
Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

# Ein Arbeitsablauf zur Analyse der Effekte von Compiler-Optimierungen auf HPC-Anwendungen

## A Workflow for Analyzing the Effect of Compiler Optimizations on HPC Applications

**Masterarbeit**

Jonas Hahnfeld
Matrikelnummer: 381478

Aachen, den 14.09.2020

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Matthias S. Müller (') |
| Zweitgutachter: | Prof. Dr. Christian Bischof (*) |
| Betreuer: | Julian Miller, M.Sc. (')<br>Dr. Christian Terboven (')<br>Dr. Christian Iwainsky (*) |

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Fachgebiet Scientific Computing, TU Darmstadt

communicated by Prof. Dr. Matthias S. Müller

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 14.09.2020

# Kurzfassung

Moderne HPC-Anwendungen werden hauptsächlich in Sprachen höherer Stufe geschrieben. Dies hilft der Programmierproduktivität, indem der Compiler zur Abstraktion genutzt wird. Es erlaubt außerdem, viele Systeme und Architekturen mit einer einzigen Code-Basis zu bedienen. Für portable Performance verlassen Entwickler sich auf den Compiler, um ihren Code zu optimieren. Allerdings können die Ergebnisse sich sehr zwischen verschiedenen Compilern und ihren Versionen unterscheiden.

Diese Arbeit beschreibt einen strukturierten Arbeitsablauf, um diese Unterschiede zu lokalisieren. Dafür werden Profiling-Daten genutzt, um zwei Ausführungen des gleichen Codes zu vergleichen. Danach beschreibt die Arbeit Methoden für detaillierte Analyseschritte. Insbesondere werden Muster bei Kerneln eingeführt, um die Unterschiede mit Compiler-Optimierungen in Verbindung zu bringen. Das erlaubt es, ihre Effekte zu analysieren und damit die Ursache für den Unterschied zu verstehen.

Abschließend präsentiert diese Arbeit Fallstudien und Analyseergebnisse des Arbeitsablaufs. Damit wird die Anwendbarkeit für die gewählten Benchmarks aus einer breiten Spanne der Simulationsgebiete demonstriert. Für einige Fallstudien ist es möglich, Änderungen der Compiler-Parameter oder am Quellcode abzuleiten. Das führt zu Verbesserungen der Performance von bis zu 14 %.

**Stichwörter:** HPC, Compiler-Optimierungen, Performance, Kernel-Eigenschaften

# Abstract

Modern HPC applications are mainly written in higher-level languages. This helps programming productivity by using the compiler as a means of abstraction. It also allows to target many systems and architectures with a single code base. For portable performance, developers rely on the compiler to optimize their code. However, the results can vary greatly between different compilers and their versions.

This thesis describes a structured workflow to first locate such differences. For that, it uses profiling data to compare two executions of the same code. Afterwards, the thesis delineates methods for detailed analysis steps. In particular, it introduces patterns of kernels to relate the differences to compiler optimizations. This allows to analyze their effects and thereby the cause of the difference.

Finally, the thesis presents case studies and analysis results of the workflow. This demonstrates the applicability for the chosen benchmarks from a wide range of simulation domains. For some case studies, it is possible to derive changes of compiler parameters or the source codes. This leads to performance improvements of up to 14 %.

**Keywords:** HPC, Compiler Optimizations, Performance, Kernel Properties

# Acknowledgements

Working on my own topic and ideas was a great source of motivation over the past months. I am grateful to Prof. Dr. Matthias S. Müller and Prof. Dr. Christian Bischof for supervising this thesis. Without doubt, the workflow evolved over time from the rough sketch it was initially. I would like to thank Julian Miller, Dr. Christian Terboven, and Dr. Christian Iwainsky for their constructive feedback. The open discussions made me think in very diverse ways and contributed to what this thesis is today.

Finally many thanks to my parents for their support and final proof-reading!

# Contents

# List of Definitions

# List of Figures

# List of Listings

# List of Tables

# 1. Introduction

When developing simulations, scientists are oftentimes faced with the following requirements: The application should deliver results quickly because computing resources are limited and expensive. At the same time, the simulation should also be able to run on multiple HPC systems. The latter includes different hardware architectures and configurations as well as varying software environments.

To reach these goals, one approach is to write portable code and rely on the compiler for optimizations. Doing so allows to target many architectures at once and adapt to changes more easily. Compilers are also beneficial for programming productivity because the scientists can work with higher-level languages. The compiler then acts as a level of abstraction for the translation to machine code. In that context, compiler optimizations are most important for the sequential performance of the generated code. This also builds the foundation for the computational parts of parallel applications.

However, reality shows that different compilers have different optimization strategies. This can have various reasons, which are usually invisible to the application developer: Firstly, compilers implement transformations for the same optimization goal in different ways. This leads to small deviations when dealing with real applications. Additionally, many optimizations are guarded with heuristics or controlled by thresholds. These are usually subject to the compiler's tuning for a particular set of applications. Still, they can have a huge impact on the optimization of all other codes.

For the developer, these differences primarily manifest as varying performance. However, it can be hard to relate the observed behavior with the causes mentioned above. This is because these implementation details are usually hidden from the user. Nevertheless, understanding the causes has many advantages: On the one hand, it allows to find approaches to make full use of the compiler. In some cases, this could result in more portable code that achieves similar or better performance. On the other hand, it can help to improve the compiler's optimization pipeline for other codes as well.

There are only few results on this topic known in literature. To the best of my knowledge, there currently exists no general approach to analyze the effect of compiler optimizations. In particular, this means only domain experts are able to understand the compiler's influence on performance. To that end, my thesis makes the following contributions to enable this analysis by application developers:

1. I propose a workflow to locate differences between the observable runtime behavior of two executables. Afterwards, I describe steps to analyze the found differences and understand their causes.

2. To ease the analysis, I present patterns in the source code of typical HPC

applications. For these, I show possible optimizations, which allows to relate differences to the machine code generated by the compiler.

3. Finally, I evaluate the two approaches with case studies of real benchmark applications. I describe possible improvements for the analyzed code or the compilation options to demonstrate the applicability.

## 1.1. Environment for Measurements

Later chapters will include runtime measurements to present practical considerations. For this, I execute on the node `login18-t` provided by RWTH Aachen University as part of the HPC environment. The system is dedicated to tuning and has the advantage to allow interactive access in contrast to waiting for batch jobs. This is especially important for the workflow described in Chapter 3, which requires iterative changes.

As mentioned before, compiler optimizations are mostly concerned with sequential performance. For that reason, I focus on execution on a single core. This also facilitates analysis of the results, because tools do not need to handle parallelism. I run on an Intel Xeon Platinum 8160 processor, which is clocked at 2.1 GHz and provides three level of caches: The L1D cache is private to each core and has a size of 32 KB. The L2 cache is also private with a size of 1 MB while the L3 cache is shared with 1.375 MB per core. The processor microarchitecture has been code-named Skylake and was launched in 2017.

For compilation I use the same Skylake system running CentOS 7.7.1908. When comparing different compilers, I employ the following versions unless noted otherwise:

- Clang 10.0.0, labeled as `clang`,
- the GNU Compiler Collection in version 10.1.0, shown as `gcc`, and
- version 19.1.1.217 of the Intel Compiler, labeled as `icc`.

## 1.2. Structure

The remainder of this thesis is structured as follows: Chapter 2 begins with short explanations of background knowledge. It starts with a brief introduction about the phases of modern compilers. The chapter continues with the advantages of an Intermediate Representation (IR) and possible optimization passes before generating machine code. Finally, I discuss the difficulties of ordering optimizations and related work in literature.

In Chapter 3, I introduce my workflow as the central contribution of this thesis. Its description is independent of the specific benchmarks discussed later. This makes it possible to apply the workflow in general and beyond the scope of this thesis. The explanations are also agnostic of the used tools to gather the needed data. Nevertheless, later sections contain practical considerations for currently available software.

Afterwards, Chapter 4 presents properties of kernels (see Definition 5). This aims to help with the analysis phase of the workflow. The properties are generically based

on patterns in source code, but again independent of real applications. For a practical discussion, I present small example code snippets and their properties.

Finally, I evaluate the applicability of the workflow in Chapter 5 by means of case studies. These include a program from the NAS Parallel Benchmarks and mini-applications developed by the Department of Energy (DoE). The case studies are chosen to show the different usage scenarios of the workflow. I present select results from applying the workflow and discuss performance improvements of up to 14 %.

I draw conclusions in Chapter 6 and summarize my findings. This includes a discussion how scientists could apply my work to their applications. Finally, I mention possible extensions of the workflow for broader usage.

# 2. State of the Art

This thesis aims to examine the effect of compiler optimizations. To that end, the following chapter establishes required background knowledge: It starts with an explanation of compiler phases in Section 2.1. In particular, I give a short overview of lexical, syntax, and semantic analysis in modern compilers. Afterwards, Section 2.2 discusses the concept and advantages of an Intermediate Representation. It also describes some target independent optimizations that are of relevance for later parts of this thesis. The third Section 2.3 continues with the translation to machine instructions. This concludes the stages to compile source code into a binary for the target architecture.

In the second part of this chapter, Section 2.4 briefly discusses how optimizations are composed in current compilers: In many cases, the effectiveness of optimizations depends on the order in which transformations are applied. The employed heuristics are one of the reasons for varying performance between different compilers mentioned before. Finally, Section 2.5 closes with references to related research to tackle this problem.

## 2.1. Compiler Phases

Compilers are an important part of modern software development: They allow to write portable applications in higher-level languages. However, hardware, in the form of processors, needs machine code for execution. The task to translate between these two representations is left to the compiler. The resulting binary can be optimized for different metrics, for example code size or memory usage. For HPC applications, the primary criterion is typically performance during execution.

To improve on this metric, researchers and industry have worked on a number of optimization techniques. They are responsible to produce fast code that runs well on the target architecture. However, this makes modern compilers themselves very complex pieces of software. To deal with this situation, literature suggests a modular design with several phases [16, 32].

In the beginning the input file is processed by *Lexical Analysis*. This splits the source code into *tokens*, which may have additional attributes. For example, a variable identifier is characterized by its name. Afterwards, *Syntax Analysis* constructs a derivation tree of the token stream. The underlying context-free grammar specifies the syntax of the programming language. The result is usually referred to as Abstract Syntax Tree (AST).

The AST is then the representation used in the third phase, *Semantic Analysis*. Its purpose is to infer additional knowledge about the compiled program. For example, this includes type checking, which cannot be expressed using context-free grammars. Finally, the compiler needs to generate machine code from the AST.

```
1 int f() {
2   int x = 1;
3   int y = 2;
4   x = x + y;
5   return x;
6 }
```

(a) multiple assignments

```
1 int f() {
2   int x1 = 1;
3   int y = 2;
4   int x2 = x1 + y;
5   return x2;
6 }
```

(b) single assignment

Listing 2.1: Example illustrating Static Single Assignment.

## 2.2. Intermediate Representation and Optimizations

In most modern compilers, generation of machine code is preceded by the translation into an Intermediate Representation (IR). It is usually closer to actual execution than the AST, which facilitates optimizations. Still, the IR provides a clear separation from the final machine code described in Section 2.3. This offers the advantage that optimizations can be written independently of the target architecture. As a result, it is less work to port the compiler to a new architecture because these transformations can be reused.

The IR also allows to choose a design well-suited for optimizations. Literature suggests that a Static Single Assignment (SSA) form simplifies transformations [32]. The idea of SSA is that each variable is only assigned once and never changed. As an example, consider the C code of a function `f` in Listing 2.1. In the left code, the variable `x` is assigned twice: It is initialized in line 2 and later updated to the result of the addition with `y` in line 4. The latter requires the compiler to distinguish between the two uses of the variable in the assignment. This is different in the right code where `x` has been replaced by two new variables `x1` and `x2`.

The advantage of the SSA form is the reduced complexity of certain analyses and optimizations. For example in line 4 of Listing 2.1b, the value of `x2` is determined by `x1 + y`. These two variables are initialized to constant values in lines 2 and 3. As such the compiler can propagate their values and assign the constant 3 to `x2`. This is easily possible because SSA guarantees that the values of the variables never change after assignment.

In a similar way, the compiler can perform other optimizations on the IR. One important requirement is that these transformations are independent of the target architecture. However, it is still allowed that certain parameters depend on the final code generation. For example an optimization could behave differently if a required machine instruction is not supported.

The following sections present optimizations whose understanding is of relevance for this thesis. The mentioned transformations are only a small subset of the optimizations described in literature. An overview of other basic transformations can be found in “A Catalogue of Optimizing Transformations”, compiled by Allen and Cocke in 1971 [19]. More literature on the early work of program optimization is listed in [18].

### 2.2.1. Dead Code Elimination

Using the previously described constant propagation, the compiler can sometimes infer that part of the code is never executed. Such regions are designated *dead* and can be removed without altering the program behavior during execution [19]. Accordingly, the optimization is referred to as Dead Code Elimination (DCE).

A closely related optimization is Dead Store Elimination (DSE). As the name suggests, it aims to remove dead stores to memory instead of executable code. This is based on the observation that programs might contain stores to the same memory location without intervening reads or other synchronization. In such case, the last store determines the value in memory and the previous accesses can be removed. As for DCE, the program behavior stays unchanged but performance might improve due to fewer memory accesses.

### 2.2.2. Inlining

Another optimization possible at IR level is function inlining (called "procedure integration" in [19]). This is motivated by the fact that function calls incur a small overhead for multiple reasons: Firstly, execution must jump to a different code address and at the end return to the next instruction after the call. Modern processors have specialized hardware units to predict branches, but their capabilities are still limited. Secondly, arguments must be passed according to the calling convention, which sometimes includes pushing to the stack. Finally, the called function needs to setup its own stack frame for storing temporary values and saved register values.

To inline a function, the compiler copies the code into the calling context. In doing so, it can deal with parameters and return values once instead of passing them at run time. Additionally, there is only one stack frame of the surrounding function, which can be enlarged as necessary. The inlining might also enable further intra-procedural optimizations where code analysis is limited to a single function [19]. For example, it might be possible to propagate constant arguments and remove dead code after inlining a function into a particular caller.

### 2.2.3. Loop Optimizations

Loops are another control flow structure that is subject to optimization. Two well known transformations include Loop Invariant Code Motion (LICM) and loop unrolling. The first attempts to move code out of the loop that is invariant for all iterations. In particular, this includes computations that are independent of the loop variable. This avoids repeated evaluations and makes it possible to keep the result in registers [19].

Unrolling, on the other hand, is concerned with replicating the loop body, which enables further optimizations. For example, it might be possible to eliminate repeated checks of the loop header [19]. The replication also reduces the number of jump instructions executed during run time. Additionally, unrolling might be a prerequisite for vectorization as discussed in the next section.

### 2.2.4. Vectorization

Many architectures today have specialized instructions to operate on vectors of data. According to Flynn's taxonomy [29], they belong to the category of Single Instruction Multiple Data (SIMD). To make use of these instructions, the compiler has to structure the code accordingly. This is usually performed on the IR because many aspects are independent of the concrete architecture.

There are two major vectorization techniques known in literature: Loop vectorization extends the transformations discussed in the previous section. For that, it moves operations from consecutive loop iterations into vector registers. In contrast, the Superword Level Parallelism (SLP) vectorizer works on combining scalar operations into vector instructions [42]. For handling of loops, this requires unrolling of the loop body to expose the parallelism. However, the idea of SLP also applies to operations outside of loops if they can be reordered without affecting the result. The two techniques target different opportunities and can be implemented as separate passes in the same compiler.

## 2.3. Machine Code

Finally, the compiler needs to generate machine code by translating the IR for the target architecture. This stage is usually split into multiple smaller steps: Instruction selection is concerned with choosing appropriate operations from the ISA. Additionally, the compiler performs instruction scheduling to increase Instruction-Level Parallelism (ILP) and thereby performance. At the end, Register Allocation (RA) assigns hardware registers to the instructions.

During or after this pipeline, it is possible to apply *Peephole Optimization* [47]. This refers to replacing instruction sequences with faster alternatives yielding equivalent results. For that, the compiler analyzes a sliding window of operations, called the *peephole*. Possible transformations include the removal of redundant instructions or simplifications of jumps [16].

The compiler also needs to lower the vectorized code to use SIMD instructions. This complements the target independent transformations described in Section 2.2.4. Moreover, the compiler might use specialized instructions, for example to speed up multiply-accumulate operations:

$$(\text{-})\, a \pm (b \cdot c) \tag{2.1}$$

In the case of addition, this is also referred to as fused multiply-add (FMA).

## 2.4. Composing Optimizations

The first transformations for optimizing compilers were described already in the 1960s and early 1970s [49, 47, 28, 17, 19]. Since then a large number of analyses and optimizations have been proposed in literature. This requires the engineers to choose a set of transformations suitable for their compiler: Integrating many transformations

costs a lot of effort and potentially introduces bugs. Consequently, compilers usually include transformations that are beneficial for their users' needs.

Once implemented, the compiler also has to decide when to apply a given transformation. For example, inlining as described in Section 2.2.2 duplicates code if a function is called multiple times. This increases the overall code size and is usually not advisable for large functions. On the other hand, it enables further optimizations as discussed above.

The solution implemented by most production compilers is to employ heuristics. These try to estimate the possible benefit of applying a transformation. Based on this approximation, the compiler decides if the optimization should be performed. Additionally, many transformations include cost models to steer individual parameters. For example, the number of unrolled loop iterations may depend on the length of the loop body [19].

These decisions are further complicated by a sequential optimization pipeline: In current compilers, optimizations are applied in a fixed order. As a result, transformations need to work on the result of their predecessors. This implies that compiler writers need to take interaction of transformations into account when designing the pipeline. Unfortunately, these relationships are non-trivial and very hard to predict for all possible codes.

The heuristics and the optimization pipeline are unique to every compiler. In fact, they usually change from one release to another as improvements are made. This is why their generated code can differ significantly, leading to varying performance during execution.

The mentioned heuristics usually make better decisions the more information they have available. One technique to gather data from another source is Profile-Guided Optimization (PGO): Here, the compiler gets a profile from an instrumented execution of the application. This gives more precise information about the frequency of branches and function calls. With Link-Time Optimization (LTO), there is an additional run of the optimization pipeline when linking. This means the whole program is visible to the pipeline, which makes Interprocedural Optimization (IPO) passes more effective.

Another related approach is binary optimization, which is applied after the executable has been linked. One recent example is the BOLT optimizer presented by Facebook engineers [52]. It also relies on sample-based profiling and reorganizes the compiled code to improve performance. For example, it tries to locate hot functions close to each other, which reduces pressure on the instruction cache.

## 2.5. Related Work

The idea of automatically searching for the best performing code dates back more than 20 years: In 1997, Bilmes et al. described PHiPAC as a methodology to write portable C code. The developed routines are parametrized and scripts search for the values with the best performance [24]. Soon after, Whaley and Dongarra published their work on Automatically Tuned Linear Algebra Software (ATLAS) [58]. They concentrate on optimizing the general matrix-matrix multiplication. This can be

used to produce fast versions of the other routines in BLAS level 3 [27].

With the increasing maturity of optimizing compilers, other research work focuses on finding the best set of options. A promising technique is "iterative compilation" which evaluates the generated code after compilation [25]. Unfortunately, testing all possible combinations is infeasible due to the large search space [40]. As one solution, Triantafyllis et al. developed Optimization-Space Exploration (OSE) [57]. Their idea is to make use of the existing compiler heuristics to limit the number of combinations to explore.

The approach of iterative compilation can also be used to address another problem described in the previous Section 2.4: Cooper et al. describe "adaptive optimizing compilers" to explore different orderings of optimizations [26]. Other works employ machine learning to improve compiler heuristics [56] or to mitigate the problem by predicting beneficial orderings [41].

In addition to the compilation of particular applications or libraries, researchers have also looked at improving the set of standard optimization settings. Haneda et al. report results with randomly generated settings [34]. Hoste and Eeckhout present the framework Compiler Optimization Level Exploration (COLE) [36]. It allows to find Pareto optimal settings for multiple metrics, such as compilation time as well as execution time.

All of the work cited so far is concerned with automatic procedures. Their shared goal is to find an optimum to some objective function. Unfortunately, this does not help to understand the effect of compiler optimization. To the best of my knowledge, this area has not been explored yet.

One step into this direction has been taken by Granston and Holler [31]: Their tool "Dr. Options" recommends compiler options based on information about the application. This includes data supplied by the user, the compiler, and optionally a profiling run. The tool matches this information with encoded feedback from tuning experts and outputs the recommendation. However, the work's focus is very narrow by targeting Hewlett-Packard's PA-RISC architecture and compiler. Additionally, the recommendations will be most useful for few standard cases that the tool was developed for. I envision that such tool would be less helpful for more advanced and yet unseen problems.

# 3. Workflow

As seen in the previous chapter, compilers perform many optimizations guarded by heuristics. These optimizations are important to get good performance from portable source code. As such compilers writers have implemented many different optimizations and strategies. Various flags are available to enable and tweak the passes or groups thereof. This leads to varying performance for a given source code when compiling into executable binaries. However, the exact causes for the performance differences are not well understood in detail.

To study the optimizations' effect, I present a workflow to compare the execution of two binaries. To that end, Section 3.1 establishes the needed terminology used in the description. Based on the given definitions, this allows to formally define the goals of the workflow and discuss scenarios not handled in this thesis. The proceeding sections explain the workflow in detail: Section 3.2 describes how to compare two profiles measured by the workflow. Afterwards, I show how to refine the profiling and compare the generated instructions in Section 3.3. Finally, Section 3.4 lists possible optimizations to improve the performance. I give practical considerations for applying the developed workflow in Section 3.5.

## 3.1. Terminology and Goals

The purpose of this section is to establish terminology to describe the proposed workflow. This will involve interaction between multiple entities, designated as follows:

- The person applying the workflow is called a "performance analyst". This may be abbreviated as *analyst* for short.
- Compiling source code results in object code for a target architecture. After linking, it is possible to get a binary that can be executed on a computer system. For this thesis, I use the terms *binary* and *executable* interchangeably.

Finally, a binary may be run on a system comprised of both hardware and software. The following two definitions characterize the environments during build and run time. They are important to get reproducible measurements and results.

**Definition 1** (build configuration)**.** The *build configuration* captures the way of compiling source code into an executable. This includes the used software and their versions as well as all options passed to it. It also comprises the hardware and how it is configured at the time of compilation.

*Remark.* The hardware is relevant because it might influence options implicitly during build. For example, many compilers support a flag that optimizes for the current architecture. That is, source code compiled with the same flags is actually optimized for different systems.

**Definition 2** (run environment)**.** The *run environment* captures where and how an application is executed. The first part is made up of the system hardware and the installed operating system. Additionally, it is defined by the resources actually used for execution. This includes the processing elements and accessed partitions of the available memory. Finally, it comprises the application configuration, its parameters, and the input data.

A run environment is *reproducible* if the system is idle except for required background services, and measures are taken to minimize the jitter (for example by pinning the execution).

In this thesis, I will use the measured runtime to compare the performance of two executables. This metric has the advantage that it is generally available and can be measured at different levels. Additionally, it is less ambiguous than the number of operations or retired instructions. Based on the established notion above, this allows to define "differences in performance":

**Definition 3** (difference)**.** Given two binaries compiled from the same source code with unique build configurations, executed in the same reproducible run environment. The measured performance shows a *difference* if one binary is statistically and reproducibly faster than the other. The difference is said to be *significant* if its relative value is larger than some defined threshold.

This definition is very central to the thesis and the developed workflow. As such, I would like to highlight three important aspects in above definition:

1. The given definition is agnostic of the granularity used for measuring the runtime. Hence, it can be applied to binaries as a whole, but also to parts of them such as functions or even instructions.

2. A difference is measured despite the same source code and the reproducible run environment. This implies it must be caused by a change in the build configuration.

3. The notion of "significance" depends on an externally defined threshold. This can be used as a parameter of the workflow that steers which differences are analyzed. As a result, the workflow can be applied with a very flexible degree of detail.

### 3.1.1. Goals of the Workflow

With the defined terminology, it is now possible to express the goals of the proposed workflow: The starting point is a significant difference of the total runtime between two binaries from the same source code. By applying the workflow, the aim is to locate differences more precisely, allowing to analyze and understand their causes. Ultimately, a thorough understanding enables the analyst to improve the performance.

However, it is initially not important what part of the build configuration causes the difference. According to the definition there are several parts that could influence performance:

- The difference can be related to the used software and their versions. Most importantly, this means the compiler, which is the focus of this thesis. In a broader sense, the software also includes linked libraries, which I will not look at. These can be difficult to analyze for two reasons: Firstly, the libraries are sometimes beyond control of the application developer. This holds for runtime libraries employed by OpenMP and MPI that are tightly integrated with the system. Additionally, source code for optimized vendor libraries like Basic Linear Algebra Subprograms (BLAS) is rarely available to the end user. The latter makes it impossible to apply the proposed workflow which relies on recompilation.
- Obviously, compiler options influence the optimization passes and thereby performance. In the most basic case, this includes standard options to set the optimization level via `-O` or target a specific architecture. However, most compilers also support many advanced flags to override heuristics and their decisions.
- Finally, the hardware can indirectly influence choices of the compiler, in particular for the architecture's support for certain instructions. However, the same effect can often be achieved by adding explicit options. As such, I neglect this degree of freedom by choosing a fixed system for building and running.

To summarize I focus on differences caused by the compiler and its options for this thesis. Also the workflow does not aim to propose improvements in the compiler by itself. Still, this can sometimes be a secondary result after fully understanding the difference.

### 3.1.2. Overview of the Workflow

The following section briefly paraphrases the proposed workflow to give a high-level overview. Figure 3.1 shows a corresponding graphical representation. More detailed explanations of the steps are found in Sections 3.2 to 3.4.

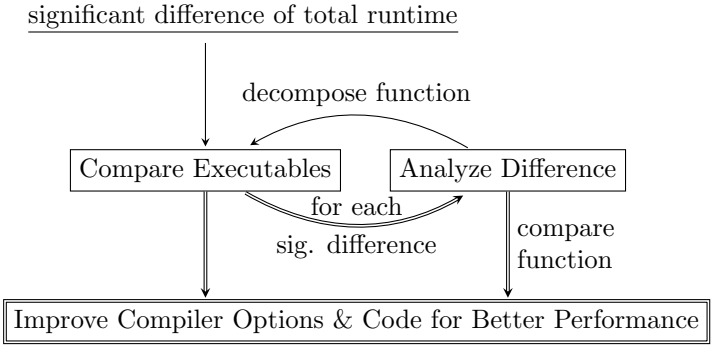


Figure 3.1.: Overview of the workflow for analyzing the effect of compiler optimizations.

As mentioned above, the workflow starts after identifying a significant difference of the total runtime between two binaries. To isolate the difference, the workflow follows a strategy of divide-and-conquer: In a first step, the two executables are compared by means of profiling data. This allows to identify performance differences at function level and decide which are significant. Afterwards, these differences are analyzed and recursively decomposed into smaller functions. By repeatedly profiling the binaries, this gradually increases the level of detail.

After a finite number of steps, the workflow isolates (possibly multiple) differences. In most cases, these are located in small parts of the application's source code. As such, it is tractable to analyze each significant difference in greater detail. In this thesis, I propose to eventually compare the functions' assembly instructions. This allows to relate the measured differences with optimizations performed by the compiler.

## 3.2. Comparing Executables

The following section describes how to find differences between two executables. A visual representation is given in Figure 3.2. The first step is to compare the total runtime of the two binaries, which has two reasons:

- For the initial profile, this step ensures that the profiling tool did not add disproportionately large overhead. In particular, the significant difference observed initially must still be present.
- After recompilation, it is important to verify that the previous differences are still visible.

In the second case, the comparison ensures consistent results while applying the workflow: In later stages of the analysis, the workflow describes modifications to refine the level of detail. This is needed for the strategy of divide-and-conquer mentioned before. However, the modifications can negatively influence the runtime of the

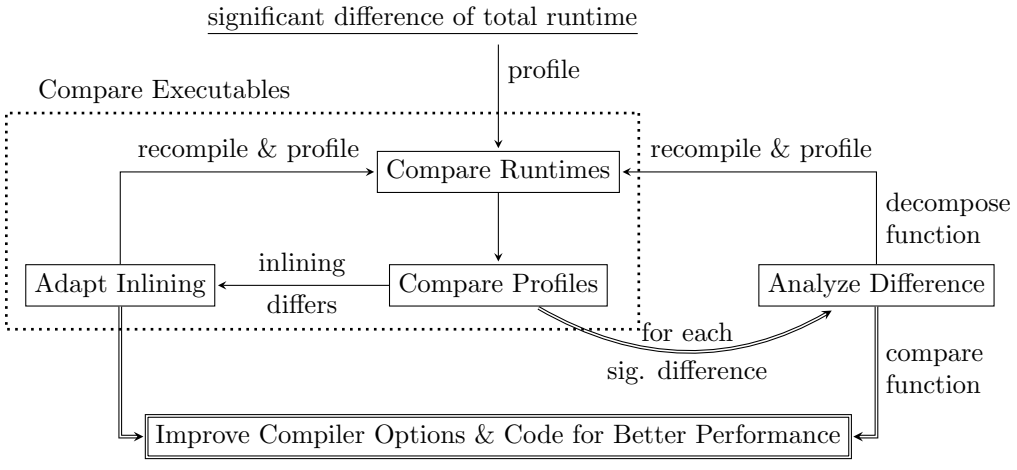


Figure 3.2.: Workflow for analyzing the effect of compiler optimizations, details for step "Compare Executables".

recompiled binaries. In this step of the feedback loop, it is thus important to compare the runtimes to previous executions:

Assume that the runtime of the faster executable increased significantly compared to previous runs. If the difference to the other runtime is now smaller than before, the modification is the primary reason. As such, it might make sense to apply the opposite modification to the other binary. Concrete cases of this are discussed for each of the modifications described later in the chapter.

### 3.2.1. Compare Profiles

The second step is to compare the runtimes of individual functions from the two executables. This requires that the two profiles contain "matching" data as defined in the following:

**Definition 4** (comparable profiles, topmost & matching function sets)**.** A *profile* captures the aggregated time spent in functions of a binary during execution. These functions can be sorted into a list by their respective runtime. Every non-empty prefix of this ordered list is called a *topmost function set*.

Two profiles are *comparable* if they have a common topmost function set. The largest common set is called the *matching function set*.

*Remark.* If it exists, the matching function set of two profiles is unique: Topmost function sets for a specific profile are prefixes of an ordered list and form a total order. Thus there is a unique largest set that is common for both profiles.

In this thesis, the workflow has no further requirements except the aggregated runtime per function. That is, it relies on flat profiles without call graph information. This is motivated by the fact that the profiles are measured with the same input data. As such, the workflow assumes that both executions call the functions in the same order and equally often. This allows to use profiling tools based on sampling, as discussed further in Section 3.5.

For profiles with common function sets, it is possible to compare the runtimes of the contained functions. To illustrate this concept and the previous definitions, consider a source code with functions `A`, `B`, `C`, and `D`. Suppose there are four different build configurations, and runs of the compiled binaries result in corresponding profiles. Table 3.1 lists the visible functions in each of the profiles. The entries are sorted by the function's runtime, which is denoted in parentheses.

The first two profiles are clearly comparable: They have the common topmost function sets $\{\texttt{A}, \texttt{B}\}$ and $\{\texttt{A}, \texttt{B}, \texttt{C}\}$. The latter is also the matching function set, which in this case comprises all visible functions. The runtime differences between functions `A` and `B` in the two profiles are negligible, while `C` has the same runtime in both.

The first and third profile have $\{\texttt{A}\}$ and $\{\texttt{A}, \texttt{B}\}$ in common while the second and third only have $\{\texttt{A}, \texttt{B}\}$. For both combinations, $\{\texttt{A}, \texttt{B}\}$ is the matching function set and the runtime differences for these functions are again small. Function `C` is not included because the third profile has function `D` with a larger runtime. This means comparison of the matching function set would miss the potential difference between function `C` in the first two profiles (30 s) and the sum of `D` and `C` in the third (37 s).

| | | | |
|---|---|---|---|
| `A` (40 s) | `B` (40 s) | `A` (40 s) | `B` (40 s) |
| `B` (39 s) | `A` (39 s) | `B` (39 s) | `D` (18 s) |
| `C` (30 s) | `C` (30 s) | `D` (25 s) | `C` (12 s) |
| | | `C` (12 s) | `A` (10 s) |
| (a) Profile I | (b) Profile II | (c) Profile III | (d) Profile IV |

Table 3.1.: Four profiles to illustrate the concept of matching functions. Functions are ordered by their runtime. Function `D` is not visible in the first two profiles.

The fourth profile has no common topmost function set with the first one and only {`B`} with the second. This is because is has function `D` in the second position, which is not visible in the first two profiles. One reason could be that it was inlined into function `C`, which would match the given runtimes. With the third, the profile shares the common function set {`A`, `B`, `C`, `D`}. All functions taken together also form the matching function set and the profiles are comparable. However, the runtimes for function `C` differ by 28 % while function `A` is four times faster in Profile IV.

When comparing two profiles, the workflow considers the functions from their matching function set: By requiring topmost function sets, it is ensured that no function is "skipped" after sorting by runtime. In the above case, comparison of all functions from the second and third profile would have neglected function `D` while finding a spurious difference in `C`. At the same time, the workflow aims to compare as many functions as possible. That is why it considers the largest common set, the matching function set.

It is apparent that the matching function set can comprise varying parts of a profile. In the best case, the profiles capture the same set of functions in possibly different orderings. This is the situation for the first two or the last two profiles from the previous example. Their functions can be compared without restrictions. For every significant difference, Section 3.3 explains how to analyze in detail.

The above example also discusses profiles where some function is not visible. This is oftentimes related to different inlining strategies. As mentioned before, the profiles might still be comparable, even if their matching function set does not contain all functions. However, this becomes a problem if the difference is located in one of the missed functions. It is also possible that two profiles are not comparable at all. In both cases, it is necessary to first adapt the inlining as explained in the following section.

### 3.2.2. Adapt Function Inlining

Ideally, the matching function set contains all functions from the two profiles. This would imply that the set covers the complete runtime. As such the workflow is guaranteed to find a difference by comparing the functions. However, this is not the case if a function is only visible in one of the profiles.

There are two ways to make the matching function set cover a larger part of the runtime: Either increase the number of functions visible in both profiles or have

bigger functions. This is closely related to the compilers' decisions whether to inline functions. Incidentally, inlined functions are usually those that are "invisible" in the profiles. Thus, the workflow could either propose to disable inlining of particular functions or force additional inlining.

As discussed before, the workflow builds on profiling the runtime spent in functions. This means less inlining and having more distinguishable functions provide a more fine-grained view of the executable, which should be preferred. However, replacing inlined code with a function call might prevent other optimizations. Furthermore, it introduces additional instructions and overhead. This is because of the calling conventions and resulting register usage as mentioned in Section 2.2.2.

It is possible that after recompilation the binary with less inlining is significantly slower than before. As mentioned above, it then makes sense to apply the opposite modification for the other executable. Here this means forcing additional inlining, which could lead to improved performance. If that is observed, such result is particularly important for the previously slower binary: The modification mimics inlining decisions from the faster executable, which were previously different. To verify such scenario, the additional inlining should also be applied to the original source code. If performance is similarly improved, the modification is the first result of the workflow.

## 3.3. Analyzing Differences in Functions

The previous section explained how to identify significant differences at function level. To follow the strategy of divide-and-conquer, the workflow proceeds with analyzing them until successful. Figure 3.3 gives a graphical representation of this part of the workflow.

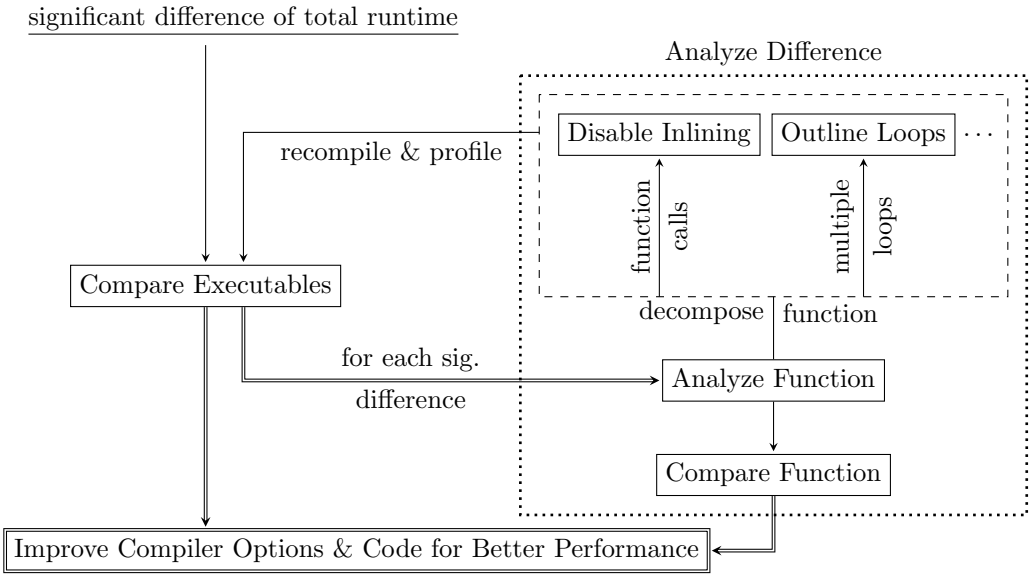


Figure 3.3.: Workflow for analyzing the effect of compiler optimizations, details for step "Analyze Difference".

If possible, the workflow first tries to decompose the identified function. This aims to make the function reasonably small to analyze the contained instructions. The way to decompose a particular function highly depends on its structure. Concretely, I describe how to handle inlined function calls and multiple loops.

### 3.3.1. Disable Inlining of Function Calls

As discussed in Section 2.2.2, function inlining can lead to better performance for multiple reasons. However, it can hinder the comparison of profiles as already seen in the previous section. In addition, inlining also complicates the analysis of a difference because it mixes instructions from different parts of the source code. Thus, it is helpful to avoid inlining of direct function calls to decompose a difference. This results in separate functions that can be clearly attributed in the profile. Thereby, it becomes possible to compare their runtimes as described before.

It was mentioned in Section 3.2.2 that such changes can negatively impact performance. For this reason, it is advisable to start with functions that are called only few times. Particularly functions called inside of loop bodies executed many times can incur a high overhead. In a first step, these function calls should be left to the compiler to optimize as needed. Instead, the loops themselves can be handled as described in the next Section 3.3.2.

To illustrate the concept, consider the example code in Listing 3.1. If the functions `workA`, `workB`, `update`, and `exchange` are defined in the same translation unit, the compiler is likely to inline the calls. As a result, a profile will only show the time spent in the surrounding function `f`. This makes it hard to localize the cause for the difference, which could be related with any of the four functions.

To get more fine grained information, the first step is to avoid inlining of the call to `update`. This should incur little overhead since the function call is outside of the loops and only executed once. Still, the effect on performance should be evaluated by

```
1  void f(int *len, int N) {
2    for (int i = 0; i < N; i++) {
3      for (int j = 0; j < len[i]; j++) {
4        workA(i, j);
5        workB(i, j);
6      }
7    }
8
9    update();
10
11   for (int i = 0; i < N; i++) {
12     exchange(i);
13   }
14 }
```

Listing 3.1: Example of a function with multiple loops and function calls.

comparing the total runtime as described in the beginning of Section 3.2. In contrast, the functions `workA`, `workB`, and `exchange` are called repeatedly. Not inlining them can also inhibit other optimizations such as vectorization. This is more likely to result in a measurable impact on performance. I describe conditions for still handling them in Section 3.3.3.

### 3.3.2. Outline Loops

The same idea as before can also be used to decompose functions with multiple loops: Instead of avoiding function inlining, I propose to actively outline loops into separate functions. After recompilation, the individual loops are visible in the profiling data via their outlined functions. This makes it possible to identify the loops that cause the differing execution times of the original function. These can be analyzed further by the workflow.

As an example, reconsider the function `f` in Listing 3.1. In total there are three `for` loops, but the one from lines 3 to 6 is nested inside another loop. To minimize the added overhead to call the outlined functions, it is advisable to focus on outer loops first. In the discussed example, the function `f` contains two outer loops: the first one from lines 2 to 7 and the other from lines 11 to 13. Outlining them into their own functions results in the code as presented in Listing 3.2. This makes the individual timings visible in a function-level profile. For that to work, care has to be taken that the compiler does not inline them again during optimization.

```
1  void f_loop1(int *len, int N) {
2    for (int i = 0; i < N; i++) {
3      for (int j = 0; j < len[i]; j++) {
4        workA(i, j);
5        workB(i, j);
6      }
7    }
8  }
9
10 void f_loop2(int N) {
11   for (int i = 0; i < N; i++) {
12     exchange(i);
13   }
14 }
15
16 void f(int *len, int N) {
17   f_loop1(len, N);
18   update();
19   f_loop2(N);
20 }
```

Listing 3.2: Outlined loops of the previous example listing.

### 3.3.3. Decompose Loop Bodies

By outlining loops as described above, the workflow is able to identify differences for individual loops. However, the loop bodies might still be too large for further analysis. For example, assume that the workflow identifies a difference in the function `f_loop1` of Listing 3.2. In that case, it is still unknown whether the difference is caused by `workA` or `workB`. A similar situation can arise if the loop contains multiple nested loops.

To get more data from such loops, it is possible to attempt further decomposition of the loop body. This includes both disabling the inlining of function calls and the outlining of (nested) loops as described in the previous two sections. However, close attention must be paid to ensure consistent results: As the loop body is executed multiple times, such decomposition might strongly influence performance. For that reason, it is important that the analyzed difference is still visible after recompilation. If this is not the case, the decomposition has to be reversed and the function must be analyzed as a whole.

### 3.3.4. Compare Function

At some point of the process, the analyzed function cannot be decomposed further as described before or not without hiding the difference. For the rest of this thesis, I will refer to such functions as kernels:

**Definition 5** (kernel)**.** A *kernel* is a function that contains

- at most a single outer loop, with possibly multiple nested ones, and
- no function calls outside of loops.

To pick up the previous example, `f_loop1` and `f_loop2` in Listing 3.2 are kernels. Meanwhile the original function `f` in Listing 3.1 violates both conditions as it consists of two loops and calls `update` at function scope.

The simple structure of a kernel implies that its source code directly influences all generated machine instructions. As a result, the difference in runtime must be caused by the compiler's code generation. This opens a chance to identify the difference in the executed binary itself.

The path towards this goal depends on the contained code and possible optimization classes. As such, the following paragraphs describe techniques to gather more data: The first three are concerned with analyzing the generated machine code instructions and their execution behavior. The optimization reports described last contain information about the compiler's heuristics and decisions. However, the reports can be very detailed, so it is oftentimes useful to know what to look for. The properties of kernels described in the following Chapter 4 can help to identify important optimizations.

**Compare quantity of instructions** In some cases, the executables contain very different kinds or numbers of instructions. This can easily be detected by a static analysis of the generated code: For usage in this workflow, I propose to map each

instruction to the associated mnemonic. Afterwards, it is sufficient to count how often each of them occurs in the analyzed function.

Vectorization is one important class of cases that can be handled like this: The different instructions allow to clearly distinguish between a sequential and a vectorized loop. Another situation detected by the number of instructions is loop unrolling. In general, such transformation will generate more instructions by duplicating the loop body [19]. However, during execution, this could lead to less jumps and less overhead for the loop header.

**Identify time-consuming instructions** In many cases the quantity of instructions is unable to reveal any obvious difference. This makes it necessary to compare the generated code at assembly level. To limit the scope of the analysis, it is important to identify "relevant" instructions. For this workflow, "relevance" can be decided based on the sampling data contained in the profile: Similar to functions, the assumption is that sample counts are proportional to execution time. Thus instructions sampled more often can be considered more "relevant" than others.

In a second step, the identified instructions need to be compared between the executables. The exact procedure highly depends on the class of instructions at hand and the application domain. As such, this step relies on expert knowledge of the performance analyst.

Once understood, the found difference can often be related to expressions in the original source code. The correspondence is obvious if the function is short and there are only few instructions. Otherwise, it is advisable to rely on debugging information added by the compiler. Even in case of optimizations and moved instructions, this usually points to the region of interest.

**Include data from hardware counters** The performance of generated code is also influenced by other factors, including caches and branch predictors. To compare functions at this architectural level, it may help to look at hardware counters. Availability of such counters depends on the vendor and the particular processor. Additionally, the profiler must be able to measure events generated by the counters. If integrated with sampling, it is possible to get numbers of events per function. Similar to the execution time, this allows to compare the two binaries.

**Consult optimization reports** Many compilers offer some form of report with details about the performed optimizations. This may contain information about missed passes due to unavailable or insufficient analysis results. For example, the compiler needs precise information about pointer aliasing to preserve correctness. This is particularly important for many loop transformations that could otherwise lead to wrong results.

The report also allows to understand the influence of heuristics that guard the optimizations. As explained in Section 2.4, they control whether to apply the optimization and to what degree. However, these heuristics are often implemented with a specific type of application in mind. As such, they can produce unexpected or suboptimal results for other cases.

## 3.4. Improving the Performance

It should be clear that the outcome of the previous analysis steps can be very diverse. In the following section, I list some common scenarios that could be encountered. Applications of the workflow in later chapters will include examples of possible improvements. However, it is important to note that the mentioned cases are by no way exhaustive for all possible codes.

In some cases, the analysis shows that a particular heuristic is responsible for the increased runtime. Fortunately, most compilers offer advanced options to adapt thresholds or overwrite decisions. It may also happen that a compiler does not perform a transformation by default. This can have multiple reasons, for example that a pass was not observed beneficial in general. In such cases, there are usually flags to still enable the optimization. The advantage of this approach is that it requires no changes in the application. This means the source code itself remains portable and is easier to maintain.

In other cases, a compiler's analysis may be unable to deduce some needed knowledge. This can sometimes be solved by encoding additional guarantees into the source code. Ideally, the modifications should make use of standardized functionality to retain portability. For example, C99 [13] introduced the `restrict` qualifier for pointers:

> An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association [...] requires that all accesses to that object use, directly or indirectly, the value of that particular pointer. (ISO/IEC 9899:1999 [13, p. 109])

Thus, the compiler can assume that no independent pointer aliases the same object. Furthermore, it also conveys additional information for a future maintainer of the application.

Finally, it can happen that a compiler lacks capability to perform a needed transformation. This can be considered a weakness in the compiler's optimization pipeline. As such, the workflow may point to areas that need improvement by compiler writers. However, some transformations are very hard to detect, even for sophisticated compilers. An example could be that one of the code path multiplies the result of an expensive computation with zero. In such cases, it may be justifiable to adapt the source code and avoid the computation altogether.

## 3.5. Practical Considerations for Applying the Workflow

The workflow described in this chapter is based on comparing runtime profiles. So far, I have only mentioned the need to measure the time spent in individual functions. To conclude the presentation of the workflow, this last section discusses how to obtain this data in practice. For the purpose of this thesis, I have looked at the following tools available at the university's HPC system:

**GNU gprof** [2] analyzes the output from an instrumented executable. It requires the compiler to insert additional instructions at the beginning of each function. These

call a runtime library that saves the entered function and its caller in a table. Furthermore, the library samples the program counter and periodically records the executed instructions.

This information is used to construct a call graph as proposed by Graham et al. in their paper from 1982 [30]. Additionally, the `gprof` command supports so-called flat profiles. They show only the total time spent in each function. Finally, it allows to annotate the source code based on the sampled program counter.

**Linux perf** [9] was initially a tool to read performance counters with the Linux kernel. It can also be used to sample the program counter and aggregate the time spent in each function. Based on infrastructure in the kernel, this makes it a lightweight profiler with little overhead. Similar to the previous tool, `perf` also supports recording call graph information. To capture the needed data, the default mode is to use the stack frame pointed to by the frame pointer.

**Intel VTune Profiler** [4] is a proprietary tool for performance analysis. Among other modes, it is able to sample the current instruction during execution of a binary. This information can be used to estimate the time spent in each function and detect hotspots. As the others, it also captures additional data to construct a call graph.

As expected for a profiler, all tools sample the program counter and aggregate time at function level. However, during the first experiments I noticed that inlined functions are handled differently: GNU gprof lists functions as emitted by the compiler, in particular not those that were inlined. The `perf` command has the same default when profiling without call graph data. When using debugging information to construct the call tree, it also shows information about inlined functions. The latter seems to be the default for Intel VTune unless disabled by a command line option.

Still, the information about inlined functions is not accurate for more complex cases: I observed profiles that either miss functions completely or attribute too little time for inlined code. This is more likely to happen if the compiler mixes instructions from multiple code locations.

It could be that future tools will improve and deliver more precise information. In the extreme case, the data would be accurately related to individual lines of code. This would greatly simplify the workflow without the need to decompose functions. However, for this thesis, it should be possible to apply the workflow with tools available today. This is why it only relies on profiles with non-inlined functions and decomposes them if needed.

Out of the tools mentioned above, `gprof` has the disadvantage that it needs support from the compiler. It works for the GNU and Intel compilers, but not for the Clang compiler by the LLVM project. As such, it may be unsuitable for analysis of arbitrary binaries. In a similar fashion, Intel VTune only supports the x86 architecture.

### 3.5.1. Profiling with Linux `perf`

For these reasons I will use Linux `perf` when applying the workflow in the following chapters. It has low overhead and runs on a variety of platforms, including x86,

PowerPC, and ARM. Being freely available, it often comes pre-installed in many HPC environments running GNU/Linux. However, the described workflow remains generic and agnostic of the used profiler. As such it can be applied with any other tool that provides function-level timings.

The choice of the profiling tool also influences how to compile the executable for profiling: Some tools require additional flags to the compiler to enable instrumentation, for example `gprof`. For `perf`, no such options are needed and profiling works with any binary that includes function tables. However, debugging symbols are helpful to analyze the results. In particular, they provide references from the assembly instructions to the original source code. For most compilers, debugging symbols can be enabled by passing the flag `-g`.

To profile an application with `perf`, it suffices to prepend the command with `perf record`. By default, this selects the `cycles` event to sample timing information. However, `perf` supports many other events that can be configured via the parameter `-e`, short for `--event`. This includes the symbolic names `cache-misses` and `branch-misses` that will be used in Chapter 5. Unless determined by parameter `-o` (for `--output`), the profiling data is stored in `perf.data`.

Afterwards, the sampling data can be analyzed using `perf report`. This command honors the parameter `-i` (for `--input`) to load a file other than `perf.data` in the current directory. If `perf` detects a terminal output, it starts with an interface for interactive analysis. Otherwise, it prints the list of functions in text form, which can also be forced by passing `--stdio`. Additionally, `perf report` supports many other flags for filtering and sorting. These are described in detail in the documentation [9].

# 4. Properties of Kernels for Optimizations

The workflow described in Chapter 3 aims to study the effect of compiler optimizations. To that end, it decomposes runtime differences between two executables by comparing profiling data. Eventually, the process arrives at a number of *kernel* functions as characterized in Definition 5. The analysis of such kernels on the other hand highly depends on its source code. This was already mentioned in Section 3.3.4 and also influences possible improvements described in Section 3.4.

Hence, this chapter defines properties of kernels and describes corresponding optimizations. The goal is to determine preconditions of compiler transformations that are important for HPC applications. This facilitates the task to understand the runtime differences decomposed by the workflow: In a first step, the analyst identifies the properties of the kernel function. They are based on source code because this is also the representation available to the compiler. As a result, the analyst obtains a list of possible optimizations to check. These are likely candidates to explain the runtime difference.

For the scope of this thesis, it is practical to restrict the properties in three ways: Firstly, the properties do not need to be exhaustive and cover all possible codes. Instead, I focus on properties and optimizations found important for HPC applications. Secondly, it suffices to determine required preconditions of the transformations. That is, the guidelines should point to a list of possible optimizations but do not need to guarantee soundness. Finally, each of the described properties is binary, either present or absent, but they are not mutually exclusive. For example, a kernel may have multiple memory access patterns as discussed in Section 4.3.

The remainder of this chapter is structured as follows: Section 4.1 covers existing classifications proposed in the literature. Afterwards, I discuss the properties grouped into three classes: Section 4.2 starts with the control flow constructs of loops and function calls. Next, Section 4.3 focuses on memory access patterns while Section 4.4 covers computational operations. Finally, I depict some of the properties and related optimizations by means of examples in Section 4.5.

## 4.1. Existing Classifications

The goal of identifying relevant properties of kernels can be seen as a classification problem. There already exists some promising literature that has worked in this area. However, most research so far has focused on approaches to structured parallel programming. They are primarily concerned with the task of “encoding” a problem into a parallel application. Instead, the described workflow is used to understand optimizations of already existing source code. Nevertheless, I discuss some approaches in the following that influenced the properties described in the later sections.

Asanovíc et al. from Lawrence Berkeley National Laboratory identify 13 dwarfs as follows: "A dwarf is an algorithmic method that captures a pattern of computation and communication." [20] In their report, the authors use the found dwarfs to classify applications that could benefit from parallelism. The goal is to design future hardware architectures and software frameworks to deliver good performance on these classes. However, the granularity of complete applications does not fit the needs of the workflow.

McCool et al. from Intel define algorithmic patterns to structure a parallel program [46]. The patterns describe how input data is used within operations to obtain output data. For example, the *map* pattern applies a function to every element of an input vector. Similarly, the *stencil* pattern defines an operation that uses data from neighbor cells to compute an output element. While closer the source code, the work is clearly tailored to developing parallel applications. The same applies to earlier work by Mattson et al. who define a pattern language to model and implement parallel programs [45].

More recently, Nugteren et al. describe algorithmic species as a way to classify affine loop nests [50]. The classification is based on information available from the polyhedral model. This has the advantage that the species can be determined automatically from source code. In a sense, this is very close to the needs of the workflow. However, the approach is limited by the restrictions from the polyhedral representation of affine loops. In particular, it is not possible to describe indexed memory accesses as discussed in Section 4.3.2.

## 4.2. Control Flow

Kernels are functions with two additional conditions as imposed by Definition 5: There may be at most a single outer loop and no function calls outside of loops. However, if the kernel contains an (outer) loop, there are no restrictions on its body. In particular, there may be nested loops and also function calls in one of the loop bodies. As such, the existence of these control flow constructs is an important property with respect to optimizations.

### 4.2.1. Loops

Loops account for a large fraction of time spent in HPC applications. Naturally, this makes optimizations very important to obtain good performance. This holds especially true for the loop body that is executed repeatedly. Important optimizations targeting the body are the topic of following sections. However, there are also compiler transformations related to the loop structure itself. Two particular examples are loop unrolling and LICM as discussed in Section 2.2.3.

### 4.2.2. Function Calls

Per Definition 5 of a kernel, function calls are only allowed inside of loops. However, this implies that any contained function call is executed repeatedly. As such, the

related overhead might become measurable and inlining is of particular importance. A detailed explanation of this optimization has been given in Section 2.2.2.

## 4.3. Memory Access Patterns

The second group of properties is related to memory access to load and store data. In many cases, the accesses exhibit a pattern that lends itself for optimization. For HPC applications, the most important classes are strided (Section 4.3.1) and indexed patterns (Section 4.3.2).

To illustrate the memory access patterns, I give small example codes for each property. All listings use a loop structure and their memory accesses depend on the loop iteration variable. This allows to keep the code short, but is not a prerequisite for the properties. Instead, the patterns are determined by the code's sequential semantics. Unrolling the loop or parts thereof still has the same memory accesses. Finally, a code may also exhibit patterns without loops, for example when loading consecutive memory from a given data structure.

### 4.3.1. Strided Memory Accesses

Many algorithms work on data that resides in memory at a constant stride of $n$ elements. This leads to a very regular pattern, which the compiler can exploit for its optimizations. The following paragraphs describe some particular forms of this access pattern and related transformations.

**unit-stride** If stride $n = 1$, the access pattern is referred to as *unit-stride*. A very basic example is the addition of two vectors as depicted in Listing 4.1. Here, the elements of the two input arrays `b` and `c` are accessed one after the other. The result of adding the two values is written back to array `a`.

One important optimization for unit-stride accesses is vectorization by means of SIMD instructions. In most cases this is a prerequisite for efficient vectorization of computational operations as discussed in Section 4.4.

```
1 for (int i = 0; i < N; i++) {
2   a[i] = b[i] + c[i];
3 }
```

Listing 4.1: Addition of two vectors as an example of a *unit-stride* memory access pattern.

**non-unit stride** If instead $n \neq 1$, the accessed memory is not contiguous and there are "gaps" in between. An example can be seen in Listing 4.2, which shows a computation on an Array of Structures (AoS). The code doubles the value of even-numbered elements, but skips over the odd-numbered offsets.

```
1 for (int i = 0; i < N; i++) {
2   data[2 * i] *= scale;
3 }
```

Listing 4.2: Computation on an Array of Structures as an example of *non-unit stride* memory access.

As for unit-stride it is sometimes possible to vectorize such memory accesses if supported by the hardware. This requires instructions to load and store non-contiguous values. These are usually referred to as *gather* for loads and *scatter* for stores.

**reuse** An orthogonal property is whether the kernel reuses values of strided memory accesses, for example across iterations of a loop. A concrete example is inspired by the *stencil* pattern described by McCool et al. [46]: Listing 4.3 shows a loop that accesses a sliding window of values from a vector.

```
1 for (int i = 1; i < N - 1; i++) {
2   new[i] = 0.25 * (old[i - 1] +
3       2 * old[i] + old[i + 1]);
4 }
```

Listing 4.3: Stencil-like operation on a sliding window of values from a vector.

An important optimization for this property is to keep reused values in registers. This can avoid repeated loads and reduce the data volume on caches and the memory bus. If multiple stores target the same memory location without an intervening read, all except the last are dead and can be removed. The related optimization is known as DSE as described in Section 2.2.1.

### 4.3.2. Indexed Memory Accesses

For some algorithms the location in memory is determined by another array of values. This is referred to as *indexed* access and represents the second important class of memory access patterns. A simple example is a matrix vector multiplication where the matrix is sparse. Listing 4.4 shows an implementation for the Compressed Row Storage (CRS) format. In the code, the required element of the vector `b` is determined by `index[j]`. Also note that the access of `M[j]` and `index[j]` is unit-stride.

Because it is less regular than strided patterns, the compiler has less opportunities to transform indexed accesses. If supported by the hardware, the compiler might again introduce gather loads or scatter stores. This is similar to the the possible transformation discussed for non-unit stride accesses. However, the instructions need to support arbitrary offsets instead of only a constant stride. If applicable this can potentially enable further transformations, as discussed in the following.

```
1 for (int i = 0; i < N; i++) {
2   a[i] = 0.0;
3   for (int j = ptr[i]; j < ptr[i + 1]; j++) {
4     a[i] += M[j] * b[index[j]];
5   }
6 }
```

Listing 4.4: Sparse matrix vector multiplication as an example for *indexed* loads. `M` is a matrix in CRS format while `a` and `b` are vectors.

## 4.4. Computational Operations

After loading data, the kernel can perform calculations on the values. For HPC applications, the first distinction is related to the used arithmetic: Real numbers are usually represented by floating point values in either single or double precision following the IEEE 754 standard [12]. However, some areas such as spectral methods require complex arithmetic with a real and imaginary part. In that case, each part is a real number and they are stored as two values next to each other in memory.

The following sections describe different kinds of operations and possible optimizations. It starts with the operations of addition, subtraction, and multiplication in Section 4.4.1. Afterwards, Section 4.4.2 looks at division and reciprocals. Finally, mathematical functions such as square roots are handled in Section 4.4.3.

### 4.4.1. Addition, Subtraction, Multiplication

These basic mathematical operations build the foundation of any HPC application. As such, current architectures support special instructions for increased performance. For one, there are single instructions working on multiple data (SIMD). Additionally, some vendors support instructions to fuse multiplications and additions (FMA).

To take advantage of these instructions, programmers usually rely on the compiler to transform their code. As such, a very important technique to get good performance for these operations is vectorization as described in Section 2.2.4. To guarantee soundness, the compiler is required to analyze data dependencies. This makes sure that the vectorized code is semantically equivalent to the sequential code. In particular, it must give the same results on all possible inputs.

After the compiler has proven soundness, it needs to find an efficient mapping to the available instructions. As a first step, all data must be available in vector registers. This requires analysis of the memory access pattern as discussed in Section 4.3. Afterwards, suitable operations must be grouped together and replaced by a vector instruction. This can be particularly tricky for operations on complex numbers as shown in the following.

The sum of two complex numbers is defined as:

$$(a + bi) + (c + di) = (a + c) + (b + d)i \tag{4.1}$$

In other words, the code needs to add the real and imaginary parts separately. As

the parts reside next to each other in memory, this is well suited for traditional vector instructions that operate on every element of a vector.

In contrast, a multiplication of two complex numbers involves more operations:

$$(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i \tag{4.2}$$

Efficient vectorization of complex multiplication is thus harder for two reasons:

1. The imaginary part of the product is composed of mixed terms. For example, $a$ is the real part of the first number, while $d$ is the imaginary part of the second. As a solution, it is possible to "shuffle" the elements of the second vector holding $c$ and $d$.
2. For the real part, the two sub-products have to be subtracted. In contrast, they need to be added for the imaginary part. This requires special vector instructions to perform the needed operations.

An example for complex arithmetic will be given in Section 4.5.1 when discussing the `zaxpy` kernel.

### 4.4.2. Division, Reciprocal

In addition to these basic operations, some algorithms also need to divide floating point values. If suitable, the generated code can make use of vector instructions as well. However, division of floating point values is much slower than addition and multiplication.

For example on Intel architectures, instructions for addition and multiplication have a latency of 3 to 5 cycles. Due to pipelining they can achieve a throughput of 1 or even 0.5 cycles per instruction [37, Appendix D.3]. In contrast, division has a latency of up to 35 cycles on 256-bit vectors and only became pipelined with the Skylake microarchitecture [37, Section 15.12]. This improved throughput to 5 cycles per instruction for single precision and 8 cycles for double precision [37, Appendix D.3].

Due to the much higher latency, some architectures provide ways to compute approximations with reduced precision. For example, a division can be replaced by a multiplication with its reciprocal:

$$a/b \approx a \cdot (1/b) \tag{4.3}$$

If needed, the approximation can be refined by a Newton-Raphson iteration [37, Sections 15.12.1 and 18.25.1]. This transformation is particularly efficient when the reciprocal can be reused for multiple computations.

### 4.4.3. Other Mathematical Functions

Finally, some HPC applications require other mathematical functions besides the operations described in the previous sections. These include square roots, trigonometric and hyperbolic functions as well as functions for exponentiation and logarithms. In the C and C++ programming languages, their definitions are provided by including

the header `math.h` or `cmath`. For Fortran programs, these functions are available as intrinsics defined by the standard [14].

Without additional options, the compiler generates the mentioned operations as function calls. Their implementation is usually provided by a system library, `libm` on POSIX systems such as GNU/Linux. However, the user can choose to omit strict requirements defined by the respective standards. For example, many compilers offer switches to neglect setting the `errno` variable on invalid arguments. This enables further optimizations because the compiler can use faster implementations.

If reduced precision is acceptable, some calls may be replaced by special hardware instructions as available. For example, Intel platforms have instructions to approximate the reciprocal of a square root. They can be used to approximate the square root function:

$$\sqrt{a} \approx a \cdot (1/\sqrt{a}) \tag{4.4}$$

Similar to the previous section, higher precision can be obtained by means of a Taylor expansion [37, Sections 15.12.3 and 18.25.3].

Another possibility is to employ vectorized versions contained in additional libraries. For example `libmvec` is available since glibc 2.22 released in August 2015 [1]. It offers vector implementations of the functionality provided by `libm` for x86_64 machines. Similarly SLEEF is another library that includes implementations for x86_64, AArch64, AArch32, and PowerPC64 [11, 55]. Additionally, some vendors also bundle specialized libraries with their commercial compiler.

## 4.5. Example Kernels

In the following, I present example kernels to discuss some properties in detail. They perform basic operations from linear algebra on vectors and matrices. This makes the code listings short and straight-forward to understand. Nevertheless, the kernels occur as building blocks in many numerical algorithms. As such, they are representative for a large set of HPC applications.

### 4.5.1. Dense Linear Algebra on Vectors

The first kernel can be seen in Listing 4.5. It works on two dense vectors $\vec{x}$ and $\vec{y}$ represented as arrays. The code consists of a `for` loop that iterates over all elements and computes $a \cdot \vec{x} + \vec{y}$. The result is assigned back to vector $\vec{y}$, overwriting the previous value. Because of its significance, this kernel is known as `axpy` operation. This refers to its name in the BLAS library [43].

```
1 for (int i = 0; i < N; i++) {
2   y[i] = a * x[i] + y[i];
3 }
```

Listing 4.5: `axpy` operation from BLAS level 1, performing $\vec{y} \leftarrow a \cdot \vec{x} + \vec{y}$.

The operation can be performed in different arithmetics as discussed in Section 4.4. Depending on the data type, the `axpy` operation is prefixed with a single letter. For this discussion, I focus on `daxpy` for double precision real values and `zaxpy` for double precision complex numbers. Apart from the arithmetic, the kernel possesses the following properties:

- From the class of control flow properties, the kernel contains a single loop. As described in Section 4.2.1, this makes loop optimizations a relevant target. In particular, loop unrolling is important as seen later on. There is no function call and therefore no possibility for inlining.
- The loop loads and stores the vector elements at unit-stride as `i` is incremented by one in every iteration. This makes it possible to load the data using vector instructions as seen in Section 4.3.1. Every value is used exactly once, so there is no reuse that the compiler could optimize for.
- Finally, the loop performs one multiplication and one addition. As there are no data dependencies between the iterations, the compiler may attempt to vectorize the code. Moreover, the operation matches the definition of FMA instructions which could be used if available.

#### Double Precision Real Values: `daxpy`

To observe the possible optimizations in practice, I present measurements of an implementation with arrays of `doubles`. As data size, I choose $N = 1024$ so that $2 \cdot N \cdot 8\,\text{Bytes} = 16,384\,\text{Bytes}$ fit into the L1D cache. To get measurable timings, the code performs the `axpy` operation 20,000,000 times. Further details on the environment have been described in Section 1.1.

Instead of runtimes, I present performance as the rate of floating point operations per second. Table 4.1 lists the average values and standard deviations of 10 repetitions. For the base version (first row), I passed the flags `-Ofast -march=native` for all compilers. The first parameter enables additional optimizations that are not valid for strict standard-compliance but may yield better performance. Targeting the `native` architecture makes the compiler optimize for that particular processor. Additionally, it is allowed to use all available instructions, most importantly SIMD instructions.

The numbers in the first row of Table 4.1 are in the range of 15 to 18 GFLOP/s. This is around one quarter of the peak performance of 67.2 GFLOP/s per core. Upon inspection, all three compilers use 256-bit vectors instead of AVX-512 instructions. The reason is that using longer registers incurs a penalty on the clock frequency. Hence, the compilers use the older AVX instructions by default in order to not regress.

| flags | `clang` | `gcc` | `icc` |
|---|---|---|---|
| `-Ofast -march=native` | $17.72 \pm 0.02$ | $15.63 \pm 0.12$ | $15.42 \pm 0.04$ |
| forcing AVX-512 | $35.79 \pm 0.57$ | $28.92 \pm 0.10$ | $27.93 \pm 0.55$ |

Table 4.1.: Measurements of the `daxpy` kernel. All numbers are in GFLOP/s.

However, for this simple kernel, the runtime is clearly bound by the vector instructions. As such, a larger vector width will improve performance, even if clock frequency degrades. For these cases, all three compilers offer advanced options to force using AVX-512 if applicable. They are named `-qopt-zmm-usage=high` for the Intel Compiler and `-mprefer-vector-width=512` for the other two. The measured performance is listed in the second row of Table 4.1. As expected, the executables are around twice as fast compared to the first row.

In both rows, the executable compiled by Clang performs 10 % to 20 % better than the other two. This is due to different unrolling decisions of the compilers: While `gcc` and `icc` do one vector computation, Clang performs up to eight before evaluating the loop header. It is possible to reach a similar decision for GCC by passing `-funroll-loops`. This improves the runtime to 16.54 GFLOP/s in the base version and to 32.04 GFLOP/s with AVX-512. For the Intel Compiler, similar options did not change the number of unrolled loop iterations.

#### Complex Arithmetic: `zaxpy`

For complex values, it is additionally important that the compiler efficiently maps the multiplication on vector instructions. These difficulties were already discussed in Section 4.4.1. For the measurements, I halve the number of elements to $N = 512$. This compensates the doubled storage requirement per element for the real and imaginary part. As before, the `axpy` operation is performed 20,000,000 times to get measurable timings.

Table 4.2 shows the average GFLOP/s and standard deviations of 10 repetitions. Similarly to Table 4.1, the first row represents the base version. In contrast to the `daxpy` results, the binary compiled by Clang is now the slowest. This is because of a regression in version 10.0.0 when generating code for the operations on complex values. For comparison, a binary compiled by the older release 9.0.1 runs at 18.78 GFLOP/s when using 256-bit vector registers.

In the second row I again force the compilers to use AVX-512 as explained above. Interestingly, this avoids the regression seen with Clang 10.0.0 which is now the fastest. As before, the runtime of the GCC version can be further reduced by enabling loop unrolling: With the option `-funroll-loops`, the performance is improved to 28.05 GFLOP/s with AVX-512.

However, it is possible to improve even further when moving closer to the machine level: Starting from the GCC version with AVX-512, I replaced the main loop by hand-written assembly code. For this I followed the ideas given at the end of Section 4.4.1: My assembly version applies shuffle instructions to swap the real and imaginary parts of `x[i]`. Additionally, it uses the instruction `vfmaddsub231pd` available on the hardware. This performs the additions and subtractions as required for

| flags | `clang` | `gcc` | `icc` |
|---|---|---|---|
| `-Ofast -march=native` | 11.71 ± 0.08 | 15.56 ± 0.15 | 17.94 ± 0.08 |
| forcing AVX-512 | 33.90 ± 0.13 | 26.48 ± 0.24 | 25.58 ± 0.45 |

Table 4.2.: Measurements of the `zaxpy` kernel. All numbers are in GFLOP/s.

the multiplication of two complex values. The full listing of the modifications can be found in Appendix A. This version achieves an average of 40.87 GFLOP/s which is around 20 % higher than the performance of Clang 10.0.0.

### 4.5.2. Sparse Matrix Vector Multiplication

As a second example, I investigate optimizations for a sparse matrix vector multiplication in CRS format. This was already presented in Listing 4.4 when discussing indexed loads in Section 4.3.2. The code is replicated in Listing 4.6 for easier reference. Note that the numbering starts with the second line to account for the function header. This makes the optimization report in Listing 4.7 easier to understand.

Similar to the `axpy` operation, the code contains an outer loop that iterates over all rows of the matrix. Additionally, there is an inner loop for every non-zero entry in row `i`. Array `a` is accessed at unit-stride depending on the outer loop variable `i`. The same holds for `ptr` where also the next element `i + 1` is needed. This constitutes a reuse as mentioned at the end of Section 4.3.1.

In the inner loop, the accesses to arrays `M` and `index` depend on `j`. It starts at an offset defined by `ptr[i]` until `ptr[i + 1]`. Still, it is unit-stride because `j` is incremented by one in every iteration. The value of `index[j]` determines the element position in array `b`. Per Section 4.3.2 this defines an indexed memory load. With regard to the computation, the inner loop performs one addition and one multiplication. As above for the `axpy` kernel, these operations are suitable for vectorization and FMA instructions.

To inspect the code generation with the different compilers, I put the kernel into a function. The required arrays `a`, `b`, `ptr`, `index`, and `M` are passed as pointers. Moreover, the function takes the number of rows `N` as a parameter. As in the previous example, I specify the flags `-Ofast -march=native` for all compilers described in Section 1.1.

The emitted assembly code shows that no compiler vectorized the kernel. The reason becomes clear when requesting optimization reports. For example, the Intel Compiler outputs an additional text file containing the lines in Listing 4.7. The remarks show that the compiler assumes dependences between the arrays. This is because the C programming language allows passing aliasing pointers which share

```
2 for (int i = 0; i < N; i++) {
3   a[i] = 0.0;
4   for (int j = ptr[i]; j < ptr[i + 1]; j++) {
5     a[i] += M[j] * b[index[j]];
6   }
7 }
```

Listing 4.6: Sparse matrix vector multiplication, replicated from Listing 4.4. Note that compared to the previous listing, the line numbers are shifted by one. This is to accommodate the function header in the actual source file `spmv.c`.

```
LOOP BEGIN at spmv.c(2,3)
   [...]
   remark #15344: loop was not vectorized: vector
      dependence prevents vectorization
   remark #15346: vector dependence: assumed FLOW
      dependence between a[i] (3:5) and M[j] (5:7)
   remark #15346: vector dependence: assumed ANTI
      dependence between M[j] (5:7) and a[i] (3:5)

   LOOP BEGIN at spmv.c(4,5)
      remark #15344: loop was not vectorized: vector
         dependence prevents vectorization
      remark #15346: vector dependence: assumed FLOW
         dependence between a[i] (5:7) and M[j] (5:7)
      remark #15346: vector dependence: assumed ANTI
         dependence between M[j] (5:7) and a[i] (5:7)
      [...]
   LOOP END
   [...]
LOOP END
```

Listing 4.7: Excerpt from the optimization report of the Intel Compiler for the sparse matrix vector multiplication kernel function.

memory. In such case, vectorized code would deliver wrong results.

The same problem also existed for the `axpy` kernel in the previous Section 4.5.1. However, that situation was easier to deal with for the compilers: Without indexed loads, the loop obviously accesses the elements 0 to $N-1$ of the two arrays `x` and `y`. To prove safeness of vectorization, it suffices to make sure that none of these elements overlap. This can be handled efficiently at runtime by comparing computed pointer values. Additionally, there were only two pointers for `axpy` whereas `spmv` takes five. To allow vectorization, the compiler would have to prove no overlap between the accesses to `a` and the four other pointers. That is because writing to the result vector could potentially modify the other data.

To remedy this situation, C99 [13] introduced the `restrict` qualifier as mentioned in Section 3.4. For a second experiment, I added the keyword to all five pointer arguments. In this code, it assures the compiler that none of the pointed memory overlaps. This enables vectorization which is now performed by all three compilers.

However, the tested compilers show different vectorization strategies: Clang and GCC use an available hardware instruction to perform the gather load of four elements. Similar to the previous section, Clang also unrolls the vectorized loop to process a total of 32 elements in one iteration. GCC only performs one vector computation on four elements before rechecking the inner loop header. In contrast, the Intel Compiler uses instructions to perform the gather in software: The generated code loads two elements in a scalar fashion and puts them at the right place of a

(smaller) vector register. This avoids the hardware instruction, which it would have been allowed to use.

The performance effect of the described differences depends on the input data. For example, consider a very sparse matrix with only few non-zero elements per row. It can happen that their number per row is smaller than that processed in one unrolled iteration. In that case, the generated code falls back to other code paths which might be slower.

Moreover, the effect of using available instructions for gather loads in hardware are not clear. Intel's "Optimization Reference Manual" devotes an entire section to discuss the reasons for and against [37, Section 15.16.4]. One of the pages also contains the following sentence:

> In performance critical applications it is advisable to evaluate both options [hardware gather and implementation in software] before choosing one.
> ([37, p. 15-67])

Such evaluation is not performed in this thesis.

# 5. Case Studies

After presenting the workflow in Chapter 3, the properties described in Chapter 4 are intended to facilitate the analysis step. To show the applicability of the two approaches, the following chapter is dedicated to the evaluation of the workflow. To that end, I present case studies for important scenarios that can be analyzed with the workflow. These stem from the possible reasons that could lead to a difference as described in Section 3.1.1:

- For differences between compilers, I study the Block Tri-diagonal (BT) solver from the NAS Parallel Benchmarks in Section 5.1. It is a pseudo-application with focus on dense linear algebra of equation systems for Computational Fluid Dynamics (CFD) [23]. Additionally, I discuss LULESH in Section 5.2, which performs computations on an unstructured mesh [5].

  The investigation of the two codes covers the main aspect from the motivation presented in Chapter 1: The case studies demonstrate how to analyze runtime differences and relate them to compiler optimizations. Furthermore, they also show that the workflow applies to both C/C++ and Fortran, which are the predominant languages for HPC codes.

- miniMD presented in Section 5.3 is an application from the field of Molecular Dynamics (MD) [7]. Its analysis shows two more possible starting points for the workflow: Section 5.3.1 investigates a difference between two versions of the GNU Compiler Collection. Additionally, I discuss the impact of different flags for the Clang compiler in Section 5.3.2.

- Finally, I investigate one difference for miniQMC in Section 5.4, a Quantum Monte Carlo (QMC) simulation [10]. From the applications presented in this thesis, it is the largest with around 20,000 lines of code. The analysis demonstrates that the workflow can successfully explain the effect of compiler optimizations in a simulation of that size. For that, it decomposes the difference which allows to relate it to the generated code of one function.

These codes represent different application domains important in the context of HPC. This demonstrates that the workflow is generally applicable and not specific to one domain. For each of the codes, I build executables with the three compilers mentioned in Section 1.1. The only exception is the BT solver because the NAS Parallel Benchmarks are written in Fortran. Hence, it is not possible to build the code with Clang, which is a compiler for C and C++ code only.

For measurements, I execute on `login18-t` as described in Section 1.1. Being an interactive system, the computing resources are shared between all users. It should therefore be avoided to block the system with long-running processes. For that reason,

| | Function | Properties |
|---|---|---|
| NAS BT | `y_solve_cell` (third loop) | loop; strided memory accesses; add., sub., mult., reciprocal |
| | `matmul_sub` | strided memory accesses; sub., mult. |
| | `matvec_sub` | strided memory accesses; sub., mult. |
| | `binvcrhs` | strided memory accesses; sub., mult., reciprocal |
| LULESH | `CalcElemShapeFunctionDerivatives` | strided memory accesses; add., sub., mult. |
| | `EvalEOSForElems` ↪ `CalcEnergyForElems` ↪ `CalcPressureForElems` | (nested) loops, function calls; strided & indexed accesses; add., sub., mult., div., `sqrt` |
| miniMD | `ForceLJ::compute_halfneigh` | loop; strided & indexed accesses; add., sub., mult., reciprocal |
| miniQMC | `DTD_BConds::computeDistances` | loops, function calls (`floor`); strided & indexed accesses; add., sub., mult., `sqrt` |

Table 5.1.: Overview of the analyzed functions by benchmark and properties of the decomposed kernels. Computational operations are abbreviated if unambiguous.

I aim for execution times of less than 5 minutes. At the same time, sampling must run long enough for meaningful profiles.

Furthermore, I do not repeat the profiling runs which enables quick iteration during the steps of the workflow. As a result, it is not possible to assess run-to-run deviations due to other load on the system. However, the measurements presented in Section 4.5.1 showed performance variations of less than 2 %. Thus, reproducible results can be expected when making sure that the system is idle and no other users disturb the measurements.

To obtain the total runtime, I rely on the integrated timing routines of the benchmarks. This has the advantage to exclude initialization methods, which are unrelated to problem solution. By comparing the runtimes of two binaries, it is possible to identify significant runtime differences between them. This represents the starting point for the workflow as discussed in Section 3.1.2. During analysis, I focus on significant differences with a relative deviation of more than 5 %.

Table 5.1 gives an overview of the analyzed functions. The third column lists the properties of the decomposed kernels. More details are presented in the corresponding sections. This also includes discussions about the reasons of the differences and possible improvements. Furthermore, I highlight important points and strengths of the workflow as well as possible weaknesses.

## 5.1. NAS Parallel Benchmarks: Block Tri-diagonal Solver

The NAS Parallel Benchmarks are developed by NASA's Advanced Supercomputing Division. The initial version was described in 1991 as "pencil and paper" specifications [23, 22]. This means the benchmarks were specified algorithmically and vendors created their own implementations. Since 1995, NAS distributes reference implementations written in Fortran. Additionally, the specification defined sets of input parameters, referred to as "classes". Over time, the group added new classes to account for the increased computing speed.

For my experiments, I use the latest version 3.4.1 available for download [8]. I compile the MPI version with Intel MPI in version 2018.4.274, but only execute sequentially with one rank. As mentioned above, I focus on the Block Tri-diagonal solver, abbreviated "BT". This pseudo-application solves "multiple, independent systems of nondiagonally-dominant, block tridiagonal equations with a $(5 \times 5)$ block size" [23, 22]. The performed computations model important parts of CFD codes.

The usage of Fortran restricts the experiments to the GNU Compiler Collection (`gcc`) and the Intel Compiler (`icc`). For these, I initially pass the compiler flags `-Ofast -march=native`. As input parameters, I use the values from class B of the benchmark. This specifies a size of $102^3$ with 200 iterations and a time increment of $\Delta\tau = 0.0003$ [22].

With serial execution, the chosen configuration provides runtimes that are neither too short for reliable measurements nor too long for repeated runs: The benchmark reports timings of 163.7 s for `gcc` and 141.8 s for `icc`. With a relative difference of around 13 %, this constitutes a significant runtime difference. In the following, I analyze the causes by applying the workflow as described in Chapter 3.

### 5.1.1. Initial Profile

As a first step, I profile the compiled executables with the help of `perf`. Doing so increases the runtimes to 167.6 s and 145.6 s, respectively. This represents an overhead of around 4 s or 2.5 % for both binaries. Table 5.2 shows the obtained data for the functions with the highest sample counts. Based on the frequency of 4,000 Hz, this can be translated into execution time listed in the third column. Note that I strip the final underscore of the function names that the compilers add for compatibility reasons.

As both profiles show the same functions, they are directly comparable. The sample counts reveal that the `gcc` versions of `y_solve_cell` and `binvcrhs` are much slower. Additionally, the functions at positions five to seven have a different order. For easier matching, I mark their function names in Table 5.2 with symbols. This makes it clear that `matmul_sub` is slower with `gcc`, but `matvec_sub` is faster than the version compiled by `icc`. The difference for `compute_rhs` is less than 4 % and therefore not studied.

During analysis, it turns out that three of the four differences are related to vectorization: `y_solve_cell`, `matmul_sub`, and `matvec_sub`. The findings leading to this conclusion and possible improvements are presented in the following section. Afterwards, I discuss possible causes for the difference in `binvcrhs`.

| Function | Samples | Time |
|---|---|---|
| `y_solve_cell` | 151946 | 38.0 s |
| `binvcrhs` | 131039 | 32.8 s |
| `z_solve_cell` | 84188 | 21.0 s |
| `x_solve_cell` | 71421 | 17.9 s |
| `matmul_sub` * | 69200 | 17.3 s |
| `compute_rhs` † | 66385 | 16.6 s |
| `matvec_sub` § | 37234 | 9.3 s |
| `x_backsubstitute` | 21119 | 5.3 s |
| `z_backsubstitute` | 17193 | 4.3 s |
| `y_backsubstitute` | 17102 | 4.3 s |

(a) GNU Compiler Collection

| Function | Samples | Time |
|---|---|---|
| `y_solve_cell` | 118111 | 29.5 s |
| `binvcrhs` | 88913 | 22.2 s |
| `z_solve_cell` | 85244 | 21.3 s |
| `x_solve_cell` | 71400 | 17.9 s |
| `compute_rhs` † | 68976 | 17.2 s |
| `matvec_sub` § | 66323 | 16.6 s |
| `matmul_sub` * | 25781 | 6.4 s |
| `x_backsubstitute` | 19899 | 5.0 s |
| `z_backsubstitute` | 17130 | 4.3 s |
| `y_backsubstitute` | 16830 | 4.2 s |

(b) Intel Compiler

Table 5.2.: Original profiles of the BT solver built with `gcc` and `icc`. Functions with less than 10,000 samples are not shown. Symbols next to function names mark corresponding entries in the two tables.

### 5.1.2. Vectorization in Three Subroutines

The function `y_solve_cell` contains multiple loops and function calls. Investigation of the generated code shows that none of the calls were inlined. This is because the routines are defined in other files and these are compiled separately. Therefore, I use the technique described in Sections 3.3.2 to outline the loops. To that end, I move three nested loops into separate functions and profile the result with `perf`. The sample counts show that around 80 % of the runtime is spent in the third loop. Moreover, this last loop also accounts for all of the original difference between `gcc` and `icc` in this function.

The body of the third loop operates entirely on double precision floating point values. It contains strided memory accesses and performs additions, subtractions, and multiplications. As described in Section 4.4.1, an important optimization of such code is vectorization. Therefore, I look at the generated instructions with suffixes `sd` and `pd`. The former operate on scalar values while latter are SIMD instructions working on "packed" data. Table 5.3 shows the found instruction mnemonics sorted by their count[1].

It can be seen that the only vector instruction for `gcc` is `vxorpd`. In contrast, `icc` also generates SIMD instructions for the computational operations. When invoking `gcc` with the option `-fopt-info-vec-missed`, the compiler confirms it "couldn't vectorize [the] loop". To solve this problem, I add the directive `!$omp simd` to the original version of `y_solve_cell`. This informs the compiler that vectorization is safe and can be performed without further analysis. After recompilation with the flag `-fopenmp-simd`, the execution time for `gcc` decreases to 155.8 s without `perf`, while `icc` is unchanged.

[1] generated using the following sequence of pipes:
```
objdump -d --no-show-raw-insn bt.gnu | \
    awk -F"\n" -v RS="\n\n" '$1 ~ /y_solve_cell_loop3_/' | tail -n +2 | \
    sed -E 's/.*:\W+(\w+).*/\1/' | grep "[sp]d$" | sort | uniq -c | sort -rn
```

| Instruction | Count |
|---|---|
| vmovsd | 87 |
| vmulsd | 72 |
| vxorpd | 25 |
| vfmadd231sd | 25 |
| vfmsub231sd | 20 |
| vaddsd | 11 |
| vfnmadd231sd | 5 |
| vfmsub132sd | 5 |
| vfmadd132sd | 5 |

(a) GNU Compiler Collection

| Instruction | Count |
|---|---|
| vmovsd | 298 |
| vmovhpd | 200 |
| vmulsd | 82 |
| vmulpd | 60 |
| vfnmsub231pd | 25 |
| vfmsub231pd | 25 |
| vfnmsub231sd | 20 |
| vfmsub231sd | 20 |

| Instruction | Count |
|---|---|
| vmovupd | 18 |
| vfmadd213pd | 15 |
| vsubsd | 10 |
| vfmadd213sd | 10 |
| vaddsd | 7 |
| vxorpd | 5 |
| vfmsub213sd | 5 |
| vaddpd | 5 |

(b) Intel Compiler

Table 5.3.: Count of instruction mnemonics with suffixes `sd` and `pd` for the third outlined loop of `y_solve_cell`[1].

Analysis of the generated code for `matmul_sub` and `matvec_sub` reveals similar vectorization issues with the GNU Compiler Collection. For `matvec_sub`, this actually gives `gcc` an advantage over the Intel Compiler. However, I focus on worse performance of `gcc` and do not investigate improvements for the binary produced by the Intel Compiler. I instead analyze `matmul_sub` in the following, which is faster with `icc` by a factor of 2.7.

Inspection of its source code shows that the function contains a fully unrolled multiplication of two $5 \times 5$ matrices. While `icc` generates vectorized code, the GNU Compiler Collection emits scalar instructions. This is because the compiler has to analyze the source code before vectorizing it. In this case, `gcc` cannot prove soundness and therefore refrains from SLP vectorization. For that reason, I rewrite the fully unrolled code as three nested loops. This allows the loop vectorizer to transform the code and reduces the runtime of the GNU Compiler Collection to 148.8 s.

However, this modification slows down the binary compiled by the Intel Compiler. The optimization report reveals that the compiler changes the order of the nested loops. This leads to a less optimal instruction stream after vectorization when compared to the original version that was fully unrolled. As a result, the total execution time increases from 141.8 s to 147.4 s.

### 5.1.3. Caching across Function Boundaries

The difference for `binvcrhs` is more difficult to explain. This is partly because the cause of the difference cannot be isolated in the generated code of a single routine: As an experiment, I compile two mixed versions with assembly code generated by the Intel Compiler. For the first, I only replace the function `binvcrhs` with assembly code from `icc`. For the second version, I use the generated assembly code for the entire source file `solve_subs.f`. This additionally includes the functions `matmul_sub` and `matvec_sub` already discussed in the previous section.

Profiling results are presented in the second and third column of Table 5.4. They show that the usage of assembly generated by `icc` reduces the execution time of

| Version | Samples | Time | Cache Misses |
|---|---|---|---|
| compiled with `gcc` | 131039 | 32.8 s | $4.35 \cdot 10^8$ |
| replacing `binvcrhs` | 107503 | 26.9 s | $3.96 \cdot 10^8$ |
| replacing `solve_subs.f` | 97128 | 24.3 s | $3.94 \cdot 10^8$ |
| compiled with `icc` | 88913 | 22.2 s | $3.64 \cdot 10^8$ |

Table 5.4.: Profiles of `binvcrhs` with mixed versions of the BT solver.

`binvcrhs`. Another slight improvement can be seen when also replacing the other functions. However, the second mixed version is still around 9 % slower compared to the runtime when compiling the complete benchmark with `icc`.

One possible explanation is the influence of hardware caches. They have a global state, which is naturally maintained across function boundaries. In this case, caching could lead to better performance when accessing data in `binvcrhs`. The amount of cached data depends on the instructions executed before entering the function. These could be different for the executables built with `gcc` and `icc`. In fact, analysis of the call sites show that this includes the functions `matmul_sub` and `matvec_sub`.

To investigate this hypothesis, I use `perf` to measure the `cache-misses` event. The last column of Table 5.4 shows the number of observed cache misses. It can be seen that the results qualitatively match the measured execution time. This provides evidence for a possible correlation between cache misses and execution time. Further analysis would be needed to fully explain the difference.

### 5.1.4. Results & Discussion

For this benchmark, the workflow identified multiple significant differences in the initial profile. The lower performance of `y_solve_cell` and `matmul_sub` could be explained with vectorization issues. For these routines, the properties described in Chapter 4 correctly identified the importance of this optimization. Afterwards, it was possible to enable vectorization by modifications in the source code: For the loop in `y_solve_cell`, I added the directive `!$omp simd`. Additionally, `gcc` vectorizes the matrix multiplication when writing as loops. In total, both modifications lead to a relative improvement of 9 %.

However, the example of `matmul_sub` also shows the difficulty with portable performance improvements: The formulation as loops is much shorter and more readable than the fully unrolled loop. This makes it easier for compilers to apply transformations, which results in lower execution times for `gcc`. On the other hand, the Intel Compiler starts reordering the loops, leading to worse performance compared to the original version. As of now, I am not aware of a portable solution with good performance for both compilers.

The investigation of `binvcrhs` hints to a conceptual limitation of the workflow: It builds on the assumption that differences between two executions are isolated in functions. This approach does not work for global influences such as caches. Similar problems could be encountered for global code layout changes. As one example, BOLT attempts to move hot functions close together as mentioned in Section 2.4.

## 5.2. LULESH

The second benchmark is LULESH, short for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics [5]. It is an implementation of the Shock Hydrodynamics Challenge Problem defined in 2011 [3]. LULESH models a hydrodynamics application with calculations on an unstructured mesh. It is written in C/C++ and has been studied extensively in literature [33, 21, 44, 15].

The current tagged release is version 2.0.3 from 2017, but I use the latest code version from GitHub [6]. For compilation, I pass the options `-Ofast -march=native`. Additionally, I explicitly disable the parallelization using MPI by defining `USE_MPI=0`. Without enabling OpenMP, this results in a serial version of the application.

In the default configuration with $30^3$ mesh points, an execution of the benchmark on a single core finishes in around 15 s. This could be too short for measurements, in particular for reliable profiling of the time spent in functions. For that reason, I increase the simulated mesh size to $45^3$ by passing the argument `-s 45`. This also used to be the default in the first version of the benchmark, but was changed for version 2.0 [38]. As expected, this leads to increased execution times and I measure 93.4 s for `clang` and 78.6 s for `icc`.

### 5.2.1. Inlining

When profiling with `perf`, the execution time increases to 95.1 s and 80.4 s respectively. This represents an overhead of less than 2.5 % and the workflow is expected to deliver consistent results. The obtained sample counts per function are shown in Table 5.5, excluding functions with less than 10,000 samples. As before, the third column of the two tables lists the corresponding execution times, based on a sampling frequency of 4,000 Hz.

The profile summary shows that Clang and the Intel Compiler made different inlining decisions. For Clang, the biggest part of the runtime is aggregated in the entry `LagrangeLeapFrog`. In contrast, the samples are spread over multiple functions for the Intel Compiler. However, the profile for Clang in Table 5.5a also shows the function `CalcElemShapeFunctionDerivatives`, which is not visible for the Intel Compiler.

Closer investigation of the source code shows that this function is called from two locations: `CalcKinematicsForElems` and `IntegrateStressForElems`. Both are visible in Table 5.5b for the profile of the executable built by the Intel Compiler. However, only `CalcKinematicsForElems` is shown in Table 5.5a for Clang. The function `IntegrateStressForElems` has instead been inlined into `LagrangeLeapFrog` mentioned above.

Static analysis of the code in `CalcElemShapeFunctionDerivatives` reveals many additions, subtractions, and multiplications. These belong to the first group of computational operations described in Section 4.4.1. This means it is likely that inlining the function enables further optimizations. For that reason, I add the attribute `always_inline` to the function declaration of `CalcElemShapeFunctionDerivatives`. After recompilation and execution with `perf`, the function is not visible anymore in the profile. Instead, the sample count of `CalcKinematicsForElems` increases slightly

| Function | Samples | Time |
|---|---|---|
| `LagrangeLeapFrog` | 267471 | 66.9 s |
| `CalcKinematicsForElems` | 34320 | 8.6 s |
| `CalcElemShapeFunctionDerivatives` | 22413 | 5.1 s |
| `__cbrt` | 11239 | 2.8 s |

(a) Clang Compiler

| Function | Samples | Time |
|---|---|---|
| `CalcHourglassControlForElems` | 60047 | 15.0 s |
| `ApplyMaterialPropertiesForElems` | 57364 | 14.3 s |
| `CalcFBHourglassForceForElems` | 46097 | 11.5 s |
| `CalcQForElems` | 37435 | 9.4 s |
| `CalcKinematicsForElems` | 37214 | 9.3 s |
| `IntegrateStressForElems` | 31604 | 7.9 s |
| `main` | 13483 | 3.4 s |

(b) Intel Compiler

Table 5.5.: Original profiles of LULESH executables built with `clang` and `icc`. Functions with less than 10,000 samples are not shown.

to 42772 samples, or 10.7 s. The counts for `LagrangeLeapFrog` and `__cbrt` remain unchanged compared to Table 5.5a within expected variations.

In a measurement without `perf`, the total execution time decreases to 88.7 s. Compared to the original runtime of 93.4 s, this represents an improvement of around 5 %. As explained in Section 3.2.2, mimicking the decisions of the other compiler is a possible outcome of the workflow. However, it provides little insight into the nature of the performed optimization after inlining. For more detailed information, it is possible to analyze the generated code of the calling functions. This is not performed in this thesis for reasons of time and space.

To continue applying the workflow, it is necessary to achieve the same inlining in both executables. For that, I add an attribute `noinline` to each of the functions visible in Table 5.5b but not in Table 5.5a. This increases the runtime of the executable built by Clang to 91.0 s without a profiler. At the same time, the total time of `icc` stays constant within expected deviations and rounding.

Runs with `perf` result in the profiles depicted in Table 5.6. The functions are sorted by sample count, which is not shown due to space constraints. It can be seen that the first seven functions are common between the two profiles. As such, the profiles are comparable according to Definition 4 in Section 3.2.1. However, four functions are at different positions, which is a first indication of possible runtime differences.

When looking at the sample counts, the biggest difference is found for the function `ApplyMaterialPropertiesForElems`: In the `clang` profile, it has 82,695 samples corresponding to around 20.7 s. In contrast, the executable built by the Intel Compiler was only sampled 57,089 times in that function. This corresponds to a runtime of 14.3 s, which is around 30 % less than `clang`. For that reason, the workflow proceeds with the analysis of this significant difference.

| clang | icc |
|---|---|
| ApplyMaterialPropertiesForElems | CalcHourglassControlForElems |
| CalcHourglassControlForElems | ApplyMaterialPropertiesForElems |
| CalcFBHourglassForceForElems | |
| CalcKinematicsForElems | CalcQForElems |
| CalcQForElems | CalcKinematicsForElems |
| IntegrateStressForElems | |
| main | |

Table 5.6.: Profile of LULESH executables after adapting function inlining. Only the first seven functions are shown, sorted by sample count, which is omitted due to space constraints.

### 5.2.2. Loop Fusion for Cache Locality

After identifying a difference, the first step is to decompose the function. In this case, `ApplyMaterialPropertiesForElems` calls the function `EvalEOSForElems`, among others. Further profiles confirm that the cause for the difference is located below that function. Therefore, I remove all other attributes `noinline` and only prevent inlining of `EvalEOSForElems`. This avoids effects from other parts of the code that could influence further analysis steps. However, I keep the `always_inline` for `CalcElemShapeFunctionDerivatives`. Together, these changes restore the runtime of `clang` without the profiler to 87.7 s. With the overhead of `perf`, I measure 90.6 s for `clang` and 80.9 s for `icc`.

To analyze the difference further, I first look at the code of `EvalEOSForElems`: It contains two outer loops, further nested loops, and several function calls. Additional profiling runs suggest that the biggest part of the runtime is spent in the function `CalcEnergyForElems`. However, when preventing inlining of this function, the total execution time increases significantly: For `clang`, I measure 94.0 s instead of 90.6 s, and 88.7 s instead of 80.9 s for the binary built with the Intel Compiler. It can be noticed that `icc` experiences a larger slowdown than `clang`. This leads to the conclusion that inlining is required to observe the optimization that causes the difference. Unfortunately, this implies further decomposition would produce inconsistent results and must not be performed.

To understand the cause of the difference, it is therefore necessary to analyze `EvalEOSForElems` together with all called functions. In addition to the mentioned `CalcEnergyForElems`, this includes the transitively called `CalcPressureForElems`. All three functions consist of several loops and perform many additions, subtractions, and multiplications. Furthermore, they contain some reciprocals, divisions, and square roots.

When taking the previous results into account, it is likely that the Intel Compiler performs an optimization across function boundaries. This hypothesis is confirmed by a detailed analysis of the generated code: It becomes clear that the compiler fuses some of the loops after `CalcPressureForElems` has been inlined into `CalcEnergyForElems`. As a result, cache locality is improved which leads to bet-

ter performance.

This transformation can also be performed in the source code directly: In the original version, `CalcEnergyForElems` has five loops and `CalcPressureForElems` another two. However, all of these loops have the same iteration space and their iterations are independent from each other. This allows to rewrite the computation with one outer loop in `CalcEnergyForElems`. To retain the original structure, the loop body calls `CalcPressureForElems` with the current iteration index as an argument.

After recompiling the code, it is possible to measure the expected improvements: Without `perf`, the executable compiled Clang now finishes in 80.4 s. This is another 9 % improvement compared to the runtime of 88.7 s from the previous section. In total, the runtime decreases by almost 14 % from the original measurement of 93.4 s. At the same time, the timings for the Intel Compiler only improve slightly to 77.9 s when compared to the original runtime of 78.6 s. This is another confirmation that the compiler already performed the optimization in the original version.

The reasoning described above also explains the increased runtime when not inlining `CalcEnergyForElems`: When analyzing the function separately, the compiler has to assume aliasing of pointer arguments. As a result, it cannot prove the soundness of fusing the loops and therefore restrains from the optimization. The same problem arises when preventing the compiler from inlining `CalcPressureForElems`: In that case, the compiler cannot analyze the code inside the called function. As result, it is not allowed to move the code because doing so could change the computed results.

### 5.2.3. Results & Discussion

By applying the workflow, it was possible to analyze the causes for two significant differences. Both were related to inlining of function calls, which enabled other optimizations. For the second difference, the important transformation was loop fusion as applied by the Intel Compiler. To make Clang benefit from this knowledge, I added one attribute `always_inline` and performed the loop fusion portably in the source code. This reduces the runtime for the studied input parameters by almost 14 %, from 93.4 s to 80.4 s.

The case study shows that the workflow can deal with inlining to analyze applications like LULESH. This is important because LULESH follows the “programming style typical in scientific C or C++ based applications” [5]. To that end, the workflow provides a structured approach to make profiles comparable. This was needed in this case because the two compilers initially made different inlining decisions. Afterwards, it is possible to compare the runtimes of functions and locate differences. To understand their causes, the workflow finally describes methods to decompose the differences.

The analysis also shows the difficulty related to optimizations across function boundaries. One reason is that the properties devised in Chapter 4 are focused on kernels according to Definition 5. This means they provide little help for optimizations like loop fusion. However, I was still able to understand the cause by manual inspection of the assembly code. This was only feasible after applying the workflow, which narrowed down the difference to the function `EvalEOSForElems`.

## 5.3. miniMD

miniMD is a Molecular Dynamics (MD) simulation modeled after the LAMMPS software package [7, 53]. It is part of the Mantevo project which aims to provide mini-applications for performance studies [35]. To that end, miniMD is considerably smaller at around 5,000 lines of C++ compared to more than 200,000 lines of LAMMPS [7]. The configuration used for this thesis is based on the sample input file distributed with the benchmark. It simulates a Lennard-Jones system of atoms without other interaction forces.

As for the BT solver from the NAS Parallel Benchmarks in Section 5.1, I compile the source code with Intel MPI but execute with only one rank. Compiler optimizations are enabled with the options `-Ofast -march=native`. Furthermore, I define `-DUSE_SIMD` for the preprocessor and pass `-fopenmp` to take `omp simd` directives into account. However, I set environment variables to allow only one thread, which makes miniMD execute routines optimized for sequential configurations.

To reach a runtime of at least 30 s, I increase the size by passing `-s 64`. This makes miniMD simulate a total of $4 \cdot 64^3 = 1048576$ atoms for 100 timesteps. Using these settings, the runtimes with `clang`, `gcc`, and `icc` range from 30 s to 36 s. However, more interesting differences are observed with different executables from the same compiler. This includes two versions of `gcc` and sets of flags for `clang`. I analyze their causes in the following to demonstrate that the workflow can handle them without modifications.

### 5.3.1. Two Versions of the GNU Compiler Collection

During early experiments, I observed a significant difference with the GNU Compiler Collection. When built with the older version 9.3.0 of the compiler, the simulation takes a total of 43.4 s. This is significantly slower than the executables by `clang` (30.4 s) and `icc` (33.1 s). The newer `gcc` in version 10.1.0 also finishes in around 35.5 s, which is much closer to the other runtimes. To investigate this improvement of around 18 %, I apply the workflow for the executables built by different versions of the GNU Compiler Collection.

Profiling with `perf` reveals that the difference is primarily caused by the function `ForceLJ::compute_halfneigh`. Static code analysis shows that the method contains nested loops, but no function calls. The loop bodies compute on floating point values and perform additions, subtractions, multiplications, and divisions. Moreover, the inner loop is marked with `#pragma omp simd`, which means that vectorization is a likely optimization.

To explain the causes for the difference, I look at the instruction mnemonics of the generated machine code. For that, I use the same methodology already presented in Section 5.1.2 for the BT solver. Comparison of the two binaries shows that `gcc` 10.1.0 generates SIMD instructions, but version 9.3.0 does not. That is despite the fact that the inner loop is annotated with a compiler directive. As such, I conclude that the reduced execution time is due to improvements in the vectorizer for this newer version of `gcc`.

### 5.3.2. Flags for the Clang Compiler

Furthermore, the binary compiled by `clang` exhibits an enormous improvement when adding the flag `-march=native`. To investigate this difference in isolation, I reduce the optimization level to `-O2`. However, I keep the options `-DUSE_SIMD -fopenmp` to enable `omp simd` directives. For this baseline version, I measure a runtime of 84.5 s, while compilation with the added `-march=native` reduces the execution time to 32.0 s. These results differ significantly by a factor of 2.64 and the cause can be analyzed with the workflow.

A profiling run with `perf` points to the same function `compute_halfneigh` as in the previous section. For this reason, I again extract the mnemonics of the generated instructions. Comparison shows that the computation is vectorized with SIMD instructions in both executables. However, the baseline version only uses instructions defined by SSE2. In contrast, `clang` also emits AVX instructions when passing the flag `-march=native`.

This is because SSE2 is included in the first specification of `x86_64`. As a result, the compiler can assume support for all processors based on this architecture. In contrast, AVX was proposed in 2008 and first products were released in 2011. For that reason, `clang` does not use these instructions unless requested, for example when invoking with `-march`. This avoids generating code by default that cannot run on older processors.

It has to be noted that other compilers generate faster executables with only SSE2: With the same baseline flags (`-O2 -DUSE_SIMD -fopenmp`), I measure 46.7 s with `gcc` and 50.3 s with the Intel Compiler. Analysis of the generated instruction mnemonics reveals that `gcc` does not vectorize the code with `-O2`, which results in the best execution time. It is possible to disable vectorization for `clang` with the flags `-fno-vectorize -fno-slp-vectorize`. However, this leads to even worse performance and a runtime of 94.7 s. As these difference are limited to suboptimal compilation flags, I did not further investigate their reasons.

### 5.3.3. Results & Discussion

In this section, I demonstrated the applicability of the workflow for executables from the same compiler. First, I investigated a difference with a newer release of the GNU Compiler Collection. Once more, the kernel properties identified vectorization as an important optimization. This helped to explain the difference with improvements in the vectorizer. The same approach could also be used to analyze the opposite case of worse performance with a newer version. This makes it possible to investigate regressions in a compiler's optimization pipeline.

Afterwards, I related a difference with `clang` to the used SIMD extensions. This explained the large improvement when adding the flag `-march=native`. In a similar manner, this approach applies to compiler flags in general. For example, it is possible to analyze the effect of individual optimizations and their parameters. Furthermore, the workflow can be used to understand the combined result from optimization levels such as `-O3`.

## 5.4. miniQMC

Finally, miniQMC contains Quantum Monte Carlo (QMC) algorithms similar to the larger simulation QMCPACK [10, 39]. It simulates the total energy of a system composed of nickel and oxygen atoms. This requires the solution of an integral over possible positions of the electrons. In miniQMC, this is implemented by randomly sampling the positions according to their probability [10].

As before, I compile with Intel MPI but only execute with a single rank. Additionally, I activate OpenMP to take advantage of annotations for vectorization. However, I use `OMP_NUM_THREADS=1` to compute with only one thread. For BLAS, the executables are linked to Intel's Math Kernel Library (MKL).

Similar to the other case studies, I choose a benchmark configuration that results in runtimes of around one minute. For that, I pass the parameter `-g "4 2 2"` on the command line when running the application. This creates $4 \times 2 \times 2$ boxes with each 16 nickel and 16 oxygen atoms. Nickel and oxygen atoms have 18 and 6 physically relevant electrons [10], respectively, which results in a total of 6144 electrons.

Using this configuration, I measure the lowest execution time of 56.0 s for `gcc` with `-Ofast -march=native`. It is closely followed by `clang` with 57.0 s when compiling with the same flags. For the Intel Compiler, the best performance results from passing `-O2 -march=native` with a runtime of 63.6 s. If instead using `-Ofast -march=native`, execution time increases to 65.1 s. Investigation with the workflow shows that the difference is due to loop unrolling. However, the more interesting difference to discuss is the one between `clang` and `icc`.

### 5.4.1. Branch Misses due to Jumps

Profiling with `perf` increases the runtimes to 58.5 s and 65.9 s. The overhead is slightly bigger for `icc`, but still below 4 %. Due to inlining, the initial profiles have no common topmost function set. It is therefore necessary to modify the source code to adapt the inlining and make the profiles comparable.

After repeated adaptions and profiling, I obtain the data shown in Table 5.7. During this process, the execution times increase slightly to 59.2 s for `clang` and 66.9 s with `icc`. However, the difference is still clearly visible and the results are expected to be consistent. From the profiles, it is apparent that the difference is related to the marked function `DTD_BConds::computeDistances`: It takes 14.9 s when compiled with `icc`, but only 7.8 s with `clang`.

To explain this difference, I look at the generated machine code. Analysis of the instruction mnemonics shows that `icc` generates 21 jumps for this function. In contrast, `clang` only emits 7 jump instructions. At runtime, jump instructions lead to more branches which hinders ILP. For that reason, modern processors have branch predictors to improve instruction pipelining.

Using hardware counters, it is possible to measure how many branches were mispredicted. This allows to assess whether the higher number of jump instructions influences performance. To that end, I use `perf` to record events for `branch-misses`. The results show that `clang` has $2.41 \cdot 10^5$ branch misses for samples in the function `DTD_BConds::computeDistances`. On the other hand, `perf` records a total of

| Function | Samples | Time |
|---|---|---|
| `MultiBsplineEval::evaluate_v` | 61259 | 15.3 s |
| `einspline_spo::set` | 58769 | 14.7 s |
| `mkl_blas_avx512_dgemm_kernel_nocopy_TN_b1` | 38822 | 9.7 s |
| `MultiBsplineEval::evaluate_vgh` | 34466 | 8.6 s |
| `mkl_blas_avx512_dgemm_kernel_0` | 32859 | 8.2 s |
| `DTD_BConds::computeDistances` † | 31039 | 7.8 s |

(a) Clang Compiler

| Function | Samples | Time |
|---|---|---|
| `MultiBsplineEval::evaluate_v` | 61404 | 15.4 s |
| `DTD_BConds::computeDistances` † | 59767 | 14.9 s |
| `einspline_spo::set` | 56828 | 14.2 s |
| `mkl_blas_avx512_dgemm_kernel_nocopy_TN_b1` | 38971 | 9.7 s |
| `MultiBsplineEval::evaluate_vgh` | 34105 | 8.5 s |
| `mkl_blas_avx512_dgemm_kernel_0` | 32628 | 8.2 s |

(b) Intel Compiler

Table 5.7.: Profiles of miniQMC executables built with `clang` and `icc` after decomposition. Functions with less than 10,000 samples are not shown. The names only list the containing `struct` or `namespace` due to space limitations. The measured samples show a significant difference for `DTD_BConds::computeDistances`, which is marked for easier reference.

$6.69 \cdot 10^8$ branch misses for `icc`. These numbers differ by a factor of more than 2,700 which explains the worse performance with the Intel Compiler.

### 5.4.2. Results & Discussion

For miniQMC, I analyzed one difference between Clang and the Intel Compiler. Similar to other benchmarks written in C++, it was possible to decompose the difference by repeated profiling. This was very effective and the found function `DTD_BConds::computeDistances` comprises less than 70 lines of code. Compared to the total size of around 20,000 lines, this corresponds to less than 0.4 % of the total simulation.

Afterwards, investigation of the generated instruction mnemonics gave the right hint. This is another example for a general strength of the workflow: With more data available, it is easier to decide on the next step when analyzing a performance difference. In this case, it was possible to measure the right hardware counters with `perf`. The results confirmed the hypothesis that the higher number of jumps influences the performance.

# 6. Conclusion

In my thesis, I presented a workflow to analyze the effect of compiler optimizations. It compares the execution of two binaries starting with a difference in total runtime. To increase the level of detail, the workflow makes use of profiling data. Based on the time spent in each function, this allows to locate the differences more precisely. The workflow recursively decomposes the functions by handling inlined function calls and loops.

After decomposing a difference, the workflow requires a more detailed analysis of the generated code. To support this step, I described properties of kernels that may be found in the source code of HPC applications. For use in the workflow, I related each property to possible transformations. This allows to focus on compiler optimizations that are likely important for a decomposed difference.

Finally, I evaluated the workflow with case studies of four benchmark applications. In measurements, all of them showed significant differences with the tested build configurations. I presented the analysis results of several differences with the workflow to demonstrate the possible usage scenarios. For all of them, the structured steps described in this thesis were successful in locating the reasons.

I analyzed multiple differences when comparing two compilers for the BT solver and LULESH. After understanding the causes, I described changes to increase performance portability. In total, the proposed modifications lead to performance improvements of up to 14 %. This demonstrates that the workflow can be used for performance tuning of portable applications. In contrast to other approaches, the analysis directly leads to parts of the code that can be improved. This is because the starting point is a faster binary from another build configuration. As such, it is guaranteed that there is a possibility to generate better code. This makes it feasible to recommend the workflow to application developers. If they come across a significant difference, the workflow represents a structured way to understand the reason.

At the same time, the initial comparison also limits the improvements that may be achieved with the workflow. Still, it is possible to exceed the performance of the initially faster binary in two cases: On the one hand, modifications could make one of the build configurations generate even better code. For example, changes could facilitate a compiler analysis which triggers additional optimizations. On the other hand, comparison could show that neither of the build configurations is the fastest for all functions. Instead, the workflow could be used to mutually improve the performance of both binaries. Further research is needed to examine if this yields usable results in the general case.

Based on the work for this thesis, the main focus of the workflow remains on improving portable performance. Nevertheless, there are two more areas of application where the workflow can be used without modifications: With the study of miniMD, I demonstrated the ability to analyze differences for the same compiler. This includes

the comparison of binaries produced by two versions of the compiler. As discussed before, this allows to investigate changes in the optimization of the code at hand. Furthermore, the comparison of different flags enables the analysis of individual optimizations and their parameters. Both analyses may be performed by an experienced application developer with basic knowledge about compilers. Without the workflow, such investigation would require thorough understanding of the complete translation from the source code to the hardware level.

The workflow is structured in a way to allow analysis of large and complex codes. In this thesis, miniQMC is the biggest of the discussed cases with around 20,000 lines of code. It could already be seen for this size that decomposing the difference becomes more important. This trend is likely to continue for even larger simulation packages used in production. Moreover, such codes often contain multiple solvers that can be employed as needed. As for performance tuning in general, this requires carefully designed test cases to analyze the code paths relevant for production runs.

The analysis of the BT solver revealed difficulties in handling differences related to caching. Similar problems could be encountered for other specialized hardware such as branch predictors. To take such global influences into account, it would be necessary to extend the workflow. This likely requires adaptions of the steps for decomposing the differences: The current workflow relies on the assumption that differences can be isolated.

In the case of caching effects, however, it is equally important what data resides in the cache when entering the function. This is primarily influenced by the code executed before the function showing the runtime difference. Similarly, the anatomy of prior conditional jumps is important for the effectiveness of branch prediction. For that reason, it would be necessary to capture this information when analyzing differences. One possibility could be to make use of an existing performance model for cache accesses. The model can be either analytical [59] or based on benchmarks [54].

Furthermore, the results from LULESH show that it is important to study the interaction between optimizations. Work in this area has previously been performed when searching for the best combination of compiler parameters. For example, the approaches by Pan and Eigenmann take the optimizations' interaction into account [51]. In a later work, Mustafa and Eigenmann define substituting optimizations [48]. They observe that, after disabling a transformation, another optimization may take its place and vice versa. By using the workflow described in this thesis, it would be possible to analyze the effect on the generated code for real applications. This could help to better understand and compare the interaction between optimizations.

# A. Assembly Version of the zaxpy Kernel

The following represents an assembly version of the main loop from the zaxpy kernel as discussed in Section 4.5.1.

```
1  zaxpy:
2    # start of the zaxpy function; code to check
3    # the arguments and perform setup ...
4
5    # real(a)
6    vbroadcastsd %xmm0, %zmm2
7    # img(a)
8    vbroadcastsd %xmm1, %zmm3
9
10   # more setup ...
11
12 .loop:
13   # x[:] stored as real(x[i]), img(x[i]), ...
14   vmovupd (%rdi,%rax), %zmm1
15
16   # swap real and imaginary parts:
17   # img(x[i]), real(x[i]), img(x[i+1]), ...
18   vshufpd $85, %zmm1, %zmm1, %zmm6
19
20   # img(a) * swap(x[:])
21   vmulpd %zmm3, %zmm6, %zmm7
22
23   # real(a) * x[:] -+ img(a) * swap(x[:])
24   # %zmm7 = %zmm2 * %zmm1 -+ %zmm7
25   vfmaddsub231pd %zmm1, %zmm2, %zmm7
26
27   # add y[:]
28   vaddpd  (%rcx,%rax), %zmm7, %zmm7
29   vmovupd %zmm7, (%rcx,%rax)
30
31   addq $64, %rax
32   cmpq %rdx, %rax
33   jne  .loop
34
35   # other versions of the loop ...
```

# Glossary

**AoS** Array of Structures. 27, 28

**AST** Abstract Syntax Tree. 5, 6

**ATLAS** Automatically Tuned Linear Algebra Software. 9

**AVX** Advanced Vector Extensions. 48

**BLAS** Basic Linear Algebra Subprograms. 10, 13, 31, 49

**CFD** Computational Fluid Dynamics. 37, 39

**COLE** Compiler Optimization Level Exploration. 10

**CRS** Compressed Row Storage. 28, 29, 34

**DCE** Dead Code Elimination. 7

**DSE** Dead Store Elimination. 7, 28

**FMA** fused multiply-add. 8, 29, 32, 34

**ILP** Instruction-Level Parallelism. 8, 49

**IPO** Interprocedural Optimization. 9

**IR** Intermediate Representation. 5, 6, 7, 8

**ISA** Instruction Set Architecture. 8

**LICM** Loop Invariant Code Motion. 7, 26

**LTO** Link-Time Optimization. 9

**MD** Molecular Dynamics. 37, 47

**MKL** Math Kernel Library. 49

**OSE** Optimization-Space Exploration. 10

**PGO** Profile-Guided Optimization. 9

**QMC** Quantum Monte Carlo. 37, 49

**RA** Register Allocation. 8

**SIMD** Single Instruction Multiple Data. 8, 29, 32, 40, 47, 48

**SLP** Superword Level Parallelism. 8, 41

**SSA** Static Single Assignment. 6

**SSE2** Streaming SIMD Extensions 2. 48

# Bibliography


[1] glibc wiki – libmvec. https://sourceware.org/glibc/wiki/libmvec. Last visited on September 6, 2020.

[2] GNU gprof. https://sourceware.org/binutils/docs/gprof/. Last visited on September 6, 2020.

[3] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[4] Intel® VTune™ Profiler. https://software.intel.com/en-us/vtune. Last visited on September 6, 2020.

[5] Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). https://computing.llnl.gov/projects/co-design/lulesh. Last visited on September 6, 2020.

[6] LLNL/LULESH: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). https://github.com/LLNL/LULESH. Last visited on September 6, 2020.

[7] Mantevo/miniMD: MiniMD Molecular Dynamics Mini-App. https://github.com/Mantevo/miniMD. Last visited on September 6, 2020.

[8] NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html. Last visited on September 6, 2020.

[9] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Last visited on September 6, 2020.

[10] QMCPACK/miniqmc Wiki. https://github.com/QMCPACK/miniqmc/wiki. Last visited on September 6, 2020.

[11] SLEEF Vectorized Math Library. https://sleef.org/. Last visited on September 6, 2020.

[12] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985.

[13] Programming languages — C. Standard ISO/IEC 9899:1999, International Organization for Standardization, December 1999.

[14] Programming languages — Fortran — Part 1: Base language. Standard ISO/IEC 1539-1:2018, International Organization for Standardization, November 2018.

[15] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 647–658, 2014.

[16] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[17] Frances E. Allen. Program Optimization. In *Annual Review in Automatic Programming*, volume 5, pages 239–307. Pergamon Press, New York, 1969.

[18] Frances E. Allen. *Bibliography on Program Optimization*. IBM Thomas J. Watson Research Division, December 1975.

[19] Frances E. Allen and John Cocke. A Catalogue of Optimizing Transformations. In Randall Rustin, editor, *Design and Optimization of Compilers*, volume 5, pages 1–30. Prentice-Hall, 1971.

[20] Krste Asanovíc, Rastislav Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[21] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 136–150, New York, NY, USA, 2017. Association for Computing Machinery.

[22] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[23] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[24] Jeff Bilmes, Krste Asanovíc, Chee-Whye Chin, and Jim Demmel. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 253–260, 1997.

[25] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative Compilation in a Non-Linear Optimisation Space. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France, October 1998.

[26] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

[27] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Ian Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[28] Mark Finkelstein. A compiler optimization technique. *The Computer Journal*, 11(1):22–25, January 1968.

[29] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[30] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[31] Elana Granston and Anne Holler. Automatic Recommendation of Compiler Options. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[32] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer Publishing Company, Incorporated, 2nd edition, 2012.

[33] Simon D. Hammond, Courtenay T. Vaughan, and Clay Hughes. Evaluating the Intel Skylake Xeon Processor for HPC Workloads. In *2018 International Conference on High Performance Computing Simulation (HPCS)*, pages 342–349, July 2018.

[34] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Generating New General Compiler Optimization Settings. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, page 161–168, New York, NY, USA, 2005. Association for Computing Machinery.

[35] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009.

[36] Kenneth Hoste and Lieven Eeckhout. COLE: Compiler Optimization Level Exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, page 165–174, New York, NY, USA, 2008. Association for Computing Machinery.

[37] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, May 2020. `https://software.intel.com/en-us/articles/intel-sdm`, Last visited on September 6, 2020.

[38] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 Updates and Changes. Technical Report LLNL-TR-641973, August 2013.

[39] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, Simone Chiesa, Bryan K Clark, Raymond C Clay, Kris T Delaney, Mark Dewing, Kenneth P Esler, Hongxia Hao, Olle Heinonen, Paul R C Kent, Jaron T Krogel, Ilkka Kylänpää, Ying Wai Li, M Graham Lopez, Ye Luo, Fionn D Malone, Richard M Martin, Amrita Mathuriya, Jeremy McMinis, Cody A Melton, Lubos Mitas, Miguel A Morales, Eric Neuscamman, William D Parker, Sergio D Pineda Flores, Nichols A Romero, Brenda M Rubenstein, Jacqueline A R Shea, Hyeondeok Shin, Luke Shulenburger, Andreas F Tillack, Joshua P Townsend, Norm M Tubman, Brett Van Der Goetz, Jordan E Vincent, D ChangMo Yang, Yubo Yang, Shuai Zhang, and Luning Zhao. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30(19):195901, April 2018.

[40] Toru Kisuki, Peter M. W. Knijnenburg, Mike F. P. O'Boyle, François Bodin, and Harry A. G. Wijshoff. A Feasibility Study in Iterative Compilation. In Constantine Polychronopoulos, Kazuki Joe Akira Fukuda, and Shinji Tomita, editors, *High Performance Computing*, pages 121–132, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[41] Sameer Kulkarni and John Cavazos. Mitigating the Compiler Optimization Phase-Ordering Problem Using Machine Learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 147–162, New York, NY, USA, 2012. Association for Computing Machinery.

[42] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 145–156, New York, NY, USA, 2000. Association for Computing Machinery.

[43] Charles L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[44] Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. A Run-Time System for Power-Constrained HPC Applications. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, pages 394–408, Cham, 2015. Springer International Publishing.

[45] Timothy G. Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Pearson Education, 2004.

[46] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[47] William M. McKeeman. Peephole Optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[48] Dheya Mustafa and Rudolf Eigenmann. PETRA: Performance Evaluation Tool for Modern Parallelizing Compilers. *International Journal of Parallel Programming*, 43(4):549–571, 2015.

[49] Jürg Nievergelt. On the Automatic Simplification of Computer Programs. *Communications of the ACM*, 8(6):366–370, June 1965.

[50] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Algorithmic Species: A Classification of Affine Loop Nests for Parallel Programming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–25, 2013.

[51] Zhelong Pan and Rudolf Eigenmann. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12 pp.–332, 2006.

[52] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2019.

[53] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 1(117):1–19, 1995.

[54] Bertrand Putigny, Brice Goglin, and Denis Barthou. A Benchmark-based Performance Model for Memory-bound HPC Applications. In *2014 International Conference on High Performance Computing Simulation (HPCS)*, pages 943–950, 2014.

[55] Naoki Shibata and Francesco Petrogalli. SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1316–1327, June 2020.

[56] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 77–90, New York, NY, USA, 2003. Association for Computing Machinery.

[57] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler Optimization-Space Exploration. In *International Symposium on Code Generation and Optimization, 2003*, pages 204–215, 2003.

[58] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998.

[59] Jingling Xue and Xavier Vera. Efficient and Accurate Analytical Modeling of Whole-Program Data Cache Behavior. *IEEE Transactions on Computers*, 53(5):547–566, May 2004.