Loop Transformations (Group 17)

1.0 Introduction

The complexity of parallel system architectures has grown significantly in recent years, yielding users the ability to run multiple programs at any given time. In order to be able to execute parallel programs, it is necessary to be able to partition the work and map the computation and data of any given algorithms onto the processors and memories of a parallel machine. In most cases, the programmer is required to partition the work themselves and purely rely on intuition or experience to perform some form of parallelism, due to the lack of easy to use parallelism tools, which we aim to rectify. Optimisation is another key aspect in computing to help reduce the load on the central processing unit (CPU) and decrease program execution times. Many techniques exist to optimising code and a few will be discussed later in this report.

Loops are of great interest to us because they are an essential and fundamental aspect in almost every programming language in existence. The overall goal is to be able to achieve the efficient use of memory by reducing load imbalances, non-local memory accesses, as well as synchronisation overhead with the use of loops in a program. It is shown that by restructuring loops in certain ways, the overall performance of any program can be increased. This restructuring of loops, is termed Loop Transformations.

This report deals with ways of optimising some of these loops with certain techniques to be able to enable parallelism or increase the performance of any given program. There is a significant amount of importance placed on this issue due to the substantial amount of performance that could be achieved based on certain transformations. Many novel programmers are unaware of temporal and spatial locality issues and have a naive approach to writing code, especially loops, which tends to be inefficient. Studies have shown that almost all popular loop transformations will give you an increase in performance with some even yielding up to a 50 percent reduction in execution time overall [1].

Our aim is to develop a simple Java program with a graphical user interface (GUI) which takes into it a certain loop and its body of statements and be able to apply a number of different transformations on the input code, to either enable parallelism or improve performance for that particular loop code. Many things need to be considered in developing this application, such as array sizes, dependency testing, nested loops and more. A novel prototype application will be built to perform some of the basic transformations and can be further developed to perform some of the more advanced transformations.

2.0 Methodology/Application Specifics

Before we understand how some loop transformations are undertaken, it is very important to understand the concept of dependency first.

2.1 Dependency

Dependency is essentially the indication that a certain line of code can only be determined, based on another certain line of code.

```
1. a = 13;
2. b = a + 4;
3. c = a+3;
4. d = a + c;
```

Consider the code to the left. It is clear that lines two and three are dependant on line one. Followed by the line four being dependant on lines one and three. This is the basic concept of code dependency.

There are two main types of dependencies. One is data dependence, as described above, and the other is control dependence. Control dependence is when the condition of an 'if' statement has lines of code following in its same block. These lines of code are control depend on the instruction within the 'if' statement. In our application and report, we are only concerned with data dependence. Data dependence encompasses three key types of dependencies. These are anti-dependence, flow dependence and output dependence.

Flow dependence is when the read operation is performed on a variable, after a write. The example above shows that the variable 'c' is written to in line 3 and read in line 4. This is an example of a flow dependence where line 4 is dependent on line 3. Of course, there are more examples of flow dependence in the above example.

Anti-dependence is similar in nature. However it is detected when there is a write operation, after a read operation. Given the example below:

```
1. c = 13;
2. b = c + 4;
3. c = a+3;
4. d = a + c;
```

Consider the code to the left. You can see that the variable 'c' is read in line 2 and written to in line 3. This then goes without saying that line 3 is anti-dependant on line 2.

Output dependence is detected when there is a write after a write operation of the same variable. This can be seen again in the above example of the variable 'c' where it is written to in lines 1 and 3.

The same principal can be applied to loops accessing array elements, which can be seen later on. However, in the case of loops, two extra types of dependencies need to be considered. These are inter-iteration dependencies and intra-iteration dependencies. Inter-iteration refers to certain iterations of a loop having dependencies on previous or future iterations of the same loop. Intra-iterations refers to an existence of a dependence within the same iteration of the loop.

The only 'true' dependence is flow dependence and cannot be eliminated. The other dependencies are a result of reusing the same memory location, and are so called 'pseudo dependencies.' There are methods to eliminate some of these by using various other techniques, such as renaming variables, however, we will not go too deeply into this as this is not our focus.

2.2 Dependency Analysis/Tests

There are many methods to detect dependencies in code. The simplest way is to read the code and spot the dependencies by eye. Other methods involve creating dependency graphs like in figure 1.

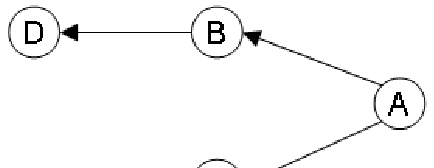


Figure 1. Dependency Graph [2]

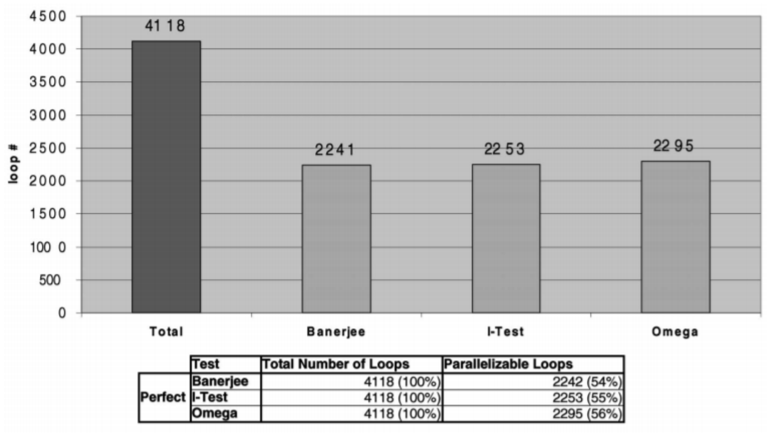
The dependency graph to the left shows how some lines of code could be represented in a graphical format. This illustrates that B cannot run without D and A cannot run without B and C. In other words, A is dependent on B and C, and B is dependent on D. This is particularly useful in depicting iteration dependencies in certain types of loop transformations (for e.g. skewing) to visualise the dependencies of the nested loop. This will be seen later on in this report. Dependency testing is relatively difficult both in theory and in practice as it requires a very deep understanding of the how the code will be executed. There are however several tests to detect if a dependency exists or not.

The most common and relatively simple to implement test is the Greatest Common Divider (GCD) test. This method essentially checks whether there is an overlapping of array elements accessed between the same line of code, or a specific line, with every other line of code. For e.g given that you have:

```
for(int i=0;i<100;i++){
a[2*i]=b[i];
c[i]=a[4*i+1];
}
```

By comparing the two array accesses to the array 'a'. We must first calculate the GCD of the variable that the jumps are being made in. I.e the number that the 'i' variable is being multiplied by. So the GCD of (2,4) which is 2. Then we calculate the difference between the two constant terms in the equation. In this case (0,1). If the first line read a[2*i+3] then we would calculate the difference between (3,1). Following on from the example, once we acquire the GCD we must see if it can divide the difference with no remainder left. I.e. 1 % 2 (if the modulus of the difference with the GCD yields a number, there is no dependence, otherwise if there is no remainder, there is a dependence) in this case yields 1. Hence there is no dependence. This is the algorithm we have used in our code to check for dependencies. The biggest limitation with this method is that it can also detect false dependencies if the loop bounds are constraint to a certain number, as this is not taken into consideration. GCD test also does not take into account the distance or direction information of the dependence and is in most cases combined with Banerjee test.

Other widely adopted, but more complicated dependence tests include the Banerjee test, as mentioned above, the I-test and the Omega test. The Banerjee test in some cases can become and exact test to know for sure if there is a dependency or not, but is mostly used as an inexact test, similar to that of the GCD test [3]. The I-test is a more enhanced version of the Banerjee test. This test in most cases can come to an exact answer and also has the capability to disprove a dependence even if the GCD and Banerjee tests fail to do so. The Omega test is based on a combination of the least remainder algorithm and Fourier-Motzkin variable elimination. It produces exact answers (yes/no) however it is also known to have worst-case exponential time complexity. All three of the discussed test can also produce direction vectors information to locate where the dependencies are arising from.



	Test	Total Number of Loops	Parallelizable Loops
Perfect	Banerjee	4118 (100%)	2242 (54%)
	I-Test	4118 (100%)	2253 (55%)
	Omega	4118 (100%)	2295 (56%)

Figure 2. shows that the performance in detecting and parallelising loops from the 'Perfect Benchmarks' package is almost equivalent with all the three major tests. However we can still see that the Omega test is able to parallelise 1% more than the I-test and 2% more than the Banerjee test.

Figure 2. Test Comparison [3]

Hence we can see that it can detect and parallelise the most loops. However it should also be kept in mind that it is also the most computationally expensive.

2.3 Loop Transformations

There exist a multitude of very different loop transformations addressing various aspects of high performance computing. Some are mainly for performance increases and others aim to enable parallelism to in turn increase performance. We will be addressing a number of popular transforms and seeing when they are applicable and how they influence performance of the program.

Loop Fusion:

Loop fusion combines two adjacent loops with identical control (i.e same number of iterations) into one loop. This is only possible if dependencies are preserved. This may improve or upset data locality based on how it is used. It will definitely reduce loop overhead and can also enable array contraction.

```
for (i = 0; i < 300; i++)
a[i] = a[i] + 3;
for (i = 0; i < 300; i++)
b[i] = b[i] + 4;
```

FUSION:

```
for (i = 0; i < 300; i++)
{
a[i] = a[i] + 3;
b[i] = b[i] + 4;
}
```

Loop Fission:

Loop fission is simply the opposite to loop fusion. This divides the loop control (i.e. number of iterations, start/stop, and increment) over the different statements in the loop body. This is only possible if there are no data dependencies that are backward in the loop. For e.g. going from one statement to a statement that references an iteration earlier in the loop. This will definitely enable other transformations, improve locality by reducing the amount of data accessed during a complete execution cycle, enables hardware prefetching and may also remove simple flow dependencies as seen on the right.

```
for (i = 0; i < 300; i++)
{
a[i] = a[i] + c;
x[i+1] = x[i] + a[i];
}
```

FISSION:

```
for (i = 0; i < 300; i++)
a[i] = a[i] + c;
for (i = 0; i < 300; i++)
x[i+1] = x[i] + a[i];
```

Loop Interchange:

Loop Interchange/Permutation is essentially just swapping the order of the loops. For e.g. (assume row major order). This is again only possible if the dependence vector of the loop nest remains lexicographically positive after the interchange, which alters the order of dependencies to match the new loop order. This is a widely used and popular transform to enable other transformations, improve spatial locality and increase parallelism.

```
for (i = 0; i < 300; i++) //i is column, j is row
{
        for (j = 0; j < 300; j++)
        {
        x = a[i][j] //access is non-uniform
        }
}
```

INTERCHANGE:

```
for (j = 0; j < 300; j++)//i is column, j is row
{
        for (i = 0; i < 300; i++)
        {
        x = a[i][j] //access is uniform
        }
}
```

Loop Unrolling:

Loop unrolling is a combination of two or more loop iterations in the body of the loop statement, along with an increase in jumps from the loop counter. This can be used in most cases. Branch penalty is also reduced by reducing the number of checks of the target variable. In some cases if the unrolled statements are independent, we can also parallelise these statements.

```
int sum =0;
for (i=0; i<N; i++)
sum += array[i];
```

UNROLLING:

```
int sum =0;
for (i=0; i<N; i+=4) {
sum += array[i];
sum += array[i+1];
sum += array[i+2];
sum += array[i+3];}
```

Loop Skewing:

Loop skewing involves skewing the execution of the inner loop by a certain factor in order to execute the statements in a tilted manner. Loop skewing is always safe to utilise given that dependencies remain in place and the only change is in the loop execution. Skewing enables us to easily parallelise code. Our goal is to go from figure 3 to figure 4, where each black dot represents an iteration, and the arrows represent the dependencies as discussed previously.

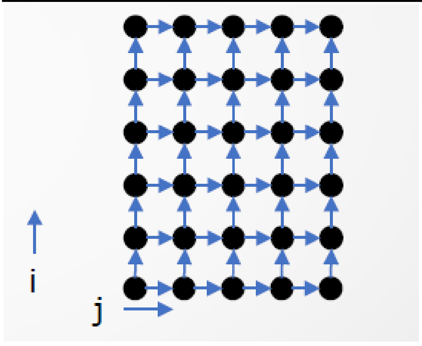


Figure 3. Un-skewed [4]

After Skewing

The arrows in red, indicate that those rows can all run in parallel via different threads.

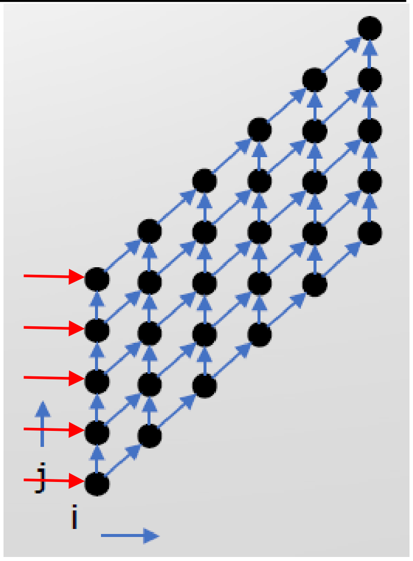


Figure 4. Skewed [4]

```
for(i = 1; i < 7; i++){
        for(j = 1; j < 6; j++){
        A[i , j] = A(i –1, j) + A(i, j –1);
        }
}
```

After SKEWING:

```
for(j = 2; j < 11; j++){
        for(i = max(1, j –6 + 1); i < min(7 –1, j –1) ; i++){
        A[i , j -i] = A(i –1, j -i) + A(i, j –1 -i);
        }
}
```

Loop Inversion:

Loop Inversion transforms a 'while' loop into a do-while loop with an initial if statement. This method is possible in most cases as we can transform any 'for' loop into a 'while' loop. The main benefit to this is in the very low level assembly code where we can save at least two jump instructions, thus saving two pipeline stalls. Although this does increase performance, this may only be noticeable in a very large number of loop iterations. You can see in the machine pseudo code that after the last iteration is run the un-optimised code will 'goto' L1 and then 'goto L2', however in the optimised code, it will not satisfy the condition and go straight to L2 without a jump, saving cycles.

```
int i, a[100];
i = 0;
while (i < 100) {
a[i] = 0;
i++;
}
```

INVERSION:

```
int i, a[100];
i = 0;
if (i < 100) {
do {
a[i] = 0;
i++;
} while (i < 100);
}
```

Machine Pseudo code:

```
L1:
if(!c) goto L2;
body;
goto L1;
L2:
```

INVERSION:

```
if(!c) goto L2;
L1:
body;
if(c) goto L1;
L2:
```

3.0 Related Work

There is a range of different parallelisation and loop transformation tools in the market for people to use. The S2P tool is a fully automated parallelisation tool that converts sequential C code, into parallel multi-threaded source code, intended for systems with a shared-memory architecture. This tool is also capable of performing data dependence analysis in order to determine if certain loops can be parallelised or not. The tool also provides a very useful GUI showing control flow graphs, caller graphs and much more. OpenMP is another semi-automatic parallelisation tool, however, it is not as versatile as S2P in the fact that it doesn't allow to identify parallel code or enable transformations. It does, however, allow for the generation of parallelised code which can then be assigned to multiple threads in order to run in parallel. This is particularly useful in loop level parallelism. Again OpenMP is an extension for C or C++ and provides a set of compiler directives to invoke certain behaviors. In the example below, the line after the '#' i.e '#pragma…' is the compiler directive. This code enables the 'for' loop to be executed in parallel with multiple threads [5].

```
int main()
{
const int N = 20000;
int i, a[N];
#pragma omp parallel for
for (i = 0; i < N; i++)
a[i] = 2 * i;
return 0;
}
```

There also exist a wide range of compilers to perform loop optimisations and transformations in the back-end. For e.g. the GCC compiler gives users the option to perform certain optimisations. The "-funroll-loops" option turns on loop unrolling, "-funswitch-loops" takes into account invariant code from within loops and optimises it out, "-fsplit-loops" splits loops into two given that certain conditions are met. These are only some of the optimisation options that are available to users. There are much more that are unrelated to loops as well [6].

The Intel Advisor 2017 is also a very powerful tool to rapidly prototype your code for correctness, performance and scalability checks, as well as dependency analysis checks. The tool also enables users to mark loops for vectorisation and threading, view loop-carried data dependencies and memory access patterns. This program can also optimize entire applications by giving users the ability to be able to vectorise code or by threading it straight away where you need to manually put in a comment like annotation, into the code, to roughly sketch out what parts you want to parallelise. The tool looks for dependencies first in the loops and then performs the vectorization and then checks for correctness [7].

Pluto is yet another automatic parallelisation tool that is based on the polyhedral model. This model is used as a way to abstract details away from high-level transformations, like parallelisation on affine loop nests or general loop nest optimisations [8]. It can also enable us to perform program analysis such as dependence analysis and also scheduling theory to enable loop tiling, (with tiling models) loop transformations, (such as those discussed above) and vectorization and parallelisation of nested loops [9].

It is clear from all these related works that a lot of them tend to be more back end or compiler based without any visibility to the user. Although some programs do provide a GUI, it may prove to be more complicated to learn and use, in order to view optimisation options in their program. It is often useful for users to see the outcome of the compiler code, after it is done with its loop optimisations for e.g., in a high-level language. In our research, we have not come across an application that does this in a high-level language, like Java for e.g. Almost all compilers perform these optimisations, however, there does not seem to be many applications to allow the user to view these optimisations.

4.0 Implementation

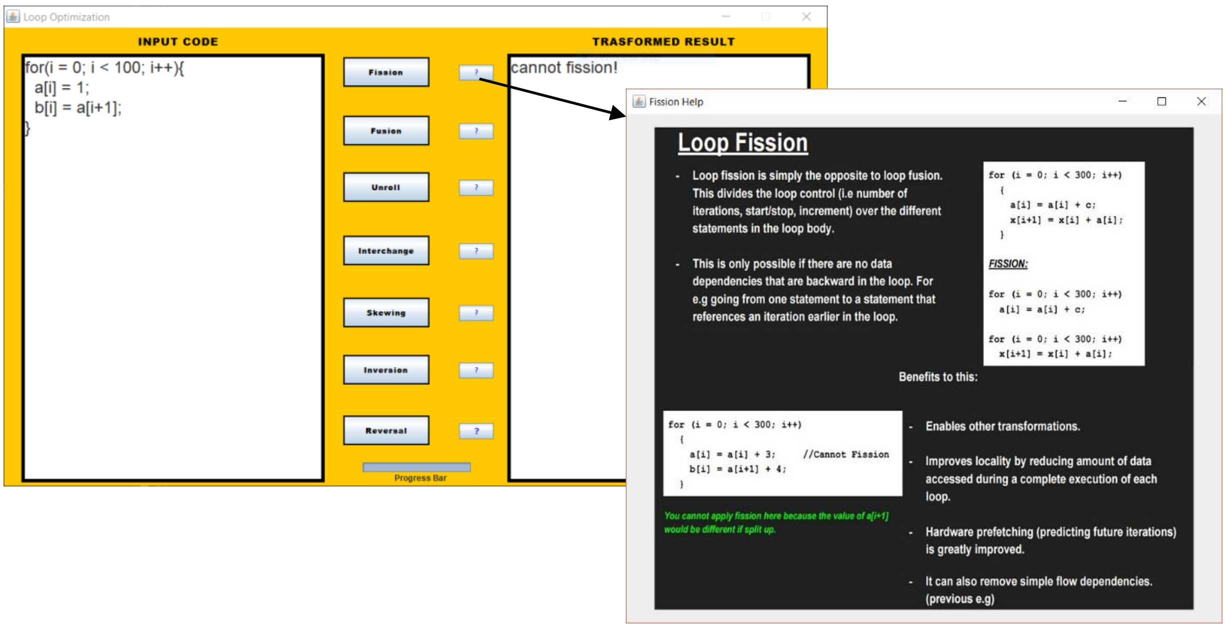
While designing our application we considered a range of different things. We needed to make sure that the user entered only Java code and not garbage. We needed to have some sort of dependency checker to see if certain transforms are viable. We also needed to have a relatively simplistic GUI so that users do not feel intimidated by too many extra toolbars or features and also make it so that it remains responsive while transformations are taking place.

In order to make sure that the user doesn't enter random garbage into our text area, we decided to run their code through an eclipse compiler and throw out error messages for invalid syntax. The user written 'loop' code is written to a Java file which is then compiled at program execution, i.e. whenever you click a certain transformation button, to check for errors.

We were also aware about the limitations of the GCD test, such as not taking into account the range and reporting false dependencies, and have found a method to work around it. We understood that the GCD test can almost always identify no dependencies, so this was our first check. Next we developed a simple algorithm to take into account the loop bounds, given that the GCD test returns true (i.e. there is a dependency.) Our last resort was to check every single combination of array access from reads and writes and look for overlaps. This ensures that our dependency checker is fairly accurate majority of the time.

Our GUI is also minimalist and uses the properties of swing worker to enable responsiveness while transforms are being performed. We also provide a sampleLoops.txt file to help users get started. We've added an extra information feature next to each transformation to teach users about the type of action they are performing. By clicking on the question mark, they will receive a slide of information about that particular transformation to enhance their knowledge, as well as understand some of the things we've considered whilst implementing the application.

The image below highlights some of these features:



```
for(i = 0; i < 100; i++){
  a[i] = 1;
  b[i] = a[i+1];
}
```

```
cannot fission!
```

Loop Fission

- Loop fission is simply the opposite to loop fusion. This divides the loop control (i.e number of iterations, start/stop, increment) over the different statements in the loop body.
- This is only possible if there are no data dependencies that are backward in the loop. For e.g going from one statement to a statement that references an iteration earlier in the loop.

```
for (i = 0; i < 300; i++)
  {
    a[i] = a[i] + c;
    x[i+1] = x[i] + a[i];
  }
```

FISSION:

```
for (i = 0; i < 300; i++)
  a[i] = a[i] + c;

for (i = 0; i < 300; i++)
  x[i+1] = x[i] + a[i];
```

Benefits to this:

```
for (i = 0; i < 300; i++)
  {
    a[i] = a[i] + 3;      //Cannot Fission
    b[i] = a[i+1] + 4;
  }
```

You cannot apply fission here because the value of a[i+1] would be different if split up.

- Enables other transformations.
- Improves locality by reducing amount of data accessed during a complete execution of each loop.
- Hardware prefetching (predicting future iterations) is greatly improved.
- It can also remove simple flow dependencies. (previous e.g)

Conclusion

To conclude we have created a simple application that allows users to input loop code and view an optimised or parallelizable version of their code. Our application can also be used as a learning tool to help users learn about loop transformations. We've considered a range of different aspects when developing the application, and are confident that it can meet our basic requirements.

References

1. Power and Energy Impact by Loop Transformations, by Hongbo Yang, Guang R. Gao, Andres Marquez, George Cai, Ziang Hu - In Proceedings of the Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques , 2000
2. "Dependency graph," Wikipedia, 23-May-2017. [Online]. Available: https://en.wikipedia.org/wiki/Dependency_graph. [Accessed: 06-Jun-2017].
3. K. Psarris and K. Kyriakopoulos, "An experimental evaluation of data dependence analysis techniques," in IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 3, pp. 196-213, March 2004. doi: 10.1109/TPDS.2004.1264806
4. T. T., L. N., and C. L., "Loop Transformations." (Group 22)
5. P. Ranadive and V.G Vaidya, "Parallelization Tools," in Center for Research in Engineering Sciences and Technology.
6. "Using the GNU Compiler Collection (GCC): Optimize Options," Using the GNU Compiler Collection (GCC): Optimize Options. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [Accessed: 06-Jun-2017].
7. A., "Intel® Advisor," Intel® Advisor | Intel® Software, 25-Oct-2016. [Online]. Available: https://software.intel.com/en-us/intel-advisor-xe/details. [Accessed: 06-Jun-2017].
8. U. K. R. Bondhugula, "PLUTO - An automatic parallelizer and locality optimizer for affine loop nests," PLUTO - An automatic loop nest parallelizer and locality optimizer for multicores. [Online]. Available: http://pluto-compiler.sourceforge.net/#contact. [Accessed: 07-Jun-2017].
9. "Polyhedral," Polyhedral Compilation. [Online]. Available: http://polyhedral.info/. [Accessed: 07-Jun-2017].

Table of Contributions:

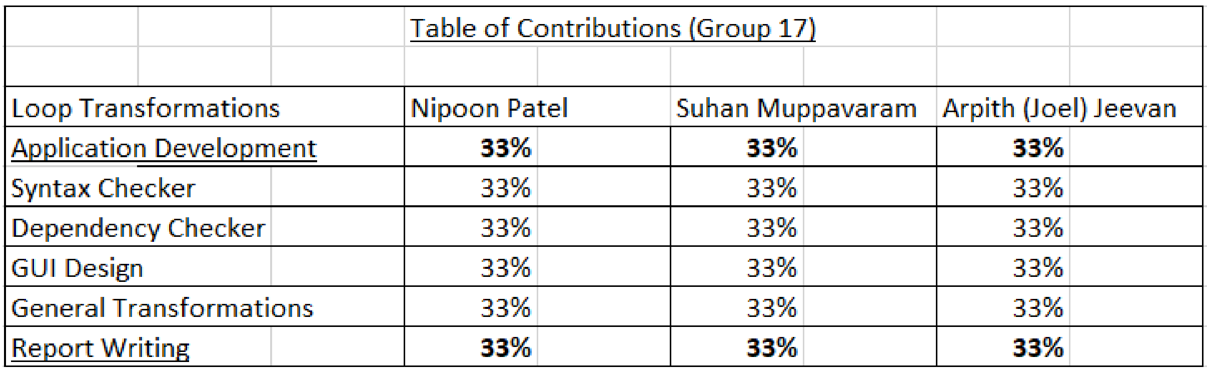


	Table of Contributions (Group 17)		
Loop Transformations	Nipoon Patel	Suhan Muppavaram	Arpith (Joel) Jeevan
Application Development	**33%**	**33%**	**33%**
Syntax Checker	33%	33%	33%
Dependency Checker	33%	33%	33%
GUI Design	33%	33%	33%
General Transformations	33%	33%	33%
Report Writing	**33%**	**33%**	**33%**

Written and Developed by:
Nipoon Patel, Suhan Muppavaram and Arpith (Joel) Jeevan