Nested data parallelism in Haskell

Amos Robinson, 3400438

June 30, 2013

Contents

1 Introduction

As transistors continue to decrease in size, as they have been for the last fifty years, more transistors are able to fit into a single processor. Historically this has led to faster processors, allowing programs to take advantage of the increased speed with no effort from the programmer. Recently, however, this increase in processor speed has come to a halt. Being unable to increase processor speed, manufacturers have instead started focussing the new transistors on multi-core processors.

The ubiquity of multi-core processors means that sequential programs can no longer take full advantage of the processor. However, writing parallel programs is significantly more difficult than writing sequential programs. Nested data parallelism lifts the burden of exploiting multi-cores from the programmer to the compiler-writer. Current implementations of nested data parallelism use impressive optimisation techniques such as fusion, but are still unable to compete with handwritten parallel code. The aim of my research is to improve the performance of nested data parallelism, making it easier for programmers to write efficient parallel code. Because nested data parallelism has focussed on purely functional languages, many well-known imperative optimisations have not been treated.

Purely functional languages, such as Haskell, are more restricted than impure imperative languages in the sense that arbitrary expressions may not perform side-effects such as destructive updates. This restriction, however, turns out to have serious benefits: an optimising compiler is able to reorder expressions to gain better performance, without any risk of changing the meaning of the program. From an engineering perspective also, the separation of side-effects from pure computation leads to clearer, easier to understand and verify code.

This document aims to review the current state of automatic parallelism techniques, nested data parallelism in particular, and the optimisations required to get them running at acceptable speeds. My overall dream is to have parallel programs that are easier to write, and execute faster than their C equivalents. So far only the first of these dreams has been realised, with Data Parallel Haskell (DPH) being able to express data parallel programs clearly and succinctly. These programs, while running significantly faster than list-based Haskell programs, cannot generally compete with hand-optimised C code.

2 Parallelism

Writing parallel programs that operate on a large amount of data can be difficult. To achieve the best performance on a multi-processor machine, programs must be written using low-level synchronisation techniques such as threads and locking. These methods are error prone, and improper use can lead to bugs, such as deadlocks and *heisenbugs*. In contrast to bugs in a sequential program, these bugs are often much harder to reproduce because of the delicate timing and communication between threads. So-called heisenbugs are a particularly hard class of bug, as they disappear and the program appears to work after debugging information is added.

There are many techniques for making parallelism easier to take advantage of, each with their own advantages and disadvantages. In general, the more restrictive a technique is, the easier it is to achieve adequate performance as there is more knowledge about the program's structure. However, these restrictions come at the expense of making it harder to express some programs. We discuss three techniques:

Flat data parallelism is able to achieve speeds comparable to C with advanced fusion techniques, but only supports regular structures, such as rectangular and cubic arrays.

Nested data parallelism is similar to flat data parallelism, but supports parallel operations over tree-like structures while keeping the load balanced over processors.

Feedback directed automatic parallelism is able to automatically parallelise any program, using profiling information from running the program to find the best places to introduce parallelism.

2.1 Flat data parallelism

Flat data parallelism is a restricted form of data parallelism that disallows nested parallelism: only the top-most computation is performed in parallel. To ensure a balanced load across parallel computations, only regular structures such as rectangular arrays can be used. Programs are expressed using combinators such as `map`, `filter`, `sum` and `fold`.

Accelerate is an *embedded domain-specific language* in Haskell, specifically for running parallel computations on graphics processors. As a restricted domain-specific language, it targets graphics processors [Chakravarty et al., 2011] and uses fusion to remove intermediate arrays. With these, and other optimisations, it is sometimes able to achieve speeds similar to hand-optimised CUDA code [McDonell et al., 2013].

Repa is a Haskell library with combinator support for regular parallel array computations [Keller et al., 2010]. Type indices are used to distinguish between so-called *delayed*

arrays and *manifest arrays*. Delayed arrays will not be reified in memory at runtime, but are instead expressed as functions of the array index. With sufficient inlining and other general purpose optimisations, delayed arrays will be removed and fused together [Lippmeier et al., 2012b]. Efficient stencil convolution is supported as a special case. To remove branching from the worker loops, the edges of a stencil are computed in different loops [Lippmeier and Keller, 2011].

Many algorithms are more naturally expressed using nested parallelism, such as the hierarchical N-body calculation [Barnes and Hut, 1986] algorithm and sparse-matrix vector multiplication. For these algorithms, flat data parallelism is insufficient.

2.2 Nested data parallelism

Nested data parallelism, as introduced in a seminal paper [Blelloch, 1990], uses a transform known as *flattening* to convert nested data parallelism into flat data parallelism while maintaining a balanced load across processors.

A language specifically for nested data parallelism, NESL [Blelloch, 1995], uses the flattening transform to remove nested parallelism. It targets a virtual machine called VCODE [Blelloch et al., 1993] which is then compiled to C to run on vector machines such as CRAY. Many example programs have been implemented in NESL [Blelloch, 1996] but it is not a general purpose language and does not support higher-order functions [Leshchinskiy, 2005] or recursive structures [Keller and Chakravarty, 1998].

Attempts have been made to introduce nested data parallelism to imperative languages such as Fortran [Au et al., 1997]. These have the benefit of taking advantage of existing compiler optimisations. The problem with targeting an impure language, however, is that arbitrary expressions may perform side-effects. This severely limits the places where parallelism may be introduced, as performing unknown side-effects in parallel will likely change the meaning of the program.

2.2.1 Data Parallel Haskell

Data Parallel Haskell (DPH) is an implementation of nested data parallelism for Haskell [Chakravarty et al., 2007]. The original flattening transform is extended to support higher-order functions [Leshchinskiy et al., 2006], a common pattern in Haskell, and arbitrary Haskell types: sum, product and recursive data types [Chakravarty and Keller, 2000].

Instead of targeting vector machines as NESL does, DPH targets commodity symmetric multi-processor (SMP) machines. Fusion is performed to remove intermediate arrays, but care must be taken not to reduce parallelism [Chakravarty and Keller, 1999].

This is solved by partitioning the program into local operations, requiring only some small chunk of the array, and global operations, requiring thread synchronisation [Keller and Chakravarty, 1999]. As long as only local operations are fused, crossing no global boundaries, no parallelism will be lost.

Unlike NESL, DPH does not require the entire program to be written using nested data parallelism, and supports partial vectorisation [Chakravarty et al., 2008]. This allows other parts of the program to be expressed naturally as sequential computations, such as drawing to the screen or reading data from disk, while the core computation can be run in parallel.

The original flattening transform also increases the time and space complexity of the program, as flattening causes data to be replicated multiple times, and if fusion does not occur correctly, such intermediate structures will be reified in memory. The time complexity problem can be fixed by using a clever representation, making replication constant time [Lippmeier et al., 2012a]. This does not solve the memory problem, however.

As with all optimisations, conversion from nested data parallelism to flat data parallelism should be semantics preserving: the original program's meaning must remain unchanged [Leshchinskiy, 2005]. This is complicated by the fact that Haskell is a lazy language, whereas, for efficiency, strict unboxed arrays are used to represent data. Several tricks are used to preserve laziness while still using strict unboxed arrays.

Flattening converts scalar operations into array operations, even for intermediate scalars [Keller et al., 2012]. This means that operations that once were simple additions on registers must now work on arrays in memory. Array fusion can often remove these intermediate structures, but fusion relies heavily on other optimisations such as inlining, which may not always occur due to sharing. Excessive inlining can also cause code blowup in the compilation process, leading to long compile times. Instead of vectorising an entire program and then relying on fusion to remove intermediate arrays, identifying scalar operations and not vectorising them has proven to be less fragile.

2.3 Feedback directed automatic parallelism

Automatic parallelism, as in Mercury [Bone, 2010], is able to parallelise more general programs than nested data parallelism, but at the cost of compiler simplicity.

Feedback directed parallelism relies on the program being run multiple times, and each time the program is recompiled. The compiler uses information from the previous runs to determine the best place to introduce parallelism.

Mercury, a logic language, has many similarities to Haskell: it is pure, statically typed, and has type inference. The purity allows the compiler to know where it is safe to introduce parallelism.

Automatic parallelism is a far more challenging problem than nested data parallelism, since the programs do not use any known set of combinators or structure. For this reason, and because many important scientific programs can be implemented with nested data parallelism [Blelloch, 1996], automatic parallelism will not be discussed further.

3 Optimisations

While the flattening transform for nested data parallelism is quite simple, it alone cannot achieve performance that competes with that of hand-optimised C code. This section discusses the optimisations required to get such performance. Most of these optimisations are general purpose, and applicable to not only nested data parallel programs.

Optimisations for pure languages can make use of the knowledge that arbitrary expressions are side-effect free. Two classes of optimisations are particularly important for nested data parallel programs in pure languages: fusion and constructor specialisation. Impure languages must be more conservative in the transformations they apply, in order to preserve the meaning of the program.

3.1 Fusion

Fusion merges array producers with consumers, removing the need to allocate intermediate arrays in memory.

3.1.1 Short-cut fusion

Short-cut fusion includes techniques such as stream fusion [Coutts et al., 2007], `fold/build` fusion [Gill et al., 1993] and functional array fusion [Chakravarty and Keller, 2001, Chakravarty and Keller, 2003]. These techniques all work by fusing adjacent producers and consumers together. This relies on general purpose optimisations such as inlining to bring producers and consumers closer.

A recent paper shows the possibility of fusion producing SIMD-vector instructions [Bik, 2004] that operate on multiple elements at a time [Mainland et al., 2013]. This allows significant speedups.

There is a balance between inlining and code blowup, however. Too much inlining and the code will become too large, leading to long compilation times and poor cache performance when a loop kernel does not fit in the instruction line. As short-cut fusion is a simple local transformation, it can be implemented using general rewrite rule facilities [Peyton Jones et al., 2001] without modifying the compiler. The local nature of short-cut fusion comes at a price, however, when a single producer has multiple consumers. As the producer cannot be inlined into both consumers without potentially duplicating work, the producer must be reified in memory.

3.1.2 Flow fusion

A new method that we have been working on is known as flow fusion [Lippmeier et al., 2013]. It uses a more global analysis and can deal with branching dataflows that short-cut fusion cannot. It analyses an entire function of combinator-based 'loops' at a time and schedules the combinators into as few loops as possible. This analysis is based on Waters' series expressions [Waters, 1991], using Shivers' loop anatomy [Shivers, 2005] to merge skeleton code for loops. This method is similar to compilation for dataflow languages [Johnston et al., 2004].

This requires more complicated implementation than short-cut fusion, in the form of a compiler plugin. As it is still very new, no research has been done into SIMD instructions.

3.1.3 Loop fusion and array contraction

Fusion in imperative languages is generally expressed as two stages: loop nests with the same index space are *fused* together, then intermediate arrays are *contracted* into scalars or smaller buffers. This requires a *loop dependency graph* [Gao et al., 1993] to be created, to calculate which loops can be fused without changing the order of writes and reads to arrays. Loops may have their index spaces reversed or otherwise modified, in order to allow further fusion. Sarkar shows that the optimal loop configuration can be found with a two-colouring of a graph [Sarkar and Gao, 1991].

Such optimisations may generally increase efficiency, but can in the worst case cause extra cache misses or register spilling. Loop fusion merges the bodies of two or more loops, which means there are often more local variables and memory references in the result, all contending for registers and cache. Some cache misses can be identified by looking at each pair of consecutive requests to the same element, and counting the number of distinct elements requested between the pair. This is called the *reuse distance* [Song et al., 2004]. With controlled fusion, loops are only fused together if the maximum reuse distance does not exceed the cache size.

In cases where a successive loop iteration uses the output of the previous loop iteration, memory reads can be reduced by reusing the register value [Wang et al., 2013].

Unimodular matrices are used to find new execution orders of loops, where the original index order cannot be parallelised [Banerjee, 1993]. For example, a 2-d loop whose (x, y)th iteration depends on the output of its $(x-1, y-1)$th iteration cannot be trivially parallelised. If the execution order is changed, however, the several threads could work diagonally through the array at the same time.

These imperative optimisations require all loops to only include trivially non-side-effecting statements. If loops had arbitrary function calls, the compiler cannot generally

know whether these function calls had side-effects, and would not be certain that re-ordering the calls would be safe. This problem does not occur in a pure language, as all functions are assured to be side-effect free.

3.2 Alias analysis

Another problem that keeps high-performance Haskell from competing with C code is incomplete alias analysis. After converting variables references to memory reads and writes, superfluous memory traffic can be reduced by storing scalar variables in registers. This requires that no other variables reference the same memory location, or *alias* with each other, as converting aliasing variables to registers can change the meaning of the program. Alias analysis is a low-level optimisation that finds sets of variables known *not* to alias, and removes superfluous memory reads and writes to them.

This can give great improvements in the runtime of loop kernels, as the majority of time would otherwise be spent updating local variables [Clifton-Everest, 2012]. However, functions generally do not know whether their arguments will alias, so local alias analysis is in some cases doomed to be pessimistic.

Another method is to introduce distinctness witnesses [Ma, 2012] as proof that two variables will not alias. These witnesses are introduced and inferred automatically by the compiler, and passed as arguments to functions. The functions can then be optimised with full assurance that arguments will not alias, leading to faster code.

The Glasgow Haskell Compiler currently only performs rudimentary alias analysis, which leads to problems with tight loops where the superfluous memory operations outweigh the loop time.

3.3 Constructor specialisation

Constructor specialisation removes superfluous allocations from loops [Bechet, 1994], where the allocation is used and thrown away immediately at the start of the next iteration.

In a purely functional language such as Haskell, loops are often expressed as recursive functions, with mutable loop variables becoming function arguments. Without mutation, however, any change to the loop variables must be allocated as new objects [Peyton Jones, 2007]. This is particularly wasteful when objects are allocated only to be used once in the next iteration, then thrown away. A solution to this is to specialise recursive functions to remove unnecessary allocation and pattern-matching of constructors. When a recursive function pattern matches or destructs one of its arguments and makes a recursive call with a constructor of that argument, the recursive call can be specialised for that particular

argument so no allocation or unboxing is necessary.

This has been implemented for both pure and impure languages such as Haskell and ML [Thiemann, 1993, Mogensen, 1993].

This kind of compile-time evaluation is also key to modern techniques such as supercompilation [Bolingbroke and Peyton Jones, 2011]. Constructor specialisation is also a generalisation of another technique, worker-wrapper transform [Gill and Hutton, 2009], that creates a specialised worker function that operates on unboxed data structures.

Excessive specialisation is also a risk, as specialisation increases the code size, potentially leading to longer compilation times and worse cache performance when a loop's code doesn't fit into the cache. I have been able to reduce this excessive specialisation in the Glasgow Haskell Compiler by first attempting to *seed* the specialisation with the call patterns the function is first called with. This reduces the number of specialisations to those that will actually be used, instead of generating all possible specialisations.

Other forms of constructor specialisation / supercompilation have been developed that are able to rewrite multiple recursive calls into a single recursion [Burstall and Darlington, 1977]. This has impressive results, but it requires user intervention and relies on domain knowledge such as commutativity of addition.

4 Conclusion

Nested data parallelism is a relatively mature field, but so far it is unable to beat the performance of hand-written C code. Recent advancements such as flow fusion [Lippmeier et al., 2013] and SIMD support for stream fusion [Mainland et al., 2013] continue to converge. It is unclear, however, whether fusion alone will be able to fix the fundamental memory space problems with flattening [Lippmeier et al., 2012a].

Improvements to nested data parallelism such as vectorisation avoidance [Keller et al., 2012] and work efficient replication [Lippmeier et al., 2012a] show promise. Further improvements could be made, however, such as avoiding the replication issue altogether when the same nested workload is shared across all threads.

Our new fusion system, flow fusion [Lippmeier et al., 2013], solves problems with existing fusion such as branching dataflow, but can only handle specific cases such as *online computations* where unbounded buffers are not necessary. Such cases are common, but by no means exclusive. For example, consider that reversing an array requires reifying the entire array in memory before being able to retrieve the first element. Further work is required to gain the best possible fusion in this case, instead of reverting to stream fusion completely.

Other problems with fusion are that excessive fusion can cause register and cache contention [Song et al., 2004]. While it is true that in these cases, fusion is still better than nothing, the ideal solution may involve some kind of loop tiling [Pike and Hilfinger, 2002], where each array is processed in buffers that fit in cache.

Incomplete alias analysis is a low-level issue that only becomes apparent once the rest of the program is perfect. In other programs, runtime is more often dominated by lack of fusion and memory usage issues. While runtime would definitely benefit from better alias analysis, my time will be better spent focussing on higher-level issues, such as better vectorisation avoidance and fusion.

References

[Au et al., 1997] Au, K., Chakravarty, M., Darlington, J., Guo, Y., Jahnichen, S., Kohler, M., Keller, G., Pfannenstiel, W., and Simons, M. (1997). Enlarging the scope of vector-based computations: extending Fortran 90 by nested data parallelism. In *Advances in Parallel and Distributed Computing, 1997. Proceedings*, pages 66–73. IEEE.

[Banerjee, 1993] Banerjee, U. (1993). *Loop transformations for restructuring compilers: the foundations*, volume 1. Springer.

[Barnes and Hut, 1986] Barnes, J. and Hut, P. (1986). A hierarchical $O(n \log n)$ force-calculation algorithm. *Nature*, 324:4.

[Bechet, 1994] Bechet, D. (1994). Limix: a partial evaluator for partially static structures. Technical report, Citeseer.

[Bik, 2004] Bik, A. J. (2004). *The software vectorization handbook*. Intel Press Hillsboro, OR.

[Blelloch, 1990] Blelloch, G. (1990). *Vector models for data-parallel computing*, volume 75. MIT press Cambridge, MA.

[Blelloch, 1995] Blelloch, G. (1995). NESL: A nested data-parallel language. Technical report, DTIC Document.

[Blelloch, 1996] Blelloch, G. (1996). Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97.

[Blelloch et al., 1993] Blelloch, G., Hardwick, J., Chatterjee, S., Sipelstein, J., and Zagha, M. (1993). *Implementation of a portable nested data-parallel language*, volume 28. ACM.

[Bolingbroke and Peyton Jones, 2011] Bolingbroke, M. and Peyton Jones, S. (2011). Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation.

[Bone, 2010] Bone, P. (2010). *Automatic Parallelisation for Mercury*. PhD thesis, University of Melbourne.

[Burstall and Darlington, 1977] Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67.

[Chakravarty and Keller, 1999] Chakravarty, M. and Keller, G. (1999). How portable is nested data parallelism?

[Chakravarty and Keller, 2000] Chakravarty, M. and Keller, G. (2000). More types for nested data parallel programming. In *ACM SIGPLAN Notices*, volume 35, pages 94–105. ACM.

[Chakravarty and Keller, 2001] Chakravarty, M. and Keller, G. (2001). Functional array fusion. In *ACM SIGPLAN Notices*, volume 36, pages 205–216. ACM.

[Chakravarty and Keller, 2003] Chakravarty, M. and Keller, G. (2003). An approach to fast arrays in Haskell. *Advanced Functional Programming*, pages 1948–1948.

[Chakravarty et al., 2008] Chakravarty, M., Leshchinskiy, R., Jones, S., and Keller, G. (2008). Partial vectorisation of Haskell programs. In *Proc ACM Workshop on Declarative Aspects of Multicore Programming, San Francisco*.

[Chakravarty et al., 2007] Chakravarty, M., Leshchinskiy, R., Jones, S., Keller, G., and Marlow, S. (2007). Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM.

[Chakravarty et al., 2011] Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., and Grover, V. (2011). Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM.

[Clifton-Everest, 2012] Clifton-Everest, R. (2012). Optimisations for the LLVM back-end of the Glasgow Haskell Compiler. Master's thesis, University of New South Wales.

[Coutts et al., 2007] Coutts, D., Leshchinskiy, R., and Stewart, D. (2007). Stream fusion: From lists to streams to nothing at all. In *Proceedings of the International Conference of Functional Programming 2007*.

[Gao et al., 1993] Gao, G., Olsen, R., Sarkar, V., and Thekkath, R. (1993). Collective loop fusion for array contraction. *Languages and Compilers for Parallel Computing*, pages 281–295.

[Gill and Hutton, 2009] Gill, A. and Hutton, G. (2009). The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251.

[Gill et al., 1993] Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993). A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA. ACM.

[Johnston et al., 2004] Johnston, W. M., Hanna, J., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34.

[Keller and Chakravarty, 1998] Keller, G. and Chakravarty, M. (1998). Flattening trees. In *Euro-Par98 Parallel Processing*, pages 709–719. Springer.

[Keller and Chakravarty, 1999] Keller, G. and Chakravarty, M. (1999). On the distributed implementation of aggregate data structures by program transformation. *Parallel and Distributed Processing*, pages 108–122.

[Keller et al., 2012] Keller, G., Chakravarty, M., Leshchinskiy, R., Lippmeier, B., and Peyton Jones, S. (2012). Vectorisation avoidance. In *Proceedings of the 2012 Haskell symposium*, pages 37–48. ACM.

[Keller et al., 2010] Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., and Lippmeier, B. (2010). Regular, shape-polymorphic, parallel arrays in haskell. *ACM Sigplan Notices*, 45(9):261–272.

[Leshchinskiy, 2005] Leshchinskiy, R. (2005). *Higher-order nested data parallelism: semantics and implementation*. PhD thesis, Universitätsbibliothek.

[Leshchinskiy et al., 2006] Leshchinskiy, R., Chakravarty, M., and Keller, G. (2006). Higher order flattening. *Computational Science–ICCS 2006*, pages 920–928.

[Lippmeier et al., 2012a] Lippmeier, B., Chakravarty, M., Keller, G., Leshchinskiy, R., and Peyton Jones, S. (2012a). Work efficient higher-order vectorisation. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 259–270. ACM.

[Lippmeier et al., 2012b] Lippmeier, B., Chakravarty, M., Keller, G., and Peyton Jones, S. (2012b). Guiding parallel array fusion with indexed types. In *Proceedings of the 2012 Haskell symposium*, pages 25–36. ACM.

[Lippmeier et al., 2013] Lippmeier, B., Chakravarty, M. M. T., Keller, G., and Robinson, A. (2013). Data flow fusion with series expressions in Haskell. In *Proceedings of the 2013 Haskell symposium*. In Submission.

[Lippmeier and Keller, 2011] Lippmeier, B. and Keller, G. (2011). Efficient parallel stencil convolution in haskell. In *ACM SIGPLAN Notices*, volume 46, pages 59–70. ACM.

[Ma, 2012] Ma, T. (2012). Type-based aliasing control for the disciplined disciple compiler. Master's thesis, University of New South Wales.

[Mainland et al., 2013] Mainland, G., Leshchinskiy, R., Jones, S. P., and Marlow, S. (2013). Haskell beats C using generalized stream fusion.

[McDonell et al., 2013] McDonell, T. L., Chakravarty, M. M., Keller, G., and Lippmeier, B. (2013). Optimising purely functional GPU programs. In *International Conference of Functional Programming*. ICFP.

[Mogensen, 1993] Mogensen, T. (1993). Constructor specialization. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 22–32. ACM.

[Peyton Jones, 2007] Peyton Jones, S. (2007). Call-pattern specialisation for Haskell programs. In *ACM SIGPLAN Notices*, volume 42, pages 327–337. ACM.

[Peyton Jones et al., 2001] Peyton Jones, S. L., Tolmach, A., and Hoare, T. (2001). Playing by the rules: rewriting as a practical optimisation technique in GHC.

[Pike and Hilfinger, 2002] Pike, G. and Hilfinger, P. N. (2002). Better tiling and array contraction for compiling scientific programs. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Computer Society Press.

[Sarkar and Gao, 1991] Sarkar, V. and Gao, G. R. (1991). Optimization of array accesses by collective loop transformations. In *Proceedings of the 5th international conference on Supercomputing*, pages 194–205. ACM.

[Shivers, 2005] Shivers, O. (2005). The anatomy of a loop: a story of scope and control. In *ACM SIGPLAN Notices*, volume 40, pages 2–14. ACM.

[Song et al., 2004] Song, Y., Xu, R., Wang, C., and Li, Z. (2004). Improving data locality by array contraction. *Computers, IEEE Transactions on*, 53(9):1073–1084.

[Thiemann, 1993] Thiemann, P. (1993). Avoiding repeated tests in pattern matching. *Static Analysis*, pages 141–152.

[Wang et al., 2013] Wang, Y., Jia, Z., Chen, R., Wang, M., Liu, D., and Shao, Z. (2013). Loop scheduling with memory access reduction subject to register constraints for DSP applications. *Software: Practice and Experience*.

[Waters, 1991] Waters, R. C. (1991). Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):52–98.