การเพิ่มสมรรถนะของการจำลองซีเอมเอสผ่านการแปลงลูป

นายธีริทธิ์ เพลินสินธุ์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตร์มหาบัณฑิต
สาขาวิชาวิทยาศาสตร์คอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2564

PERFORMANCE IMPROVEMENT OF CMS SIMULATION VIA LOOP TRANSFORMATION

Mr. Teerit Ploensin

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science Program in Computer Science

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2021

Thesis Title	PERFORMANCE IMPROVEMENT OF CMS SIMULATION VIA LOOP TRANSFORMATION
By	Mr. Teerit Ploensin
Field of Study	Computer Science
Thesis Advisor	Associate Professor Krerk Piromsopa, Ph.D.
Thesis Co-advisor	Assistant Professor Norraphat Srimanobhas, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

Dean of the Faculty of Engineering

. . . .

(Professor Supot Teachavorasinskun, D.Eng.)

THESIS COMMITTEE

. . . . Chairman

(Assistant Professor Natawut Nupairoj, Ph.D.)

. . . . Thesis Advisor

(Associate Professor Krerk Piromsopa, Ph.D.)

. . . . Thesis Co-advisor

(Assistant Professor Norraphat Srimanobhas, Ph.D.)

. . . . External Examiner

(Assistant Professor Jittat Fakcharoenphol, Ph.D.)

ธีริทธิ์ เพลินสินธุ์: การเพิ่มสมรรถนะของการจำลองซีเอมเอสผ่านการแปลงลูป. (PERFORMANCE IMPROVEMENT OF CMS SIMULATION VIA LOOP TRANSFORMATION) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : รศ. ดร. เกริก ภิรมย์โสภา, อ.ที่ปรึกษาวิทยานิพนธ์ร่วม : ผศ. ดร. นรพัทธ์ ศรีมโนภาษ 71 หน้า.

วิทยานิพนธ์เล่มนี้นำเสนอการทดลองใช้เทคนิดการเพิ่มประสิทธิภาพของลูป เพื่อการจำลองและการประมวลผลของซีเอมเอส สำหรับฟิสิกส์พลังงานสูงในซีเอมเอสซอฟต์แวร์ โดยทำการเปรียบทั้งด้านประสิทธิผลทางฟิสิกส์ และผลการทำงานของโปรแกรมในเชิงสมรรถนะ โดยเทคนิดการเพิ่มประสิทธิภาพของลูปที่เลือกใช้คือ เทคนิคการแปลงที่สอดคล้องกับลูปในแบบรูปทรงหลายเหลี่ยม หรือรูปแบบการแปลงเลียนแบบความสัมพันธ์ ซึ่งได้เลือกนำทั้งสองวิธีมาดำเนินการแปลงแบบอัตโนมัติผ่านกระบวนการประมวลผลของคอมไพเลอร์ จากการทดลองด้วยเทคนิคดังกล่าวไม่ส่งผลกระทบต่อผลการทดลองทางฟิสิกส์ อีกทั้งยังสามารถเพิ่มประสิทธิภาพของการทำงานในเชิงสมรรถนะของซอฟต์ด้อีกด้วย

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	....
สาขาวิชา	วิทยาศาสตร์ คอมพิวเตอร์	ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
ปีการศึกษา	2564	ลายมือชื่อ อ.ที่ปรึกษาร่วม	....

6170192021: MAJOR COMPUTER SCIENCE
KEYWORDS: COMPILER / LOOP OPTIMIZATION / SOURCE-TO-SOURCE TRANSFORMATION

TEERIT PLOENSIN : PERFORMANCE IMPROVEMENT OF CMS SIMULATION VIA LOOP TRANSFORMATION. ADVISOR : ASSOC.PROF. Dr. KRERK PIROMSOPA, Ph.D., THESIS COADVISOR : ASST PROF Dr. NORRAPHAT SRIMANOBHAS, Ph.D., 71 pp.

High performance processor can tackle bottleneck issues by increasing vector lengths and leveling effectiveness of memory hierarchies to address these issue. Manual optimization of code is a difficult task when having multiple architecture-dependent transformation. Our goal is to develop a tool that performs source code transformation based on loop optimization techniques, since a loop plays an important role in improving of performance in scientific simulation software. We implement an source-to-source transformation tool based libTooling, a Clang's library, based on polyhedral model to simplify a loop transformation of CMSSW building pipeline. The tool also can be used for automatically transformation. The results show that any simple loop transformations can trigger other optimizations in compilers.

Department:	Computer Engineering	Student's Signature	....
Field of Study:	Computer Science	Advisor's Signature	....
Academic Year:	2021	Co-advisor's signature	....

Acknowledgements

I would like to say thank you to Professor Krerk Piromsopa, my advisor, for guiding me into the compiler and high-performance field and giving me a chance to start graduate school in computer science. When I find myself lost with my research or even lost with the understanding of the theory, he always encourages me and gives me motivation. His ideas on research are extremely outstanding and his guidance on how to do research is extraordinary. We can talk with him not only about research and academics but also about how to manage your life, and how to change our perspective of thinking to each environment, which is all valuable for all moments in our time. Even now I adopt all of his teachings to be aware of every step of the learning process. Moreover, I would like to special thank you to Professor Norraphat Srimanobhas, my co-advisor, who gave me the opportunity to work on CMS software, and for all the guidance that he has given me during the time of my graduate study.

I thanked my beloved family who supports me in higher education, especially during the master's degree, and they always encourage me to go on with the research and for always listening to every complaint and problem on my problem.

I want to say thank you to my SPA lab friends for helping with the complex process of doing things inside the university, exchanging ideas, and discussing together in our group meeting.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter I

INTRODUCTION

For scientific applications, computational cost and time are mostly dedicated to programming loops. Scientific simulation code and its mathematical models are steadily evolved to new insights and requirements. Consequently, loop optimization is significantly important for improving execution time and reducing loops overhead. Thus, many techniques are specifically used to provide high-performance code.

Loop optimization [1; 2] can be described as a sequence of specific loop transformations that performs to the source code or intermediate representation of their languages. Normally, a transformation must preserve all dependencies. Compilers have a wide range of loop optimizations, which are different. For instance, compiler directives are used to transform from a sequential loop into a parallel one (i.e. OpenMP [3]). Directives for loop optimizations, such as vectorizing, are implemented in most mainstream compilers. Directives are popularly used by many scientific application, because of the ease of use without to modify source code of the relevant loops [4; 5]. Although compiler directives are uncomplicated to use, they are specific to a compiler (i.e. unrolling a loop in GCC uses *#pragma GCC unroll n*, while Clang uses *# pragma clang loop unroll_count(n)*). Compilers will either perform the transformation automatically or upon request via the command-line flag. Nevertheless, compilers may achieve significantly lower performance than those of manually optimized programs [6]. Moreover, not many loop transformations have been implemented in compilers. The most common implemented transformation is loop unrolling [7; 8]. Typically, other optimizations can be triggered by using compiler options such as compiler directives, or manually code transformation. Therefore, one of the simplest ways to practically optimize a loop is to restructure the source program by either permuting or skewing the loop. A better approach is to search for restructuring techniques that enable other optimizations while preserving the semantics of the original source code. This thesis investigates the use of these techniques by making manual changes, examining whether or not the optimizations can be done by the compilers, and comparing run-time performance among them.

1.1 The Compact Muon Solenoid

The Compact Muon Solenoid (CMS) [9; 10] is one of the particle detectors that is designed to detect particles from the collision produced by Large Hardron Collider (LHC). It is used to study extra dimensions and particles which may include dark matter. The detector contains a sequence of subsystems in layers of increasing distance to the minor interaction points. The detected particles by the detector are muons, photons and charged hadrons. Each layer of the detector has measurement properties of specific particle, and this is able to identify the particle type.

1.1.1 Event Simulation via The CMS Software

To encourage a predictions of physics phenomena and to understand the result from collision of the detector response, all simulation events are built from measured data to identify interaction processes. Scientific simulation code, created by physicists, is gradually developed to insightful study. The CMS Software (CMSSW) [11] is one of the overall collection of software frameworks, written in the C++ programming language, by which CMS reconstruction, simulation, and offline analysis are performed. It provides an efficient structural code in Object-Oriented Programming (OOP) design for reusing in several projects and shared algorithm development. In [12], the authors investigated that the CMSSW requires optimization to fit the resource budget which allows a single event to process faster. They also analyzed the CMSSW release with external packages have approximately 1.5 million lines of code. Moreover, they investigated the limitation of practical experience in the high-energy physics community in producing high-quality and high-performance designs for object-oriented C++ applications. Optimizing the CMSSW is therefore necessary. In such a framework, short-running tasks produce longer time consumption by the framework itself and long-running tasks can prevent compiler optimization.

1.1.2 Modular Architecture of the Framework

CMSSW [13] contains a number of modules that integrated into the executable *cmsRun*. The framework loads the plug-in and instantiates the module when it is called by the configuration file which can be configured by user in python language to module-specific parameter sets. Nevertheless, these cannot be reconfigured during the job being process. When a job is submitted, the framework normally manages to instantiate the modules. A part of code described in the configuration file is dynamically loaded. With the modular design, it should be easy to ingrate a third-party tool with the existing architecture.

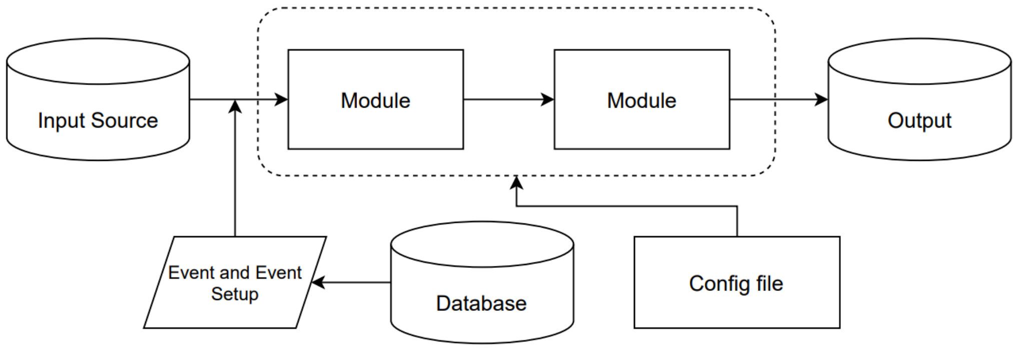

Figure 1.1: The modular architecture of CMSSW framework.

1.2 Problem statement

Loops in the CMSSW written by physicists sometimes have a form that is cumbersome to be optimized by the compiler. Many optimizations by a compiler cannot directly perform on such loops. In this case, we can write a normal loop and let compilers optimize it with generating platform-specific vector instructions. However, compilers is not smart as it is. Thus, rewriting them into appropriate pattern via loop optimization is considered, such as Polyhedral Model, a framework for performing loop transformation. In order to achieve the goal, the main contribution of this thesis is a source code transformation tool for optimizing the loop part in the CMSSW via loop transformations.

1.3 Goals

The purpose of this thesis is to build the source-to-source translator tool. The tool should be able to transform nested loops by applying the Polyhedral Model. The tool in this context should be able to:

- Generate C++ code with desired loops transformation.
- Be able to integrate with the CMS software and to be used by non-expert users.
- Be used as a study tool to investigate on intermediate representation of source program.

Chapter II

BACKGROUND OF STUDY

The previous chapter explains the motivation for performance improvement of CMSSW. As a result, the problem statements by which we describe, we will apply compiler optimization techniques to improve the execution time of the system. However, each optimization technique is different; for instance, specifying a directive can trigger a parallel code (i.e. OpenMP) [14] or using a domain-specific language to avoid the complex procedure. Then, common optimization techniques are provided. Moreover, we discuss which optimization seems to be an appropriate approach in our scenario.

Because most of the vectorization or some optimization task is relying on data dependencies among programming statements, for example, loop optimization or automation vectorization, this chapter presents data dependency which is the basis of compiler optimization. Importantly, automatic vectorization is plays an important role in our work since we will trigger it via a loop transformation. Thus, this chapter also describes the context of the technology in more detail.

2.1 Compiler Optimization

Firstly, we start with the concept of compiler optimization. Compiler optimization, commonly mentioned in textbook [2], is generally a sequence of optimizing transformations or algorithms that produce a semantically equivalent program that executes faster than the original one.

2.1.1 Compilers

A compiler is a computer program that processes a programming language into machine understandable language. This process consists of three procedures as shown in Figure 2.1.

- First, the compiler frontend checks the grammar and syntax of the source code. Then, it converts the source code and builds an intermediate representation of its language binding to the programming source code. This phase contains lexical, syntax and semantic analysis, which we know as the tokenization phase.

- Then, generated tokens from the previous step are converted to Intermediate Representation (IR) where major optimization happens. An optimizer pass at the backend performs on this part.

- Finally, the result from the previous step which is optimized IR is transformed to target architecture. This is known as the code generation phase.

In addition, the optimizer is either responsible for a variety of optimization or transformation, such as dead code elimination or code smell deletion. However, optimizations not only occur at optimizer but also can be added to compiler frontend.

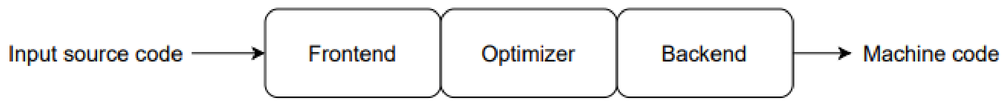

Figure 2.1: Three main phases of classical compiler design

Most common compilers contain various optimization techniques in the optimizer, which are described in the following detail:

- A combination of loop transformations, included interchange, unrolling, switching, etc.

- Dead code elimination is a techniques to reduce code size of unused code.

- Inline expansion is a technique that a body statement replaces to a function call.

These presented list are most common techniques that are implemented in most compilers. Also, compilers apply them during compilation phase. Moreover, there are other optimizations or passes can be applied by a user using flags. This thesis, we will only focus on loop optimization which transform a loop into a sequence of vector operations, thus further detail is given within the next section.

2.2 Vectorization

Vectorization is a task where mathematical operation in a loop in source code is performed the same operation in parallel by special vector hardware in CPUs. Modern high-performance architectures improve their performance by expanding vector length and additionally improving memory hierarchies in order to gain high performance result [15]. Specialized instructions are used for vector operations, known as Single Instruction Multiple Data (SIMD). It is not only useful for scientific applications but also for multimedia applications. If such a language implementation can use SIMD instructions, this brings to significantly improve of performance.

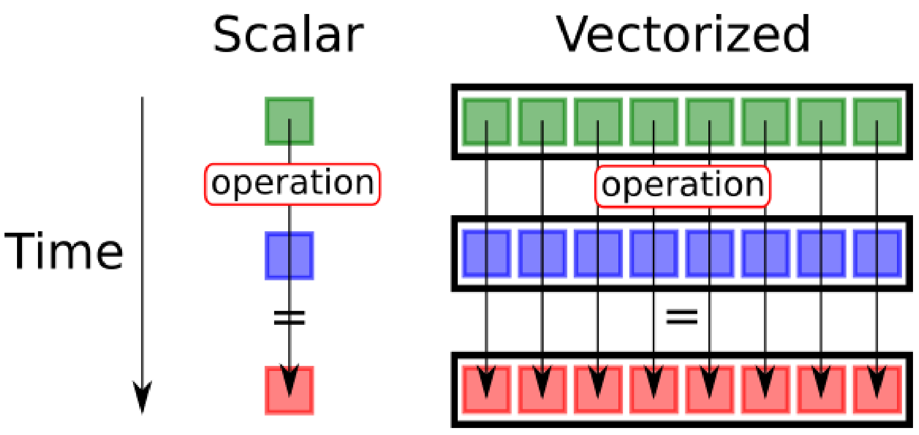

Figure 2.2: Comparison of execution between scalar sequential (scalar) and vectorization computation. Source: `https://lappweb.in2p3.fr/~paubert/ASTERICS_HPC/6-6-1-985.html`

Recently, vectorization is an important feature in boosting the performance of HPC applications. Continuously increasing vector length can create more opportunities in the optimization process of those applications. A compiler is able to automatically vectorize their code as part of optimization process. Nevertheless, complex code makes imperfectly vectorization. Moreover, inefficient utilize of memory and cache decrease performance obtained by vectorization. This thesis will focus on vectorization or optimization from compiler perspective where by code in a manner of vectoization is possible. With the simply code, a compiler can generate vector instruction from such code to take advantage of vector hardware.

2.3 Vectorizable Code

This section, we describe about vectorizable code in term of a loop. Since, vectorization mostly happens in the loop. For any compiler to perform vectorization, the loop must pass the basic requirements for vectorizable loop. Any loop that does not pass these requirements may not possible to do.

The basic requirements for vectorizable loops are given below:

- A loop number must be countable at runtime. It means that the total number of loop iterations need to be specified before a loop executes.
- A statement inside a loop need to be a single control flow. Because, branching or conditional statement prevent code from being vectorized.
- A loop should not contain function calls, because there is a complicated jumping condition between stack and memory.

2.4 Data Dependency

Previously, we conceptualized vectorization technology that evaluates code in parallel version instead of sequential one. Nevertheless, the original order of a loop may change due to the transformation. If the result of previous iteration depends on the next computation of iteration, in this case, the automatic vectorization will lead to incorrect results. This is known as data dependency [16] which occurs when an instructions use a register of memory location which other instructions uses. Dependency between statements affects the compiler's ability to optimize code for parallelism or vectorization. So to get the maximum benefit from optimization code, data dependency should be carefully managed.

Thus, dependency analysis is used to analyze data dependencies between instructions. Three types of dependencies are:

1. **True (Flow) dependence**, known as read-after-write, exists when an instruction depends on a previous instruction's result.

   ```
   x = 1;
   z = x + 1;
   ```

   The second instruction uses x after the first instruction writes. This kind of dependency is not vectorizable.

2. **Anti-dependence**, known as write-after-read, exists when instruction stores after another instruction uses it.

   ```
   y = x + 1;
   x = 2;
   ```

   The third instruction writes to x after the second instruction reads it. This kind of dependency is vectorizable. A input value is used to be an input in previous iteration, then it used to write to the next iteration.

3. **Output dependence**, known as write-after-write, exists when two instructions write the same memory location. It affects final output of a variable writes to it.

   ```
   x = 1;
   y = x + 1;

   x = 2;
   z = x + 1;
   ```

   The second and fourth instructions read an difference result due to differently store value into x. This occurs when multiple iterations can alter the single variable. Thus, this kind of dependency is not vectorizable.

2.5 Loop Optimization

Loops mainly have an impact on the performance of many programs. They are the part where execution time is mostly spent on. Therefore, loop optimization has been studied to make it faster. It is a classical subject of study [1; 2]. In addition, a combination of loop transformations can be considered to be loop optimization. A transformation of source program is an operation where it converts the input program and generates another program with equivalent to the original. Generally, loop transformations work on reshaping iteration space to gain more data locality of the target architecture. Almost of loop transformations are a technique that is designed to make vectorization or another optimization possible because sometimes the loops can not directly vectorize or optimize until applying another transformation before.

2.5.1 Loop Unrolling

Loop unrolling [1; 2; 17] is the simplest loop optimization by unrolling to avoid jumps instructions to execute. When an unrolled loop body statement is duplicated with the same number of the loop incremental. This number is called unroll factor. However, when the conditional variables of loops are not known at compiler time, a compiler adds a remaining part below the unrolled loop to handle them during execution time.

```
// Original
void OriginalLoop(int n) {
    for(int i = 0; i < n; i++) {
        statement(i);
    }
}

// Loop Unroll
void LoopUnrolling(int n) {
    for (int i = 0; i < n-2; i+=2) {
        statement(i);
        statement(i+1);
    }

    for (int i = n - (n & 2); i < n; i++) {
        statement(i);
    }
}
```

Listing 2.1: Example of loop unrolling

2.5.2 Index Set Splitting

Index Set Splitting [18] proposed by M. Griebl, P. Feautier, and C. Lengauer, is a method in which the loop nest's parallelism is maximized. They introduce it as a pre-possessing state for the polyhedral model optimization. This method is also known as a technique that divides a loop containing conditional expressions into several loops with less complex control flow [19]. The algorithm tries to distinguish the iteration space into individually parts. Then, the splitting loop can improve parallelism.

As well as in [20], they purpose a new extension of the CLooG's algorithm that divides the outer loop into small parts to removes loop bounds generated from code generation. The loop bounds sometimes blocks the compiler from others optimization, such as vectorization.

```
// Original
void OriginalLoop(int n) {
    for(int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++ {
            statement(i,j);
        }
    }
}

// Index Set Splitting
void ISS(int n) {
    for (int i = 0; i < k; i++) {
        for (int j = 0; j < m; j++) {
            statement(i,j)
        }
    }
    for (int i = k; i < n; i++) {
        for (int j = 0; j < m; j++) {
            statement(i,j)
        }
    }
}
```

Listing 2.2: Example of Index Set Splitting

2.5.3 Loop Interchange

Loop interchange [1; 17] changes the indices of the nested loop by swapping an inner loop to an outer loop. Interchanging the loop can improve the access pattern. However, there is some consideration that loop interchange may give in worse performance due to cache performance. Loop interchange is legal when the interchanged loop uses a non-computed value.

```
// Original
void OriginalLoop(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            statement(i, j);
        }
    }
}

// Loop Interchange
void LoopInterchange(int n) {
    for (int j = 0; j < m; j++) {
        for (int i = 0; i < n; i++) {
            statement(i, j);
        }
    }
}
```

Listing 2.3: Example of Loop Interchange

2.5.4 Loop Skewing

Loop skewing [17] is the loop transformation that changes the basis of the execution of an inner loop relates to an outer loop to exploit wavefront parallelism. This is very useful transformation in case the inner loop is depending on the outer loop which blocks it from running in parallel.

```
// Original
void OriginalLoop(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <=i; j++) {
            statement(i, j);
        }
    }
}

// Loop Skewing
void LoopSkewing(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j <= 2*i; j++) {
            statement(i, j-i);
        }
    }
}
```

Listing 2.4: Example of Loop Skewing

2.6 Polyhedral Model

Polyhedral model [21; 22] is an abstraction of nested loops with loop transformations in mathematical model. In general, a loop transformation can be represented in term of an affine transformation. For example, skewing is the change of basis of original dependency to obtain wavefront parallelism. Thus, the loop transformation is performed on the iteration space. Moreover, we can easily legality of transformation via composition of iteration space. Nested loops commonly are able to represent such model. Then, it represent loop condition and its iteration as points inside representation object.

As a result, polyhedral model is used to perform a loop modeling and apply any loop transformation on it. Moreover, there is a study that nested loop auto-vectorization can be guided by polyhedral model [23].

High-level concept of polyhedral model relies on three steps:

- representing the original code into the geometrical view called iteration space;
- performing a geometrical transformation in this space;
- converting the set of iteration domain back to generate code.

2.6.1 Correctness of Transformation

This section is a summary of Cooper's work [24]. There are two considerations for the correctness of a compiler. First, target code that is generated by the compiler should remain the same meaning as the original. Any operations integrated into the compiler must handle this as a baseline. In an optimizing perspective, the result generated from the compiler should have lower or equal time or space consumption. As a result, each operation uses code for some part of a program as an input, then it generates code with the same part. The output of optimizing compiler is better by some performance metric. There is the statement of the notion of observational equivalence introduced by Plotkin to formalize this notion [25].

> For two expressions, M and N, we say that M and N are closely equivalent if and only if, in any context C where both M and N are closed (that is, have no free variables), evaluating C[M] and C[N] either produces identical results or neither terminates.

Especially, most compilers aim to improve time and space consumption of the program as much as possible. Preserving program equivalence is also important. Two programs between transformed and original ones should be equivalent to the same input. In this thesis, we do not focus on how the transformed program maintain mathematical correctness but focus on the conservation of physics simulation compared to the original one. As a result, physics result validation will use as the correctness of transformation.

2.7 The Design of an Optimizing Compiler

Currently, a modern compiler supports multiple languages or target architectures. Mostly, the compiler frontend can parse in many languages, and the backend can be specifically written for any target. This brings to the new design whereby the compiler uses a common code representation in its optimizer, as known as an intermediate representation, as represented in Figure 2.3.

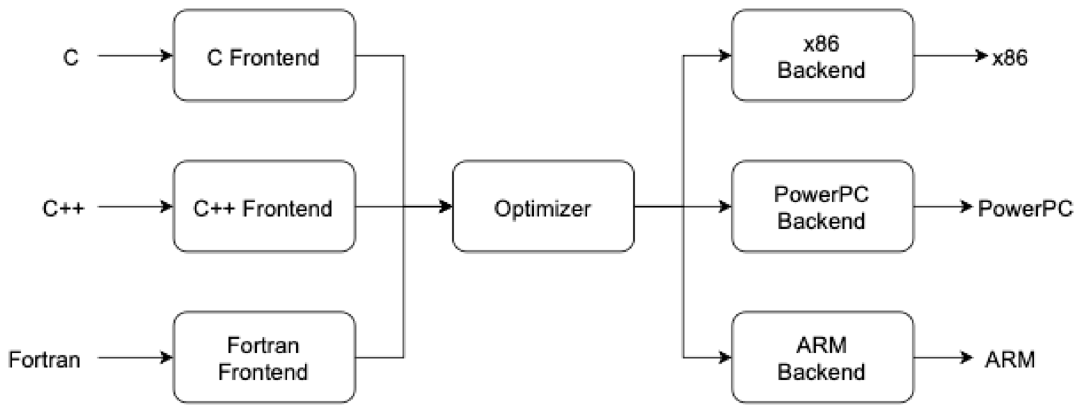

Figure 2.3: Three main phases of modern compiler design

It takes maintainability that the skill required to implement in each part are different. This simplifies the workflow for developers to enhance or maintain their part of the compiler.

Many compilers have commonly optimization such as loop optimization (i.e. unroll, interchange), automatic vectorization, and data-flow optimization (i.e. constant propagation, dead code elimination). In compilers, there are static performance analyses and heuristics that handle possibilities to apply optimization to a program. Moreover, they provide options using directives or flags to indicate operation from code. However, all available optimization possibilities or even the new design, still have limitations where every compiler's decision is not obviously optimal.

2.7.1 Limitations

Compilers can automatically tackle performance improvement of a program via static performance analyses and heuristics. However, they still fail to apply optimizations due to both indicators. There is a source-to-source transformation technique that no longer depends on the compiler's indicator. Since it gives a good balance among possibilities to optimize code and expressiveness of the technique. Moreover, the optimization only occurs in the frontend phase. However, not many tools support building frontend optimization due to the reality that it is extremely hard to handle correctness of a program. Support tools for building source-to-source transformator for C-family languages were difficult. Thus, this thesis aims to optimize the specific framework in scientific software using the source-to-source transformation. The main goal is to preserve the original source program in terms of meaning as much as possible. The following sections will describe various approaches and tools for building the source-to-source transformation tools. Also, related works will be discussed in brief details.

2.8 Source to Source Transformation

A source-to-source transformation [26; 27] is the method of taking one of a program as an input and generating another representation of it as an output with the same meaning. It mostly belongs to a compiler frontend. The goal is clearly to optimize, refactor, or migrate from one language to another language. For example, a user has a set of C++ code and performs the source-to -source transformation on it using some tool, and the resulting output is optimized C++ code to earn advantage of a loop transformation approach. Unlike the general compiler is responsible for converting a high-level programming language to a machine code that is binary, the source-to-source compiler is to convert one source code from one programming language to another language which is the same level of compilation [28; 29].

Any transformation can be done manually, but it can be integrated into a compiler as an automation procedure by modifying the compiler's data structure (e.g. abstract syntax tree) or conveniently using patterns or templates of source code fragmentation [30].

2.8.1 Abstract Syntax Tree

Abstract Syntax Tree (AST) [31] is a data structure that represents the structure of a program in a tree representation. The AST is an entry for semantic analysis of a program. Usually, semantic analysis is a compiler process where it parses a program to collect necessary information from the program. For example, it includes type checking, or even a variable is declared before use. Thus, the AST can be used to represent most procedural languages.

There are three types of elements in the AST mainly used in this thesis as described below:

- *Declaration* is a statement for name, type, and value of a symbol. Symbols also are sub-elements such as constants, variables, and functions.
- *Statement* indicates an action to be carried out that changes the state of the program such as loops, conditions, and return statements.
- *Expression* is a combination of operations and values that is evaluated due to specific rules. The result of evaluation is an integer, floating point, or string.

Chapter III

LITERATURE REVIEW

From previous chapter, we present background of study that is important to know for loop optimization and source code transformation. Even we want to propose a new tool to help physicists or developers who work on a physics simulation to optimize their code. This tool must be a source-to-source translator tool which is able to automatically transform source code. Thus, this chapter provides the survey of existing tools that can perform the source-to-source transformation. Then, we will only choose a tool that matches our criteria.

The literature review includes several frameworks using a compiler frontend to perform the source code transformation known as the source-to-source compiler. Moreover, there are many compilers that can perform these transformations by their optimizer module. Therefore, the main purpose is to find the compiler or framework that will be best match with the following criteria:

- The compiler or framework has to generate an AST, and we need an output code to remain at source level with unspecified compiler intermediate representation (IR).
- The compiler or framework has to generate output code as source level and not in a compiler-specific.
- The compiler or framework is able to perform source-to-source transformation on C++ language.
- The compiler or framework is able to integrate with the CMS Software (CMSSW).

Moreover, working on performance tuning, profiler and analyzer tools need to be consider. They can be helpful in finding bottleneck part and clear up these performance issue. This chapter also provides the profiling tools that currently used for CMSSW which are IgProf, Valgrind(Callgrind and MemCheck), and CMS's time report.

3.1 Tools for The Source-to-Source Transformation

There are a number of available compiler that is specialized at source-to-source transformation, but only some number can match with our criteria. In this section, we compare and present a list of main tools or compilers that are able to perform source-to-source.

3.1.0.1 Clang

Clang [8; 32; 33], a cross-platform C/C++/Objective-C developed by Apple, is a compiler frontend which is responsible for tokenization and generating intermediate representation of C/C++ languages. Clang also gives libraries to parse and interact with C/C++/Objective-C code at abstraction levels. It is important to note on these modules:

- **Driver**, Clang's executable is a small driver that manages execution of other tools. We only use it to run other tools such as parsing AST to feed to next the stand alone tool.
- **Preprocesing**, this stage is responsible for macro expansion, tokenization of the input source program and managing declared directive or preprocessors.
- **Parsing and Semantic Analysis** The input file is parsed during this stage, converting the token from before step into intermediate representation called parse tree. It also use semantic analysis to identify types for expressions. Moreover, this stage manage to provide warning or error at compile time of the program. The result from this stage is an Abstract Syntax Tree (AST).
- **Code Generation and Optimization** These stages handle to translate the AST from the previous step into another Intermediate Representation (IR) which is used for

backend side. This phase manage to optimize generated code and handling target-specific code generation.

Clang also provides interface to write a standalone tool for semantic and syntactic information from program. Moreover, it underling framework provides the tool that supports to run tools over single file, or multiples files and independently of the system like LibTooling.

Clang/LLVM's Tooling library (libTooling) [34; 35] contains source-to-source transformation abilities, but it has demonstrated of using as source refactoring and code generation task [36]. There are works with LibTooling which demonstrates its use for the source-to-source transformation using annotations or flags which we will describe in the next section.

Canonical example when to use LibTooling:

- a simple syntax checker
- refactoring tools

3.1.1 OP2-Clang

OP2-Clang [37] is a source-to-source translator implemented using LibTooling. The tool can specifically generate target platform code such as SIMD, OpenMP, CUDA and the combination with MPI. By design, it is easy to maintain and extend their tool to use to generate new parallelization and optimization for hardware target for their source code. For use case study of their work, it is used for generating parallel code of Airfoil CFD and Tsunami simulation application via OP2's API.

In this work, they utilize Clang's LibTooling for the source-to-source transformation. LibTooling has an API to perform operations over source code with the modification via the AST. The tool is starting by utilizing of Clang's parser to parse and generate AST of the source code. The next phase of their tool involves transforming task on the AST with specifying target. After that, the transformed AST is generated back to optimized source code. The overall architecture of OP2-Clang is shown in the red box in the Figure 3.1 [37]

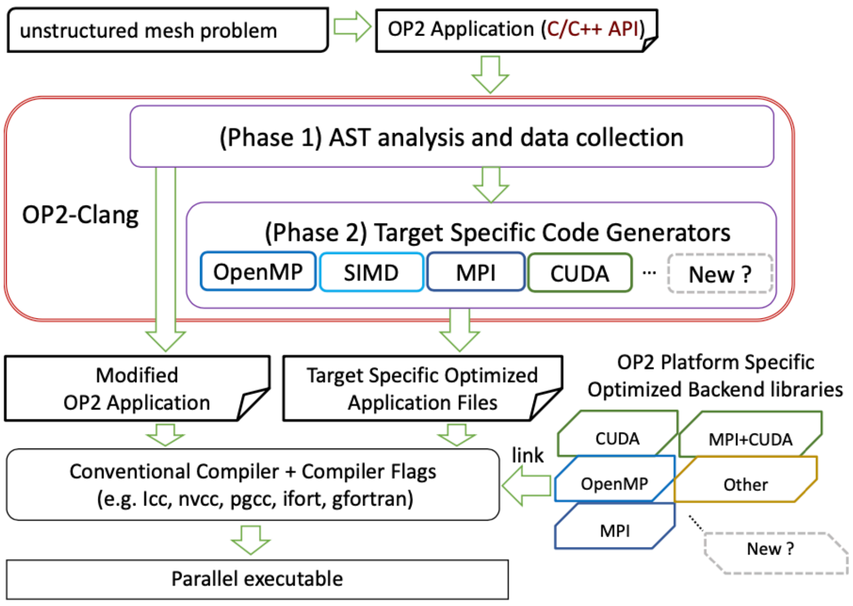
unstructured mesh problem
OP2 Application (C/C++ API)
OP2-Clang
(Phase 1) AST analysis and data collection
(Phase 2) Target Specific Code Generators
OpenMP
SIMD
MPI
CUDA
New ?
Modified OP2 Application
Target Specific Optimized Application Files
OP2 Platform Specific Optimized Backend libraries
CUDA
MPI+CUDA
OpenMP
Other
MPI
New ?
link
Conventional Compiler + Compiler Flags (e.g. Icc, nvcc, pgcc, ifort, gfortran)
Parallel executable

Figure 3.1: The high-level architecture of OP2-Clang

As shown in the figure 3.1, The tool is designed in modular design to support for generating SIMD vectorization and CUDA code. An idea of SIMD vectorization is to generate an automatic vectorized code during compilation phase with a C/C++ compiler. The function itself needs to parse via Clang's parser to gain its AST information to transform the element function. Clang's AST matcher is used to specify a part of tree in binding with the function signature. Then, it is replaced by the desired subscription to expose vectorization. Finally, vectorizable AST is generates back to vectorizable code.

3.1.2 Scout

Scout [38] is a source-to-source transformator for SIMD optimizations, e.g., auto-vectorization. Scout provides a compiler directive to trigger vectorization via a pragmatic flag where users need to annotate their own. Scout also gives both command line and graphical user interfaces to use. It utilizes the Clang front-end interface to build an AST from source code. Then, the AST is converted by an operation to optimize AST. Finally, the optimized AST is converted back to source code. Scout is compatible with SIMD instructions set like SSE or AVX. There are two main methods before the actual vectorization starts. First, the loop body is modified by loop unswitcing , function inlining to move some invariant statement to outside of it. Consequently, the loop is able to unroll from the inner side. Then, the resulting loop is vectorized using unroll and jam technique. Scout has a C front-end and a function to perform vectorization transformation. In addition, it cannot be integrated into other frameworks.

3.1.3 ROSE

ROSE [39] is a compiler under the license of BSD (Berkeley Software Distribution) that is designed for source-to-source transformation task. The motivation to develop this framework is to make a compiler to provide a meaningful source representation of the post-transformation changes comparing to intermediate representation one. Since the traditional compiler provide the result in extremely difficult representation. As a result, ROSE is developed to be source-to-source compiler that can parse C/C++ and Fortran. ROSE's API also supports both analyze, modify and rewrite to generate code after modification. ROSE provides libraries that make it easy to build wide range of tasks for optimizing source-to-source compiler. Nevertheless, ROSE can handle C/C++ language, but it is a really black box that is not flexible to modify.

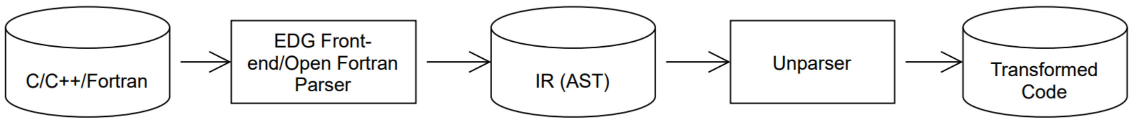

Figure 3.2: ROSE compiler infrastructure

3.1.4 OpenC++

OpenC++ [40] is also under the license of BSD. It is an open-source refactoring library and C++ compiler front-end. It is implemented to help programs to easily analyze C++ code or to perform source-to-source transformation. The flow of OpenC++ works by writing meta-level program, which adds the method to translate or analyze any C++ code. Then, OpenC++ compiler compiles code and links it to a compiler plug-in [41].

As a result, developers can simply implement their own transformation tools. They also can define new object, new syntax, and new annotation. Moreover, there are very useful tools to develop a analyzing tool for source code such as tool for producing the class-inheritance graph of a C++ program [41]. See figure 3.3.

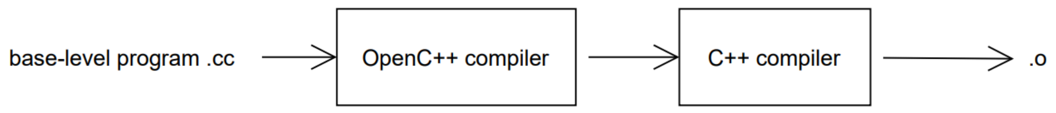

Figure 3.3: Overview of meta-level program in OpenC++ compiler

However, it is maintained by a group of volunteers, but it has not been updated since 2004. Regrading to the tool, it only handles C++ and is depend on its compiler.

3.1.5 DMS Software Re-engineering Toolkit

Design Maintenance System Software Re-engineering Toolkit [42] is a commercial compiler frontend. The compiler allows developer to apply source-to-source pattern transformations, then re-generate transformed source text. It can also be used to construct semantic analyzer to find issues and apply restructuring an existing body statement to resolve those issues. It provides a wide range of front-ends for most commonly used programming languages including C++, Java, C# and COBOL.

3.1.6 Summary

Table 3.1 presents a summary of previous introducing tools and their abilities to handle our criteria. We can see that LLVM/Clang is the only one which matches our criteria to handle C++ project in CMSSW. In addition, it is an open-source that allows us to modify what could manage certain applications for a loop transformations. Others are commercial or can not handle the three criteria required for us.

	C++	C	Source-to-source	Documentation
LLVM/Clang	✓	✓	✓	✓
Scout	✓	✓	✓	-
OP2-Clang	✓	✓	✓	-
ROSE	✓	✓	✓	✓
OpenC++	✓	-	✓	-

Table 3.1: Summary of existing tools for performing source-to-source transformation.

3.2 Profiling Tools

Profiling is a process of software that evaluates space (memory) or time consumption of a program, the usage of instruction and duration of each function calls. There are profilers and libraries of different system.

The following tools are profilers which are mostly used for C/C++ languages.

- The GNU gprof profiler [43] is the standard profiling tool for C/Unix. It is used for sampling and functional instrumentation to perform the profiling.
- OProfile [44] relies on hardware performance counters when measuring the performance of an application.
- Intel VTune [45] is a closed source tool that relies on hardware performance counters and timers. It provides the execution time spent in part of code.
- IgProf [46] is a profiler that can measure performance and memory of program. Benefit of using IgProf is completely operation in use.

In profiling method, it can figure out the parts that are time consuming and need to be optimized called hot spot. The profiling data give you high of resource usage (where resource is memory, CPU cycles and etc.), so you can know that where the problem has to consider. It leads you to focus on the correct point in the software for an optimization task. You can also measure and re-measure to verify your performance improvement. This always makes your program execution faster which is desired. Most of profiling tools work via a dynamic analysis. It is a process that they measure live executing program. In very large projects, profiling can help you find many other statistics through which many potential bugs can be spotted and sorted out.

3.2.1 Igprof

IgProf [46] is light weight profiler using in CMSSW community. L Tuura et al. [47] described that the main benefits of IgProf are lightweight, efficiency, and speed. Moreover, their work show that the performance profiler consumes only 40 MB for memory usage which can be neglected. The memory profiler consumes approximately 500 MB. Moreover, it is significantly faster than Valgrind and Callgrind.

This command below is to obtain profile statistics files. If you have run Igprof in performance profile `(-pp)` mode:

```
igprof -d -pp -z -o igprof.pp.gz myApp arg1 >& igtest.pp.log &
```

If you run igprof in memory profiling `(-mp)` mode:

```
igprof -d -mp -z -o igprof.mp.gz myApp arg1 >& igtest.mp.log &
```

The result file consists of three sections which are flat profile (cumulative), flat profile (self) and call tree profile (cumulative).

3.2.2 Callgrind

Callgrind [48] is a profiler that collects a very precise number of instructions executed, relationship between function with call counts and source line. It can also simulate

L1/L2 caches and measure cache hits/misses. The Callgrind tool counts function called and instruction executed in each call. Thus, we use Callgrind for both cache and CPU profiling. In addition, Callgrind results can be alternatively displayed by KCachegrind [49] as an analysis GUI.

To run your software to start profiling, you need to use the command below:

```
valgrind --tool=callgrind myApp arg
```

The command above will run normally with a bit slowly due to instrumentation. When it finished, it shows shortly the total number of events:

```
==22417== Events    : Ir
==22417== Collected : 8239568
==22417==
==22417== I   refs:      8,239,568
```

Valgrind will write the output file named `callgrind.out.id`. Then you run an annotation on this output file to display:

```
callgrind_annotate --auto=yes callgrind.out.id
```

The output with annotation gives instruction executed and displays the sorted list of function in highest to lowest counts. Your bottleneck function will be listed at top of file. For showing cache hits/misses, you can use with the simulate-cache option like this:

```
valgrind --tool=callgrind --simulate-cache=yes myApp arg
```

To conclude the chapter, the literature review of this the thesis provides briefly understanding related tools for source-to-source transformation, mainly the Clang compiler frontend and its infrastructure. Since, it is selected to be a suitable tool that matches our criteria. Moreover, we also describe the profiler tools which are using for performance tuning.

Chapter IV

METHODOLOGY

This chapter describes a proposed method, the design of the source-to-source transformation tool for integrating with the building process of CMSSW, and the interface of the Clang compiler to build a tool. As described in the earlier chapter, LibTooling has a convenient interface to perform operations over source code. Thus, it's suitable for the task of refactoring or code modification in this context of the problem. Next section, we will discuss our design in the high-level concept of the source-to-source transformation tool.

4.1 Concept Overview

This section presents an overview of module usage involved in a source-to-source process. The section also gives more details on transformation and supported loop transformations via the process.

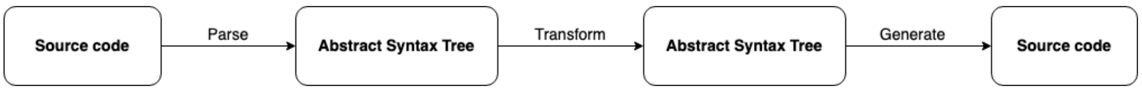

Figure 4.1: Overview of module usage during the source-to-source transformation process.

Basically, a compiler frontend will parse the source code into an Abstract Syntax Tree, sometimes it is known as an intermediate representation. During the process, when there is a need to transform or change some part of code, a transformer operates on the AST using a visitor or matcher pattern to find a node that requires changes. In the end, the modified AST is generated back to the source file via a rewriter module.

4.2 Supported Transformations

In this section, we describe the high-level concept of loop transformation on the AST. Most of the loop transformations mentioned in the background section can be directly done through the AST. There are brief details for loop transformation below.

```
ForStmt
|-DeclStmt
| `-VarDecl
|   `-IntegerLiteral
|-BinaryOperator '_Bool' '<'
| |-ImplicitCastExpr 'int' <LValueToRValue>
| | `-DeclRefExpr 'int' lvalue Var 'i' 'int'
| `-IntegerLiteral 'int' 12
|-UnaryOperator 'int' postfix '++'
| `-DeclRefExpr 'int' lvalue Var 'i' 'int'
`-ForStmt
  |-DeclStmt
  | `-VarDecl j 'int' cinit
  |   `-IntegerLiteral 'int' 0
  |-BinaryOperator '_Bool' '<'
  | |-ImplicitCastExpr 'int' <LValueToRValue>
  | | `-DeclRefExpr 'int' lvalue Var 'j' 'int'
  | `-IntegerLiteral 'int' 12
  |-UnaryOperator 'int' postfix '++'
  | `-DeclRefExpr 'int' lvalue Var 'j' 'int'
  `-DeclStmt
    `-VarDecl temp 'int' cinit
      `-IntegerLiteral 'int' 0
```

Listing 4.1: A nested loop in a pare tree representation

4.2.1 Index Set Splitting

Index Set Splitting is the technique where a loop is split into multiple loops with different condition statements. To transform with these criteria, the whole loop needs to be duplicated, then the condition statements are changed to match the criteria of the transformation. Pseudo code is described below.

```
ISS(u)   ( u is the vertex where the tree starts )
    if outer ForStmt is found then
        tmp = ForStmt
        change the condition of the loop to splitting point
        insert tmp into starting point of the loop
    else
        return
```

4.2.2 Skewing

The procedure of skewing transformation can be divided into two parts which are the inner of a nested loop and the body statement inside the parse tree of the loop mentioned in 4.1. For an inner loop, we can represent the high level of parse tree as below. Therefore, we will interested in a child *forStmt* node since it is the inner loop.

In addition, skewing is the technique that rearranges its array access. Consequently, the declaration reference expression needs to be changed on the parse tree. Thus, the transformation is a text replacement of those values to match the criteria of skewing transformation.

For a body statement tree traversing, we use a recursively depth-first search to find array access for the body statement inside a loop. Pseudo code is described below.

```
REC(u)    ( u is the vertex where the search starts )
    if (visited[u] is ImplicitCastExpr) then
        change the basis of index by i + j
        return
    else
        get children statement as v
        if v is null
            continue
        REC(v)

END REC()
```

4.3 System Design

The tool entry is to parse source code. Then, parsed source code is generated as an AST via a parser of Clang. It holds AST nodes in *ASTContext*. This AST contains the collection of required information such as data types, statements to use for tree matching.

Next, the LibTooling library provides the *ASTConsumer* interface which is an abstract interface that allows us to inherit and override a few methods of it to gain AST information in the desired way. Then, the consumed AST is modified via AST's operation.

ASTConsumer has many methods to read the AST. Clang also provides *HandleTranslationUnit(ASTContext)* is invoked when AST is parsed. In our case, we should look for loops in any function. Clang has another class for finding such loops: *clang::ast_matchers::MatchFinder::MatchCallback*. It is a callback function when the matching node is successfully found in the AST.

Thus, to transverse all the AST operations, the *RecursiveASTVisitor*, *MatchFinder*, and *TranslationUnitDecl* classes are all used for building the tool which are described in the following steps:

1. Firstly, AST is generated from an input source, then it is passed to an implementation of another interface. When writing the tool-based libTooling, the entry point of the AST is *FrontendAction* which is the class that gives specific actions for execution. Clang provides the *ASTFrontendAction* interface to run tools over the AST which responsible for executing the action. Thus, we can extend *ASTFrontendAction* to implement our own logic of transformation.

2. Next, a method to detect part of the AST is implemented which is used for handling statements and function declarations contained loops to transform. This method is basically done by implementing *RecursiveASTVisitor* interface to pull out the relevant information from the AST. Moreover, it is possible to gain AST's information via *MatchFinder* as well.

3. Then, an desired operation is applied to the AST. This method is done by the implementation of *ASTConsumer* interface which is used to specify modification on the AST. Finally, *Rewriter* interface is implemented to modify the input source code. It is also responsible for code generation task to generate code back to high-level source language.

The design briefly shows how source-to-source transformation based-LibTooling works as a standalone tool. First, we perform an analysis of the AST and find out where to apply transformations, then the *Rewriter* is used to apply transformations. We implement this tool as a preprocessor that processes source code to transform specific loops before compilation. Figure 4.2 and 4.3 represent the way source-to-source transformation performs the action at the preprocessing stage.

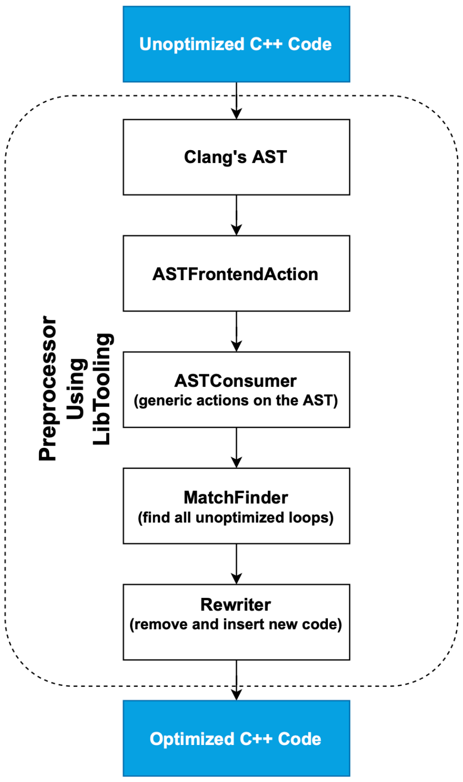
Unoptimized C++ Code
Preprocessor Using LibTooling
Clang's AST
ASTFrontendAction
ASTConsumer (generic actions on the AST)
MatchFinder (find all unoptimized loops)
Rewriter (remove and insert new code)
Optimized C++ Code

Figure 4.2: The high-level architecture of the source-to-source transformation tool using Clang's libraries.

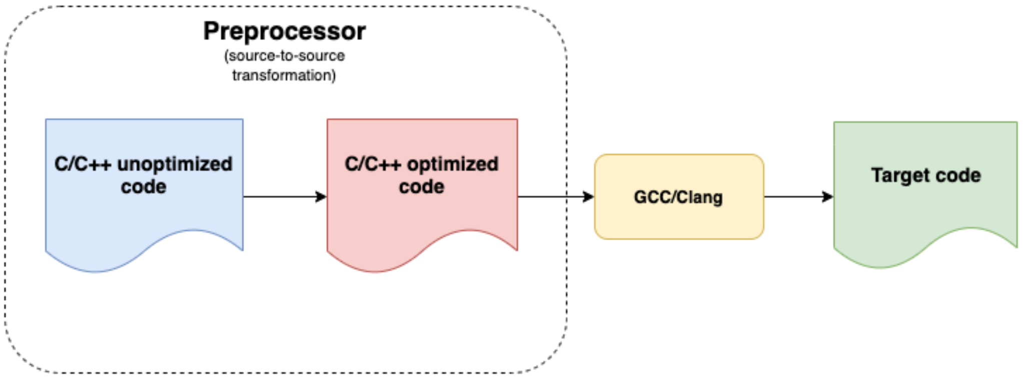

Figure 4.3: The full sequence of steps when compiling a C/C++ program.

4.4 System Implementation

We have provide system implementation details in appendix part. This section just shortly tells the step of the implemenation below:

1. Loop matches
2. AST traversal
3. AST operation
4. Code generation

4.5 Performance Analysis

There are three different tools for performance analysis which are the CMS framework, IgProf, and Callgrind. CMSSW provides timing and memory measurement tools that allow us to execute to measure memory usage and time consumption. However, IgProf is used for providing a precise level of detail. The use of IgProf is mainly in performance mode to measure the time of source transformation. Moreover, Callgrind is used for a very precise executed instruction counting.

Chapter V

RESULTS

5.1 Preliminary Results

Most compilers contain command-line options to control code generation by selecting a sequence of compiler optimizations that aim to maximize performance or to reduce code size as an alternative. Usually, CMSSW is compiled using the GNU GCC compiler by default with level two optimization (*-O2*). The *-O* level option tells GCC to turn on compiler optimization when the specified value of level is in effect. With *gcc -O2*, the compiler will explicitly invoke level two optimization, which improves the performance of the output binary, while avoiding numerical accuracy issues.

From evaluating optimizations that apply to several loops in the **DataFormats/ TrackReco** and the **DataFormats/Candidate** package for reconstructed data. The current packages consist of 35 loops targeting the 3 different loop optimization discussed in the previous section. Table 5.1 represents an overview of the loop optimizations and the number of test programs that target these optimizations.

Loop optimization	**Number of an optimized pattern**
Loop unrolling	25
Index set splitting	9
Loop reordering	1

Table 5.1: The number of benchmark programs that can be applied with loop optimizations.

Function name	**Initialization**	**Loop optimizations (Skewing + Splitting)**
fillCovariance	49,617,744	13,250,193
TrackExtra	24,437,033	20,044,796
TrackBase	11,768,945	7,108,025
VertexCompositePtrCandidate	11,995,424	2,249,142
VertexCompositeCandidate	62,268	58,788

Table 5.2: Number of instructions executed for a selection of functions from CMSSW_10_2_3 compiled with GCC 7.3.1.

5.1.1 Evaluation

First, we analyze our performance by using Callgrind, a profiling tool, to collect the number of instructions executed. For all events in the result refers to instruction fetch. Table 5.2 represents results for a selection of functions contained loops. The results are divided into two parts: initialization and loop optimizations.

GCC compilation option is used (e.g.,*gcc -O2*) to select fewer optimizations. Nevertheless, manually applying loop optimizations to high-level source code is significantly smaller. As you can see in Table 5.2, the number of instruction executed in total by selected function which calls from caller is significantly reduced. This is because loop optimizations might enable other loop optimizations. In our case, loop optimizations that applied in test programs are designed to let the compiler translate it into platform-specific vector instructions. Other than that, with appropriate data structures, the compiler can generate more instruction-level parallelism by loop unrolling. The unrolled loops use fewer instructions, because they reduce the number jump instruction. Thus, there are fewer loop tests and branches.

Even though GCC's optimization can improve performance in application, there

Optimization description	Initialization	Optimized loops	Speed up
Index set Splitting	125.34	79.67	1.57
Unrolled with directive	122	76.34	1.59
Loop reordering	112	64.67	1.73

Table 5.3: Average execution times of repeatedly selected loop patterns in nanoseconds of CMSSW_10_2_3 compiled with GCC 7.3.1.

are many conditions to consider. In most cases, GCC needs to confirm that there is no loop-carried dependencies, which would prevent consecutive iterations. Thus, loop in a program needs to be restructured to improve loop performance.

Then, we evaluate the performance improvement in terms of timing speed up. Table 5.3 shows the average execution time per event processed has been improved both high-level loop optimizations, which are index set splitting, loop reordering, and low-level loop optimization, which is being done at code generation phase.

5.2 System Integration Testing Results

This section shows the measure of efficiency in our source-to-source transformation tool using test cases from the whole **DataFormats** package which contains many kinds of loops such as nested loop, for, while loops etc.

Our tool is able to parse the whole **DataFormats** that are written in C++ language. Parsing time is not our point of interest, so we will not measure then at this point. The experiment focuses on the performance impact of transformed loops. Also, the correctness of simulation results should preserve as well as the original one. The next section we will discuss on the correctness of physics results.

5.2.1 Correctness

To validate the correctness of our transformations, the result of physics simulation should be the same meaning as original one. In our tool, the transformation algorithm with an underlying polyhedral model generates correct code before building and compiling the whole package. We then build our experiment package from transformed original source code. The result from the Figure 5.1 shows transformed code does not change the simulation of the physics result when compared to the original one. We can see in Figure 5.1 that both versions of the source code give the same result.

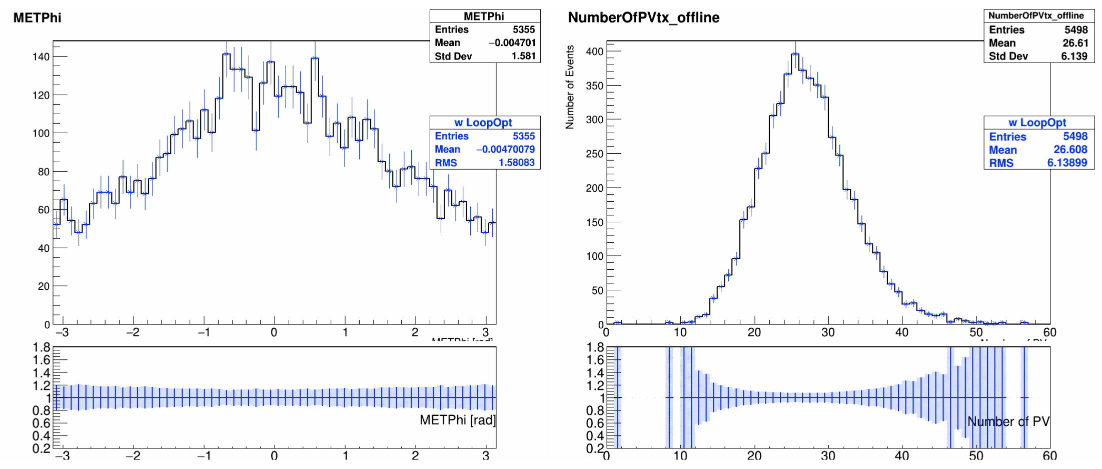

Figure 5.1: Physics result among loop optimization and original code

5.2.2 Performance

The profitability tests are performed where the loop transformation process increases or decreases the performance in the case where time consumption in function contained loops. As one can see from Table 5.4, we get signification time improvement for loop splitting and loop skewing. However, we did not achieve significant speedups for loop skewing alone. Normally, loop skewing aim to reshape the iteration space to find possible parallelism. However, this transformation does not gain benefit from compiler optimization in GCC version 7.4.1. It produces target code that differs significantly in term of performance. Consequently, this gave the result in slower of applying loop skewing alone.

Then, we investigate overall runtime performance through minor change of loop

Function	Initialization	Loop optimizations (Skewing + Splitting)	Speed up
fillCovariance	1.35	1.09	1.24
Vertex::fillCovariance	1.08	0.75	1.44
TrackExtra::TrackExtra	0.75	0.65	1.15
TrackBase::TrackBase	0.30	0.30	1.00
GsfComponent5D::covariance	0.46	0.46	1.00
VertexCompositePtrCandidate::fillVertex Covariance	1.52	1.92	0.77

Table 5.4: Time spent of called function in CMSSW_10_6_8 compiled with GCC 7.4.1 via igprof performance profiling

transformations to our source programs. The amount of improvement is shown in Table 5.5. The result shows that there is slightly runtime improvement after we apply a sequence of loop transformation into our source programs.

Batch	Initialization (-O2)	Loop optimizations (Skewing + Splitting)
Trial 1	106.49	105.34
Trial 2	107.88	103.24
Trial 3	111.08	107.46
average	108.48	105.34

Table 5.5: Time usage measurement per 100 events in seconds on CMSSW_10_2_3.

Chapter VI

SUMMARY & DISCUSSION

6.1 Summary of Findings

To conclude, we study the concepts of existing loop optimization techniques. We present novel code transformations, which resolve into better loop optimizations of large complicated software like CMSSW. We investigate the optimization level two option of GCC compiler in CMSSW. Since level two optimization (*-O2*) is a full set optimization. The compiler will highly generate optimized code. However, we find from the case study that using *-O2* in CMSSW is not an effective solution for the program speed. An important problem is that using *-O2* alone does not normally give the expected performance without invoking other optimization techniques. The reason is that compiler cannot simplify the complexity of loops. Thus, it does not gain any benefit from this option. Consequently, we apply a sequence of loop transformation in out case study. This allows compiler to perform better optimizations. Our measurement is based on speed up and the number of instruction executed.

From the proof of concept, we then build the tool for performing transformation task. Consequently, this experiment shows that the tool is possible to transform nested loops successfully. First, it looks up a loop that matches to our criteria and applies appropriate transformation. Moreover, we showed that our transformation tool can be used for CMSSW scenario.

As a result, the tool can successfully transform the loop into an appropriate form for compiler to optimize it. In this case, GCC translates it into platform-specific vector instructions. Therefore, the loop is vectorized during compilation phase. Moreover, the tool does not affect physics simulation results which is represented in the Figure 5.1. From loop optimization perspective, the source code remains the same in meaning, but the performance gain is slightly different where loop transformation is better in improving performance.

However, the result as shown in 5.5, we cannot say that the amount of improvement is gained from loop transformation since CMSSW is organized in many modules working together. Moreover, the per module time of CMSSW cannot subtract from IgProf profiler.

6.2 Limitation of the tool

There is a limitation to loop skewing and index set splitting. The pattern needs to be aligned with the following form. So, the matcher in the tool can match the parsing tree from source program.

```
for(int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++ {
        statement(i,j);
    }
}
```

Moreover, the body statement inside the loop needs to contain only one statement with a two-dimensional access pattern or else it will meet the exception case of the program. Skewing transformation cannot apply since the body statement is too complicated for the program to handle like the example below.

```
for(index i = 0; i < dimension4D; ++i) {
    for(index j = 0; j <= i; ++j) {
      if( i == dimension || j == dimension ) {
        covariance_[ idx ++ ] = 0.0;
      } else {
        covariance_[ idx ++ ] = err( i, j );
      }
    }
}
```

6.3 Future Work

In the future, we aim to add more transformation possibilities for the same loop pattern. Also, it includes more deeply nested than double loops such as triple nested loops with three different indices variable.

Furthermore, the modern compiler architecture has many embedded optimization techniques inside. Consequently, loop optimization can also perform during the compilation phase via optimizer in the compiler backend pass. Thus, we can add more optimization in both compiler's frontend and backend.

REFERENCES

[1] R. Allen and K. Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers, 2001.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler Transformations for High-Performance Computing," ACM Comput. Surv., vol. 26, p. 345–420, Dec. 1994.

[3] L. Dagum and R. Menon, "OpenMP: an industry standard api for shared-memory programming," Computational Science & Engineering, IEEE, vol. 5, no. 1, pp. 46–55, 1998.

[4] I. J. Bertolacci, M. M. Strout, B. R. de Supinski, T. R. W. Scogland, E. C. Davis, and C. M. Olschanowsky, "Extending OpenMP to Facilitate Loop Optimization," in IWOMP, 2018.

[5] M. Kruse and H. Finkel, "User-Directed Loop-Transformations in Clang," 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 49–58, 2018.

[6] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop Transformations: Convexity, Pruning and Optimization," ACM SIGPLAN Notices, vol. 46, pp. 549–562, 05 2011.

[7] "Loop-Specific Pragmas." [Online]. Available from: `https://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html`. [Accessed July 3, 2020].

[8] "Code Transformation Metadata." [Online]. Available from: `https://llvm.org/docs/TransformMetadata.html#loop-unrolling`. [accessed July 3, 2020].

[9] S. Chatrchyan et al., "The CMS Experiment at the CERN LHC," JINST, vol. 3, p. S08004, 2008.

[10] I.-M. Gregor and A. Straessner, The LHC Detectors, pp. 57–94. Cham: Springer International Publishing, 2015.

[11] A. Petrilli and A. Hervé, "CMS Computing Model: The "CMS Computing Model RTAG"," 12 2004.

[12] L. A. Tuura, V. Innocente, and G. Eulisse, "Analysing CMS software performance using IgProf, OProfile and callgrind," J. Phys. Conf. Ser., vol. 119, p. 042030, 2008.

[13] The CMSSW Documentation Suite, The CMS Offline Workbook, [accessed June 23, 2020]. [Online]. Available from: `https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBook`.

[14] OpenMP Architecture Review Board, "OpenMP application program interface version 5.0," 11 2018.

[15] Y. Lebras, "Code optimization based on source to source transformations using profile guided metrics," July 2019.

[16] J. Hennessy, Computer Architecture : A Quantitative Approach. San Francisco, CA: Morgan Kaufmann Publishers, 2003.

[17] E. Laforest, "Ece 1754 survey of loop transformation techniques," 2010.

[18] M. Griebl, P. Feautrier, and C. Lengauer, "Index set splitting," International Journal of Parallel Programming, vol. 28, pp. 607–631, 2004.

[19] C. Barton, A. Tal, B. Blainey, and J. N. Amaral, "Generalized index-set splitting," in Compiler Construction (R. Bodik, ed.), (Berlin, Heidelberg), pp. 106–120, Springer Berlin Heidelberg, 2005.

[20] H. Razanajato, V. Loechner, and C. Bastoul, "Splitting Polyhedra to Generate More Efficient Code," Jan. 2017.

[21] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques, (Juan-les-Pins, France), pp. 7–16, September 2004.

[22] M. Griebl, C. Lengauer, and S. Wetzel, "Code Generation in the Polytope Model," in In IEEE PACT, pp. 106–111, IEEE Computer Society Press, 1998.

[23] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, PACT '09, (USA), p. 327–337, IEEE Computer Society, 2009.

[24] K. Cooper, K. Mckinley, and L. Torczon, "Compiler-based code-improvement techniques," pp. 3–4, 06 2020.

[25] G. Plotkin, "Call-by-name, call-by-value and the λ-calculus," Theoretical Computer Science, vol. 1, no. 2, pp. 125 – 159, 1975.

[26] D. B. Loveman, "Program Improvement by Source-to-Source Transformation," J. ACM, vol. 24, p. 121–145, Jan. 1977.

[27] M. Ward, "Proving program refinements and transformations," 1986.

[28] S. Malabarba, P. Devanbu, and A. Stearns, "Mohca-java: a tool for c++ to java conversion support," 04 1999.

[29] P. Bhatt, H. Taneja, and K. Taneja, "Ssccj: System for source to source conversion from c++ to java for efficient computing in iot era," 2020.

[30] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," Software: Practice and Experience, vol. 46, 08 2015.

[31] D. Thain, Introduction to Compilers and Language Design. United States: Douglas Thain, 2020.

[32] "New LLVM C Family Front-end." `https://llvm.org/devmtg/2007-05/09-Naroff-CFE.pdf`. Accessed: 2021-07-25.

[33] "Clang: A C Language Family Front-end for LLVM." `http://www.clang.llvm.org`. Accessed: 2021-01-28.

[34] "How To Setup Clang Tooling for LLVM." [Online]. Available from: `https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html`. [Accessed June 22, 2020].

[35] G. Balogh, G. Mudalige, I. Reguly, S. Antao, and C. Bertolli, "Op2-clang: A source-to-source translator using clang/llvm libtooling," pp. 59–70, 11 2018.

[36] E. Bendersky, "Modern source-to-source transformation with tooling," 2014.

[37] G. Balogh, G. Mudalige, I. Reguly, S. Antao, and C. Bertolli, "Op2-clang: A source-to-source translator using clang/llvm libtooling," in 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 59–70, Nov 2018.

[38] O. Krzikalla, K. Feldhoff, R. Müller-Pfefferkorn, and W. E. Nagel, "Scout: A Source-to-Source Transformator for SIMD-Optimizations," in Euro-Par 2011: Parallel Processing Workshops (M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. L. Scott, J. L. Traff, G. Vallée, and J. Weidendorfer, eds.), (Berlin, Heidelberg), pp. 137–145, Springer Berlin Heidelberg, 2012.

[39] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in Cetus users and compiler infrastructure workshop, in conjunction with PACT, vol. 2011, p. 1, Citeseer, 2011.

[40] "OpenC++ Home Page." [Online]. Available from: `https://chibash.github.io/public/opencxx/`. [accessed August 12, 2021].

[41] S. Chiba, "A metaobject protocol for C++," in Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '95, (New York, NY, USA), p. 285–299, Association for Computing Machinery, 1995.

[42] "DMS software reengineering toolkit home page." [Online]. Available from: `http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html`. [Accessed August 10, 2021].

[43] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN '82, (New York, NY, USA), p. 120–126, Association for Computing Machinery, 1982.

[44] "OProfile home page." [Online]. Available from: `https://oprofile.sourceforge.net`. [Accessed July 10, 2020].

[45] "Intel VTune Performance Analyzer home page." [Online]. Available from: `https://software.intel.com/en-us/vtune`. [Accessed July 10, 2020].

[46] G. Eulisse and L. Tuura, "IgProf profiling tool," 2005.

[47] L. Tuura, V. Innocente, and G. Eulisse, "Analysing cms software performance using igprof, oprofile and callgrind," Journal of Physics: Conference Series, vol. 119, p. 042030, 07 2008.

[48] J. Weidendorfer, "Sequential Performance Analysis with Callgrind and KCachegrind," in Tools for High Performance Computing (M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, eds.), (Berlin, Heidelberg), pp. 93–113, Springer Berlin Heidelberg, 2008.

[49] "KCachegrind home page." `https://sourceforge.net/projects/kcachegrind`. Accessed: 2021-01-28.

Appendix I

RELATED SOFTWARE SOURCE CODE

A.1 CMakeList

This given list below is CMakeLists for the standalone clang tool.

```
cmake_minimum_required (VERSION 3.0)

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
project (S2S)


############################################################
#    base
############################################################

set(CMAKE_CONFIGURATION_TYPES "Debug;Release" CACHE STRING "Configs" FORCE)
set(CMAKE_SUPPRESS_REGENERATION TRUE)

############################################################
#    setting Project informations
############################################################

# Clang libraries
set(LIBRARY_LIST clangFrontend  clangSerialization clangDriver
clangParse clangRewriteFrontend clangStaticAnalyzerFrontend clangSema)
set(LIBRARY_LIST ${LIBRARY_LIST} clangAnalysis clangEdit clangAST
clangLex clangBasic clangTooling clangRewrite clangASTMatchers clangToolingCore)
set(COMPONENT_LIST mcparser bitreader support mc option)

############################################################
#    generate makefiles
############################################################
```

```
find_package(LLVM REQUIRED CONFIG)
message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}")
message(STATUS "Using LLVMConfig.cmake in: ${LLVM_DIR}")

include_directories(${LLVM_INCLUDE_DIRS})
message(STATUS "Found LLVM Include ${LLVM_INCLUDE_DIRS}")
include_directories(${CLANG_INCLUDE_DIRS})
message(STATUS "Found Clang Include ${LLVM_INCLUDE_DIRS}")

if(LLVM_BUILD_MAIN_SRC_DIR)
  include_directories(${LLVM_BUILD_MAIN_SRC_DIR}/tools/clang/include)
  include_directories(${LLVM_BUILD_BINARY_DIR}/tools/clang/include)
endif()
link_directories(${LLVM_LIBRARY_DIRS})
add_definitions(${LLVM_DEFINITIONS})

add_definitions(
-D__STDC_LIMIT_MACROS
-D__STDC_CONSTANT_MACROS
)

add_executable(clang-skewing skewing.cc)
add_executable(clang-splitting splitting.cc)

foreach(exec_name
    clang-skewing
    clang-splitting
   )

  if ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    foreach(link_lib IN LISTS LIBRARY_LIST)
      target_link_libraries(${exec_name} optimized ${link_lib})
      target_link_libraries(${exec_name} debug     ${link_lib})
    endforeach()
  else()
    target_link_libraries(${exec_name} ${LIBRARY_LIST})
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14 -Wno-unused-parameter
    -fno-strict-aliasing -fno-exceptions -fno-rtti")
    #set(CMAKE_EXE_LINKER_FLAGS "-static -static-libgcc -static-libstdc++")
  endif()

  #llvm_map_components_to_libnames(llvm_libs ${COMPONENT_LIST})
  #target_link_libraries(${exec_name} ${llvm_libs})
  target_link_libraries(${exec_name}
    LLVMX86AsmParser # MC, MCParser, Support, X86Desc, X86Info
    LLVMX86Desc # MC, Support, X86AsmPrinter, X86Info
    LLVMX86AsmPrinter # MC, Support, X86Utils
    LLVMX86Info # MC, Support, Target
    LLVMX86Utils # Core, Support
    LLVMipo
    LLVMScalarOpts
    LLVMInstCombine
    LLVMTransformUtils
    LLVMAnalysis
    LLVMTarget
    LLVMOption # Support
    LLVMMCParser # MC, Support
    LLVMMC # Object, Support
    LLVMObject # BitReader, Core, Support
    LLVMBitReader # Core, Support
    LLVMCore # Support
    LLVMSupport
  )
endforeach()

message(STATUS "User selected libraries = ${LIBRARY_LIST}")
message(STATUS "User selected components = ${COMPONENT_LIST}")
message(STATUS "    = ${llvm_libs}")
```

Listing A.1: CMakeList for building Clang libTooling

This given list below is to generate a compilation database for CMSSW.

```
cmake_minimum_required(VERSION 3.0.0)

project(CMSSW)

# Bring headers into the project
include_directories(
     "/work/home/tploensin/CMSSW_10_6_8_patch1_New1/src"
     "/work/app/cms/slc7_amd64_gcc700/cms/cmssw-patch/CMSSW_10_6_8_patch1/src"
     "/work/app/cms/slc7_amd64_gcc700/lcg/root/6.14.09-pafccj3/include"
     "/work/app/cms/slc7_amd64_gcc700/external/pcre/8.37-pafccj/include"
     "/work/app/cms/slc7_amd64_gcc700/external/boost/1.67.0-pafccj/include"
```

```
    "/work/app/cms/slc7_amd64_gcc700/external/bz2lib/1.0.6-pafccj/include"
    "/work/app/cms/slc7_amd64_gcc700/external/clhep/2.4.0.0-pafccj/include"
    "/work/app/cms/slc7_amd64_gcc700/external/gsl/2.2.1-nmpfii2/include"
    "/work/app/cms/slc7_amd64_gcc700/external/libuuid/2.22.2-pafccj/include"
    "/work/app/cms/slc7_amd64_gcc700/external/tbb/2019_U3-pafccj/include"
    "/work/app/cms/slc7_amd64_gcc700/external/xz/5.2.2-pafccj/include"
    "/work/app/cms/slc7_amd64_gcc700/external/zlib-x86_64/1.2.11-pafccj/include"
    "/work/app/cms/slc7_amd64_gcc700/external/md5/1.0.0-pafccj/include"
    "/work/app/cms/slc7_amd64_gcc700/external/OpenBLAS/0.3.5-nmpfii2/include"
    "/work/app/cms/slc7_amd64_gcc700/external/tinyxml2/6.2.0-pafccj/include"
)

# Export compile commands
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
set(CMAKE_CXX_STANDARD 17)

file(GLOB source
    "*.cc"
    "*.cpp"
    "DataFormats/*/*.cc"
)

add_executable(Server ${source_files})
```

Listing A.2: CMakeList for building compilation database in CMSSW

A.2 Python scripting

The python script below is used for running the tool for CMSSW project.

```
import sys
import getopt
import subprocess
import os
import glob

basepath = '/work/home/tploensin/CMSSW_10_6_8_patch1/src/DataFormats'

path_executable =
'/work/home/tploensin/CMSSW_10_6_8_patch1/src/clang-transform/build/'

path_compile_commands =
'/work/home/tploensin/CMSSW_10_6_8_patch1/src/build/compile_commands.json'

exclude_path = ['/work/home/tploensin/CMSSW_10_6_8_patch1/src/DataFormats/Provenance',
    '/work/home/tploensin/CMSSW_10_6_8_patch1/src/DataFormats/ParticleFlowReco',
    '/work/home/tploensin/CMSSW_10_6_8_patch1/src/DataFormats/ParticleFlowCandidate']


def get_target_file():
    file_list = []
    for dir_name in os.listdir(basepath):
        dir_path = os.path.join(basepath, dir_name)
        if not os.path.isdir(dir_path):
            continue
        # print(dir_path)
        for sub_dir_path in os.listdir(dir_path):
            sub_path = os.path.join(basepath, dir_name, sub_dir_path)

            if (sub_dir_path == 'src'):
                for src_name in os.listdir(sub_path):
                    path = os.path.join(sub_path, src_name)
                    compilation_db = '-p ' + path_compile_commands

                if (src_name.endswith('.cc')):
                    file_list.append(path + ' ' + compilation_db)
    return file_list


def run(argv):
    inputfile = ''
    outputfile = ''
    try:
        opts, args = getopt.getopt(argv, "hi:o:", ["ifile=", "ofile="])
    except getopt.GetoptError:
        print('test.py -i <inputfile> -o <outputfile>')
        sys.exit(2)
    for opt, arg in opts:
        if opt == '-h':
            print('test.py -i <inputfile> -o <outputfile>')
            sys.exit()
        elif opt in ("-i", "--ifile"):
            inputfile = arg
        elif opt in ("-o", "--ofile"):
            outputfile = arg

    for file in get_target_file():

        print("Process: " + file)
        cmd = path_executable + inputfile + ' ' + file + ' ' + outputfile
        popen = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True)
        popen.wait()
        popen.stdout.read()
        print("Finished! \n")


if __name__ == "__main__":
    run(sys.argv[1:])
```

Listing A.3: Python script to process the whole CMSSW's *dataformats* package

A.3 Source-to-source transformation tool

This section provides an interface and implementation for the source-to-source translator. The tool contains below list.

- commonAction
- skewingTool
- splittingTool
- main

Next, the commonAction part is entry that the source file is comsumed by the AST-Conusmer of Clang's tooling. The ToolName mentioned in the code can be replace by your specific action.

```
#include "clang/Frontend/FrontendAction.h"

#include "tool.h"

class Action : public clang::ASTFrontendAction {
public:
  using ASTConsumerPointer = std::unique_ptr<clang::ASTConsumer>;

  Action(bool DoRewrite, const std::string &RewriteSuffix)
      : DoRewrite(DoRewrite), RewriteSuffix(RewriteSuffix) {}

  ASTConsumerPointer CreateASTConsumer(clang::CompilerInstance &Compiler,
                                       llvm::StringRef Filename) override {
    return std::make_unique<ToolName::Consumer>(DoRewrite, RewriteSuffix);
  }

private:
  bool DoRewrite;
  std::string RewriteSuffix;
};
```

Listing A.4: clang::ASTFrontendAction for the tool

Finally, we implement our logic under MatchCallback, because the transformation occurs only if the matcher can find the specific pattern given by a user.

The pattern that we need to specify for the matcher interface is given below.

```
// clang-format off
  const auto Matcher =
      forStmt(isExpansionInMainFile(),
          hasDescendant(
        forStmt(
          hasLoopInit(
            declStmt(hasSingleDecl(varDecl().bind("initVar")))),
              hasIncrement(unaryOperator(hasOperatorName("++"),
                hasUnaryOperand(declRefExpr(to(varDecl().bind("incVar")))))),
                  hasCondition(binaryOperator(hasOperatorName("<="),
                    hasLHS(ignoringParenImpCasts(declRefExpr(
                      to(varDecl().bind("condVar"))))),
                    hasRHS(expr()))))
            .bind("inner")))
          .bind("outer");
  // clang-format on

  MatchFinder.addMatcher(Matcher, &Handler);
```

Listing A.5: Matcher expression in term of Clang's Tooling

The equivalent expression to the high-level is:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j <= i; j++) {
      statement(i, j)
  }
}
```

Next, the action for splitting needs to implement over the MatchResult from the matcher. The list is given below.

```
// Runs the MatchHandler's action.
void run(const MatchResult &Result) {
  auto &Context = *Result.Context;
  /* Loop transformation algorithm start here... */
  const auto &Stmt = Result.Nodes.getNodeAs<clang::ForStmt>("outer");
  assert(Stmt != nullptr);

  clang::CharSourceRange forStmtRange = clang::CharSourceRange::getTokenRange(
      Stmt->getLocStart(), Stmt->getLocEnd());

  std::string Str =
      clang::Lexer::getSourceText(forStmtRange, Context.getSourceManager(),
                                  Context.getLangOpts())
          .str();

  if (Str.find(DIMENSIONS) != std::string::npos) {
    transformSplitting(Str, Stmt, DIMENSIONS, Context);
  }

  if (Str.find(DIMENSIONS_4D) != std::string::npos) {
    transformSplitting(Str, Stmt, DIMENSIONS_4D, Context);
  }

  if (Str.find(DIMENSIONS_5) != std::string::npos) {
    transformSplitting(Str, Stmt, DIMENSIONS_5, Context);
  }
}

void transformSplitting(std::string loopCondition, const clang::ForStmt *stmt,
                        std::string dimension, clang::ASTContext &Context) {
  findAndReplaceAll(loopCondition, dimension, "< 2;");
  const auto BeginLocationStmt = stmt->getBeginLoc();
  const auto FixItStmt = clang::FixItHint::CreateInsertion(
      BeginLocationStmt, loopCondition + "\n");

  const auto Init = stmt->getInit();
  const auto EndLocationInit = Init->getLocEnd();
  const clang::SourceRange SourceRange(EndLocationInit, EndLocationInit);
  const auto FixItInit =
      clang::FixItHint::CreateInsertion(EndLocationInit, "+2");

  auto &DiagnosticsEngine = Context.getDiagnostics();

  RewriterPointer Rewriter;
  if (DoRewrite) {
    Rewriter = createRewriter(DiagnosticsEngine, Context);
  }

  const auto ID = DiagnosticsEngine.getCustomDiagID(
      clang::DiagnosticsEngine::Warning,
      "This should probably be transformed via splitting");

  DiagnosticsEngine.Report(BeginLocationStmt, ID).AddFixItHint(FixItStmt);
  DiagnosticsEngine.Report(EndLocationInit, ID).AddFixItHint(FixItInit);

  if (DoRewrite) {
    assert(Rewriter != nullptr);
    Rewriter->WriteFixedFiles();
  }
}
```

Listing A.6: Splitting action implements on Clang's tooling interface

The given list below is a listing of skewing action.

```
void run(const MatchResult &Result) {
  auto &Context = *Result.Context;

  // match inner loop
  const auto &Stmt = Result.Nodes.getNodeAs<clang::ForStmt>("inner");
  assert(Stmt != nullptr);

  const auto Init = Stmt->getInit();
  const auto Cond = Stmt->getCond();
  const auto Body = Stmt->getBody();

  // extract lhs value from loop's condition
  llvm::StringRef ref = clang::Lexer::getSourceText(
      clang::CharSourceRange::getCharRange(Cond->getLocStart(),
                                           Cond->getLocEnd()),
      Context.getSourceManager(), Context.getLangOpts());

  auto LhsCondVar = ref.str().substr(0, 1);

  const auto EndLocationCond = Cond->getLocEnd();
  const auto FixItCond =
      clang::FixItHint::CreateInsertion(EndLocationCond, "2*");
  // FIXME: change hard code to use variable from syntax tree
  const auto EndLocationInit = Init->getLocEnd();
  const clang::SourceRange SourceRange(EndLocationInit, EndLocationInit);
  const auto FixItInit =
      clang::FixItHint::CreateInsertion(EndLocationInit, "+i");

  auto &DiagnosticsEngine = Context.getDiagnostics();

  RewriterPointer Rewriter;
  if (DoRewrite) {
    Rewriter = createRewriter(DiagnosticsEngine, Context);
  }

  const auto ID = DiagnosticsEngine.getCustomDiagID(
      clang::DiagnosticsEngine::Warning,
      "This should probably be transformed via skewing");

  DiagnosticsEngine.Report(EndLocationCond, ID).AddFixItHint(FixItCond);
  DiagnosticsEngine.Report(EndLocationInit, ID).AddFixItHint(FixItInit);
  auto exitCond = false;
  handleBodyStmt(Body, DiagnosticsEngine, exitCond);

  if (DoRewrite && exitCond) {
    assert(Rewriter != nullptr);
    Rewriter->WriteFixedFiles();
  }
}

void handleBodyStmt(const clang::Stmt *stmt,
                    clang::DiagnosticsEngine &DiagnosticsEngine,
                    bool &exitCond) {
  std::string type = stmt->getStmtClassName();

  if (type == "BinaryOperator") {
    auto cs = llvm::dyn_cast<clang::BinaryOperator>(stmt);
    for (auto child : cs->Stmt::children()) {
      if (child == NULL)
        continue;

      handleBodyStmt(child, DiagnosticsEngine, exitCond);
    }
```

```
} else if (type == "CXXOperatorCallExpr") {
  auto cs = llvm::dyn_cast<clang::CXXOperatorCallExpr>(stmt);
  for (auto child : cs->Stmt::children()) {
    if (child == NULL)
      continue;

    handleBodyStmt(child, DiagnosticsEngine, exitCond);
  }
} else if (type == "ImplicitCastExpr") {
  auto cs = llvm::dyn_cast<clang::ImplicitCastExpr>(stmt);
  for (auto child : cs->Stmt::children()) {
    if (child == NULL)
      continue;

    auto childCS = llvm::dyn_cast<clang::DeclRefExpr>(child);
    std::string childType = child->getStmtClassName();

    if (childType == "DeclRefExpr") {
      handleBodyStmt(child, DiagnosticsEngine, exitCond);
      break;
    }
  }
} else if (type == "DeclRefExpr") {
  auto cs = llvm::dyn_cast<clang::DeclRefExpr>(stmt);
  auto varName = cs->getNameInfo().getAsString();

  if (varName == "j") {
    const auto EndLocationCond = cs->getEndLoc();
    const auto FixItCond =
        clang::FixItHint::CreateReplacement(EndLocationCond, "j-i");
    const auto ID = DiagnosticsEngine.getCustomDiagID(
        clang::DiagnosticsEngine::Warning,
        "This should probably be transformed via skewing");
    DiagnosticsEngine.Report(EndLocationCond, ID).AddFixItHint(FixItCond);
    exitCond = true;
  }
} else if (type == "CompoundStmt") {
  auto cs = llvm::dyn_cast<clang::CompoundStmt>(stmt);
  for (auto child : cs->Stmt::children()) {
    if (child == NULL)
      continue;
      handleBodyStmt(child, DiagnosticsEngine, exitCond);
    }
  }
}
```

Listing A.7: Skewing action implements on Clang's tooling interface

Appendix II

LIST OF PUBLICATIONS

B.1 International Conference Proceeding

1. Ploensin, T., Piromsopa, K., & Srimanobhas, N. (2021). Code Transformation Impact on Compiler-based Optimization: A Case Study in the CMSSW. In 2021 11th International Conference on Applied Physics and Mathematics (ICAPM 2021). Shanghai, China. https://doi.org/10.1088/1742-6596/1936/1/012023

Biography

Teerit Ploensin was born in Saraburi on May, 1996. He graduated from Princess Chulabhorn College school and then went to Chulalongkorn University where he received B.Sc in Physics. His field of interest includes various topics in optimization, software engineering, web development, and cloud technology.