Cross Component Optimisation in a High Level Category-Based Language

T.J. Ashby^{1,2}, A.D. Kennedy¹, and M.F.P. O’Boyle²

¹ School of Physics
University of Edinburgh
Scotland

² Institute for Computer Systems Architecture
University of Edinburgh
Scotland

Abstract. High level programming languages offer many benefits in terms of ease of use, encapsulation etc. However, they historically suffer from poor performance. In this paper we investigate improving the performance of a numerical code written in a high-level language by using cross-component optimisation. We compare the results with traditional approaches such as the use of high performance libraries or Fortran. We demonstrate that our cross-component optimisation is highly effective, with a speed-up of up to 1.43 over a program augmented with calls to the ATLAS BLAS library, and 1.5 over a pure Fortran equivalent.

1 Introduction

Superscalar architectures rely on instruction level parallelism (ILP) to get high performance. For codes with large data sets, bandwidth limitations to off-chip memory can mean that the full parallelism of the CPU is not used during streaming operations, such as processing large arrays, because operands/results are not delivered/stored quickly enough. Optimising codes for cache locality ameliorates the situation by reducing the pressure on memory bandwidth, in effect increasing the available parallelism and removing the barrier to performance.

Modularity and encapsulation are fundamental tenets of software engineering, and we wish to adhere to them by writing programs in a high level way using layered composition of components. However, we do not wish to sacrifice performance. It is possible to capitalise on encapsulation and re-use by extensively optimising selected components for performance. Data locality is a program-wide property though, so optimisations must break the barriers introduced by modularity to maximise it. Hence we have two opposing strategies – encapsulate and heavily optimise individual components, or target global locality and optimise less extensively across components on a per program basis.

In this paper we present a case study of cross-component optimisation for a numerical benchmark that makes heavy use of abstraction and other advanced language features such as lexical scoping, dynamic allocation of objects etc, and consists mostly of vector operations. The language used is Aldor [3]. We

show that the resulting program outperforms both a version that makes calls to highly tuned components instead of using our optimisations, and an equivalent program written purely in Fortran and compiled from source using Intel's Fortran compiler. The highly tuned components are the Level 1 binaries of the high-performance ATLAS BLAS library [12].

2 Language

Our approach is based on aggressive compiler optimisations for Aldor, a strict, statically typed (with type inference), mixed functional/imperative language. Aldor is similar to the ML family [9], with advanced module constructs including limited dependent types. The current Aldor compiler generates an intermediate representation (called FOAM), optimises it, and then translates it to C code that is linked against a small runtime library. This means that the language can be used anywhere that a C compiler is available as long as the run-time system is ported. The run-time system provides, amongst other things, initialisation of certain objects and garbage collection.

The core language of Aldor is very small, but offers great scope for abstraction and modularisation by building types. The language supports *domains* which are basic abstract data types, equivalent to modules in ML, and these are typed using *categories*, which are themselves roughly equivalent to (named) ML signatures. A category can inherit from and extend a simpler category, and the benchmark makes heavy use of user defined hierarchical categories to describe the rich mathematical structure of linear algebra (see sec. 3 for an example of category inheritance). This degree of abstraction is coupled with flexible syntax including overloading of both application syntax and the standard binary operators. The end result is very readable code that closely resembles the mathematical formulation.

Aldor incorporates a *whole program optimisation* strategy. This means that all the FOAM code used for a given executable must be available when it is linked, and thus can be analysed to perform extensive cross component optimisation at link-time. Aldor's language model is thus an important enabler for the exploitation of parallelism on single core chips.

3 Benchmark

In order to demonstrate the fundamentals of our approach, we discuss its application to a class of algorithms, being the family of iterative linear solvers (CG, BiCG etc.), with the specific example here being a variant of QMR [5]. The QMR algorithm consists of three nested parts – *basis vector generation*, which is called from the *search vector update*, which in turn is called from the *solution vector update*. The last of these is called repeatedly from a simple enclosing loop that iterates until some condition on the solution vector has been reached.

Each part of the algorithm maintains some state, and when called updates this state using the code extracts given in fig. 1. For more information, we

refer the reader to [10]. The extracts consist of expressions using and updating lexically scoped variables from their environment, with tokens in upper case letters denoting sparse matrices, (numbered) lower case letters denoting vectors, and phonetic Greek letters denoting scalars. The expressions are essentially a list of function calls attached to the overloaded binary operators (including the overloading of juxtaposition), using the functions exported by the domains. So, the components that we are optimising are the basic binary functions provided by the abstract data types. Each of the functions contains, when unoptimised, a fairly simple loop (or scalar manipulation), and allocates a new object to hold its result.

Basis vector generation

```

1. w2 := AH w1 - conj(alpha) * w1
   - conj(beta) * w2;
2. deltaTemp := (v2 * w2);
3. deltaOld := delta;
4. delta := deltaTemp/gamma^2;
5. (v1, v2) := (v2/gamma, v1);
6. (w1, w2) := (w2/conj(gamma), w1);
7. beta := (gamma * delta)/deltaOld;
8. u := A v1;
9. alpha := (u * w1)/delta;
10. v2 := u - alpha * v1 - beta * v2;
11. gamma := norm(v2);

```

Search vector update

```

1. p2 := eta3 * (v2 - eta1 * p1
   - eta2 * p2);
2. (p1, p2) := (p2, p1);

```

Solution vector update

```

1. x := x + zeta * p1;

```

Fig. 1. Sections of Aldor code for the three parts of QMR.

Sparse matrix-vector multiplication for certain sparse matrices can be computed by means of a “stencil” operation. We use these types of matrices in our example, so the use of the matrices in lines 1 and 8 from the basis vector generation extract are actually calls to stencil routines. The stencil we use is a 7 point Laplacian:

$$u_{i,j,k} := \kappa(v_{i+1,j,k} + v_{i-1,j,k} + v_{i,j+1,k} + v_{i,j-1,k} + v_{i,j,k+1} + v_{i,j,k-1} - 6v_{i,j,k})$$

The domain implementing this stencil belongs to the user defined category `LinearOperator` (which is derived from `LinearAlgebra`, `LinearSpace`, `Module`, `AbelianGroup` and `AbelianMonoid`, all of which are also user defined), to which any other matrix representation would also belong. There is some re-use of the vector in the application of a stencil (see [8]) that can be exploited for non-stationary iterative methods, but the vectors have to be very large before tiling has an effect, so we have not yet investigated this optimisation. Instead, we look for locality across all the basic functions including the stencils. The code consists mostly of simple operations on vectors, and so we maximise locality by minimising the number of vector traversals in the program.

4 Transformations

The Aldor compiler has a number of standard optimisations that operate on FOAM code. These include aggressive inlining, which helps to remove the layering of abstraction. The ordinary compiler does not however yet include fusion and the subsequent removal of temporaries. We have developed and present here four different stages of transformation where each stage builds on the previous with more aggressive optimisation. The baseline presented here is not the most naive translation of the code – instead it uses BLAS routines written in Aldor. This allows a more meaningful comparison with the ATLAS augmented and Fortran versions.

- Baseline: Uses BLAS routines written in Aldor. That is, `axpy`, `nrm2` etc. These already give destructive updates of vectors and some limited fusion.
- Level 1: Fully exploits locally available fusion in the vector operations without touching the stencil.
- Level 2: Fuses the action of the stencil with other vector operations within the basis vector generation.
- Level 3: Aggressively rearranges the code in the basis vector generation so that both stencil applications happen simultaneously, and the normalisation of the new v_1 vector is delayed until it is used. This latter optimisation is a direct trade of extra FLOPS for a saving in bandwidth.

Table 1. Breakdown of vector operations in basis vector generation.

Baseline	Level 1	Level 2	Level 3
1. stencil (line 1)	stencil (line 1)	lines 1, 2, 5, 6 (inc. stencil)	lines 1, 2, (5), 6, 8, 9 (inc. both stencils)
2. 1st <code>axpy</code> (line 1)	lines 2,5,6 and 1	lines 8, 9	lines 10, 11
3. 2nd <code>axpy</code> (line 1)	stencil (line 8)	lines 10, 11	
4. inner prod. (line 2)	inner prod. (line 9)		
5. scale (line 5)	lines 10, 11		
6. scale (line 6)			
7. stencil (line 8)			
8. inner prod. (line 9)			
8. 1st <code>axpy</code> (line 10)			
9. 2nd <code>axpy</code> (line 10)			
10. norm (line 10)			

The operations for the basis vector generation are broken down in more detail in table 1. The total number of vector traversals for this part of the algorithm along with count for the other parts of the algorithm is given in table 2. A full breakdown of the fusion in the other parts has been omitted due to space limitations.

Table 2. Total vector traversals count.

	Baseline	Level 1	Level 2	Level 3
basis vector generation	19	13	10	7
Search vector update	5	3	3	3
Solution vector update	2	2	2	2
Total	26	18	15	12

5 Results

5.1 Control Experiments

By way of comparison with our transformations we present results for two different approaches. The first substitutes the BLAS routines written in Aldor with calls to the ATLAS library. This amounts to extreme localised tuning of the basic vector operations in the algorithm to compare against cross-component optimisation. The second codes the entire problem in Fortran, based on a modification of the freely available QMRpack [4].

QMRpack was originally structured to use call-backs to invoke the user provided matrix-vector multiplication routines. This enables modularity for use with different operators, but introduces an artificial barrier to optimisation. To provide a fairer comparison with our approach, the stencil routine is now called directly from the main routine. The source files for the main routine and all the library subroutines (including the BLAS routines) are concatenated together before compilation. QMRpack does not try and separate the Lanczos process from the QR decomposition. Hence, the code just consists of a series of vector and scalar operations and does not attempt to mimic the extra structure in the Aldor code or any of the functional programming features. Indeed, it is written in Fortran 77.

5.2 Test Environment

The test environment comprised a 1 GHz Pentium III desktop machine (Coppermine with 256kB L2 cache) running Linux RedHat 7.1, the Intel C compiler `icc` and Intel Fortran compiler `ifc`, both version 8.0, the ATLAS BLAS binaries for Pentium III release 3.4.2, and the Aldor compiler version 1.0. All timing information was generated using `gprof` version 2.11 and the compiler profile generation switches provided by `icc` and `ifc`.

All compiler flags were chosen to try and maximise the performance of the code. The experiments are conducted using complex double floats (4 words = 16 bytes each) as the scalars, based on 3 dimensional regular grids with equal side lengths. Hence the size of an individual vector is $side^3 \times 16$ bytes. The problem sizes run from 15, which is just after the total data set no longer fits into cache, up until 40.

Fig. 2. Plot of problem size vs. speed-up over baseline.

5.3 Discussion

Fig. 2 shows a ratio plot comparing the total execution times of various programs against that of the baseline. For the ATLAS augmented version, the executable spends from $\approx 45\%$ of its time in the ATLAS routines for the smallest dataset size, rising steadily to $\approx 67\%$ for the largest. As we can see, the Fortran code does well on the first data point. This seems reasonable, as the whole problem fits into the L2 cache until it reaches size 13, and superior overall scheduling (and prefetching) would make itself felt near the bottom end of the scale. Nonetheless, when the problem size reaches 20 the transformations of Level 3 outperform all the other approaches, and as the size of the dataset increases the relative gains improve.

The speed-up is up to 1.43 over the ATLAS version and up to 1.5 over Fortran, demonstrating significant gains for cross-component optimisation.

Latency, Prefetching and Code Generation. Due to the limits on out-of-order execution and the C compiler strategy for issuing prefetches, we may not be realising the full benefit of our optimisations. Both `ifc` and `icc` issue prefetches, but `ifc` issues more for the Fortran code, particularly in the vector expressions. Presumably the Fortran version is consequently closer to being simply bandwidth limited. As far as we can tell, `ifc` does no restructuring of the loops though. We wish to investigate improving our current approach by adding iterative optimisation to find transformations that reduce the suspected latency penalties.

The performance of the Fortran code is substantially less than the transformed Aldor at larger problem sizes, showing that the gains in terms of bandwidth outweigh the introduced latency costs. Although it is not possible to examine the ATLAS binaries, we assume that they are more or less perfectly scheduled and prefetched, and the same conclusion holds.

6 Related Work

Reducing the impact of modularisation on performance has been tackled many times in many different settings. To our knowledge though, the combination of language type, approach and target of optimisation presented here are unique. The most popular target for this sort of work is `C++`, with examples including expression templates [11], library annotations, and extensive analysis and pre-processing of source code. Expression templates are pragmatic (lots of people use `C++`) but ad hoc; the analysis of the code that can be done and hence the optimisations is limited, the semantic analysis is inadequate, and coupling the transformation system to an engine that searches for good transformations would be hard. Library annotations for domain specific languages work at a higher level [7]; we wish to invasively fuse operations on objects, not rewrite expressions at the ADT level. The approach taken in [1] is probably closest to our own; however, that project appears to be coming from the bottom up rather than the top down, in that they already support various optimisations and are edging toward effectively defining their own specialised language by adding parsing and semantic analysis of a language piggybacked onto `C++`. However, we do not require a user to specify the transformations as re-write rules. [2] is an automatic system for generating parsing and transformation tools similar to those used in [1]. Something like it might be used to generate a system for performing the transformations that we use, but we are focused on the transformations rather than developing a general tool. Also, although our transformations are quite specific, they are based purely on information that can be recovered from FOAM rather than requiring extra semantic input in terms of grammars and rules provided by a library programmer.

The work in [13] appears to be at a lower level than ours, and addresses a problem that is already dealt with by the existing Aldor optimisations. Future work on optimising Generic Java is likely to be of greater interest.

7 Conclusion

The results show that optimising whole program locality for ILP is important for this class of application. Hence, using an elegant language with lot of abstraction doesn’t necessarily imply poor performance if the language infrastructure supports cross-component optimisation well. Indeed, focusing on cross-component optimisations can bring important speedups compared to traditional approaches, even if the result is almost certainly crudely scheduled and prefetched compared to individual extensively tuned assembly routines.

The most important immediate extension of this work is to fully investigate the interaction of the transformations with the functional features of Aldor. Following this we wish to investigate the impact of our optimisations on a wider set of codes with more complex operators, such as the red-black preconditioned QCD Wilson–Dirac operator, and more specialised solvers [6], as well as other benchmarks from the field of scientific computing.

References

1. Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Dave Binkley and Paolo Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press. (To appear).
2. Kei Davis and Dan Quinlan. ROSE II: An optimizing code transformer for C++ object-oriented array class libraries. In *Proceedings of the Third World Multiconference on Systemics, Cybernetics and Informatics (SCI'99) and the Fifth International Conference on Information Systems Analysis and Synthesis (ISAS'99)*.
3. Stephen Watt et al. Aldor. <http://www.aldor.org>.
4. R. W. Freund and N. M. Nachtigal. QMRpack. <http://www.netlib.org/linalg/qmr/>.
5. R. W. Freund and N. M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. In R. Beauwens and P. de Groen, editors, *Iterative Methods in Linear Algebra*, pages 151–154. Elsevier Science Publishers, 1992.
6. A. Frommer and B. Medeke. Exploiting structure in Krylov subspace methods for the Wilson fermion matrix. In A. Sydow, editor, *15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 3, pages 485–490. Wissenschaft und Technik Verlag, 1997.
7. Samuel Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. In *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation*.
8. Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32. IEEE Computer Society, 2000.
9. Robert Harper Robin Milner, Mads Tofte and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
10. T.J.Ashby, A.D.Kennedy, and M.F.P.O'Boyle. An iterative solver package in Aldor. Submitted to: Proceedings of the Third International Workshop on Numerical Analysis and Lattice QCD, November 2003.
11. Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
12. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
13. Peng Wu, Sam Midkiff, José Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 109–118. ACM Press, 1999.