Loop optimizations in C and C++ compilers: an overview

Réka Kovács, Zoltán Porkoláb

Eötvös Loránd University
rekanikolett@caesar.elte.hu
gsd@inf.elte.hu

Abstract

The evolution of computer hardware in the past decades has truly been remarkable. From scalar instruction execution through superscalar and vector to parallel, processors are able to reach astonishing speeds – if programmed accordingly. Now, writing programs that take all the hardware details into consideration for the sake of efficiency is extremely difficult and error-prone. Therefore we increasingly rely on compilers to do the heavy-lifting for us.

A significant part of optimizations done by compilers are loop optimizations. Loops are inherently expensive parts of a program in terms of run time, and it is important that they exploit superscalar and vector instructions. In this paper, we give an overview of the scientific literature on loop optimization technology, and summarize the status of current implementations in the most widely used C and C++ compilers in the industry.

Keywords: loops, optimization, compilers, C, C++

MSC: 68N20 Compilers and interpreters

1. Introduction

The Illiac IV, completed in 1966, was the first massively parallel supercomputer [13]. It marked the first milestone in a decades-long period that would see computing machines become unbelievably fast and increasingly complex. To harness the capabilities of such ever more parallel systems, researchers started writing tools

that could transform sequential programs (typically written in Fortran for scientific applications) to their own parallel equivalents.

By the mid 70's, a group of University of Illinois researchers led by David Kuck developed the Parafrase [18] tool, which pioneered the most influential ideas on automatic program transformations including dependence testing [19] and the first loop transformations [32].

In the late 70's, researchers led by Ken Kennedy at Rice University started the Parallel Fortran Compiler (PFC) [2] project. The authors' initial goal was to extend Parafrase, but they ended up implementing a completely new system, furthering the theory of data dependence [4] and inventing effective algorithms for a number of transformations including vector code generation, the handling of conditional statements [3], loop interchange, and other new transformations [5].

In this paper, we take a step aside, and instead of discussing the optimization of Fortran programs, for which most of the classical algorithms have been invented, we take a look at how C and C++ compiler writers are coping with the challenge. C family languages are very similar to Fortran, but notorious for the lack of constraints imposed on the programmer, which makes their analysis and optimization undoubtedly more difficult.

The structure of this paper is as follows. Section 2 describes the challenges C and C++ compiler developers face in contrast to classic Fortran optimization, and lists some of the strategies used to mitigate these issues. In Section 3, we give a status report of loop optimizations in the two open-source compilers most heavily used in the industry. Finally, in Section 4, we survey the latest research papers in the field of loop optimizations.

2. Adapting classic algorithms for C/C++

2.1. Challenges in optimizing C/C++

In their 1988 paper, [6] Allen and Johnson pointed out a number of considerations that make the vectorization and parallelization of C code difficult, as opposed to Fortran, the language that inspired most of the classic loop transformations in the literature. These concerns pose a challenge in optimizing programs written in C and C++ to the present day:

- **Pointers instead of subscripts.** C/C++ programs often address memory using pointer variables rather than arrays and explicit subscripts. Because of this, it is extremely difficult to decide whether two statements refer to the same section of memory or not.

- **Sequential operators.** Conditional operators and operators with side effects (e.g. ++) are inherently sequential. Vectorization of such operations require them to be either transformed or removed.

- **Loops.** The *for* statement used in C family languages is less restricted than the *DO* loop of Fortran, for which most of the classic loop transformations

were developed. This makes its vectorization considerably more difficult. The loop can contain operations that almost arbitrarily change the loop variable, and the loop body can contain branching statements.

- **Small functions.** Function calls can hide information that is necessary for optimization. Modern compilers often run optimizations together with inlining in an iterative fashion, trying to regain some information lost to the fine modularity encouraged in C and C++.

- **Argument aliasing.** Unline Fortran, function parameters in C and C++ are allowed to point into the same section of memory. Aliasing prohibits vectorization, but can only be checked at run-time, resulting in a high run-time cost.

- **Volatile.** In C, *volatile* variables represent values that may change outside of the context of the program, even if the change cannot be "seen" from the source code. Obviously, such language constructs are very difficult to optimize.

- **Address-of operator.** The & operator allows the programmer to take the address of any variable and modify it. This greatly increases the analysis needed for optimization.

2.2. Compiler strategies for C/C++ optimization

Inlining. The problems listed in the previous section are relatively hard to handle in compilers. Suprisingly, a large portion of the problems can be removed by the judicious inlining of function calls. Some of the benefits of inlining:

- If the body of a function call is available in the caller function, the compiler no longer has to calculate its effects conservatively, assuming the worst case. It can use the actual function body, making the analysis more precise, and allowing more optimizations to happen.

- Some of the argument aliasing problems disappear when the origins of arrays become visible.

- Function calls are inherently sequential operations. Their removal helps vectorization in its own right.

A high-level IR. Low-level intermediate representations had long been the norm for C [16] and Fortran [29] compilers before their vectorizing versions began gaining ground. Lowering the code too early can introduce unnecessary complexity in the analyses that precede optimizations. For example, the ability to analyze loops and subscripts is crucial for loop optimizations, and breaking down the loop into `goto`s and pointers would make it considerably more difficult. Other information such as the `volatile` modifier also get lost or obfuscated after the lowering phase.

Loop conversion.   The C `for` loop is a fairly unconstrained language construct: the increment and termination conditions can have side effects, bounds and strides can change during execution, and control can enter and leave the loop body. Because of this, most C compilers perform a *doloop conversion*, when they attempt to transform the unconstrained `for` loop into a more regular `DO`-like form. It often makes sense to do the high-level transformations on this representation as well.

3. Loop optimization in modern compilers

The most widely used C and C++ compilers include both open-source (GCC, LLVM) and commercial (Microsoft Visual C++, IBM XL, Intel C Compiler) products. Unfortunately, there is limited information available for closed-source application, thus in this section we decided to review the status of loop transformations in GCC and LLVM. Both of these compilers are heavily used in the industry, and have a populous base of active contributors.

3.1. LLVM

The LLVM optimization pipeline [24] consists of 3 stages. The *Canonicalization* phase removes and simplifies the IR as much as possible by running scalar optimizations. The second part is called the *Inliner cycle*, as it runs a set of scalar simplification passes, a set of simple loop optimization passes, and the inliner itself iteratively. The primary goal of this part is to simplify the IR, through a cycle of exposing and exploiting simplification opportunities. After the Inliner cycle, the *Target Specialization* phase is run, which deliberately increases code complexity in order to take advantage of target-specific features as make the code as fast as possible on the target.
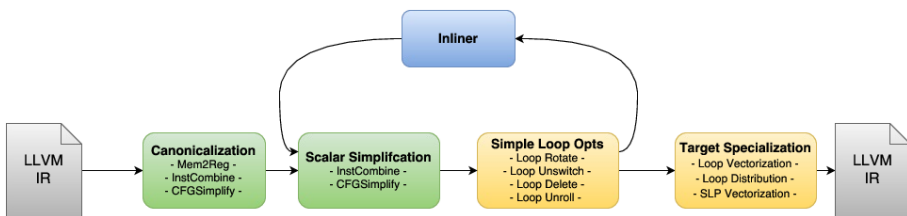

Figure 1: The LLVM optimization pipeline

   LLVM supports various *pragmas* that allow users to guide the optimization process, which saves them the trouble of performing the optimizations by hand. Alternatively, they can rely on the *heuristics* in the compiler that strive to achieve similar performance gains.
   The optimization infrastructure is modular, passes can be switched on and off on demand [17]. The currently available loop transformation passes are the following: loop unrolling, loop unswitching, loop interchange, detection of `memcpy` and `memset`

idioms, deletion side-effect-free loops, loop distribution, and loop vectorization. Members of the open-source community are working on adding loop fusion to the list [7].

As part of the modular structure of the optimizer, a common infrastructure is available to optimizations in the form of certain passes that perform analyses (`LoopInfo`, `ScalarEvolution`) and normalizing transformations (`LoopRotate`, `LoopSimplify`, `IndVarSimplify`) on the loops.

Many of the mentioned loop transformations are disabled by default, as they are either experimental in nature or not mature enough to be used by the wide public. Such transformations are e.g. `LoopInterchange` and `LoopUnrollAndJam`.

The order of the loop optimizations is fixed within the pipeline. This may result in conflicts or less profitable sequences of transformations. Additionally, because scalar and loop passes are run in cycles, they often interfere with each other by destroying canonical structures and invalidating analysis results.

A recent proposal plans to switch to a single integrated `LoopOptimizationPass` that would not interact with scalar optimizations, making it simpler. Similarly, the introduction of a loop tree intermediate representation could make loop modifications easier and might also help the profitability analysis. This idea is inspired by red-green trees [23] used in the Roslyn C# compiler.

3.2. GCC

As the early days of GCC date back to the 80's, its old monolithic structure made it hard to keep it aligned with the forefront of optimization research for a long time. However, its loop optimizer was almost completely re-written in the early 2000's [11]. Its new modular structure is similar to that of LLVM's, starting with an *initialization* pass, followed by several *optimization* passes, and ended with a trivial *finalization* phase that de-allocates any data structures used.

During initialization, the optimizer runs the induction variable, scalar evolution, and data dependence analyses [8] to gather necessary information about the loop, and performs preliminary transformations that simplify and canonicalize it. The optimization passes include loop unswitching, loop distribution, and two types of auto-vectorization [22]. The middle block of `pass_graphite` transformations refers to the polyhedral framework GRAPHITE that comes with GCC, but is unfortunately turned off by default, due to a lack of resources for maintenance.

In spite of its long history, GCC was still not very good at optimizing loop nests in 2017 [9]. This was mainly caused by

- a lack of traditional loop nest transformations, including loop interchange, unroll-and-jam, loop fusion and scalar expansion,

- transformations in need of a revision, and possibly
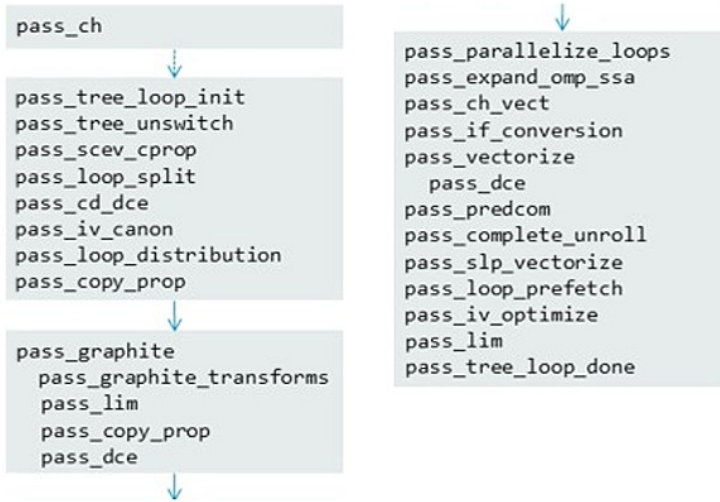
- a suboptimal arrangement of passes.

Figure 2: The GCC optimization pipeline

Since 2017, members of the community have started adding some of the missing transformations to the compiler, e.g. loop interchange [14], but others e.g. loop fusion and scalar expansion are still future work.

Similarly to LLVM, the latest proposal to the pass arrangement problem is a single loop transformation pass with a unified cost model.

4. Current trends in optimization research

Transformation ordering.   One of the main research directions in the past few years concerns the choice of loop transformations and their ordering. With ever more complex machines, the performance gap between hand-tuned and compiler-generated code is getting wider. [31] presents a system and language named Locus that uses empirical search to automatically generate valid transformation sequences and then return the list of steps to the best variant. The source code needs to be annotated. [33] gives a template of scheduling algorithms with configurable constraints and objectives for the optimization process. The template considers multiple levels of parallelism and memory hierarchies and models both temporal and spatial effects. [10] describes a similar loop transformation scheduling approach using dataflow graphs. [30] recognizes that some combinations of loop optimizations can create memory access patterns that interfere with hardware prefetching. They give an algorithm to decide whether a loop nest should be optimized mainly for temporal or mainly for spatial locality, taking hardware prefetching into account.

Straight-line code vectorization. The past few years saw significant new developments in the field of straight-line code vectorization. The original Superword-Level Parallelism algorithm (SLP) [20] was designed for contiguous memory access patterns that can be packed greedily into vector instructions, without expensive data reordering movements. Throttled SLP [26] attempts to identify statements harmful to vectorization and stop the process earlier if that leads to better results. SuperGraph SLP [25] operates on larger code regions and is able to vectorize previously unreachable code. Look-ahead SLP [28] extends SLP to commutative operations, and is implemented in both LLVM and GCC. The latest development, SuperNode SLP [27] enables straight-line vectorization for expressions involving a commutative operator and its inverse.

Improving individual transformations. Other research efforts target the improvement of individual optimizations. [21] gives an algorithm to locate where to perform code duplication in order to enable optimizations that are limited by control flow merges. [1] describes a software prefetching algorithm for indirect memory accesses. [12] shows how to discover scalar reduction patterns and how it was implemented in an LLVM pass. [15] created a framework to enable collaboration between different kinds of dependency analyses.

5. Conclusion

In the age when hardware evolution makes machines ever more complex, compiler optimizations become ever more important, even for the simplest applications. This paper gave a short history of parallel hardware and compiler optimizations, followed by a discussion of hardships that the C and C++ languages pose to compiler writers. We gave a status report on the loop optimizing capabilities of the most popular open-source compilers for these languages, GCC and LLVM. In the end, we reviewed the latest research directions in the field of loop optimization research.

References

[1] S. Ainsworth, T. M. Jones: *Software prefetching for indirect memory accesses*, in: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2017, pp. 305–317,
DOI: https://doi.org/10.1145/3319393.

[2] J. R. Allen, K. Kennedy: *PFC: A program to convert Fortran to parallel form*, tech. rep., 1982.

[3] J. R. Allen, K. Kennedy, C. Porterfield, J. Warren: *Conversion of control dependence to data dependence*, in: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1983, pp. 177–189,
DOI: https://doi.org/10.1145/567067.567085.

[4] J. R. Allen: *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*, AAI8314916, PhD thesis, USA, 1983,
DOI: https://doi.org/10.5555/910630.

[5] R. Allen: *K. Kennedy*, Automatic translation of FORTRAN programs to vector form. A CM Transactzons on Programming Languages and Systems 9.2 (1987), pp. 491–542,
DOI: https://doi.org/10.1145/29873.29875.

[6] R. Allen, S. Johnson: *Compiling C for vectorization, parallelization, and inline expansion*, ACM SIGPLAN Notices 23.7 (1988), pp. 241–249.

[7] K. Barton: *Loop Fusion, Loop Distribution and their Place in the Loop Optimization Pipeline*, LLVM Developers' Meeting, 2019,
URL: https://www.youtube.com/watch?v=-JQr9aNagQo.

[8] D. Berlin, D. Edelsohn, S. Pop: *High-level loop optimizations for GCC*, in: Proceedings of the 2004 GCC Developers Summit, Citeseer, 2004, pp. 37–54.

[9] B. Cheng: *Revisit the loop optimization infrastructure in GCC*, GNU Tools Cauldron, 2017,
URL: https://slideslive.com/38902330/revisit-the-loop-optimization-infrastructure-in-gcc.

[10] E. C. Davis, M. M. Strout, C. Olschanowsky: *Transforming loop chains via macro dataflow graphs*, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 265–277,
DOI: https://doi.org/10.1145/3168832.

[11] Z. Dvorák: *A New Loop Optimizer for GCC*, in: GCC Developers Summit, Citeseer, 2003, p. 43.

[12] P. Ginsbach, M. F. O'Boyle: *Discovery and exploitation of general reductions: a constraint based approach*, in: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2017, pp. 269–280.

[13] R. M. Hord: *The Illiac IV: the first supercomputer*, Springer Science & Business Media, 2013.

[14] *Introduce loop interchange pass and enable it at -O3.* https://gcc.gnu.org/ml/gcc-patches/2017-12/msg00360.html, Accessed: 2020-05-24.

[15] N. P. Johnson, J. Fix, S. R. Beard, et al.: *A collaborative dependence analysis framework*, in: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2017, pp. 148–159.

[16] S. C. Johnson: *A portable compiler: theory and practice*, in: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1978, pp. 97–104,
DOI: https://doi.org/10.1145/512760.512771.

[17] M. Kruse: *Loop Optimizations in LLVM: the Good, the Bad, and the Ugly*, LLVM Developers' Meeting, 2018,
URL: https://www.youtube.com/watch?v=QpvZt9w-Jik.

[18] D. J. Kuck: *Automatic program restructuring for high-speed computation*, in: International Conference on Parallel Processing, Springer, 1981, pp. 66–84,
DOI: https://doi.org/10.1007/BFb0105110.

[19] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe: *Dependence graphs and compiler optimizations*, in: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1981, pp. 207–218,
DOI: https://doi.org/10.1145/567532.567555.

[20] S. LARSEN, S. AMARASINGHE: *Exploiting superword level parallelism with multimedia instruction sets*, Acm Sigplan Notices 35.5 (2000), pp. 145–156,
DOI: https://doi.org/10.1145/349299.349320.

[21] D. LEOPOLDSEDER, L. STADLER, T. WÜRTHINGER, ET AL.: *Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations*, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 126–137,
DOI: https://doi.org/10.1145/3168811.

[22] D. NAISHLOS: *Autovectorization in GCC*, in: Proceedings of the 2004 GCC Developers Summit, 2004, pp. 105–118.

[23] *Persistence, Facades and Roslyn's Red-Green Trees*, https://docs.microsoft.com/en-gb/archive/blogs/ericlippert/persistence-facades-and-roslyns-red-green-trees, Accessed: 2020-05-24.

[24] *Polly: The Architecture.* https://polly.llvm.org/docs/Architecture.html, Accessed: 2020-05-24.

[25] V. PORPODAS: *Supergraph-slp auto-vectorization*, in: 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2017, pp. 330–342,
DOI: https://doi.org/10.1109/PACT.2017.21.

[26] V. PORPODAS, T. M. JONES: *Throttling automatic vectorization: When less is more*, in: 2015 International Conference on Parallel Architecture and Compilation (PACT), IEEE, 2015, pp. 432–444,
DOI: https://doi.org/10.1109/PACT.2015.32.

[27] V. PORPODAS, R. C. ROCHA, E. BREVNOV, L. F. GÓES, T. MATTSON: *Super-Node SLP: optimized vectorization for code sequences containing operators and their inverse elements*, in: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2019, pp. 206–216,
DOI: https://doi.org/10.1109/CGO.2019.8661192.

[28] V. PORPODAS, R. C. ROCHA, L. F. GÓES: *Look-ahead SLP: Auto-vectorization in the Presence of Commutative Operations*, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 163–174,
DOI: https://doi.org/10.1145/3168807.

[29] R. G. SCARBOROUGH, H. G. KOLSKY: *A vectorizing Fortran compiler*, IBM Journal of Research and Development 30.2 (1986), pp. 163–171,
DOI: https://doi.org/10.1109/10.1147/rd.302.0163.

[30] S. SIOUTAS, S. STUIJK, H. CORPORAAL, T. BASTEN, L. SOMERS: *Loop transformations leveraging hardware prefetching*, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 254–264,
DOI: https://doi.org/10.1145/3168823.

[31] S. T. TEIXEIRA, C. ANCOURT, D. PADUA, W. GROPP: *Locus: a system and a language for program optimization*, in: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2019, pp. 217–228,
DOI: http://doi.acm.org/10.1145/2737924.2738003.

[32] M. J. WOLFE: *High performance compilers for parallel computing*, Addison-Wesley Longman Publishing Co., Inc., 1995.

[33] O. ZINENKO, S. VERDOOLAEGE, C. REDDY, ET AL.: *Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling*, in: Proceedings of the 27th International Conference on Compiler Construction, 2018, pp. 3–13,
DOI: https://doi.org/10.1145/3178372.3179507.