An active linear algebra library using delayed evaluation and runtime code generation

[Extended Abstract]

Francis P Russell, Michael R Mellor, Paul H J Kelly and 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 idea of delaying evaluation of expressions built using library calls, then generating code at runtime for the particular compositions that occur. We explore this idea with a dense linear algebra library for C++. The key optimisations in this context are loop fusion and array contraction.

Our library automatically fuses loops, identifies unnecessary intermediate temporaries, and contracts temporary arrays to scalars. Performance is evaluated using a benchmark suite of linear solvers from ITL (the Iterative Template Library), and is compared with MTL (the Matrix Template Library). Excluding runtime compilation overheads (caching means they occur only on the first iteration), for larger matrix sizes, performance matches or exceeds MTL – and in some cases is more than 60% faster.

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 [5], who observed that active libraries break the abstractions common in conventional compilers.

This paper presents a prototype linear algebra library which we have developed in order to explore one interesting approach to building active libraries. The idea is to use a combination of delayed evaluation and runtime code generation 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 commonly 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 obtain improved performance over a conventional library, it is important that the generated code should on average, 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.

This approach has the advantages that:

- There is no need to analyse the client source code.
- The library user is not tied to a particular compiler.
- The interface of the library is not over complicated by the concerns of achieving high performance.
- We can perform optimisations across both statement and procedural bounds.
- The code generated for a recipe is isolated from client-side code - it is not interwoven with non-library code.

This last point is particularly important, as we shall see: because the structure of the code for a recipe is restricted in form, we can introduce compilation passes specially targeted to achieve particular effects.

The disadvantages of this approach are, the overheads 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 [6] (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 4).
4. We implemented two optimisation passes which transform the code prior to compilation: loop fusion, and array contraction (see Section 5).
5. We introduce a scheme 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 6).
6. We evaluate the effectiveness of the approach using a suite of iterative linear system solvers, taken from the Iterative Template Library (see Section 7).

The contributions we make with this work are as follows:

- Compared to the widely used Matrix Template Library [7], we demonstrate performance improvements of up to 64% across our benchmark suite of dense linear iterative solvers from the Iterative Template Library. Performance depends on platform, but on a 3.2GHz Pentium 4 (with 2MB cache) using the Intel C Compiler, average improvement across the suite was 27%, once cached complied code was available.
- 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.

We discuss related work in Section 8.

2. DELAYING EVALUATION

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

This information may consist of values such as matrix or vector sizes, or the various relationships between successive

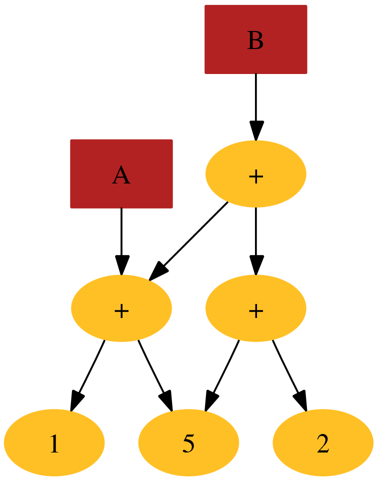


Figure 1: An example DAG. The rectangular nodes denote handles held by the library client.

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 3) 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.

Delayed evaluation in the library we developed works as follows:

- Delayed expressions built using library calls are represented as Directed Acyclic Graphs (DAGs).
- Nodes in the DAG represent either data values (literals) or operations to be performed on them.
- Arcs in the DAG point to the values required before a node can be evaluated.
- Handles held by the library client may also hold references to nodes in the expression DAG.
- Evaluation of the DAG involves replacing non-literal nodes with literals.
- When a node no longer has any nodes or handles depending on it, it deletes itself.

An example DAG is illustrated in Figure 1. The node representing the addition of the values *5* and *2* has no handles referencing it. This makes it in effect, unnamed. When the expression DAG is evaluated it is possible to optimise out this value entirely (its value is not required outside the runtime generated code). For expression DAGs representing matrix or vector operations, this enables us to reduce memory usage and improve cache utilisation.

Delayed evaluation also gives us the ability to optimise across successive library calls. This is called *Cross Component Optimisation* and is something typical linear algebra libraries are incapable of doing.

Work by Ashby[1] has shown the effectiveness of cross component optimisation when applied to Level 1 Basic Linear Algebra Subprograms (BLAS) routines implemented in the language Aldor.

Unfortunately, with each successive level of BLAS, the improved performance available has been accompanied by an increase in complexity. BLAS level 3 functions typically take large a number of operands and perform a large number of more primitive operations simultaneously.

The burden then falls on the the library client programmer to structure their algorithms to make the most effective use of the BLAS interface. Code using this interface becomes more complex both to read and understand, than that using a simpler interface more oriented to the domain.

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

3. 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.

Code generated by the library we developed is specialised in the same way. The constant loop bounds and array sizes make the code more amenable to the optimisations we apply later. These are described in Section 5.

4. 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 be 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:

- It is impossible to detect the situation where the presence of commutative operations allow two differently structured delayed expression DAGs to be used in place of each other.

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

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

Hashing occurs as follows:

- Each DAG node in the sorted list is assigned a value corresponding to its position in the list.
- A hash value is calculated for each node corresponding to its type and the other nodes in the DAG it depends on. References to other nodes are hashed using the numerical values previously assigned to each node.
- 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.

If the maximum number of nodes a node can refer to 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 does not significantly effect 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 two DAGs constructed in an identical order, 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 in the same order.

The code caching mechanism discussed, whilst 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.

5. LOOP FUSION AND ARRAY CONTRACTION

We implemented two optimisations using the TaskGraph back-end, SUIF. A brief description of these transformations follow.

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.

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.

The loop fusion pass implemented in our library requires that the loop bounds be constant. We can afford this limitation because our runtime generated code has already been specialised with loop bound information. Our loop fuser does not posses a model of cache locality to determine which loop fusions are likely to lead to improved performance. Despite this, visual inspection of the code generated during execution of the iterative solvers indicates that the fused loops commonly use the same data. This is most likely due to the structure of the dependencies involved in the operations required for the iterative solvers.

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.

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 be optimised away. For details of other memory access transformations, consult Bacon et al.[2].

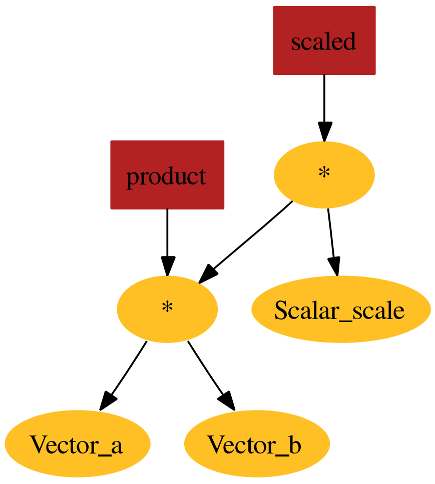
6. LIVENESS ANALYSIS

When analysing the runtime generated code produced by the iterative solvers, it became apparent that a large number of vectors were being passed in as parameters. We realised that by designing a system designed to recover runtime information, we had lost the ability to use static information.

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

This operation can be represented with the following DAG:



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.

7. PERFORMANCE EVALUATION

We evaluated the performance of the library we developed using solvers from the ITL set of templated iterative solvers running on dense matrices of different sizes. We used the Intel C compiler for runtime code generation, and the Intel C++ compiler for compiling the MTL benchmarks. We will discuss the observed effects of the different optimisation

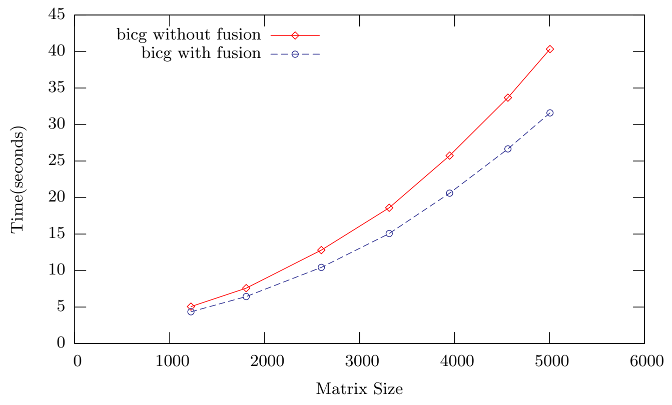


Figure 2: 256 iterations of the BiConjugate Gradient (BiCG) solver running on architecture 1 with and without loop fusion, including compilation overhead.

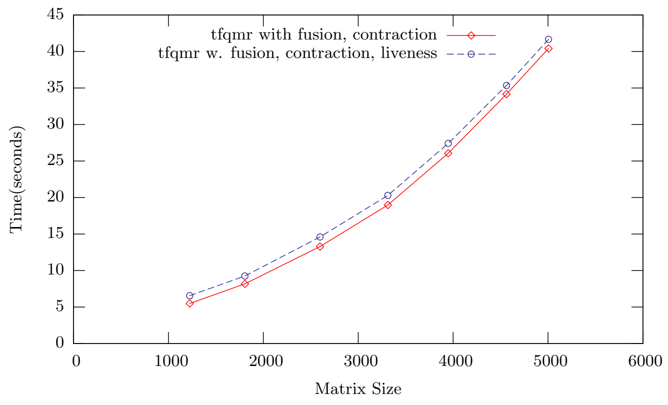


Figure 3: 256 iterations of the Transpose Free Quasi-Minimal Residual (TFQMR) solver running on architecture 1 with and without the liveness analysis enabled, including compilation overhead.

methods we implemented, and the conclude with a comparison against the same benchmarks using MTL.

We evaluated the performance of the solvers on two architectures:

1. Pentium IV processor running at 3.2GHz with Hyperthreading, 2048 KB L2 cache and 1 GB RAM.
2. Pentium IV processor running at 3.0GHz with Hyperthreading, 512 KB L2 cache and 1 GB RAM.

The first optimisation implemented was loop fusion. The majority of benchmarks did not show any noticeable improvement with this optimisation. Visual inspection of the runtime generated code showed multiple loop fusions had occurred between vector-vector operations but not between matrix-vector operations. As we were working with dense matrices, we believe the lack of improvement was due to the fact that the vector-vector operations were $O(n)$ and the matrix-vector multiplies present in each solver were $O(n^2)$.

The exception to this occurred with the BiConjugate Gradient solver. In this case 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 2.

The second optimisation implemented was array contraction. We only evaluated this in the presence of loop fusion as the former is often 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 on vectors, and these only affected the vector-vector operations. This is not surprising seeing that only one matrix was used during the execution of the linear solvers and as it was required for all iterations, could not be optimised away in any way. We believe that were we to extend the library to handle sparse matrices, we would be able to see greater benefits from both the loop fusion and array contraction passes.

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

The runtime liveness analysis mechanism was able to find vectors in three of the five iterative solvers that could be allocated locally to the runtime generated code. The three solvers had an average of two vectors that could be optimised away, located in repeatedly executed code. Unfortunately, usage of the liveness analysis mechanism resulted in an overall decrease in performance. We discovered this to be because the liveness mechanism resulted in extra constant overhead due to more compiler invocations at the start of the iterative solver. This was due to the statistical nature of the liveness prediction, and the fact that as it changed its estimates with regard to whether a value was live or dead, a greater number of runtime generated code fragments had to be produced. Figure 3 shows the constant overhead of the runtime liveness mechanism running on the Transpose Free Quasi-Minimal Residual solver.

We also compared the library we developed against the Matrix Template Library, 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 found the performance increase we obtained to be architecture specific.

On architecture 1 (excluding compilation overhead) we only obtained an average of 2% speedup across the solver and matrix sizes we tested. The best speedup we obtained on this architecture (excluding compilation) was on the BiConjugate Gradient solver, which had a 38% speedup on a 5005x5005 matrix. It should be noted that the BiConjugate Gradient solver was the one for which loop fusion provided a significant benefit.

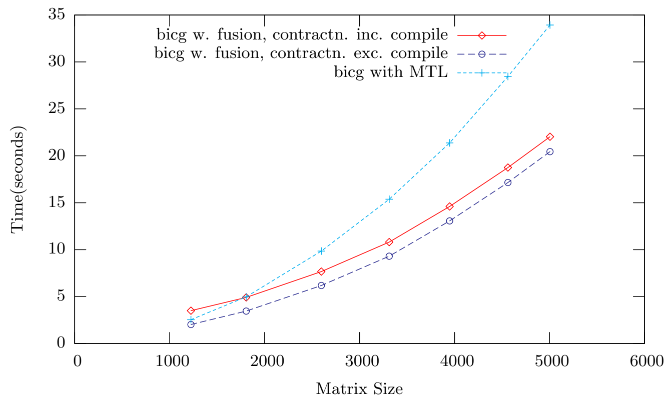


Figure 4: 256 iterations of the BiConjugate Gradient (BiCG) solver using our library and MTL, running on architecture 2. Execution time for our library is shown with and without runtime compilation overhead.

On architecture 2 (excluding compilation overhead) we obtained an average 27% speedup across all iterative solvers and matrix sizes. The best speedup we obtained was again on the BiConjugate Gradient solver, which obtained a 64% speedup on a 5005x5005 matrix. A comparison of the BiConjugate Gradient solver against MTL running on architecture 2 is shown in Figure 4.

In the figures just quoted, we excluded the runtime compilation overhead, leaving just the performance increase in the numerical operations. As the iterative solvers use code caching, the runtime compilation overhead is independent of the number of iterations executed. Depending on the number of iterations executed, the performance results including compilation overhead would vary. Furthermore, mechanisms such as a persistent code cache could allow the compilation overheads to be significantly reduced. These overheads will be discussed in Section 9.

Figure 5 shows the execution time of Transpose Free Quasi-Minimal Residual solver running on architecture 1 with MTL and the library we developed. Figure 6 shows the execution time of the same benchmark running on architecture 2. For our library, we show the execution time including and excluding the runtime compilation overhead.

Our results appear to show that cache size is extremely important with respect to the performance we can obtain from our runtime code generation technique. On out first architecture, we were unable to achieve any significant performance increase over MTL but on architecture 2, which had a 4x larger L2 cache, the increases were much greater. We believe this is due to the Intel C Compiler being better able to utilise the larger cache sizes, although we have not yet managed to determine what characteristics of the runtime generated code allowed it to be optimised more effectively than the same benchmark using MTL. In the full version of this paper, we also intend to analyse the performance of these benchmarks running on the AMD Opteron architecture.

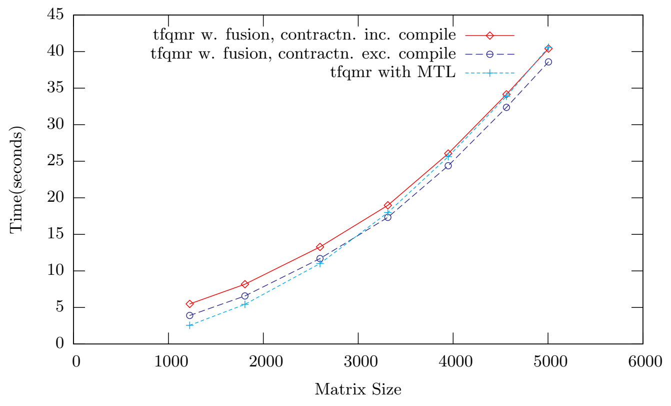


Figure 5: 256 iterations of the Transpose Free Quasi-Minimal Residual (TFQMR) solver using our library and MTL, running on architecture 1. Execution time for our library is shown with and without runtime compilation overhead.

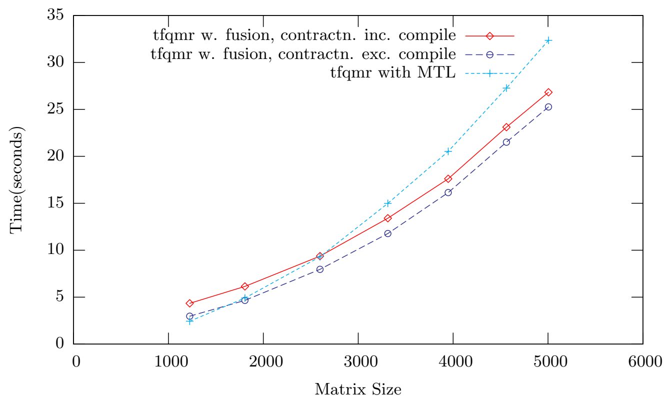


Figure 6: 256 iterations of the Transpose Free Quasi-Minimal Residual (TFQMR) solver using our library and MTL, running on architecture 2. Execution time for our library is shown with and without runtime compilation overhead.

8. RELATED WORK

Work related to our research includes that done by Beckmann on delayed evaluation in a parallel linear algebra library[4] in which delayed evaluation is used to optimise data placement. In our work, we use delayed evaluation to help optimise the code to be executed. Other work includes that done on the TaskGraph library[3] which demonstrates the effectiveness of specialisation and runtime code generation as a means to improving performance of various applications. TaskGraph is also the code generation mechanism for our library.

Work by Ashby[1] has shown the effectiveness of cross component optimisation when applied to Level 1 BLAS routines. It is these form of optimisations that we have tried to exploit to develop a technique for generating high performance code without sacrificing interface simplicity.

9. CONCLUSIONS AND FURTHER WORK

One conclusion that can be made from this work is the importance of cross component optimisation. Numerical libraries such as BLAS have had to adopt a complex interface to obtain the performance they provide. Libraries such as MTL have used 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[5].

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 (we do not believe GCC or ICC will perform array contraction or loop fusion) or the difficulty of performing these optimisations across interprocedural bounds.

Whilst 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. A number of these overheads can be minimised. Two techniques to reduce these overheads are:

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 statically compiled code 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.

Other future work for this research includes:

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 limited methods of loop fusion and array contraction. Other optimisations could improve the code's performance further, and/or reduce the effect the quality of the vendor compiler used to compile the runtime generated code has on the performance of the resulting runtime generated object code.

10. REFERENCES

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

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

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

[4] O. Beckmann and P. H. J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In *LCR98: Languages, Compilers and Run-time Systems for Scalable Computers*, number 1511 in LNCS, pages 123–138. Springer-Verlag, May 1998.

[5] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In *Generic Programming. Proceedings*, number 1766 in LNCS, pages 25–39, 2000.

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

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