# DESOLA: an Active Linear Algebra Library Using Delayed Evaluation and Runtime Code Generation [⋆]

Francis P. Russell, Michael R. Mellor, Paul H. J. Kelly, Olav Beckmann

*Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK*

**Abstract**

Active libraries can be defined as libraries which play an active part in the compilation, in particular, the optimisation of their client code. This paper explores the implementation of an active dense linear algebra library by delaying evaluation of expressions built using library calls, then generating code at runtime for the compositions that occur. The key optimisations in this context are loop fusion and array contraction.

Our prototype C++ implementation, DESOLA, automatically fuses loops arising from different client calls, identifies unnecessary intermediate temporaries, and contracts temporary arrays to scalars. Performance is evaluated using a benchmark suite of linear solvers from ITL (Iterative Template Library), and is compared with MTL (Matrix Template Library), ATLAS (Automatically Tuned Linear Algebra) and IMKL (Intel Math Kernel Library). Excluding runtime compilation overheads (caching means they occur only on the first iteration), for larger matrix sizes, performance matches or exceeds MTL; when fusion of matrix operations occurs, performance exceeds that of ATLAS and IMKL.

*Key words:* runtime code generation, delayed evaluation, active libraries, numerical libraries

[⋆] This paper is an extended version of a submission presented at the 2006 Library-Centric Software Design Workshop entitled "An Active Linear Algebra Library Using Delayed Evaluation and Runtime Code Generation" [20]. Changes include the integration of related work into a new Background section, the addition of a Discussion section and fresh results. Since the LCSD'06 paper, the library described has been modified to decrease overhead.

## 1 Introduction

The idea of an "active library" is that, just as the library extends the language available to the programmer for problem solving, so the library should also extend the compiler. The term was coined by Czarnecki et al. [7], who observed that active libraries break the abstractions common in conventional compilers. Active libraries are described in detail by Veldhuizen and Gannon [26].

This paper presents DESOLA (Delayed Evaluation Self Optimising Linear Algebra), a prototype linear algebra library which we have developed in order to explore one interesting approach to building active libraries. The idea is to:

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

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

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

One aspect of this approach is that the library interface remains isolated from the concerns of achieving high performance. As we discuss later in Section 2.1, the evolution of high performance numerical libraries has been accompanied by a corresponding increase in the complexity of their interfaces. By allowing the library implementation to take more responsibility for optimisation, we aim to provide a more appropriate interface to the user, a similar goal to the Matrix Template Library [21].

Another aspect of this approach is that the code generated for a recipe is isolated from client-side code - it is not interwoven with non-library code. This is particularly important because the structure of the code for a recipe is restricted in form which enables us to introduce compilation passes specially targeted to achieve particular effects.

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

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

Our exploration covers the following ground:

(1) We present an implementation of a C++ library for dense linear algebra which provides functionality sufficient to operate with the majority of methods available in the Iterative Template Library [16] (ITL), a set of templated linear iterative solvers for C++.
(2) This implementation delays execution, generates code for delayed recipes at runtime, and then invokes a vendor C compiler at runtime - entirely transparently to the library user.
(3) To avoid repeated compilation of recurring recipes, we cache compiled code fragments (see Section 5).
(4) We implemented two optimisation passes which transform the code prior to compilation: loop fusion, and array contraction (see Section 6).
(5) We introduce a scheme to attempt to predict, statistically, which intermediate variables are likely to be used after recipe execution; this is used to increase opportunities for array contraction (see Section 7).
(6) We evaluate the effectiveness of the approach using a suite of iterative linear system solvers, taken from the Iterative Template Library (see Section 8).

Although the exploration of these techniques has used only dense linear algebra, we believe these techniques are more widely applicable. Dense linear algebra provides a simple domain in which to investigate, understand and demonstrate these ideas. Other domains we believe may benefit from these techniques include sparse linear algebra (work in progress) and image processing [6].

The contributions we make with this paper are as follows:

- Compared to the widely used Matrix Template Library [21], we demonstrate performance improvements across our benchmark suite of dense linear iterative solvers from the Iterative Template Library.
- We present a cache architecture that finds applicable pre-compiled code quickly, and which supports annotations for adaptive re-optimisation.

- Using our experience with this library, we discuss some of the design issues involved in using the delayed-evaluation, runtime code generation technique.

## 2 Background

We begin with numerical libraries and issues related to their performance, then the techniques we employ in DESOLA to enable it to be "active", and lastly, the compiler optimisations we use to improve the performance of our runtime generated code. We assume that the reader already has a basic familiarity with dependence analysis. Those unfamiliar are invited to consult Bacon et al. [2].

### *2.1 BLAS*

The Basic Linear Algebra Subprograms [15] are routines written in Fortran which provide basic building blocks for vector and matrix operations. They are frequently used in high-performance numerical computing. Processor vendors commonly supply tuned BLAS implementations targeted towards the architectures they manufacture.

BLAS routines are classified into three levels:

**Level 1** The original set of BLAS routines. They perform vector, scalar and vector-vector operations.
**Level 2** Matrix-vector operations.
**Level 3** Matrix-matrix operations.

BLAS functions typically 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 placed on the programmer to ensure that they use the correct routines to do this.

In their paper on BLAS level 2 [11], 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. They state that this is due to the BLAS level 1 interface inhibiting optimisations. In particular, on vector machines, the full nature of the matrix-vector operations is not apparent to the compiler, hence the development of Level 2 BLAS which encapsulated matrix-vector functionality. Similarly, in their paper on Level 3 BLAS [10], Dongarra et al. note that Level 2 BLAS does not translate well to computers with a memory hierarchy because data is not reused effectively. Level 3 BLAS allows higher performance by using blocked algorithms, which exploit the memory hierarchy. The evolution of BLAS clearly shows that by optimising across a series of smaller operations, benefits can be achieved that were previously unavailable.

The BLAS interfaces were designed for performance. Each function, especially those at the higher levels, takes a large number of parameters, and performs a number of more fundamental operations simultaneously. While 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, and the more complicated the development of numerical software.

Blackford et al. have proposed an updated BLAS [5] which adds new operations, and extends existing ones included some which perform level 1 and level 2 BLAS operations simultaneously. Although new BLAS interfaces can be devised each time a particular set of computations are recognised as performance critical kernels, we believe such an approach is not sustainable.

The addition of new composed kernels with each extension of the BLAS interface enables the reduction of memory traffic and provides greater scope for optimisation. It also indicates the need for a mechanism that enables the composition and optimisation of arbitrary combinations of kernels as can occur in different numerical applications.

One of the aims of this project is to investigate whether our approach allows the creation of high performance numerical libraries. We want to be able to create composed and optimised kernels without having to sacrifice interface usability as in the case of BLAS. Another technique is C++ template metaprogramming as used by the Matrix Template Library [21], described in Section 2.3.

## *2.2 Cross Component Optimisation*

Good software engineering practice dictates that software should be written in a modular manner. Software components often simplify testing and debugging, enable abstractions to be defined at component boundaries and promote encapsulation and re-usability.

In the context of cross component optimisation, we define components as any form of software building block from procedure calls to library code. While the use of components in software engineering has many useful benefits, it is often the case that components inhibit the optimisations that can be applied when they are composed together. When we talk about cross-component optimisation, we refer to optimisations that improve the performance of a composition of software components.

We have already discussed the evolution of BLAS, and the benefits possible by optimising across different linear algebra operations. Work showing the effectiveness of cross component optimisation is presented by Ashby et al. [1] in which the performance of an iterative solver, is analysed when implemented with ATLAS, Fortran and Aldor.

The Aldor algorithm implementation of the iterative solver uses level 1 BLAS routines, also implemented in 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.

A comparison is made between the solvers using ATLAS, Fortran and optimised/unoptimised Aldor implementations. Analysis of the optimised Aldor solvers showed that the compiler had 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 not analysed.

It is these types of optimisations we wish to be able to exploit in DESOLA. In particular, we aim to exploit loop fusion and array contraction. In order to do this in a C++ library without using a custom compiler, we perform these operations at runtime. This has both benefits and disadvantages compared to the approach taken by Ashby et al. [1]. By delaying optimisation and code generation until runtime, we incur a significant overhead, however, we also get access to runtime values that would have been unknown at link time.

## *2.3 Template Metaprogramming and the Matrix Template Library*

Template metaprogramming [23] is a C++ technique that uses the compiler as a compile-time interpreter. Runtime *for* loops and *if* statements are replaced by compile-time template specialisation and recursion. These techniques have proved useful for writing high performance numerical libraries in C++ such as Blitz++ [25] and MTL [21].

A technique called expression templates [24] allows C++ libraries to control the parsing of expressions. This is extremely useful for numerical applications where a naive C++ implementation of matrices and vectors results in code that contains numerous loops and temporary array allocations.

One numerical library using these techniques is Blitz++ [25] which uses template metaprogramming to perform optimisations such as loop interchange, collapsing and partial unrolling of inner loops, hoisting of invariant stride computations and tiling to optimise cache use.

It is worth noting that libraries such as MTL and Blitz++ use template metaprogramming to overcome the usual restrictions imposed by software engineering abstractions. The object code for performing numerical operations is generated in the client executable at compile time, rather than in a precompiled library that it links against.

We now describe the Matrix Template Library [21], a state of the art numerical library for C++ using template metaprogramming techniques. We compare the performance of DESOLA to MTL using a suite of dense iterative solvers later in this paper. The Matrix Template Library 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 [22], 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(y, 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 while maintaining abstractions. Thus, MTL demonstrates that high performance numerical code in C++ using abstractions is possible.

MTL has the same goals as our research, to provide a numerical library with a simple interface capable of achieving high performance. MTL uses template metaprogramming to achieve greater control over the generated code than other libraries. In contrast, our approach which uses runtime code generation to achieve this greater level of control.

## *2.4 Delayed Evaluation, Self-Optimising Software Components*

Beckmann and Kelly describe a delayed-evaluation self-optimising linear algebra library [4] 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.

The authors also present a strategy 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 further and whether or not its last usage was sub-optimal.

Beckmann and Kelly's library effectively demonstrates that delayed-evaluation self-optimising software components can obtain context information only present at runtime. It also shows that the collected runtime context information can be used to improve performance.

Further work by Liniker et al. [18] 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++ [9], a set of templates for iterative solving methods in C++.

We use a similar approach to attempt to improve performance in DESOLA. However, we use runtime code generation and transformation, as opposed to data placement, as the mechanism for improving the performance of our code.

## *2.5 Runtime Code Generation*

Runtime code generation enables applications to specialise themselves more effectively to their input. DESOLA uses runtime code generation to allow it to generate more optimised and specialised code than would be possible at runtime. We briefly compare the TaskGraph approach to two other systems for runtime code generation.

**Fabius [17]** A prototype compiler developed to research the notion of deferred compilation. The Fabius compiler compiles a rudimentary, strict, first-order functional language.

**Tick C [12]** A superset of ANSI C designed to allow high level, efficient machine-independent specification of dynamic code.

Comparing TaskGraph, Tick C and Fabius shows the variability in possible approaches to runtime code generation. They differ in a number of important aspects including:

**Programmer Involvement** In the TaskGraph system, the programmer has to explicitly construct the runtime generated code and controls the optimisation. Tick C also requires the programmer to specify the runtime generated code using the additional language constructs. Fabius does this less explicitly, using programmer hints and syntactic cues to determine what code should be subjected to runtime code generation.

**Overhead** TaskGraph runtime code generation incurs a significant overhead. TaskGraph requires that the abstract syntax tree of the runtime generated code be manipulated at runtime and invokes a stand-alone C compiler to perform its compilation. Fabius uses a much more “lightweight” approach. Compiled code performing code generation in Fabius is “hardwired” to produce code for a small portion of the input program. tcc [19], a compiler for

Tick C has two runtime backends allowing the user to choose between speed or quality of code generation. The higher quality code generation processes an intermediate form to provide efficient register allocation while the faster backend generates code without analysis.

**Language Support** Tick C takes the approach of extending the C language thus requiring a custom compiler to support it. Fabius uses programmer hints to determine what code should be generated at runtime, and as such also depends upon language support. The TaskGraph approach implements a C-like sub-language in C++ using macros and templates and avoids the requirement of a custom C++ compiler.

**Portability** Fabius and Tick C portability depends on their respective compilers and code generation mechanisms. TaskGraph uses a stand-alone C compiler to compile its code and thus is portable to any platform where a C compiler already exists. This also enables us to leverage the optimisation abilities of commercial C compilers such as ICC.

Previous work by Beckmann [3] using the TaskGraph library demonstrates the effectiveness of specialisation and runtime code generation as a mechanism for improving the performance of various applications. The TaskGraph library is used to generate specialised code for the application of a convolution filter to an image. As the size and the values of the convolution matrix are known at the runtime code generation stage, the two inner loops of the convolution can be unrolled and specialised with the values of the matrix elements.

With previous research involving TaskGraph showing promising results with runtime code generation, we decided to see if the performance benefits provided by runtime code generation could be employed in a numerical library in a transparent manner.

## *2.6 Loop Fusion*

Loop fusion [2] can improve performance by reducing loop overhead, increasing instruction parallelism and improving locality in the registers, data cache, TLB or page 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.

In general, optimal loop fusion for maximising parallelism or locality is NP-complete [8]. However, there exist optimal algorithms for subsets of the loop fusion problem and heuristics for the more general problem. Kennedy has proposed a greedy algorithm for maximising memory reuse between loops [14]. Gao et al. have proposed a heuristic for fusing loops to maximise array contraction [13]. Yi and Kennedy have proposed a transformation they call *dependence hoisting* that combines loop interchange and fusion for nested loops to improve memory hierarchy performance [29].

We use loop fusion in DESOLA in an attempt to improve cache locality and reduce loop overhead. In particular, we observe that the runtime generated code contains large numbers of fusible vector-vector operations. Loop fusion facilitates array contraction [2], which we implement as well.

### *2.7 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 often facilitated by loop fusion. 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.

We use array contraction in DESOLA to attempt to reduce the memory usage of the loops we have fused together. Our experimental results presented later show that after loop fusion, we were able to remove significant numbers of temporary vectors from our benchmark applications.

## 3 Delaying Evaluation

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

This information may consist of values such as matrix or vector sizes, or the various relationships between successive library calls. Knowledge of dynamic values such as matrix and vector sizes allows us to improve the performance of the implementation of operations using these objects. For example, the runtime code generation system (see Section 4) can use this information to specialise the generated code. One specialisation we do is with loop bounds. We incorporate dynamically known sizes of vectors and matrices as constants in the runtime-generated code.

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

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

An example DAG is illustrated in Figure 1. The rectangular node represents a handle held by the library client, and the other nodes represent delayed expressions. The three multiplication nodes do not have a handle referencing them. This makes them in effect, unnamed. When the expression DAG is evaluated, it is possible to optimise away the storage for these values entirely (their values are not required outside the runtime generated code). For expression DAGs involving matrix and vector operations, this enables us to reduce memory usage and improve cache utilisation.

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

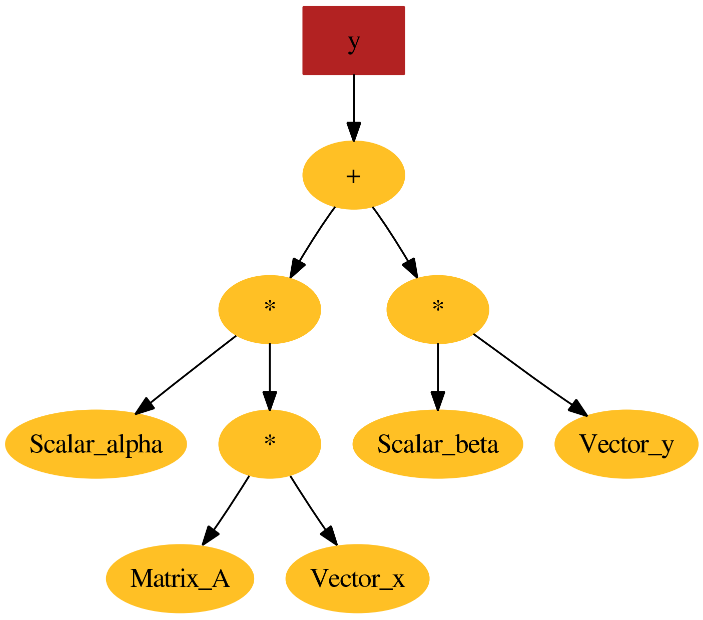

Fig. 1. An example DAG. The rectangular node denotes a handle held by the library client. The expression represents the matrix-vector multiply function from Level 2 BLAS, $y = \alpha Ax + \beta y$.

## 4 Runtime Code Generation

Runtime code generation is performed using the TaskGraph [3] system. The TaskGraph library is a C++ library for dynamic code generation. A TaskGraph represents a fragment of code which can be constructed and manipulated at runtime, compiled, dynamically linked back into the host application and executed. TaskGraph enables optimisation with respect to:

**Runtime Parameters** 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 sub-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.

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

Code generated by DESOLA is specialised to matrix and vector sizes as in the example above. The constant loop bounds and array sizes make it much simpler to apply our loop fusion and array contraction optimisations. These are described in Section 6.

## 5 Code Caching

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

Related work by Beckmann [4] on the efficient placement of data in a parallel linear algebra library cached execution plans in order to improve performance. We adopt a similar strategy in order to reuse previously compiled code. We maintain a cache of previously encountered recipes along with the compiled code required to execute them. As any caching system would be invoked at every force point within a program using the library, it was essential that checking for cache hits would be as computationally inexpensive as possible.

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

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

This causes two limitations:

- We do not detect where the presence of commutative operations allow two differently structured delayed expression DAGs to be used in place of each other.
- As there can be more than one valid topological sort of a DAG, it is possible for multiple identically structured expression DAGs to exist in the code cache.

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

Hashing occurs as follows:

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

Isomorphism checking works similarly:

- Nodes in the sorted lists for each graph are assigned a value corresponding to their location in their list.

- Both lists are checked to be the same size.
- The corresponding nodes from both lists are checked to be the same type, and any nodes they reference are checked to see if they have been assigned the same numerical value.

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

As the maximum number of other nodes a node can depend on is bounded (maximum of two for a library with only unary and binary operators) then both hashing and isomorphism checking between delayed expression DAGs can be performed in linear time with respect to the number of nodes in the DAG.

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

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

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

## 6 Loop Fusion and Array Contraction

We implemented two optimisations using the TaskGraph back-end, SUIF [28]. Both loop fusion and array contraction are applied to the runtime generated code. As loop fusion often facilitates array contraction, the loop fusion pass precedes the array contraction pass.

Loop fusion [2] can lead to an improvement in performance when the fused loops use the same data. As the data is only loaded into the cache once, the fused loops take less time to execute than the sequential loops. Alternatively, if the fused loops use different data, it can lead to poorer performance, as the data used by the fused loop displace each each other in the cache. Loop fusion is described in more detail in Section 2.6.

A brief example involving two vector additions. Before loop fusion:

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

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

After loop fusion:

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

In this example, after fusion, the value stored in vector $a$ can be reused for the calculation of $e$.

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

As discussed in Section 4 we employ a heuristic to generate code where loops are likely to reference the same data. Visual inspection of the code generated during execution of the iterative solvers indicates that the fused loops commonly use the same data. We believe this is likely due to the structure of the dependencies involved in the operations required for the iterative solvers.

We follow loop fusion by 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. Array contraction is described in more detail in Section 2.7. We also provide results on the number of array contractions we perform on our benchmarks in Section 8.

Another example. Before array contraction:

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

After array contraction:

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

Here, the array *a* can be reduced to a scalar value as long as it is not required by any code following the two fused loops.

We use this to technique to optimise away temporary matrices or vectors in the runtime generated code. This is important because the DAG representation of the delayed operations does not hold information on what memory can be reused. However, we can determine whether or not each node in the DAG is referenced by the client code, and if it is not, it can be allocated locally to the runtime generated code and possibly optimised away. For details of other memory access transformations, consult Bacon et al. [2].

## 7 Liveness Analysis

When analysing the runtime generated code produced by the iterative solvers, it became apparent that a large number of vectors were passed in as parameters. Their initial values were not being used by the runtime generated code, instead, they were being passed in so they could be assigned to and propagate values out of the runtime generated code.

On further investigation we discovered that a number of these vectors were not used outside the runtime generated code, but our library could not determine this because handles to the vectors were still held by the iterative solver. This meant that our library could not optimise them away because they were not being allocated locally to the runtime generated code. We realised that by designing a system to recover runtime information, we had lost the ability to use static information, in particular the liveness information of variables.

Consider the following code that takes two vectors, finds their cross product, scales the result and prints it:

```
    void printScaledCrossProduct(Vector<float> a,
                                 Vector<float> b,
```

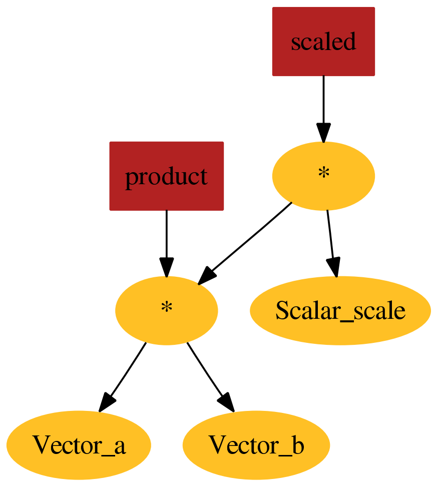

Fig. 2. A DAG representing the cross product of two vectors multiplied by a scalar. Upon evaluating the multiplication, the library must assume that the cross product could be used later as the client still holds a handle to it.

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

This operation can be represented with the DAG in Figure 2. The value pointed to by the handle *product* is never required by the library client. From the client's perspective the value is dead, but the library must assume that any value which has a handle may be required later on. Values required by the library client cannot be allocated locally to the runtime generated code, and therefore cannot be optimised away through techniques such as array contraction. Runtime liveness analysis permits the library to make estimates about the liveness of nodes in repeatedly executed DAGs, and allow them to be allocated locally to runtime generated code if it is believed they are dead, regardless of whether they have a handle.

Having already developed a system for recognising repeatedly executed delayed expression DAGs, we developed a similar mechanism for associating collected liveness information with expression DAGs.

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

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

## 8 Performance Evaluation

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

We compare the performance of our library against the Matrix Template Library [21], Intel's Math Kernel Library and ATLAS [27]. We compare against MTL because it has a similar goal of trying to provide high performance code in C++ with an elegant interface. Comparisons against ATLAS and Intel's MKL are provided as a performance baseline.

ITL already provides support for using MTL as its numerical library. We adapted ITL's existing interface for Sparse BLAS to dense BLAS allowing the ITL solvers to work with ATLAS and Intel's Math Kernel Library. We analysed the performance of five iterative solvers suitable for asymmetric matrices, namely Conjugate Gradient Squared, BiConjugate Gradient, BiConjugate Gradient Stabilised, Quasi-Minimal Residual and Transpose Free Quasi-Minimal Residual. The Chebyshev Iteration and Preconditioned Richardson solvers were not benchmarked due to the need to generate coefficient matrices with appropriate spectral properties. We did not implement the Generalised Conjugate Residual and Generalised Minimum Residual because they required additional matrix-class functionality. Additionally, all benchmarks used the identity preconditioner as all other ITL preconditioners required MTL-specific functionality.

The version of ITL used was 4.0.0-1. The version of MTL used was 2.1.2-23 of MTL2. Version 10.1 of the Intel C and C++ Compilers was used for both the runtime compiled code generated by our library and the compilation of

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

Table 1
The number of compiler invocations in each iterative solver, the total compiler overhead in seconds and total execution time (including compilation) for 256 iterations of each solver with a problem size of 500 and 5000 for architecture 1. Liveness analysis (Section 7) is disabled.

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

In order to show trends more clearly, we show throughput. The number of floating point operations required have been estimated from the ITL implementation of the iterative solvers. It is important to note that our library requires that we invoke a compiler at runtime and hence incur a compilation overhead. We omit this from the throughput graphs as the relative effect of this overhead is dependent on numerous factors including the size of the matrices involved and the number of iterations the solvers are run for. An indication of the compilation overhead for one of our architectures is given in Table 1.

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

(1) Intel Xeon "Clovertown" processor running at 2.66GHz, 4096 KB L2 cache with 4 GB RAM running 64-bit Ubuntu 7.04.
(2) Pentium IV "Prescott" processor running at 3.2GHz with Hyper-threading

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

Table 2
The options supplied to Intel C/C++ compilers and their meanings.

disabled, 2048 KB L2 cache with 2 GB RAM running 32-bit Ubuntu 7.04.

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

We obtained significant speedups from loop fusion between matrix-vector multiply operations on two benchmarks, the BiConjugate Gradient solver and the Quasi-Minimal Residual solver. The first required no modification to ITL, but the latter[1] required minor changes to ITL. We observed that the checks for QMR breakdown forced the evaluation of a matrix-vector multiply before a second matrix-vector multiply, making fusion impossible. Data dependencies permitted moving the second (transpose) matrix-vector multiply to be adjacent to the first, enabling fusion. We note that as this move disregarded the control dependence between the transpose matrix-vector multiply and the QMR breakdown check, our changes also had the minor effect of the multiply being executed unnecessarily in the instance of the breakdown of the solver.

In both these cases the loop fuser was able to fuse a matrix-vector multiply and a transpose matrix-vector multiply with the result that the matrix involved was only iterated over once for both operations. A graph of the speedup obtained across matrix sizes is shown in Figure 3.

[1] the QMR matrix-vector multiply fusion result was not yet achieved in the LCSD'06 paper [20].

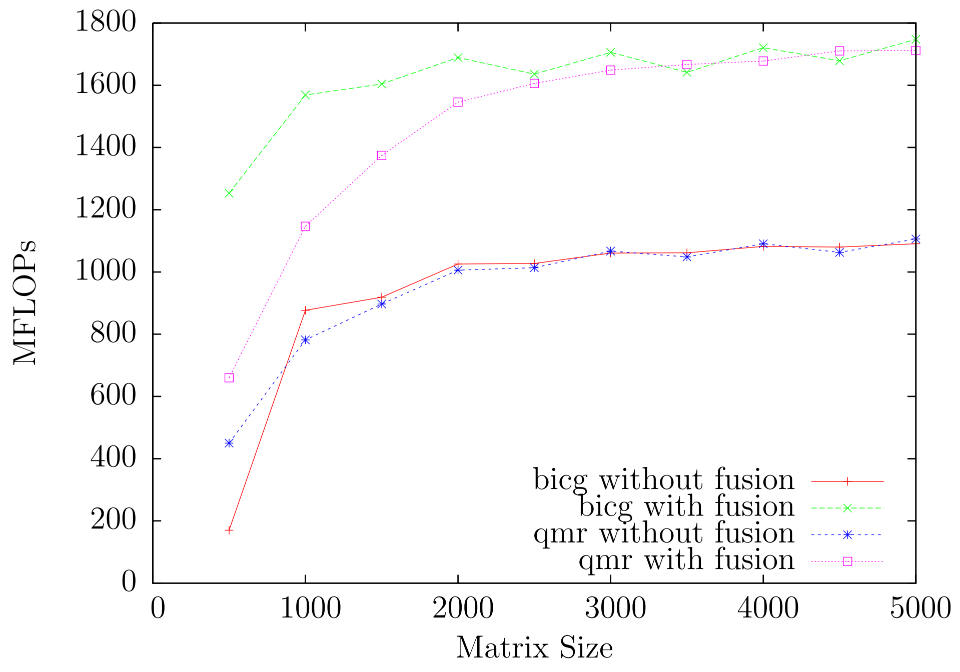

Fig. 3. Throughput of the BiConjugate Gradient (BiCG) and Quasi-Minimal Residual solvers running on architecture 2 with and without loop fusion.

The second optimisation implemented was array contraction. We only evaluated this in the presence of loop fusion as the former is facilitated by the latter. The array contraction pass did not show any noticeable improvement on any of the benchmarks applications. On visual inspection of the runtime generated code we found that the array contractions had occurred in all the iterative solvers. The number of array contractions for each iterative solver are listed in Table 3. However, these only occurred on vectors, and affected only vector-vector operations. This is not surprising since only one matrix is used during the execution of the linear solvers and as it was required for all iterations, could not be optimised away in any way. Future work includes extending our library to sparse matrices, which would make the effect of the loop fusion passes and array contraction more apparent as the cost of the matrix-vector multiply operations becomes $O(n)$ instead of $O(n^2)$.

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

The runtime liveness analysis mechanism was able to find vectors in three of the five iterative solvers that could be allocated locally to the runtime generated code. The three solvers had an average of two vectors that could be optimised away, located in repeatedly executed code. Unfortunately, usage of the liveness analysis mechanism resulted in an overall decrease in performance. We discovered this to be because the liveness mechanism resulted in extra

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

Table 3
Number of array contractions occurring in each iterative solver. *Total array contractions* refers to the number of array contractions performed in code generated during the execution of the solver. *Array contractions in repeatedly executed code* refers to the number of array contractions that occurred in code executed by the solver each iteration. Liveness analysis (Section 7) is disabled.

constant overhead due to more compiler invocations at the start of the iterative solver. This is due to the statistical nature of the liveness prediction, and the fact that as it changed its estimates with regard to whether a value was live or dead, a greater number of runtime generated code fragments had to be compiled.

We also compared DESOLA against the Matrix Template Library, Intel's Math Kernel Library and ATLAS, running the same benchmarks. We enabled the loop fusion and array contraction optimisations, but did not enable the runtime liveness analysis mechanism because of the overhead already discussed.

We observe that on the BiCG and QMR benchmarks, on which we were able to perform matrix-vector loop fusion, we outperform the other implementations on both architectures at matrix sizes above 1500. We show the performance of the different implementations with the BiCG benchmark running on architecture 1 in Figure 4 and QMR on architecture 2 in Figure 5. Performance results for the QMR and BiCG benchmarks are similar on both architectures.

On architecture 1, we note that IMKL and ATLAS appear to perform particularly well with matrix sizes smaller than 1000. On the BiCGSTAB, TFQMR and CGS benchmarks, we can outperform MTL and ATLAS for matrix sizes above 2000, but do not achieve the performance of IMKL. Performance comparisons for the TFQMR benchmark are shown in Figure 6. Results for the BiCGSTAB and CGS benchmarks are similar.

On architecture 2, on the BiCGSTAB, TFQMR and CGS benchmarks, we outperform IMKL and MTL at matrix sizes above 1500. ATLAS consistently outperforms all other implementations at all matrix sizes. Performance comparisons for the TFQMR benchmark are shown in Figure 7. Results for the

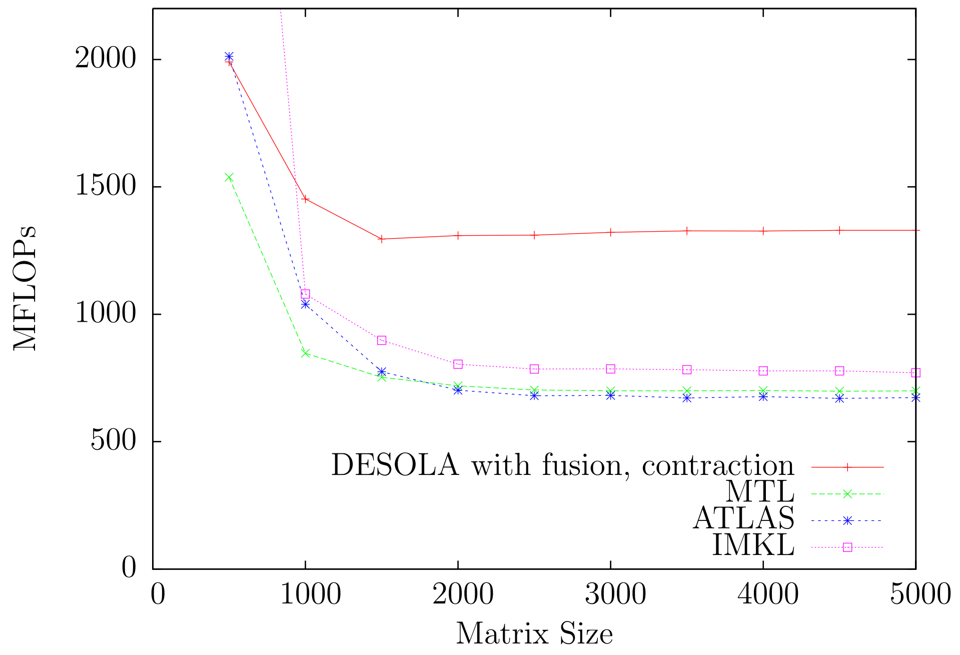

Fig. 4. Throughput of the BiConjugate Gradient (BiCG) solver using DESOLA, MTL, ATLAS and IMKL running on architecture 1. Estimated throughput for IMKL at matrix size 500 is 4504 MFLOPs. Throughput for DESOLA ignores the constant compilation overhead (see Table 1).

BiCGSTAB and CGS benchmarks are similar.

Once again, we stress that these graphs ignore the compilation overhead which is a constant in the case of the iterative solvers (see Table 1). Therefore, the relative effects of this overhead on performance will vary depending on the size of the problem, and the number of iterations required to meet convergence. We also note that mechanisms such as a persistent code cache could allow the compilation overheads to be significantly reduced. These overheads will be discussed in Section 10.

## 9 Discussion

In this section we discuss our work and its relationship to other work in this area and other questions that have been raised.

### *9.1 Is this a job for the compiler?*

Ideally, at least some of the work we do in DESOLA could be performed by a conventional compiler. The techniques we seek to exploit such as loop fusion

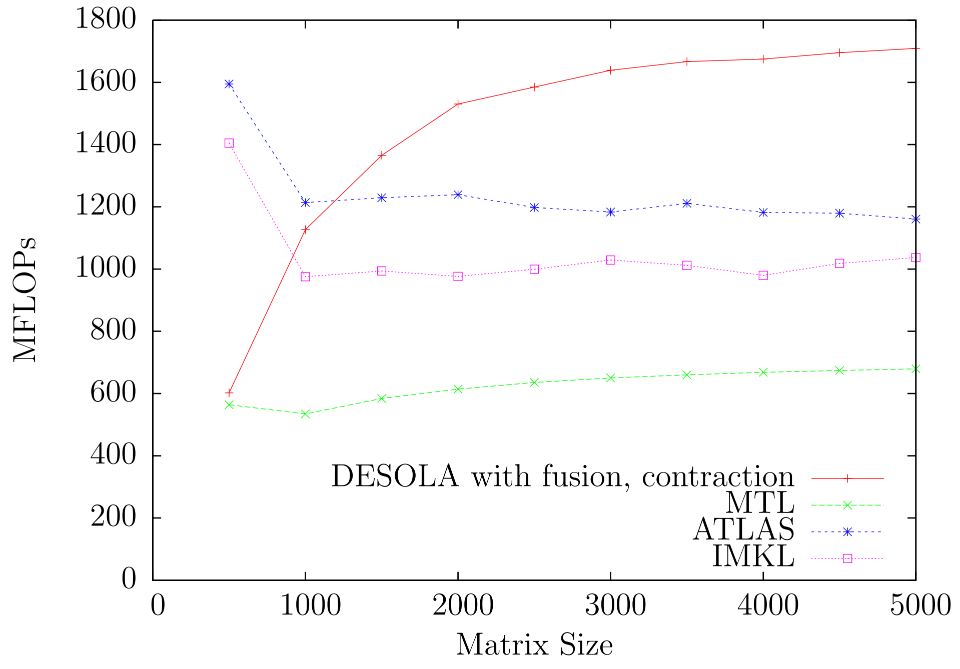

Fig. 5. Throughput of the Quasi-Minimal Residual (QMR) solver using DESOLA, MTL, ATLAS and IMKL running on architecture 2. Throughput for DESOLA ignores the constant compilation overhead (see Table 1).

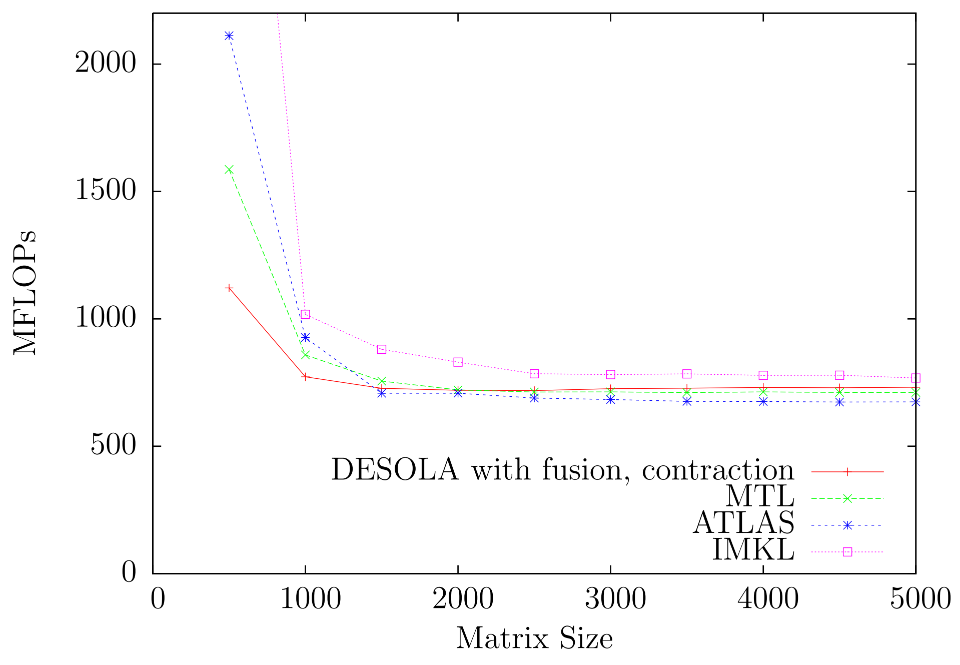

Fig. 6. Throughput of the Transpose Free Quasi-Minimal Residual (TFQMR) solver using DESOLA, MTL, ATLAS and IMKL running on architecture 1. Estimated throughput for IMKL at matrix size 500 is 4233 MFLOPs. Throughput for DESOLA ignores the constant compilation overhead (see Table 1).

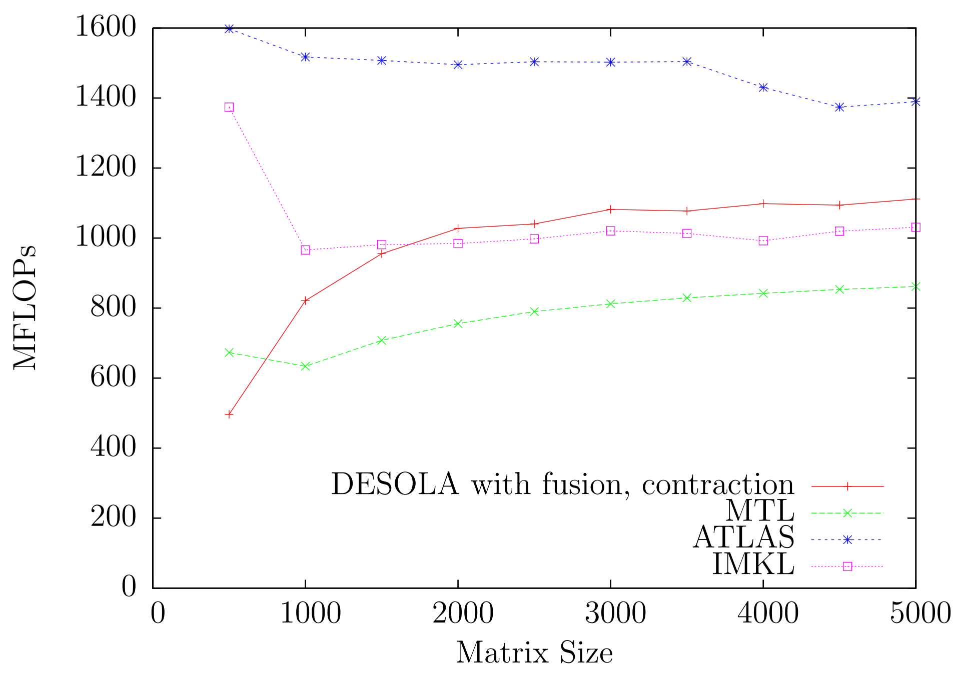

Fig. 7. Throughput of the Transpose Free Quasi-Minimal Residual (TFQMR) solver using DESOLA, MTL, ATLAS and IMKL running on architecture 2. Throughput for DESOLA ignores the constant compilation overhead (see Table 1).

and array contraction are well known and have been researched for a significant period. However, from the output of ICC running on our runtime-generated code, we were able to observe two aspects of ICC's behaviour. Firstly, when compiling code targeting machines with SSE support, ICC would choose not to fuse performance critical loops in our BiCG and QMR benchmarks due to the belief that it would reduce performance. Secondly, when we told ICC to target an architecture without SSE support, it would perform loop fusion in our QMR and BiCG benchmarks, but would fuse vector-matrix and vector-vector operations together rather than the matrix-vector operations required to significantly improve performance.

This indicates a flaw in ICC's fusion heuristics rather than in ICC's ability to fuse loops. As our runtime-generated code takes a restricted form, we can choose to apply optimisations targeted to achieve specific effects. Our library enables us to deliver optimisations that the compiler used for code generation may not yet possess or have the ability to apply effectively.

The ability of our system to optimise across library calls regardless of their location in the client application allows for a greater scope of optimisations to be performed. However, at least some inter-statement optimisation should be possible with MTL. As MTL generates code at compile time inside the client application rather than in a library, it should also be possible for a compiler to exploit inter-statement optimisations between MTL calls. However, until compilers can apply well known techniques like loop fusion and array contraction effectively, we believe our library provides a useful means to bring these optimisations to numerical code irrespective of compiler quality.

### *9.2 What is the difference between our approach and JIT?*

Our approach to generating code has also been compared to Just-in-Time compilation. JIT compilation can make effective use of runtime information to optimise execution performance in the same way we seek to. The quality of the JIT compiler is of course an important issue. JIT compilers typically do not have access to a high level representation of the code making it much more difficult to apply the optimisations we want. Furthermore, we would also need the JIT to optimise across specific sequences of library calls.

It is also possible to argue that a static analysis beforehand could be used to determine information required for the desired optimisations and this information passed to the JIT compiler at runtime. This would provide the benefits of the applied optimisations without the full cost of a runtime analysis. Of course, all these approaches require a C++ JIT compiler whereas one of our aims was to create a C++ library with high portability rather than modify the compiler.

## 10 Conclusions and Further Work

We have presented DESOLA, a prototype library that allows the composition and optimisation of arbitrary sequences of kernels. The experimental results from Section 8 show that for larger matrix sizes we match or exceed the performance of MTL, and when fusion of matrix operations occurs, we exceed the performance of ATLAS and IMKL.

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

Numerical libraries such as BLAS have had to adopt a complex interface to obtain the performance they provide. Libraries such as MTL use unconventional techniques to work around the limitations of conventional libraries to provide both simplicity and performance. The library we developed also uses unconventional techniques, namely delayed evaluation and runtime code generation, to work around these limitations. The effectiveness of this approach provides more compelling evidence towards the benefits of Active Libraries [7].

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

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

**Persistent code caching** This would allow cached code fragments to persist across multiple executions of the same program and avoid compilation overheads on future runs.

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

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

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

Other plans for future work for this research include:

**Sparse Matrices** Linear iterative solvers using sparse matrices have many more applications than those using dense ones, and would allow the benefits of loop fusion and array contraction to be further investigated.

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

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

**Parallelisation** This provides a number of interesting research topics. Loop fusion can inhibit parallelisation when sequential and parallel loops are fused. We need to able to choose when to fuse and when to parallelise taking into account these interactions. In addition, there is also the issue of data alignment when parallelisation is considered in a distributed memory setting. Work by Beckmann and Kelly [4] has already investigated these issues in the context of delayed evaluation.

## 11 Acknowledgements

This work was supported by the EPSRC through a DTA studentship and grant no. EP/E002412-1. We thank the anonymous referees for their very helpful and constructive feedback.

## References

[1] T. J. Ashby, A. D. Kennedy, M. F. P. O'Boyle, Cross component optimisation in a high level category-based language, in: M. Danelutto, M. Vanneschi, D. Laforenza (eds.), Euro-Par, vol. 3149 of Lecture Notes in Computer Science, Springer, 2004.

[2] D. F. Bacon, S. L. Graham, O. J. Sharp, Compiler transformations for high-performance computing, ACM Computing Surveys 26 (4) (1994) 345–420. URL http://citeseer.ist.psu.edu/bacon93compiler.html

[3] O. Beckmann, A. Houghton, M. R. Mellor, P. H. J. Kelly, Runtime code generation in C++ as a foundation for domain-specific optimisation, in: C. Lengauer, D. S. Batory, C. Consel, M. Odersky (eds.), Domain-Specific Program Generation, vol. 3016 of Lecture Notes in Computer Science, Springer, 2003.

[4] O. Beckmann, P. H. J. Kelly, Efficient interprocedural data placement optimisation in a parallel library, in: LCR98: Languages, Compilers and Run-time Systems for Scalable Computers, No. 1511 in LNCS, Springer-Verlag, 1998.

[5] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington,

R. C. Whaley, An updated set of basic linear algebra subprograms (BLAS), ACM Trans. Math. Softw. 28 (2) (2002) 135–151.

[6] J. L. T. Cornwall, P. H. J. Kelly, P. Parsonage, B. Nicoletti, Explicit dependence metadata in an active visual effects library, in: The 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Springer LNCS, 2007.

[7] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, T. L. Veldhuizen, Generative programming and active libraries, in: Selected Papers from the International Seminar on Generic Programming, No. 1766 in LNCS, Springer-Verlag, 2000.

[8] A. Darte, On the complexity of loop fusion, in: PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, Washington, DC, USA, 1999.

[9] J. Dongarra, A. Lumsdaine, R. Pozo, K. Remington, IML++ v. 1.2: Iterative methods library reference guide, Tech. rep., Knoxville, TN 37996, USA (1996). URL http://citeseer.ist.psu.edu/article/dongarra96iml.html

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

[11] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software 14 (1) (1988) 1–17.
URL http://citeseer.ist.psu.edu/dongarra86extended.html

[12] D. R. Engler, W. C. Hsieh, M. F. Kaashoek, C: a language for high-level, efficient, and machine-independent dynamic code generation, in: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 1996.

[13] G. R. Gao, R. Olsen, V. Sarkar, R. Thekkath, Collective loop fusion for array contraction, in: U. Banerjee, D. Gelernter, A. Nicolau, D. A. Padua (eds.), LCPC, vol. 757 of Lecture Notes in Computer Science, Springer, 1992.

[14] K. Kennedy, Fast greedy weighted fusion, in: ICS '00: Proceedings of the 14th international conference on Supercomputing, ACM Press, New York, NY, USA, 2000.

[15] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, Basic Linear Algebra Subprograms for Fortran usage, ACM Trans. Math. Softw. 5 (3) (1979) 308–323.

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

[17] M. Leone, P. Lee, Lightweight Run-Time Code Generation, in: Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Technical Report 94/9, Department of Computer Science, University of Melbourne, 1994.
URL citeseer.ist.psu.edu/leone94lightweight.html

[18] P. Liniker, O. Beckmann, P. H. J. Kelly, Delayed evaluation, self-optimising software components as a programming model, in: B. Monien, R. Feldmann (eds.), Euro-Par, vol. 2400 of Lecture Notes in Computer Science, Springer, 2002.

[19] M. Poletto, D. R. Engler, M. F. Kaashoek, tcc: a system for fast, flexible, and high-level dynamic code generation, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Design and Implementation (PLDI '97), Las Vegas, Nevada, 1997.

[20] F. P. Russell, M. R. Mellor, P. H. J. Kelly, O. Beckmann, An active linear algebra library using delayed evaluation and runtime code generation, in: Proceedings of the Second International Workshop on Library-Centric Software Design (LCSD'06), 2006.

[21] J. G. Siek, A. Lumsdaine, The matrix template library: A generic programming approach to high performance numerical linear algebra, in: D. Caromel, R. R. Oldehoeft, M. Tholburn (eds.), ISCOPE, vol. 1505 of Lecture Notes in Computer Science, Springer, 1998.

[22] J. G. Siek, A. Lumsdaine, A rational approach to portable high performance: The basic linear algebra instruction set (BLAIS) and the fixed algorithm size template (FAST) library, in: S. Demeyer, J. Bosch (eds.), ECOOP Workshops, vol. 1543 of Lecture Notes in Computer Science, Springer, 1998.

[23] T. Veldhuizen, Using C++ template metaprograms, C++ Report 7 (4) (1995) 36–43, reprinted in C++ Gems: Programming Pearls from the C++ Report, ed. Stanley Lippman, Cambridge University Press.

[24] T. L. Veldhuizen, Expression templates, C++ Report 7 (5) (1995) 26–31, reprinted in C++ Gems: Programming Pearls from the C++ Report, ed. Stanley Lippman, Cambridge University Press.

[25] T. L. Veldhuizen, Arrays in Blitz++, in: D. Caromel, R. R. Oldehoeft, M. Tholburn (eds.), ISCOPE, vol. 1505 of Lecture Notes in Computer Science, Springer, 1998.

[26] T. L. Veldhuizen, D. Gannon, 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), SIAM Press, 1998.

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

[28] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, J. L. Hennessy, SUIF: An infrastructure for research on parallelizing and optimizing compilers, SIGPLAN Notices 29 (12) (1994) 31–37.
URL http://citeseer.ist.psu.edu/wilson94suif.html

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