CODE OPTIMIZATION FOR PERFECTLY NESTED LOOPS

Vandit Sharma(161225)

Kamal Kant(161210)

Under the supervision of:

DR. RAVINDRA BHATT

Department of Computer Science and Engineering and Information Technology

Jaypee University of Information and Technology, Waknaghat, Solan-173234 Himachal Pradesh

CANDIDATE DECLARATION

I hereby declare that the work presented in this report entitled

‘Code Optimization’ in partial fulfilments of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering submitted in the department of Computer Science and Engineering/ Information Technology, Jaypee University of Information Technology Waknaghat, is an authentic record of my own work carried out over a period of July, 2019 to November, 2019 under the supervision of Dr. Ravindra Bhatt .

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

[illegible]	Kamal
Vandit Sharma , 161225	Kamal Kant , 161210

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

[illegible]

Dr. Ravindra Bhatt

Assistant Professor (Senior Grade)

Computer Science and Engineering/Information Technology

Dated:

ACKNOWLEDGEMENT

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. We would like to extend our sincere thanks to all of them.

We are highly indebted to Dr. Ravindra Bhatt for their guidance and constant supervision and as well as for necessary information regarding the project and also for their support in completing the project.

We would like to express our gratitude towards our parents and Jaypee University of Information Technology for their kind co – operation and encouragement which helped us in completion of the project.

Our thanks and appreciation also go to our colleague in developing the project and people who have willingly helped us out with their abilities.

TABLE OF CONTENTS

FIGURE DETAILS

ABSTRACT-

Until the mid-1980s, numerous compiler journalists thought about optimization as a highlight that ought to be added to the compiler simply after its different parts were functioning commendably. This encouraged a differentiation between debugging compilers and optimizing compilers. An investigating compiler underscored brisk accumulation at the cost of code quality. These compilers didn't altogether improve the code, so a solid correspondence stayed between the source code and the executable code. This rearranged the errand of mapping a run-time mistake to a particular line of source code; henceforth, the term debugging compiler. Interestingly, an advancing compiler centers around improving the running time of the executable code to the detriment of aggregate time. Investing more energy in assemblage regularly delivers better code. Since the optimizer frequently moves tasks around, the mapping from source code to executable code is less straightforward, and investigating is, in like manner, harder. As “risc” processors have moved into the commercial center (and as “risc” usage procedures were applied to “cisc” structures), a greater amount of the weight for improving run-time execution has fallen on compilers. To increment execution, processor engineers have gone to highlights that require more support from the compiler. These incorporate defer spaces on branches, non-blocking memory tasks, expanded utilization of pipelines, and expanded numbers of useful units. Processors have gotten considerably more execution delicate to both high level issues of program design and structure and to low-level subtleties of scheduling and resource allocation. As the exhibition space and gap made by optimization has developed, the requirement for code quality have developed, to the point where optimization has become a normal piece and expected part of present-day compilers.

The normal incorporation of an optimizer, thus, changes nature in which both the front end and the “back-end” work. Optimization further protects the front end from execution concerns. To a degree, this improves the generation of “ir” age in the front end. Simultaneously, optimization changes the code that the back-end forms. Present day optimizers accept that the back end will deal with asset distribution; accordingly, they

ordinarily target an admired machine that has a boundless stockpile of registers, memory, and practical units. This, thusly, has set more weight on the strategies utilized in the compiler's back end.

In the event that compilers are to bear a lot of obligation regarding run-time execution, they should incorporate optimizers. As we will see, the instruments of optimization likewise assume a huge job in the compiler's “back end”. Therefore, it is significant to present optimization and investigate a portion of the issues that it raises before examining the methods utilized in a compiler's back end.

1.1) INTRODUCTION-

The objective ofthe compiler's middle agent, or an “optimizer”, isto change the program made by the front end into a program that figures the equivalent and brings about a superior way. Here, "better" can take on numerous implications. It as a rule suggests quicker code, yet it may infer progressively conservative code.Thedeveloper might need the enhancer to create code that expends less power when it runsor costs less to rununder some model for asset bookkeeping. All of these objectives fall into the domain of optimization. Open doors for streamlining emerge from numerous sources. To make this exchange concrete, consider the wasteful aspects that can emerge in executing source-language reflections. Since the front end deciphers source codeinto “ir” without performing broad investigation of the encompassing setting, assuming any, it will regularly create “ir” that handles the broadest instance of a build. The analyser can decide a portion of the setting by performing extra investigation; but unfortunately, some setting can't be known at compiletime.

A compiler’s middleagent function is to enhance the program made by the front end in such a way that it makes it superior and better. Here, “better” can take various meaning. It may suggest faster speed of execution, or make code progressively conservative code. The developer might need to create code that uses less memory or processor clocks. These objectives fall under the umbrella of code optimization. Code optimization finds at assemble time data about runtime conduct of the program and improve the generated code. Previous model are widely recognized for accelerating the execution of the code as its primary objective. Nowadays , compilers have concentrated on the run timespeed of the arrangedcode. Prominence is given to speed which comes to the determent of room bigger code. This offers a great exchange off in calculation configuration of time versus space.

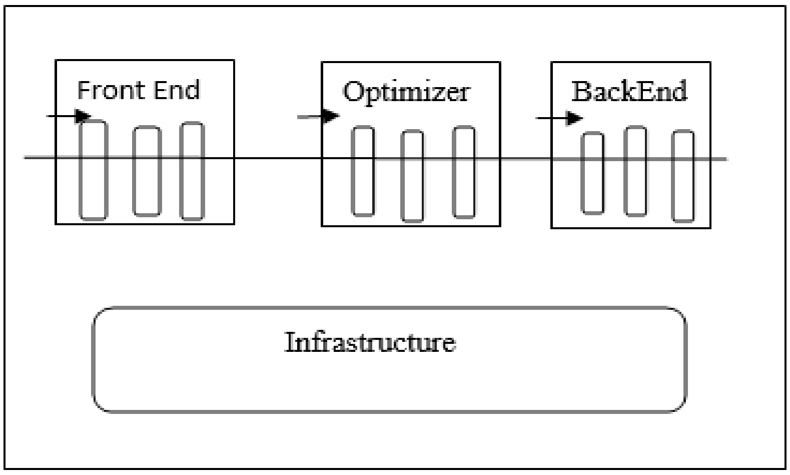


Fig. 1.1 -Schematic diagram of a code optimizer[5]

Think about what happens whenthe compiler has to produce code for an arrayreference, for example, A [i, j]. Without explicit information about A, i, and j and the encircling environment, the compiler must produce thefull articulation for tending to a two-dimensionalarray. This calculation would be

$$\begin{aligned} &\mathit{address}(A) \\ &+ (i - \mathit{low}1(A)) \times (\mathit{high}2(A) - \mathit{low}2(A) + 1) \times \mathit{size} \\ &+ (j - \mathit{low}2(A)) \times \mathit{size} \end{aligned}$$

where address is the run-time address of the primary component of the array, low i(A) furthermore, high i(A) are the lower and upper limits, separately, of A's ith measurement, furthermore, size is the size of a component of A. This calculation is costly. The compiler's capacity to improve this code depends straightforwardly on its investigation of the code and the encompassing setting.

If the array A has been declared locally, with known limits and a recognized type, then the compiler can frequentlyand simply perform large parts of this address computation at compiletime and simply use the resultat run time. For e.g. if we take the following term in concern i.e.

$(high2(A) - low2(A) + 1) \times size$

This can be figured at compiletime and preserved as aconstant at run time (For example, using a load I,). Alongside, other parts of the computation can beimproved. If the compiler is able to identify that the location A[i , j] that occurs inside the loop nest where i and j change in an arranged in order and in well-implied manner, it may substitute some of the integer multiplications inthe calculations with summations and a renovation is renamed as operator strength reduction.

This basic example shows both one inspiration for optimization as well as the essential activity of a code optimizing agent. The objective of codeoptimization is to find, at assemble time, data about the run-time conduct of the program and to utilize that data to improve thecode generatedby the compiler. Improvementcan take numerous structures. The past models all centered around the most widely recognized objective of optimization: accelerating the executionof the incorporated code. For certain applications, be that as it may, the size of the arranged code is significant. Models incorporate code that will be resolved to peruse just memory, wheresize is a fiscal requirement, or code that will be transmitted over a restricted transmission capacity interchanges channel before it executes, where size directly affects time to culmination. Optimization, for theseapplications, should create code that consumes lessspace. In certain circumstances, the client needs to upgrade for other criteria, for example, register use, memory use, control utilization, or reaction to continuous occasions. Generally, advancing compilers have concentrated fundamentally on the run-time speed of the arranged code. The vast majority of our dialog of optimization will bargain withspeed. This prominence on speed frequently comes to the detriment of room bigger code size from reproduced activities. This is a great exchange off in calculation configuration—time versus space. In view of the rising significance of code space and information space in installed and intuitive registering, we will center a portion of the dialog on strategies that either decrease the space prerequisites of the program or, in any event, don't expand them altogether. Optimization is a huge and detailedsubject. This part, and the two that pursue, give a short prologue to the subject. This section lays the basis.

Code optimization is projected to improve thequality of code produced bythecompiler and for this purpose it contains analysis and transformations. It uses various analyses like data flow analysis to help compiler in discovering the opportunities for transformations and the safeties required to apply them efficiently. But analysis is just the prolog part, complier rewrites the code in order to improve it.

Speed of processors has improved in the last decades by a big margin. However, memory speed has been left behind. To speed the gap , memory hierarchies which had cache memories were introduced . Due to this we have an added latency to access data. Many solutions through various means have been sought to reduce the latency. To make full use of the available hardware structure compiler techniques have been developed. Therefore we have two type of transformations – loopand datalayout transformations. Here we work upon perfectly nested loops and ways to improve their locality of reference.

Perfectly nested loops are heart of various applications – imaging, deep learning, multimedia – are often memory intensive. If all the assignment instructions are inside the innermost loop the nested loop is a perfectly nested loops. A deepperfectly nested loop is required by the convolutionkernel. The important and unsolved challenge remains to discover the most efficient way to execute a perfectly nested loop. The main reason for this is that the joint search space of hardwarecombined with the ways to execute the loops is vast.

When we choose a specific way of executing the loops that leadsto a particularexecution method this significantly impacts the computationpatterns which in turn affects the power and performance of execution. This is an important step, because if not done carefully the benefits may as well be negative.

1.1.1)**ENABLING OTHER TRANSFORMATIOMS-**

LOOP ENROLLING-

This technique to enable transformation is one of the oldest one. Compiler imitates the loop's body and manipulates the logic that will be used to control the number of performing iterations. If the body of the loop is replaced by the compiler itself (for example by a factor of 4), it will require only single iteration i.e. quarter the number of evaluations.Let's consider value of n i.e. say 250, and after dividing it by unrolling factor and if result is even, then the loop unrolling is in a simple state.

LOOP UNSWITCHING-

Loop invariant control flow operation are heaved out of a using loop unswitching. In another case if any "if then else" predicate is not variant inside the loop, then "if the else" statements are pulled out of the loop and the loop is rewritten and creates a tailored copy of the loop in the inner side of new "if then else" statements and is valid only for small loop transformations. Loop unswitching allows compiler to alter the bodies of loops which otherwise is a difficult task to achieve, this is an enabling transformation technique. Afterward there is very less control flow in rest of the loops. Unswitching allows execution in fewer branches only and operations to support these branches.

RENAMING-

Above all transformations use reordering and rewriting of the operations in a code. These techniques help the code to achieve a right code shape so tat it can be exposed to the opportunities for optimization.

REDUDANCY ELIMINATION-

LCM shows up for instance of code movement. It broadens the information stream approach spearheaded with accessible articulations to a system that brings together code movement and repetition disposal. Lifting dispenses with indistinguishable activities to lessen code size; this decrease doesn't, ordinarily, decline the quantity of activities that the program executes.

1.2) PROBLEM STATEMENT

Most loops used in real world programming inculcate the use of imperfectly nested loops. For instance codes involving matrix factorization which use three nested loops. They all compute the same result but their performance can vary from machine to machine. Some methods like polyhedral methods cannot be directly applied on them. We can apply loop distribution to transform an imperfect nested loop to a perfect nested loop. However , loop distribution is not always applicable and may result into wrong conversion.

Therefore we face two problems.

1. How to specify transformations of imperfectly nested loops?
2. How to find legal and desirable transformations?

“Iterator-based” approach allows to explore orderings left by the automatic optimizers. But they too work. They are/were bound to produce incorrect results. So in essence we already have algorithms for perfectly nested loops but not for imperfectly nested loops – which is the case most of the times.

The issue is to transform an imperfectly nested loop to perfectly nested without trading off simplicity, meaning of code and also improving the locality of reference. Up till now there were no automatic tools to identify imperfect nested loops. Most of the algorithms present are dependent upon the machine, the data structure and the algorithm used. No literature could tile all the codes automatically.

Since the statement in a loop is executed many times , we need a way of identifying a particular dynamic instance of given statement.

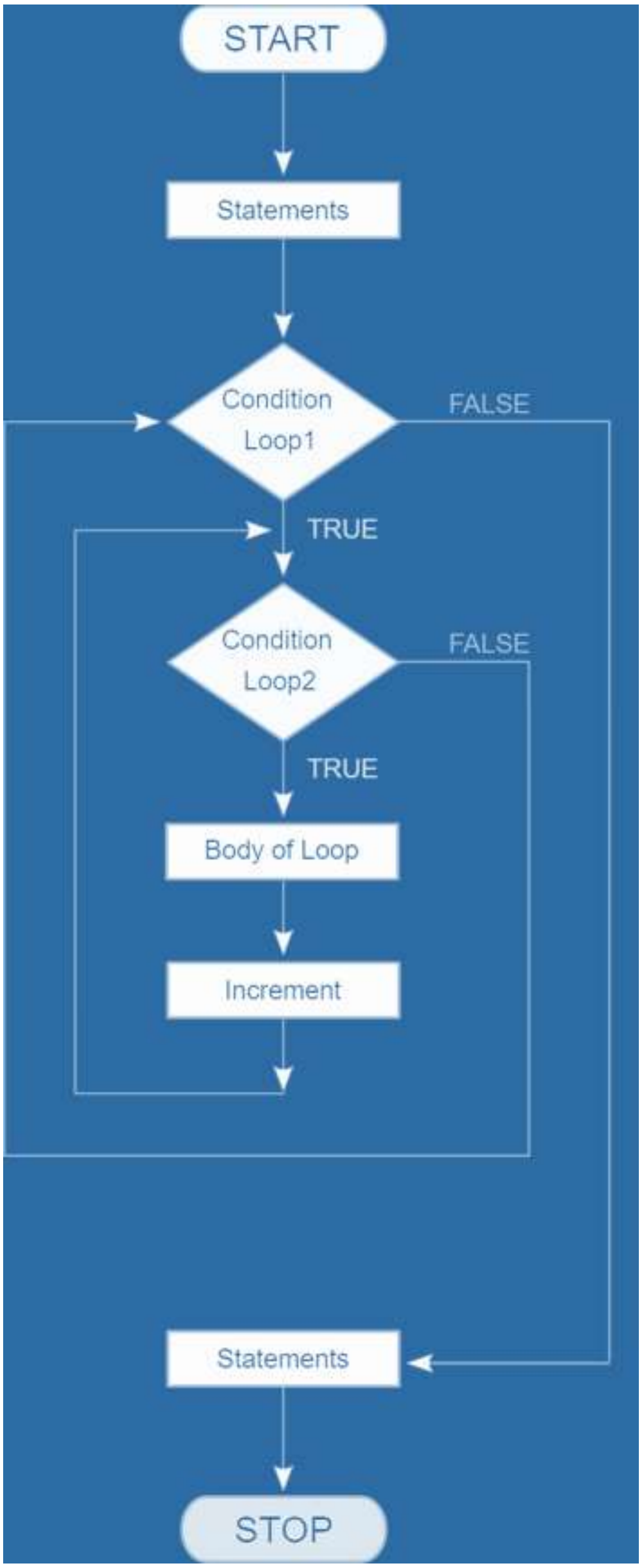


Figure 1.2 Flowchart of a perfectly nested loop

1.3) OBJECTIVES

The things which we will try to achieve are:

1. To evaluate the use of loop re-ordering optimizations.
2. To elucidate the effect(s) of the implemented ordering and examine its advantages.

1.4) METHODOLOGY

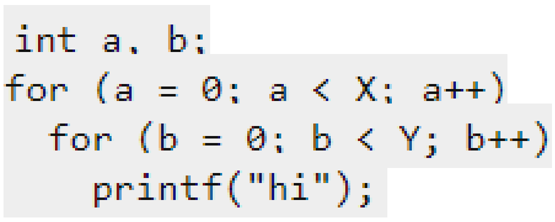
There are various techniques to implement nested loop optimization for imperfectly nested loops. Many papers have attempted to solve various aspects of this issue.

The usual way to solve is to use the technique developed by the community for scheduling statements in loop nests on arrays. It is straight forward to use these mappings as loop transformations.

Its imperative to discuss the polyhedral model as many of the dense array computations use this. This framework can often produce definite portrayals of the arrangement of executions of explanations in the loop.

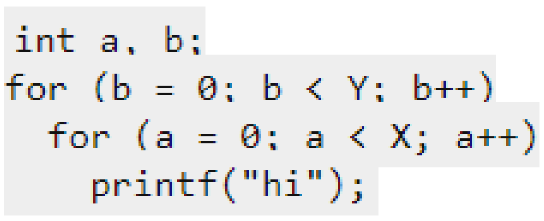
The first thought which occurs is to reduce the number of jumps in a nested loop. This can be achieved by reordering nested loop. This will involve less code jumps. Steve McConnell gave this idea in code complete.

Look at the following figure. The number of jumps in case 1 are less than the jumps in case 2 if X < Y.



```
int a, b;
for (a = 0; a < X; a++)
  for (b = 0; b < Y; b++)
    printf("hi");
```

Figure 1.3 Case 1



```
int a, b;
for (b = 0; b < Y; b++)
  for (a = 0; a < X; a++)
    printf("hi");
```

Figure 1.4 Case 2

This makes the code a little bit faster. Through mathematical induction we can prove for deeper nested loops. Hence while writing nested loops the outermost loop must be the least changing and the innermost the most changing.

Our strategy will involve to assign the statements of a loop to a statement iteration space and map them to a larger iteration space called the product space. It will be the Cartesian product of the two. The functions involve the use of transformations – code sinking and loop fusion. We will see how this framework is used to tile loops and how we can determine functions that enable tiling.

There are various advantages to our approach. Because we use a product space we can easily neglect the syntactic structure of the code. Also we can avoid the search for sequence of transformations as we determine directly the functions that help us in tiling.

1.5) ORGANISATION

There are different ways to achieve our goal. Loop optimization can be either machine dependent or machine independent we have a large range of methods and applications.

Machine Independent Optimization involves loop unrolling and loop unswitching. Though there are only a few algorithms but we still can attain a level of optimizations.

Machine Dependent Optimization involves exploiting hardware features of the system, manage and hide latency and manage bounded machine resources.

1.5.1) APPLICATIONS

TRIANGULAR SOLVE

A triangular matrix in linear algebra is a square matrix. They are used in various numerical analysis as matrix equations are easier to solve.

It is of various types

1. Lower triangular if the entries above the main diagonal are zero
2. Upper triangular if the entries below the main diagonal are zero
3. Diagonal matrix if its both upper and lower triangular

Upper triangular matrices from a subalgebra of the associative algebra of square matrix for a given size.[6]

$$L = \begin{bmatrix} \ell_{1,1} & & & & 0 \\ \ell_{2,1} & \ell_{2,2} & & & \\ \ell_{3,1} & \ell_{3,2} & \ddots & & \\ \vdots & \vdots & \ddots & \ddots & \\ \ell_{n,1} & \ell_{n,2} & \ldots & \ell_{n,n-1} & \ell_{n,n} \end{bmatrix}$$

Figure 1.5 Lower Triangular[6]

$$U = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \ldots & u_{1,n} \\ & u_{2,2} & u_{2,3} & \ldots & u_{2,n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & u_{n-1,n} \\ 0 & & & & u_{n,n} \end{bmatrix}$$

Figure 1.6 Upper Triangular[6]

CHOLESKY DECOMPOSITION OR FACTORIZATION

Of a Hermitian positivedefinitematrix it is a decomposition into the product of lower triangular matrix and its conjugatetranspose.

For numerical solution of Linear equations of the form Ax = b , Cholesky decomposition is used. It is utilized for numerical solidness and prevalent productivity. For computation if applicable it is used in favour of LU decomposition as it involves small error and no strategy for pivoting is necessary.

The computational complexity is O(n^3) in general.[6]

$$\begin{pmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 6 & 1 & 0 \\ -8 & 5 & 3 \end{pmatrix} \begin{pmatrix} 2 & 6 & -8 \\ 0 & 1 & 5 \\ 0 & 0 & 3 \end{pmatrix}$$

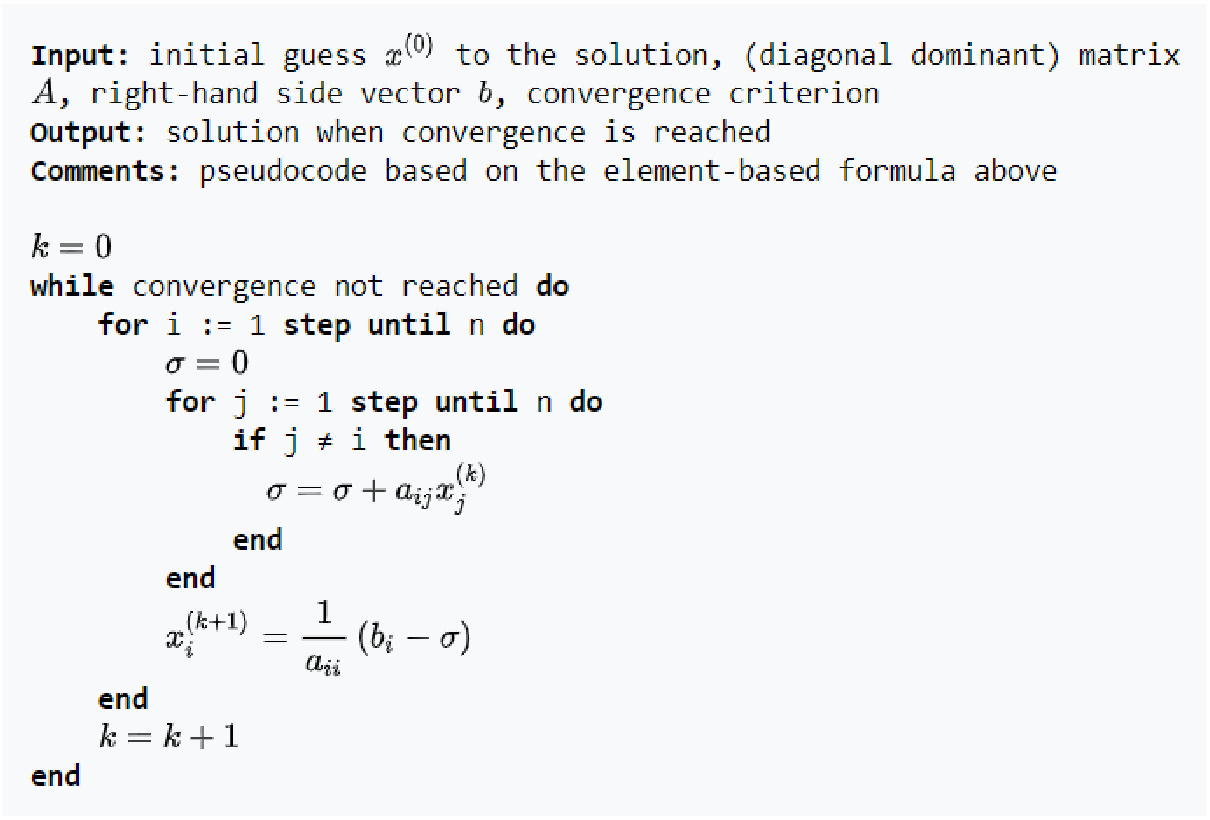
Figure 1.7 Cholesky Decomposition of a Symmetric Matrix[6]

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k} L_{j,k}^*},$$
$$L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k}^* \right) \quad \text{for } i > j.$$

Figure 1.8 Formula for computing Cholesky Decomposition[6]

JACOBI KERNEL

Its is a stencil code to update array elements in view of a specific pattern. Codes of computer simulation involves the use of various stencil codes. Jacobi in particular is used to find solutions of a strictly diagonally system of linear equations in linear algebra. In this each diagonal element is solved and value is stored. This method is repeated until the value converges. It was named after Carl Gustav Jacob Jacobi.[6]



```
Input: initial guess $x^{(0)}$ to the solution, (diagonal dominant) matrix
$A$, right-hand side vector $b$, convergence criterion
Output: solution when convergence is reached
Comments: pseudocode based on the element-based formula above

$k = 0$
while convergence not reached do
    for i := 1 step until n do
        $\sigma = 0$
        for j := 1 step until n do
            if j ≠ i then
                $\sigma = \sigma + a_{ij} x_j^{(k)}$
            end
        end
        $x_i^{(k+1)} = \frac{1}{a_{ii}} (b_i - \sigma)$
    end
    $k = k + 1$
end
```

Figure 1.9 Algorithm for computing solutions using Jacobi method[6]

2.1)LITERATURE SURVEY

Looptransformations are important for compiling high performance code for new era. Previous work has been focused on loops in which all assignment statements are in the inner most loops(Perfectly nested). In real world coding the idea of perfectly nested loops is farfetched. In some cases we can convert imperfectly nested loops to perfectly nested by loopdistribution but this is not always possible.

A goodoptimization of densearray codesdepends upon the execution of nested loops. As many important computer operations operate on nested loops and multidimensional arrays like dynamic programming algorithms, differential equation solvers and linear equation solvers. Aim of loop optimization is to improve locality of reference for most of its data accesses to be done by faster memory.

Numerical Linear Algebra community have written libraries of programs such as the BasicLinear AlgebraSubroutines(BLAS) and LAPACK. They however are only fruitful when linear system or eigen-solvers are needed but are useless to solve partial differential equations.

Compiler community has worked on a technology which does not care about the algorithm in use. It tends to be brought in use with no confinement in the problem domain.[4]

Most of the work done in this field is focused on perfectlynested loops that manipulate arrays. If the same memory locationis accessedby two or more iterations then the loop is said to carry algorithmic reuse.

SGI MIPSPro compilers incorporate tiling which changes the order of the iterations executed and also perform linearloop transformation such as skewing and reverse enable tiling. They help the compilers to produce good code for perfectly nested loops.

Many methods are available to transform imperfectly nested loops to perfectly nested loops in a program. The easiest method is to transformeach maximal perfectlynested loop separately.

SGI MIPSPro compiler adopts more assertive tactic. It follows two steps:

1. Converts an imerfectly nestedloop to perfectly nested loop by applying code sinking loop fusion loopfission.
2. Undertake localityenhancement techniques for the resulting best perfectly nestedloops

Wolf employed erudite heuristics to guide various transformations in the SGI MIPSPro compiler. However it does not perform well like the hand written code in the LAPACK library.

Data shackling was proposed by Kodukula et al to overcome various problems and adversities in compilers. It involved choosing compiler blocks data arrays and setting an order in which they are brought into the cache. When the block is brought into the cache all the statements in it are executed. This approach does not involve tiling of loops.

But again it does not explain how it can be used forJacobi or Gauss-Seidel relaxation codes that make multipletraversals over an array. Pugh and Rosser developed an algorithm called iteration space splicing. It however did not address tiling.[3]

For imperfectly nested loops there are special purpose algorithms proposed in the literature. Example, after a specific sequence of loop transformations Carr has shown that factorization codes can be tiled. Specific programs with a particular structure having a sequence of perfectly nested loops have a technique for tiling relaxation codes like Jacobi proposed by Song and Li. But in matrix factorizations this strategy fails.

Our strategy as developed in the previous frame work is to improve locality of imperfectly nested loops. According to figure each statement is assigned a space called statement iteration space. These areembedded in product space. The productspace is the cartesian product of the individual statement iteration spaces. Functions given in figure depict codesinking and loop fusion that convertimperfectly nested loops to perfectlynested loops.

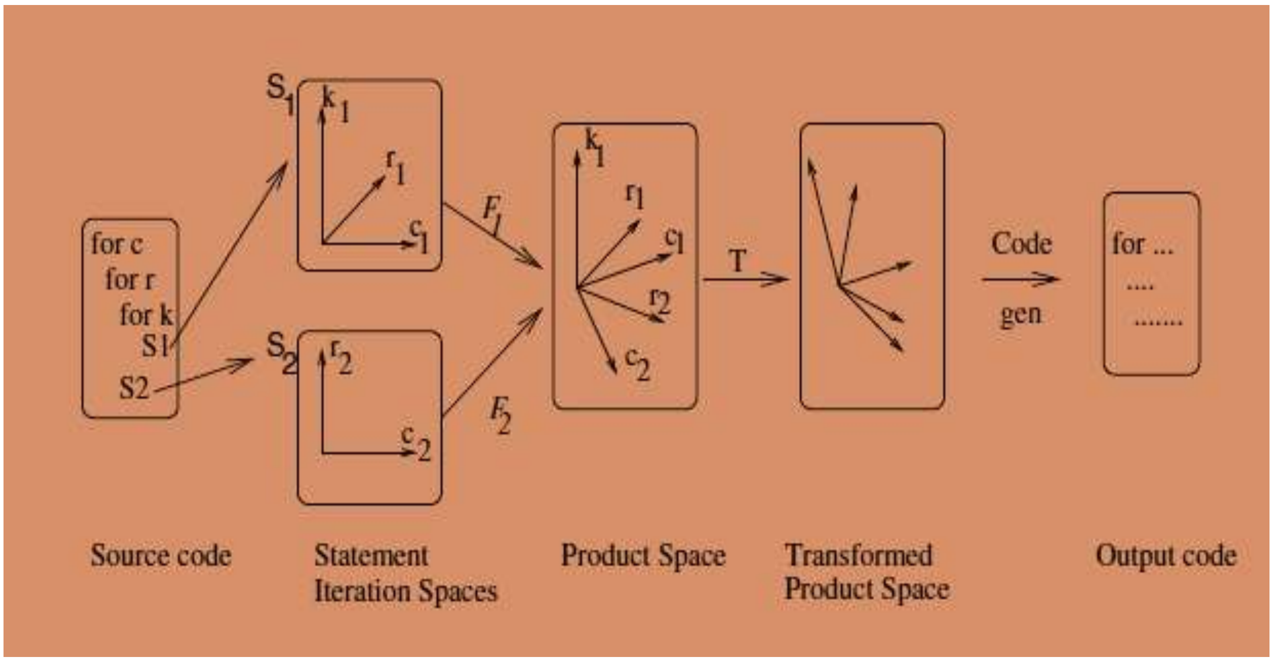


Figure 2.1 Tiling Imperfectly Nested Loops[1]

3.1) SOFTWARE REQUIREMENTS

We need the only basic compilers for our project. We will involve the use of the following compilers and software.

1. Code::Blocks C/C++ version 13.12
2. Clang++
3. GCC
4. Intel Advisor(2020)

Minimum system requirements should be:

1. 1 GB RAM
2. 9 – 58 GB Free hard disk space
3. Basic GPU
4. Intel Pentium or Compatible, 1.6 GHz minimum
5. 1024 X 768 or high resolution monitor
6. Microsoft windows 7,8,8.1(32 bit and 64 bit)

Code::Blocks C/C++

As the implementation is elementary implementation C/C++ has a collection of vast libraries which makes it easier to use. Also its faster than JAVA and Python.

Code::Blocks is a free open source cross platform IDE. It is a well maintained and an efficient software which also has an optional Make support.

Clang++

Clang is used in Objective Cpp, Cpp, C, Objective C as a compiler front end. It is also used for various frame works like OpenCL etc. It reduces memory footprint and also is highly compatible with GCC.

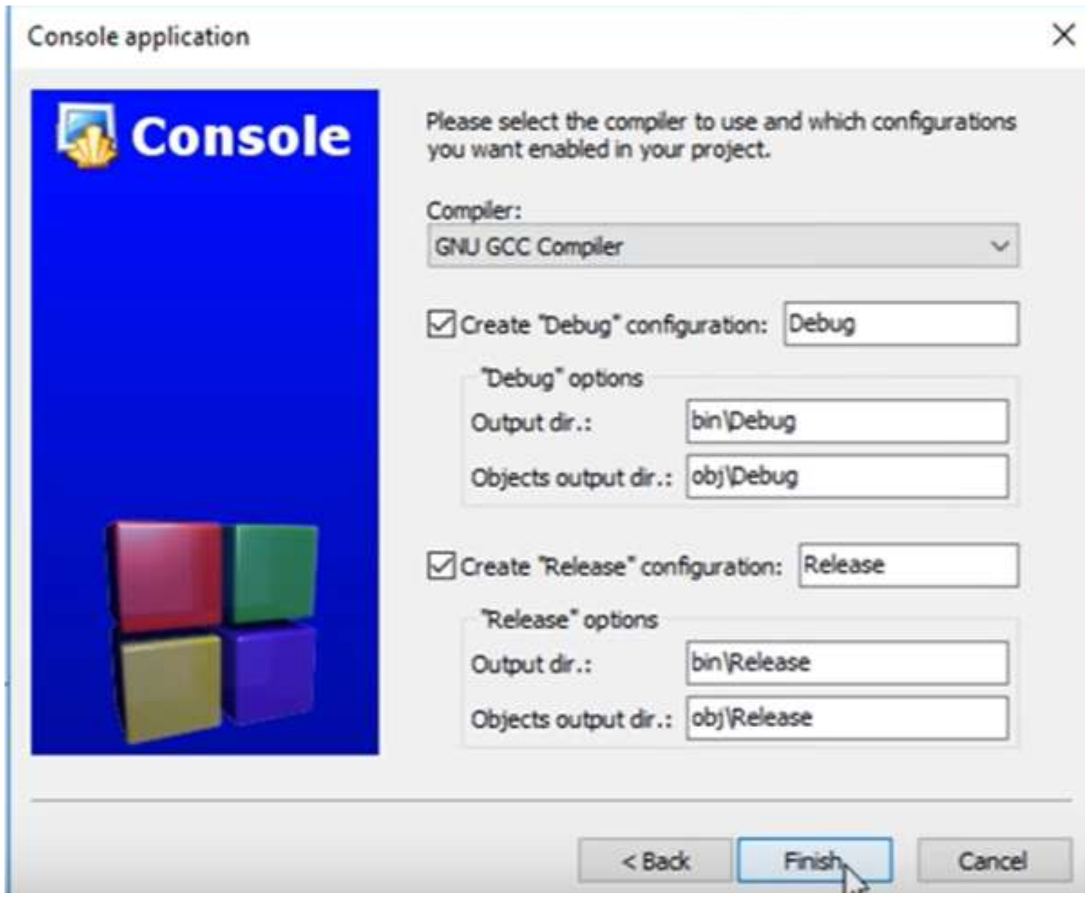
GCC(GNU Compiler Collection)

It is a project undertaken by GNU which supports various languages. It is a key component for various projects using GNU and Linux kernel. Its stable version is 9.2 released on 12 August 2019.It can be used with various architecture set. It is a free software which supports various languages and environments.

Intel Advisor

It is a standalone GUI assistance tool for a SIMD vectorization and shared memory. It is used for software development in C, Cpp, C# and Fortran. It is a freeware profiler tool with support to command line interface.

3.2) MAIN CONFIGURATIONS


Console application

Please select the compiler to use and which configurations you want enabled in your project.

Compiler: GNU GCC Compiler

☑ Create "Debug" configuration: Debug

"Debug" options

Output dir.: bin\Debug

Objects output dir.: obj\Debug

☑ Create "Release" configuration: Release

"Release" options

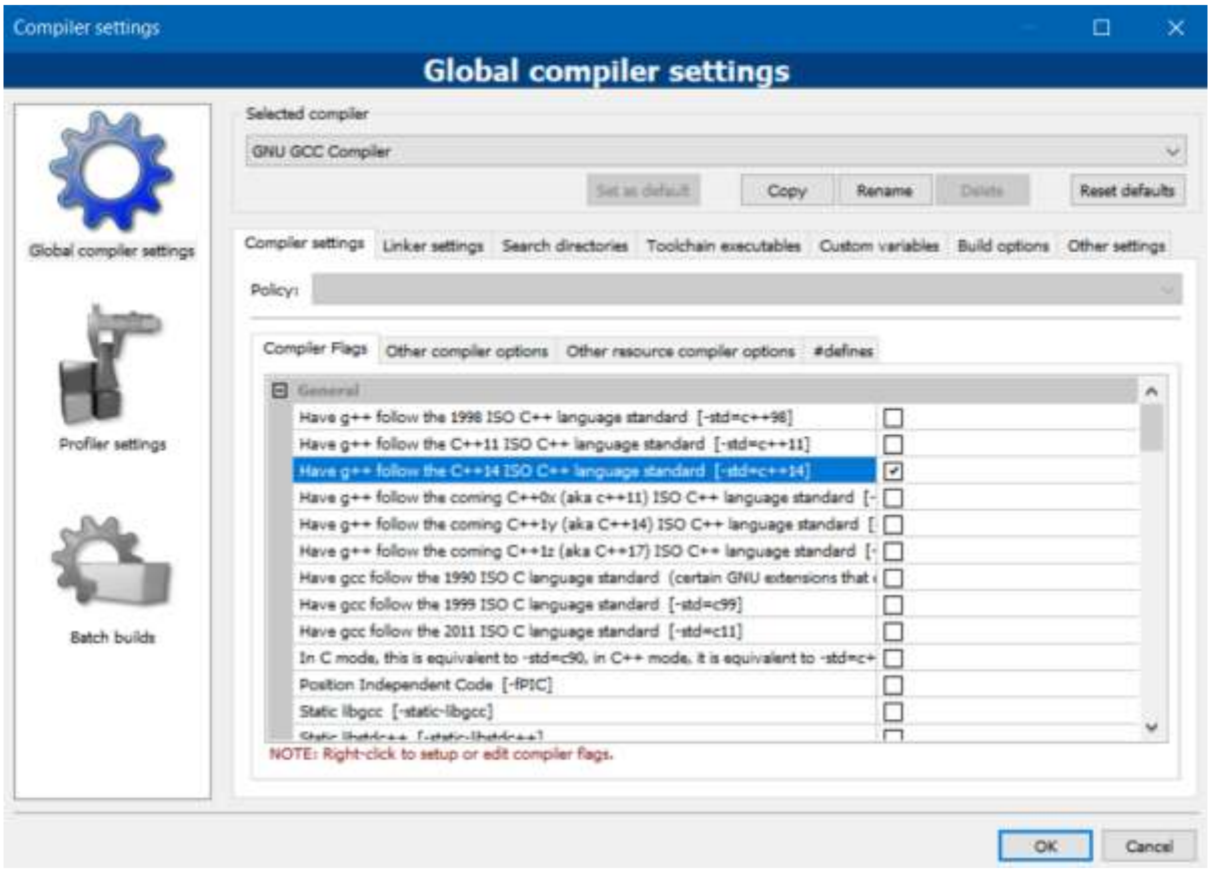
Output dir.: bin\Release

Objects output dir.: obj\Release

Figure 3.1 Console Application Window

In the dialog given above we have option to select our desired compiler. This option appears before creating a project. Make sure to select the GNU GCC Compiler. Do not change the Debug options. Select the Create Debug Configuration checkbox and create Release configuration checkbox.

Click on Finish when done.



GNU GCC Compiler

- [ ] Have g++ follow the 1998 ISO C++ language standard [-std=c++98]
- [ ] Have g++ follow the C++11 ISO C++ language standard [-std=c++11]
- [✓] Have g++ follow the C++14 ISO C++ language standard [-std=c++14]
- [ ] Have g++ follow the coming C++0x (aka c++11) ISO C++ language standard [-
- [ ] Have g++ follow the coming C++1y (aka C++14) ISO C++ language standard [
- [ ] Have g++ follow the coming C++1z (aka C++17) ISO C++ language standard [-
- [ ] Have gcc follow the 1990 ISO C language standard (certain GNU extensions that
- [ ] Have gcc follow the 1999 ISO C language standard [-std=c99]
- [ ] Have gcc follow the 2011 ISO C language standard [-std=c11]
- [ ] In C mode, this is equivalent to -std=c90, in C++ mode, it is equivalent to -std=c+
- [ ] Position Independent Code [-fPIC]
- [ ] Static libgcc [-static-libgcc]
- [ ] Static libstdc++ [-static-libstdc++]

NOTE: Right-click to setup or edit compiler flags.

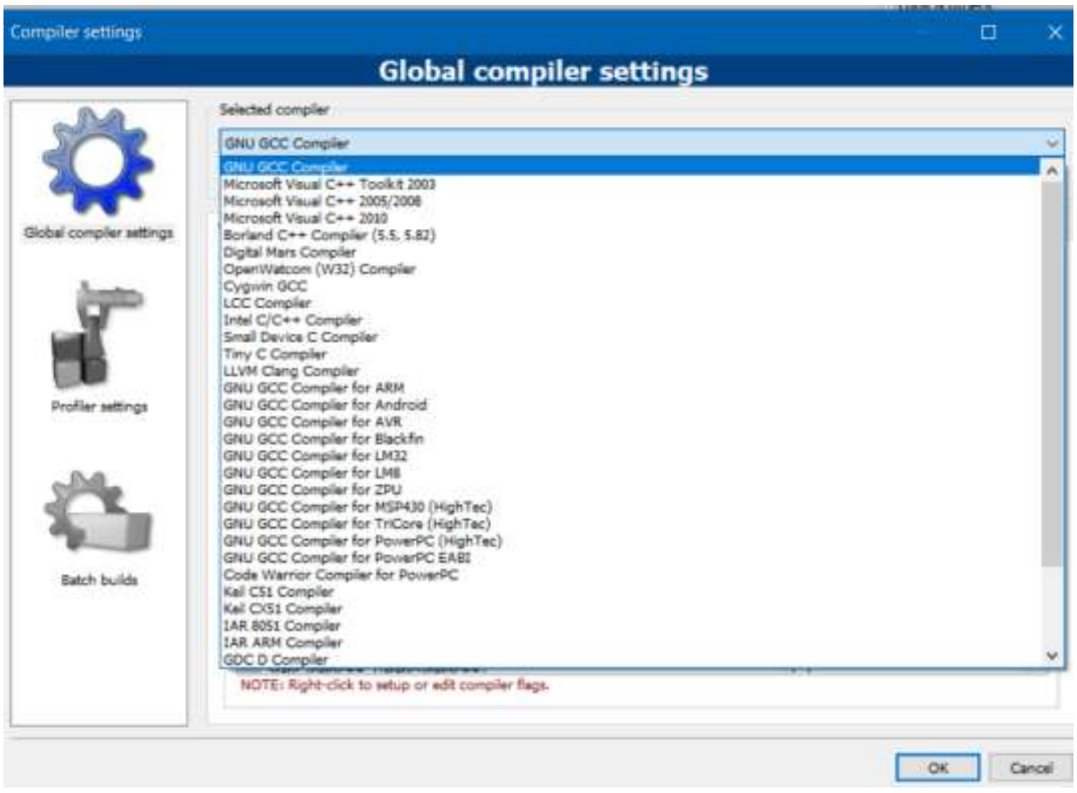
Figure 3.2 Compiler settings window

Go to Code::Blocks Settings Tab > Compiler Menu Option. Global compiler settings window will open. In this window we can see various options regarding the settings of the compiler. We have three main tabs which are:

1. Global Compiler Settings
2. Profiler Settings
3. Batch Builds

Each have there own default settings which can be changed according to the user. In Global Compiler settings there are various tabs related to the settings of the compiler. We have the tab compiler flags which allows to activate the components of the c++ compiler.

Also we have an option to select different available compilers.



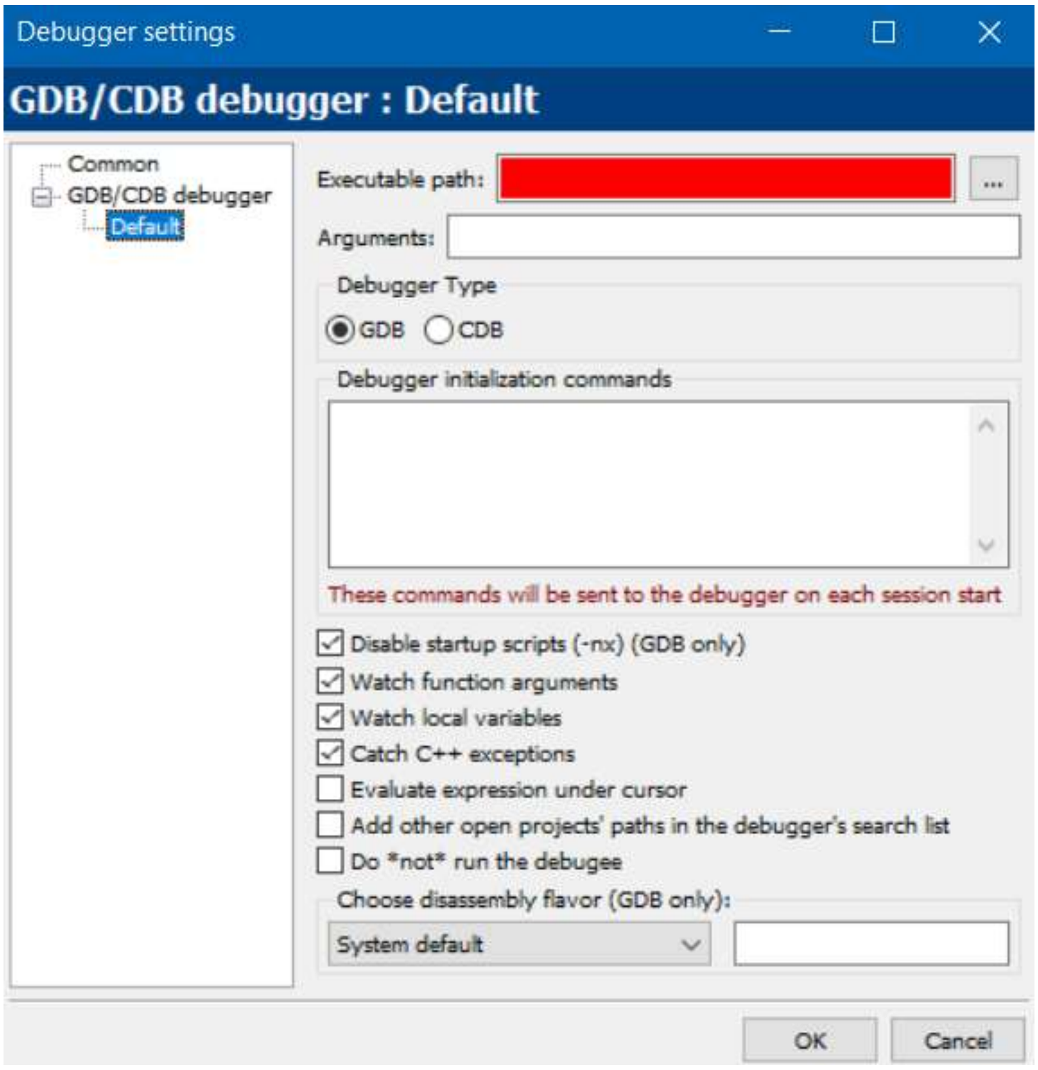
Selected compiler: GNU GCC Compiler

NOTE: Right-click to setup or edit compiler flags.

Figure 3.3 Selected Compiler dropdown menu

From this down you can select any of the available compilers for use. In our case we will be using the default GNU GCC Compiler.

Next we are going to see the debugger settings.



Debugger settings

GDB/CDB debugger : Default

Executable path: 🟥

Arguments:

Debugger Type

[✓ GDB] CDB

Debugger initialization commands

These commands will be sent to the debugger on each session start

- [✓] Disable startup scripts (-nx) (GDB only)
- [✓] Watch function arguments
- [✓] Watch local variables
- [✓] Catch C++ exceptions
- [ ] Evaluate expression under cursor
- [ ] Add other open projects' paths in the debugger's search list
- [ ] Do *not* run the debugee

Choose disassembly flavor (GDB only): System default

Figure 3.4 Debugger Settings Window

Ensure the debugger settings looks exactly like the figure given above. These settings are extremely crucial and need not be fidgeted with.

4)SYSTEM DEVELOPMENT

4.1) Mathematical Analysis

The first thought which occurs is to reduce the number of jumps in a nested loop. This can be achieved by reordering nested loop. We need to decide which loop should come out and which should not. This will involve less code jumps.

Suppose we have a nested loop structure. Let A,B,C,…N be number of times each a nested loop runs. Therefore in nested form, segment of innermost loops runs $A + A * B + A * B * C + \ldots * N$.

Now we have to prove that $A+A*B+A*B*C\ldots N\ldots$. Is smallest for $A<B<C\ldots<N$ (1)

We use Mathematical Induction,

For p(0):

Let $X=10,Y=5$;

$X+X*Y=60$

According to the problem statement via Mathematical Induction

$X=A,Y=B$

Hence,

$Y+Y*X=55$ hence, p(0) is true for $Y < X$

For p(1):

$X=600,Y=400$

$X+X*Y=2,40,600$

$Y+Y*X=2,40,400$

For Larger N=5 i.e. p(k)

Assume, $X=100,Y=80,Z=95,P=85,Q=75$

For given ordering the code runs 4,910,368,100 times.

For new ordering(Q,Y,P,Z,X) the code runs 4,893,966,075 times.

Difference after optimization is 16,402,025

Hence statement 1 is correct and proved.

Therefore the difference increases for larger values of N.

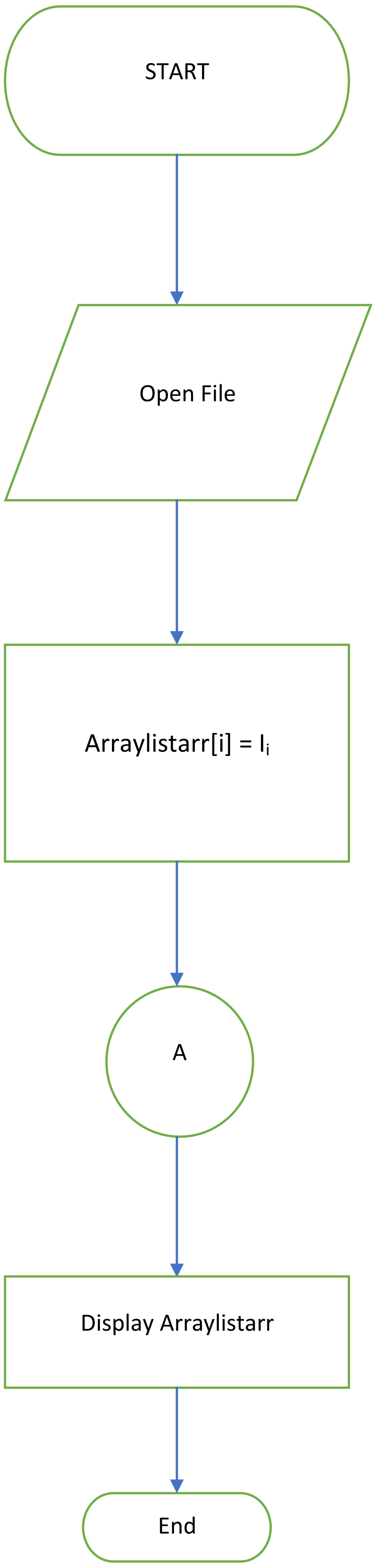
Thus output will consist of the best possible ordering of the nested loops involved in the users code.

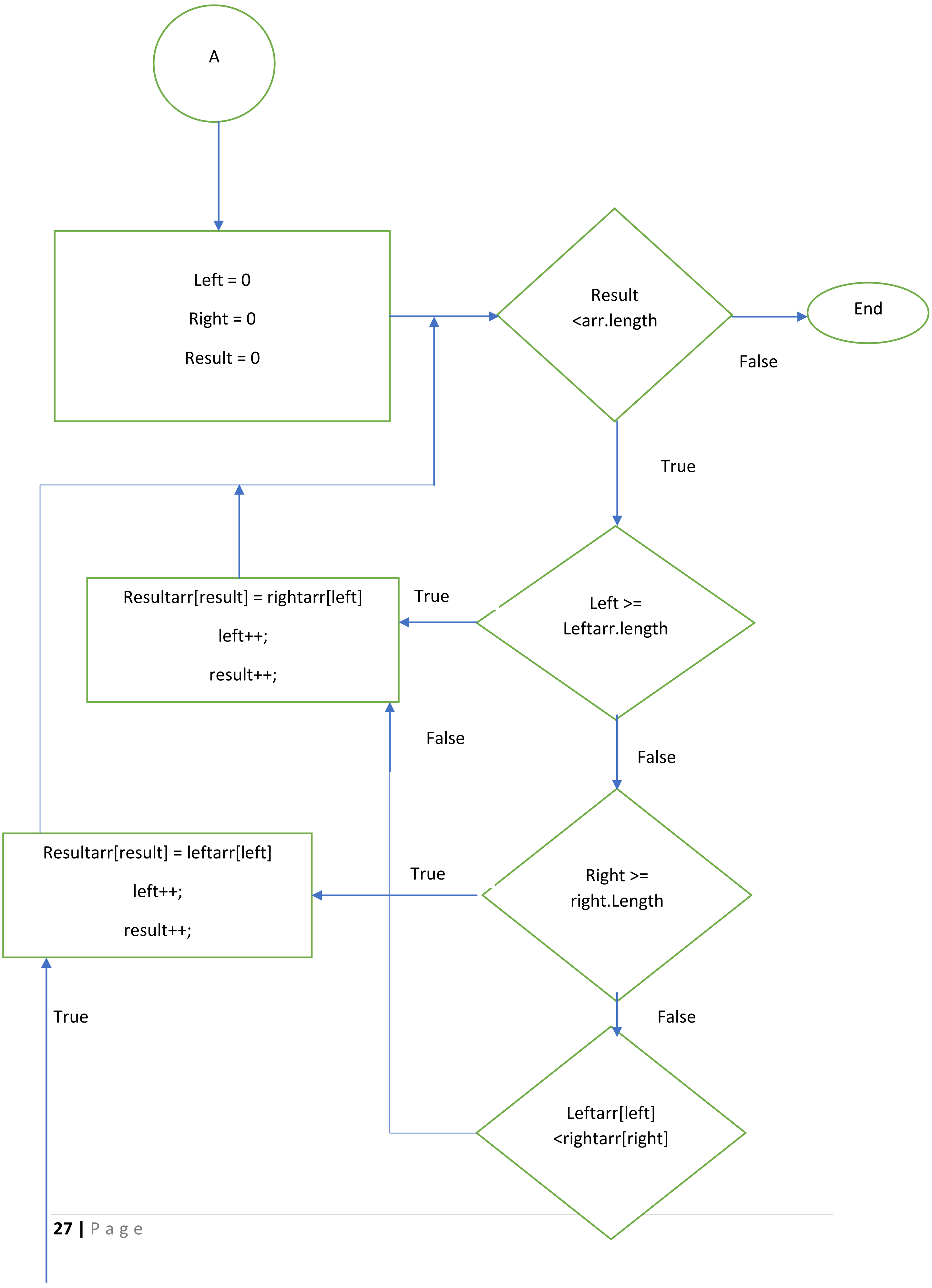
The ordering will be such that the number of iterations are minimum.

4.2) Algorithm

1. Open and read file which consists the code.
2. Find the block which is having the nested loops.
3. Traverse through the block and find the number of nested loops
4. For each loop store its number of iterations.
5. Let for N loops I_1, I_2, I_3, …..I_N be the number of iterations of each.
6. Use these values to find the order of iterations

Figure 4.1 Flowchart 1




A
Left = 0
Right = 0
Result = 0
Result
<arr.length
End
False
True
Resultarr[result] = rightarr[left]
left++;
result++;
True
Left >=
Leftarr.length
False
False
Resultarr[result] = leftarr[left]
left++;
result++;
True
Right >=
right.Length
True
False
Leftarr[left]
<rightarr[right]

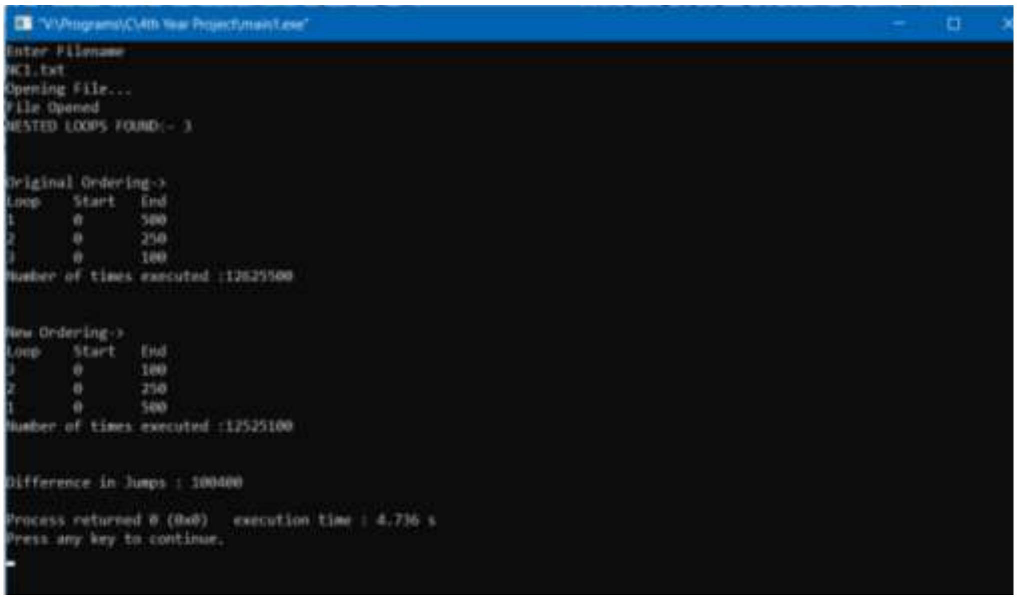
Therefore using mathematical induction we can infer that for a given iteration I1 + I2 * I1 it will be minimal if I1 < I2 and I2 – I1 is very large.This idea can be extended to deeper loops. To find the minimum iterations sort the obtained list in ascending order.If the ordering output is I1,I2,I3,….IN this means that I1 < I2 < I3 < I4….< IN.

5) PERFORMANCE ANALYSIS

5.1) Output

The following screen shot of code tells the user the best order in which the given loops must be put in. The difference in jumps conveys that the new ordering will be effective.

Case 1:



```
"V:\Programs\C\4th Year Project\main1.exe"
Enter Filename
NC1.txt
Opening File...
File Opened
NESTED LOOPS FOUND:- 3


Original Ordering->
Loop    Start   End
1       0       500
2       0       250
3       0       100
Number of times executed :12625500


New Ordering->
Loop    Start   End
3       0       100
2       0       250
1       0       500
Number of times executed :12525100


Difference in Jumps : 100400

Process returned 0 (0x0)   execution time : 4.736 s
Press any key to continue.
```

Fig 5.1 Case 1 Orderingwith three loops

Output Explanation :

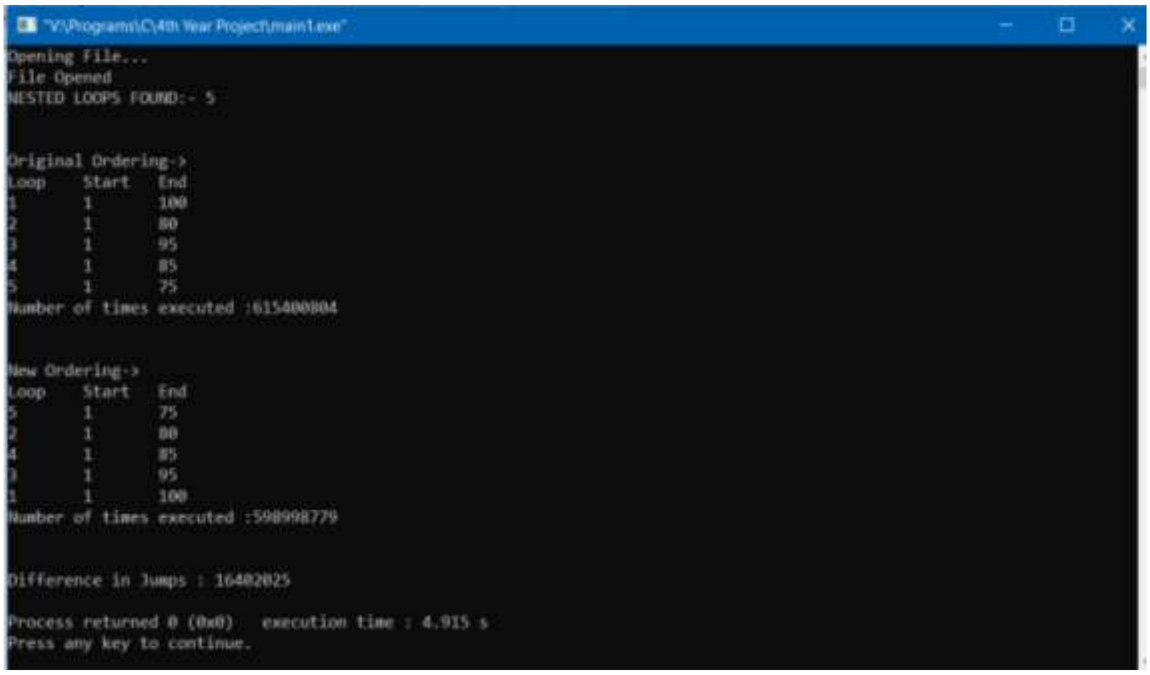
The first part of the code reads the input file "NC1.txt". It parses the file and finds the number of NESTED LOOPS present. The function "findLoop(FILE *)" parses the file(ln 105-137). The number of times the original ordering of loops have run the innermost line is (in this case) : 1,26,25,500 (> 10^7).

The second part of the code involves finding the correct order of loops according to the algorithm. The function "findOrder()" is called (ln 66-84).

With new ordering the number of jumps of the innermost line goes to :1,25,25,100 (> 10^7). However, the difference between them is striking – 1,00,400(>10^5).

We increased the number of nested loops to five. The results were same with the new ordering.

Case 2:



```
"V:\Programs\C\4th Year Project\main1.exe"
Opening File...
File Opened
NESTED LOOPS FOUND:- 5


Original Ordering->
Loop     Start   End
1        1       100
2        1       80
3        1       95
4        1       85
5        1       75
Number of times executed :615400804


New Ordering->
Loop     Start   End
5        1       75
2        1       80
4        1       85
3        1       95
1        1       100
Number of times executed :598998779


Difference in Jumps : 16402025

Process returned 0 (0x0)   execution time : 4.915 s
Press any key to continue.
```

Fig 5.2 Case 2Ordering with five loops

Here as expected the difference is of 1,64,02,025(61,54,00,804 – 59,89,98,779).

Also the difference also increases as we increase the number of loops.

5.2) Comparative Analysis

We can only understand the effectiveness of this algorithm when we know how it affects the machine or the program. Even though loop optimization does not require any hardware specification we can still see it effects on the time elapsed and the DRAM bandwidth of a system. We use Intel Advisor – a freeware and a propriety software. We can visualize actual performance against hardware-imposed ceilings(rooflines)- such as memory bandwidth and compute capacity- which provides as ideal road map of potential optimization steps. It allows to focus on the loops that have room for improvement and helps to deliver the biggest performance payoff.

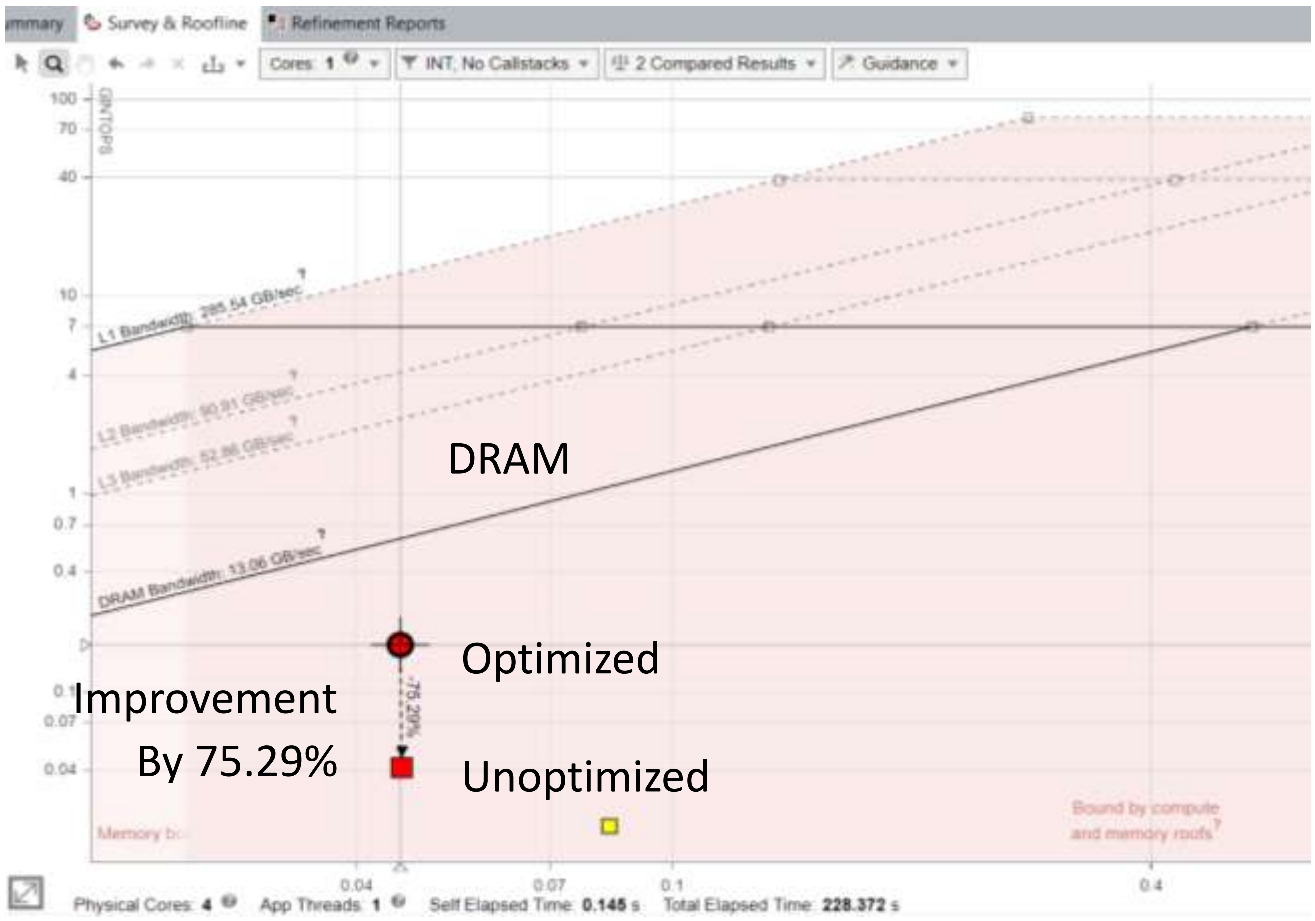
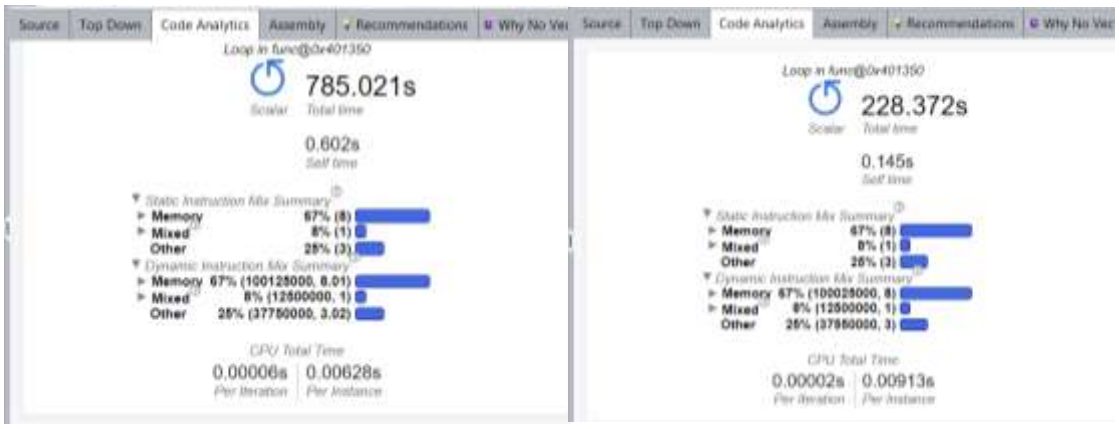


Fig 5.3 Performance versus Intensity Graph (Effect of Optimization)

This is performance versus intensity graph of case 1(With and without optimization). The more close is the red circle the more optimized an application(or code) is. Here the improvement is by almost 75.29%. This means our optimization algorithm works and improves performance of the code.

The below given figures gives a deeper insight on effects of optimization.



```
Loop in func@0x401350
785.021s
Scalar   Total time
0.602s
Self time
▼ Static Instruction Mix Summary
 ▸ Memory                       67% (8)
 ▸ Mixed                         8% (1)
   Other                        25% (3)
▼ Dynamic Instruction Mix Summary
 ▸ Memory  67% (100125000, 8.01)
 ▸ Mixed         8% (12500000, 1)
   Other      25% (37750000, 3.02)
CPU Total Time
0.00006s | 0.00628s
Per Iteration | Per Instance
```

```
Loop in func@0x401350
228.372s
Scalar   Total time
0.145s
Self time
▼ Static Instruction Mix Summary
 ▸ Memory                       67% (8)
 ▸ Mixed                         8% (1)
   Other                        25% (3)
▼ Dynamic Instruction Mix Summary
 ▸ Memory  67% (100025000, 8)
 ▸ Mixed        8% (12500000, 1)
   Other     25% (37550000, 3)
CPU Total Time
0.00002s | 0.00913s
Per Iteration | Per Instance
```

Fig 5.4 Code Analytics of Unoptimized and Optimized Code respectively

If we look closely the total time of unoptimized code is 785.021 seconds and that of our optimized code is 228.372 seconds. This goes on to show that our optimization is correct and we have achieved it without changing the meaning of the code.

6.1) CONCLUSION

As stated an optimization algorithm is efficient if it improves the performance of the code without changing its meaning. Here, we not only achieved optimization without changing the meaning of the code but also this type of optimization is not processor dependent. We extended on idea of reducing the number of loop jumps. We proved it with mathematical induction and developed a tool which executes the said algorithm and shows the user an effective order of loops to achieve maximum optimization. The use of Intel Advisor further cements our claim and proves that we can achieve some amount of optimization without much involvement of external software or hardware.

6.2) FUTURE WORK

LOOP FUSION AND USE OF SCALARS IN DERICHE IMAGE FILTERING -

In addition to producing aChapel version of the correct “i/j/k” tiling for the “Nussinov” algorithm, we have begunto explore the use of our technique for optimization of other codes. Wehave written a Chapel version of the Deriche benchmark from the “PolyBench” benchmark set, and are using this code to explore combinations of iterator-based iteration-space changes with data-space transformationsimplemented as Chapel classes.

We can further investigate optimizations not showed in the benchmark code, for example, loop combination. On the off chance that we present iterators, we can either recreate the first execution request, or investigate a request that should deliver lower memory traffic: Instead of going completely through each estimation of the informative input array twice to make y1andy2, we can emphasizethrough the segments (“I” dimension) just once in the initial three stages of the algorithm, making a column each of y1 and y2 , and afterward utilizing these lines, before moving to the following section. So also, we could consolidate the last three stages of the algorithm to create and promptly use sections. We would trust this would increment execution by reducing the main-memorytraffic .

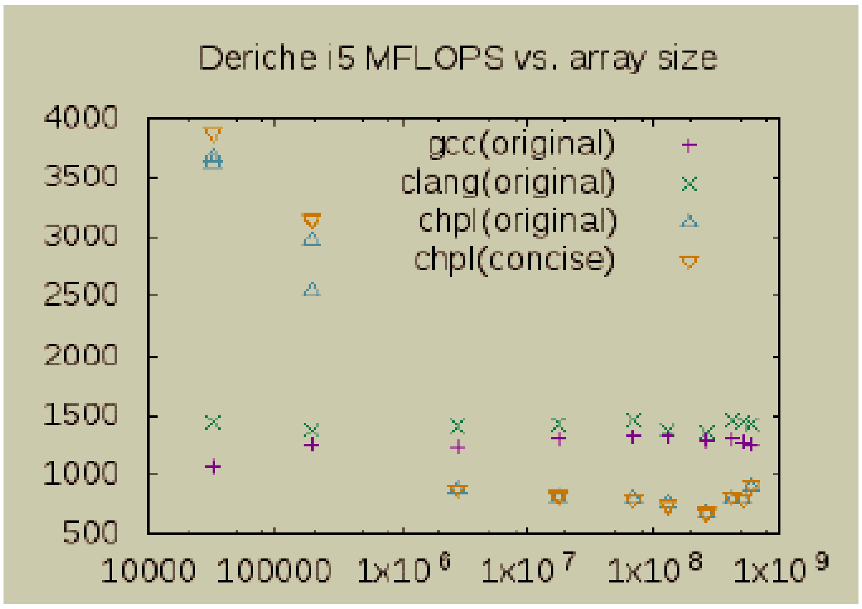


Fig 6.2 – Performance Deriche Code[3]

Appendix A: Code

FileRead.h

```
// Helper functions to LOAD file
void closeFIle(FILE *fptr);
FILE* readFile(char filename[])
{
    FILE *fptr;
    char c;
    char s[100];
    int i=0;
    printf("Opening File...\n");
    //Open file to read only
    fptr = fopen(filename, "r");
    if (fptr == NULL)
    {
        printf("Cannot open file \n");
        exit(0);
    }
    // Read contents from file
    printf("File Opened\n");
    return fptr;
}
void closeFile(FILE *fptr)
```

```
{

    fclose(fptr);

}
```

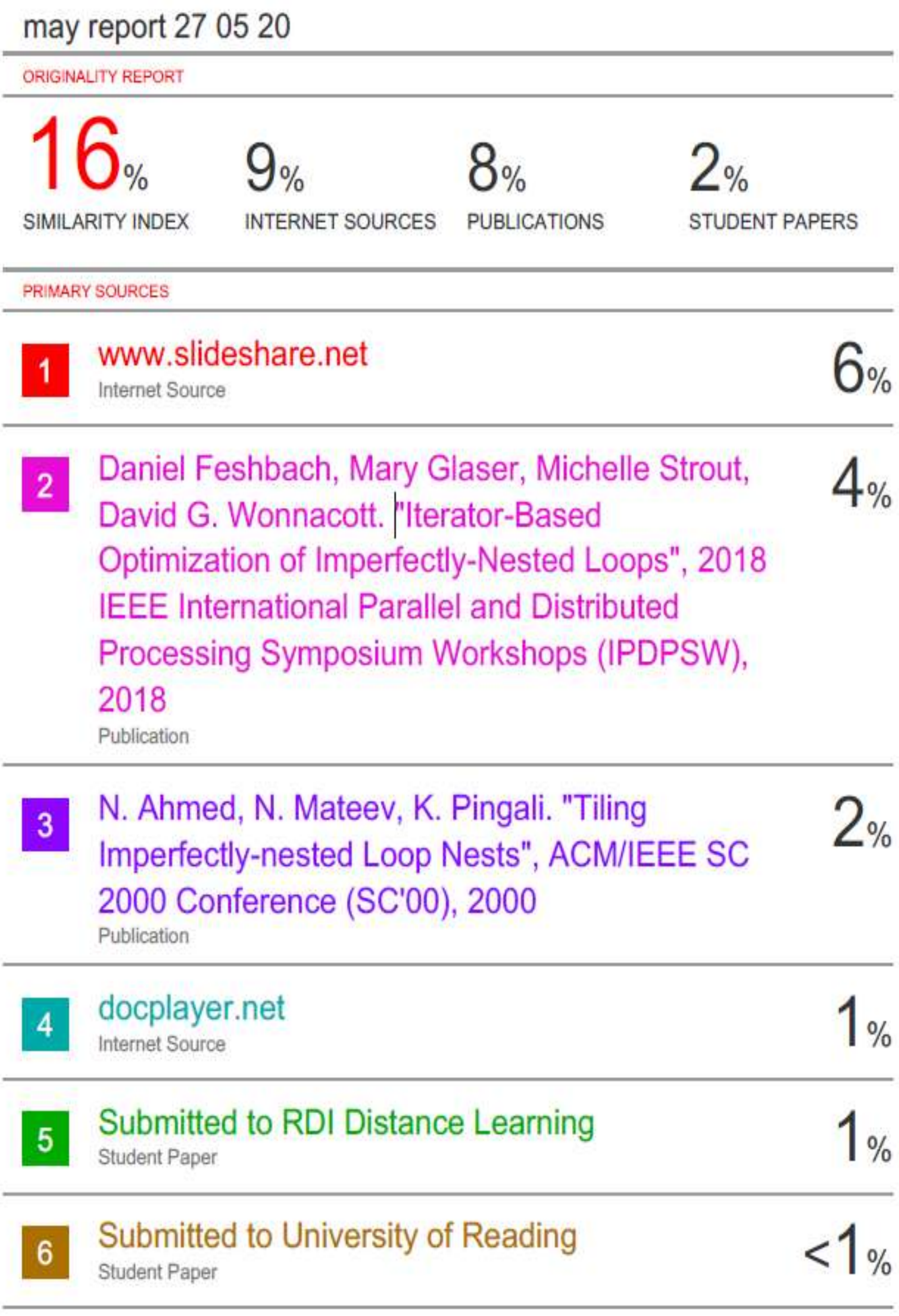
Main.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "FileRead.h"
#define MAX 100

FILE * getFile();
int findLoop(FILE *fptr);
void reset(char *s);
void getItr(FILE *fptr);


int val = 0;
int valArr[MAX][3];

int main()
{
    FILE *fptr = getFile();
    getItr(fptr);
    int num_loops = findLoop(fptr);
    printf("NESTED LOOPS FOUND:- %d\n\n\n",num_loops);



    int i = 0;int exe_number = 0;int temp = 1;

    printf("Original Ordering->\n");
    printf("Loop\tStart\tEnd\n");

    for(int i = 0;i < num_loops;i++)
    {

printf("%d\t%d\t%d\n",valArr[i][2],valArr[i][0],valArr[i][1]
);

        temp = temp * valArr[i][1];
        exe_number = exe_number + temp;
    }
    temp = 1;
    printf("Number of times executed :%d\n",exe_number);


    findOrder(num_loops);


    int nexe_number = 0;

    printf("\n\nNew Ordering->\n");
    printf("Loop\tStart\tEnd\n");

    for(int i = 0;i < num_loops;i++)
    {
printf("%d\t%d\t%d\n",valArr[i][2],valArr[i][0],valArr[i][1]
);
        temp = temp * valArr[i][1];
        nexe_number = nexe_number + temp;
    }


    printf("Number of times executed :%d\n",nexe_number);


    printf("\n\nDifference in Jumps : %d\n",exe_number -
nexe_number);


    closeFile(fptr);
}
void findOrder(int n)
{
    int i, j;
   for (i = 0; i < n-1; i++)
   {
    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
    {
        if (valArr[j][1] - valArr[j][0] > valArr[j + 1][1] -
valArr[j + 1][0])
        {
            swap(&valArr[j][1], &valArr[j+1][1]);
            swap(&valArr[j][0], &valArr[j+1][0]);
            swap(&valArr[j][2], &valArr[j+1][2]);
        }
    }

   }
```

```
}
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

FILE * getFile()
{
    char filename[100];
    printf("Enter Filename\n");
    scanf("%s",&filename);

    FILE *fptr;

    fptr = readFile(filename);
    return fptr;
}

int findLoop(FILE *fptr)
{
  char c;
  int i = 0;
  char s[MAX];
  char str1[MAX];
  int flag = 0;
  int count_loops = 0;
  c = fgetc(fptr);

  // The Input File follows a specific format to write code
  while(c != '*')
  {
      c = fgetc(fptr);
  }


   while(fscanf(fptr,"*%s",str1) == 1)
   {
    // If encounters a for
       if(strcmp(str1,"for") == 0)
       {
          // Start counting "for" loops
          count_loops++;
          reset(&str1);
       }
       c = fgetc(fptr);
   }
  // Drinks number of loops
  return count_loops;
}

void reset(char *s)
{
    int i = 0;
    val = 0;
    for(int i = 0;i < MAX;i++)
    {
        s[i] = '/0';
    }
}
void getItr(FILE *fptr)
{
    char c;
    //int valArr[MAX][2];
    int x,y;
    int i = 0;
    int *ptr;
    ptr = &valArr[0][0];
    c = fgetc(fptr);

    while(c != '/')
    {
        c = fgetc(fptr);
    }
    //printf("%c",c);
    while(fscanf(fptr,"/%d %d",&x,&y) == 2)
    {
        valArr[i][0] = x;
        valArr[i][1] = y;
        valArr[i][2] = i + 1;
        //printf("%d %d\n",y,valArr[i][0]);
        c = fgetc(fptr);
        i++;
    }

}
```

REFERENCES

1. Iterator-Based Optimization of Imperfectly-Nested Loops By Daniel Feshbach , Mary Glaser , Michelle Strout , David G. Wonnacott
2. J.-F. Collard, Reasoning About Program Transformations: Imperative Programming and Flow of Data
3. Tiling Imperfectly nested Loop Nests by Nawaaz Ahmed, Nikolay Mateev and Keshav Pingali
4. Transformations for Imperfectly Nested Loops by Induprakas Kodukula , Keshav Pingali
5. Engineering a compiler by Keith D Cooper and Linda Torczon
6. William Gilbert Strang, Introduction to Linear Algebra
7. Code Complete by Steve McConnell
8. Intel Advisor Documentation



may report 27 05 20

ORIGINALITY REPORT

16%	9%	8%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.slideshare.net Internet Source	6%
2	Daniel Feshbach, Mary Glaser, Michelle Strout, David G. Wonnacott. "Iterator-Based Optimization of Imperfectly-Nested Loops", 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2018 Publication	4%
3	N. Ahmed, N. Mateev, K. Pingali. "Tiling Imperfectly-nested Loop Nests", ACM/IEEE SC 2000 Conference (SC'00), 2000 Publication	2%
4	docplayer.net Internet Source	1%
5	Submitted to RDI Distance Learning Student Paper	1%
6	Submitted to University of Reading Student Paper	<1%

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: 17 July 2020

Type of Document (Tick): PhD Thesis | M.Tech Dissertation/ Report | [✓ B.Tech Project Report] | Paper

Name: Kamal Kant **Department:** CSE **Enrolment No** 161210

Contact No. 7018659865 **E-mail.** hapsvnkamal@gmail.com

Name of the Supervisor: Dr. Ravindra Bhatt

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): CODE OPTIMIZATION FOR PERFECTLY NESTED LOOPS

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

- Total No. of Pages = 49
- Total No. of Preliminary pages =
- Total No. of pages accommodate bibliography/references = 2

Kamal

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at ....16..........(%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

Ravindra Bhatt

(Signature of Guide/Supervisor) **Signature of HOD**

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	**Excluded**	**Similarity Index (%)**	**Generated Plagiarism Report Details** (Title, Abstract & Chapters)	
	• All Preliminary Pages • Bibliography/Images/Quotes • 14 Words String		Word Counts	
Report Generated on			Character Counts	
		Submission ID	Total Pages Scanned	
			File Size	

Checked by
Name & Signature **Librarian**

....

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: 17nJuly 2020

Type of Document (Tick): PhD Thesis | M.Tech Dissertation/ Report | [✓ B.Tech Project Report] | Paper

Name: Vandit Sharma **Department:** CSE **Enrolment No** 161225

Contact No. 9805109181 **E-mail.** vaandit@gmail.com

Name of the Supervisor: Dr. Ravindra Bhatt

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): CODE OPTIMIZATION FOR PERFECTLY NESTED LOOPS

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

- Total No. of Pages = 49
- Total No. of Preliminary pages =
- Total No. of pages accommodate bibliography/references = 2

[illegible]

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at 16 (%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

[illegible]

(Signature of Guide/Supervisor) **Signature of HOD**

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	**Excluded**	**Similarity Index (%)**	**Generated Plagiarism Report Details** (Title, Abstract & Chapters)	
	• All Preliminary Pages • Bibliography/Images/Quotes • 14 Words String		Word Counts	
Report Generated on			Character Counts	
		Submission ID	Total Pages Scanned	
			File Size	

Checked by
Name & Signature **Librarian**

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com