraction enabled.

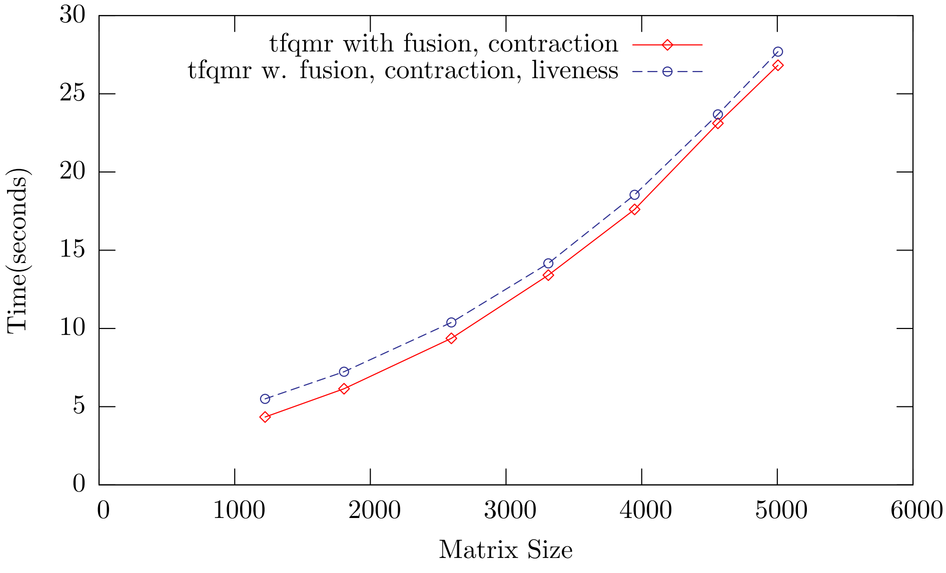


(a) qmr on rays

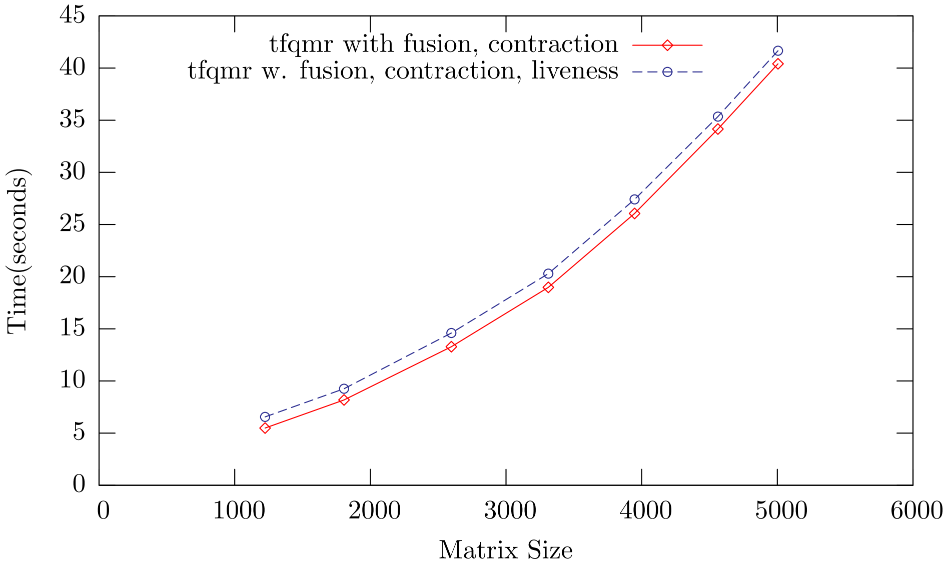


(b) qmr on vertices

Figure C.14: Time to execute 256 iterations of the Quasi-Minimal Residual solver with and without liveness analysis enabled (lower is better). Both solvers have loop fusion and array contraction enabled.



(a) tfqmr on rays



(b) tfqmr on vertices

Figure C.15: Time to execute 256 iterations of the Transpose Free Quasi-Minimal Residual solver with and without liveness analysis enabled (lower is better). Both solvers have loop fusion and array contraction enabled.