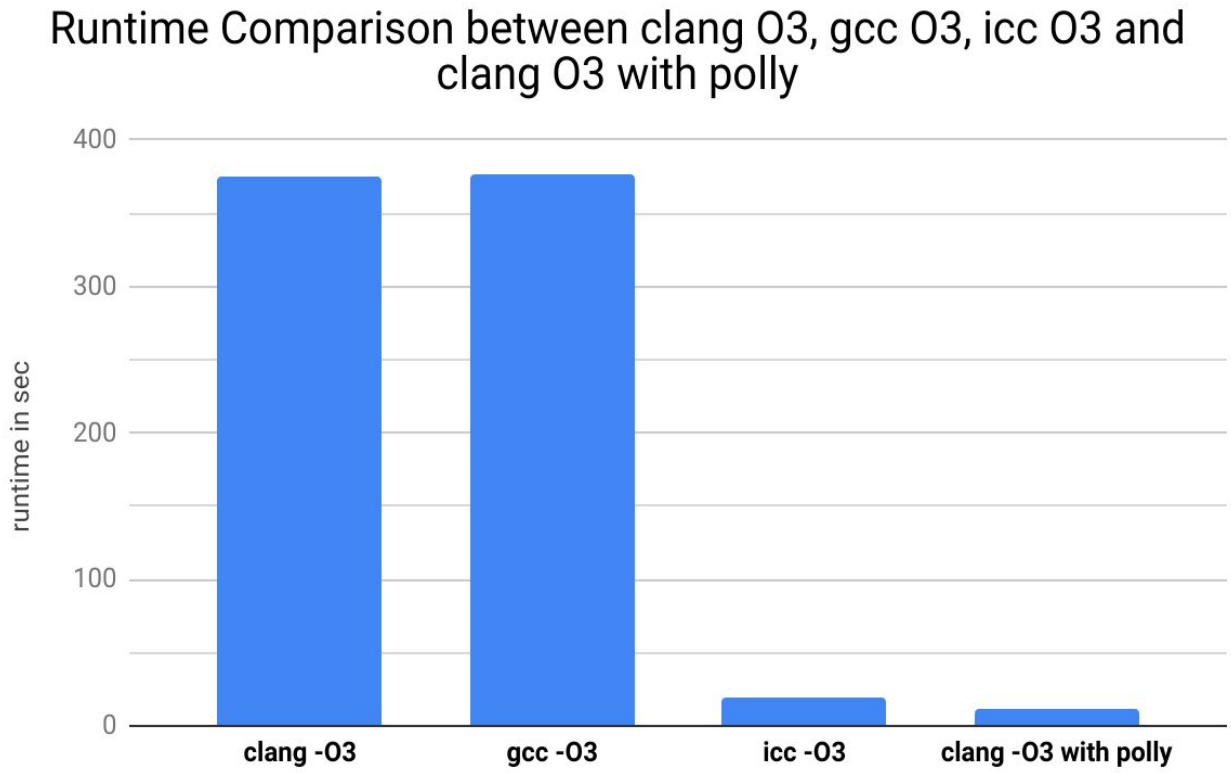
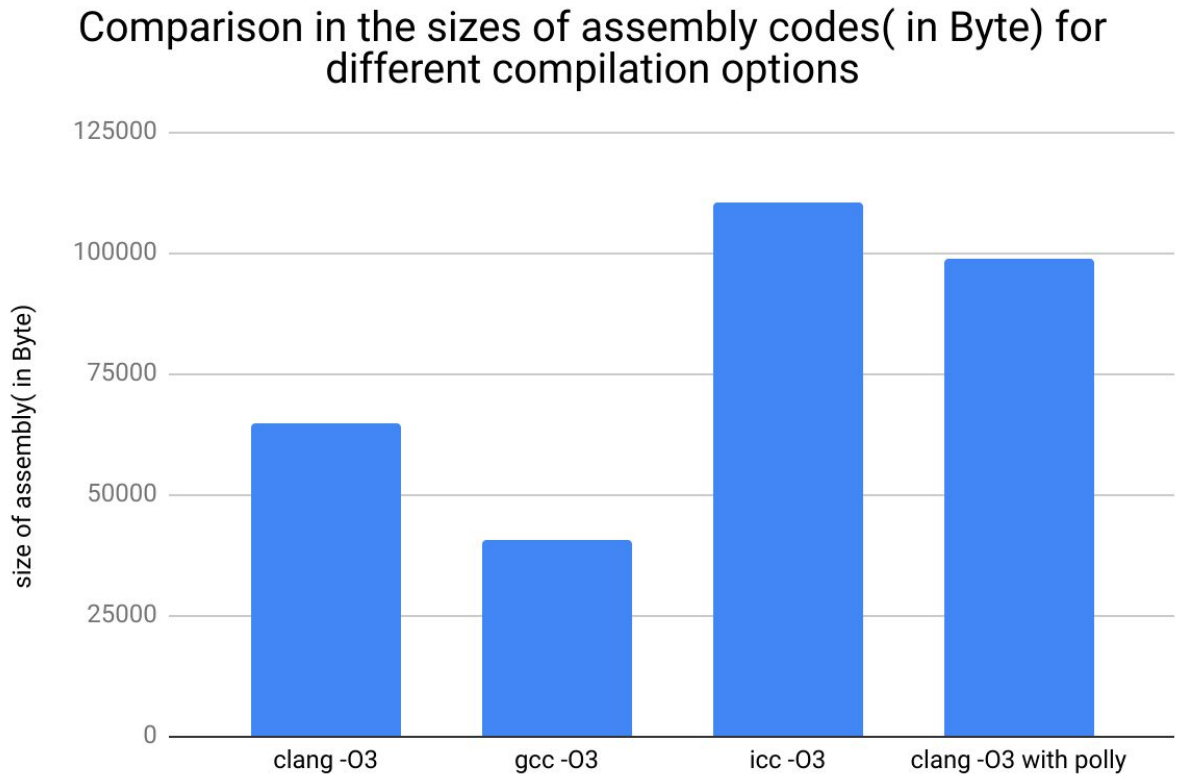
COL 729: Lab 3 Report

Part A: Compilation of gemm program using gcc, icc and clang(with and without Polly) and Observing respective runtimes



Compilation Option	Execution Time(in Seconds)
Clang -O3	377.53
gcc -O3	381.33
Icc -O3	18.45
Clang -O3 with Polly	12.17

Part B: Analysis of the Assembly Codes generated in above four cases



Analysis of x86 assembly generated by gcc -O3

gcc - O3 does not explores the possibility of loop vectorization, loop unrolling, loop tiling etc. which are considered to be the most significant and popular loop optimization techniques. The assembly code generated by gcc -O3 doesn't add anything tricky to the original straightforward implementation of the matrix multiplication routine of gemm. A high level idea of what gcc -O3 does can be inferred by looking into the assembly code generated and that is as follows:

```
For i = 0 upto 4000
        For j = 0 upto 4000
                C[ i , j ] = C[ i , j ] * beta
                For k = 0 upto 4000
                        C[ i , j ] = C[ i , j ] + alpha * A[ i , k ] * B[ k , j ]
                End for
        End for
End for
```

Analysis of x86 assembly generated by clang -O3

Clang -O3 also doesn't explore the possibility of sophisticated loop optimization techniques like loop vectorization and loop tiling but introduces loop unrolling which gcc -O3 could not. The runtime comparison between gcc -O3 and clang -O3 possibly reflect this fact. The high level overview of what clang -O3 does to gemm program is as follows:

```
For i = 0 upto 4000
        For j = 0 upto 4000
                C[ i , j ] = C[ i , j ] * beta
                For k = 0 ; k < 4000 ; k = k +2
                        C[ i , j ] = C[ i , j ] + alpha * A[ i , k ] * B[ k , j ]
                        C[ i , j ] = C[ i , j ] + alpha * A[ i , k + 1 ] * B[ k + 1 , j ]
                End for
        End for
End for
```

What clang -O3 does can not be claimed to be enough aggressive from a performance(execution time) optimization perspective. It unrolls the inner loop only two times without taking any advantage of the parallelization opportunity offered by the hardware by means of vectorization.

Analysis of x86 assembly generated by icc -O3

Intel compiler, when applied with the highest optimization level, applies loop tiling or loop blocking in order to reduce the number of cache misses. In this particular kind of program where a lot of memory accesses take place, the uncontrolled cache misses can severely affect the runtime. This can be validated from the runtime comparison chart presented above in the first section. Intel compiler applies loop splitting also to split the two statements. It vectorizes the first statement with a vector width of 2(possibly). The high level overview of how intel compiler optimizes the gemm program can be derived from the following description:

```
For i = 0 upto 4000
        For j = 0 upto 4000 in a step of 8( j = j + 8)
                vector< C[ i , j ] , C[ i , j + 1 ] > =
                vector_multiply( vector< C[ i , j ] , C[ i , j + 1 ] >, <  beta, beta, beta, beta>
                vector< C[ i , j + 2 ] , C[ i , j + 3 ] > =
                vector_multiply( vector< C[ i , j + 2 ] , C[ i , j + 3 ] >, <  beta, beta, beta, beta>
                vector< C[ i , j + 4 ] , C[ i , j + 1 + 5] > =
```

```
            vector_multiply( vector< C[ i , j + 4 ] , C[ i , j + 5 ] >, <   beta, beta, beta, beta>
            vector< C[ i , j + 6 ] , C[ i , j + 7 ] > =
            vector_multiply( vector< C[ i , j + 6 ] , C[ i , j + 7 ] >, <   beta, beta, beta, beta>
        End for
End for
For ii = 0 ; ii < 4000 ; ii = ii + 128
        For jj = 0 ; jj < 4000 ; jj = jj + 128
            For kk = 0 ; kk < 4000 ; kk = kk + 128
                For tile_i = 0 ; tile_i < min(ii + B, 4000) ; tile_i++
                    For tile_j = 0 ; tile_j < min(jj + B, 4000) ; tile_j++
                        For tile_k = 0 ; tile_k < min(kk + B, 4000) ; tile_k++
                            C[ tile_i , tile_j ] += alpha * A[ tile_i][tile_k]
                                                    * B[ tile_k][tile_j]
```

Analysis of x86 assembly generated by clang -O3 with Polly

When gemm is compiled with clang with the polly, it exhibits the best result among all other compilation options presented in this report. Polly applies vectorization, unrolling and tiling and also checks when it can apply them or when the memory layout of matrices are overlapping and it can not apply some of the techniques. The high level overview of what polly does is as follows:
(only the kernel-gemm part is described. That is the heart of gemm)

```
If the memory layouts of the A, B, C matrices are not overlapping, then
        For ti = 0 upto 125
            For tk = 0 upto 32
                For tj = 0 upto 125
                    ii = ti * 32 + tk
                    vector< C[ ii , tj * 32 + 0], C[ ii , tj * 32 + 1]> *= beta
                    vector< C[ ii , tj * 32 + 2], C[ ii , tj * 32 + 3]> *= beta
                    vector< C[ ii , tj * 32 + 4], C[ ii , tj * 32 + 5]> *= beta
                    vector< C[ ii , tj * 32 + 6], C[ ii , tj * 32 + 7]> *= beta
                    vector< C[ ii , tj * 32 + 8], C[ ii , tj * 32 + 9]> *= beta
                    vector< C[ ii , tj * 32 + 10], C[ ii , tj * 32 + 11]> *= beta
                    vector< C[ ii , tj * 32 + 12], C[ ii , tj * 32 + 13]> *= beta
                    vector< C[ ii , tj * 32 + 14], C[ ii , tj * 32 + 15]> *= beta
                    ……..
                    vector< C[ ii , tj * 32 + 30], C[ ii , tj * 32 + 31]> *= beta
            End for
        End for
```

```
For j tiling is 0 upto 4
        For k tiling is 0 upto 10
                For i tiling is 0 upto 63
                        For j = 0 upto 4000 insteps of 4
                                For i = 0 upto 4000 in steps of 4
                                        For k = 0 upto 4000
                                                vector< C[ i , j ] , C[ i , j + 1] +=
                                                vector< alpha * A[ i , k ] * B[ k , j ] ,
                                                        Alpha * A[ i , k ] * B[ k , j + 1 ]> ;

                                                vector< C[ i , j + 2 ] , C[ i , j + 3 ] +=
                                                vector< alpha * A[ i , k ] * B[ k , j + 2 ] ,
                                                        Alpha * A[ i , k ] * B[ k , j + 3 ]> ;

                                                ……….
                                                vector< C[ i + 3 , j + 2 ] , C[ i + 3 , j + 3 ] +=
                                                vector< alpha * A[ i + 3 , k ] * B[ k , j + 2 ] ,
                                                        Alpha * A[ i + 3 , k ] * B[ k , j + 3 ]> ;
                                        End for
                                End for
                        End for
                End for
        End for
End for
Else
        For i = 0 upto 4000
                For j = 0 upto 4000
                        C[ i , j ] = C[ i , j ] * beta;
                        For k = 0 upto 4000( insteps of 2)
                                C[ i , j ] += alpha * A[ i , k ] * B[ k , j ];
                                C[ i , j ] += alpha * A[ i , k + 1 ] * B[ k + 1 , j ];
                        End for
                End for
        End for
End if
```

More efficient implementation than Polly

The code for this has been attached with the submission folder of this Assignment. The approach which has been followed to construct the efficient implementation is as follows:

First the understanding which is developed after studying and investigating the LLVM bit-code as well as the x86 assembly code generated by clang -O3 Polly is utilized to create such an implementation which tries to mimic all the optimizations applied by Polly. As Polly lacks detailed knowledge about the underlying processor architecture and memory hierarchy, it uses some hard-coded vectorization width while vectorizing the loops. Intel intrinsic instructions are used to enforce desired control over vectorization widths. With some incremental changes, the implementation becomes faster than polly. *The average runtime found is : 7.88 Seconds*.

A Comparative Study of Run Time Behaviours of the x86 Assemblies

Compilation opt.	Cache References	Cache Misses	Cycles per Inst.
Clang -O3	59,43,25,18,989	11,24,73,04,317	0.40
Clang -O3 with polly	26,47,39,773	5,10,17,630	2.69
gcc -O3	58,95,95,24,223	11,17,12,50,028	0.49
Icc -O3	32,14,31,271	12,66,15,305	2.21

Part C: Key Learnings about the Strengths of Polly Framework

Optimizing a program like gemm is very important because it is a frequently occurring dense matrix multiplication routine. gcc and clang do not modify the loop structure significantly and introduces no SIMD instructions. Polly can mitigate these limitations. It can apply Strip Mining to change the loop structure in a way such that the locality is improved and the trivially vectorizable loops are exposed. It replaces trivially vectorizable loops with SIMD instructions. This way Polly achieves what ICC with highest optimization achieves. To minimize the scalar loads needed to initialize any vector, Polly can employ Code Hoisting, identify invariant loads and improve the runtime significantly. All these important loop optimizations are automatically applied by Polly. The only manual component, in this case, is the externally provided schedule. Even in a low-level program, optimizations can be performed by Polly when provided with only polyhedral

schedule. External optimizers like PLuTo can be used to expose the parallelism and followed by that Polly can be employed to create OpenMP code that takes advantage of the exposed parallelism. The optimization process offered by Polly is not bound to any specific high-level programming language and does not need the input code to obey any special syntax. Polly can target multi-core systems as well as heterogeneous platforms with several cores and accelerators.

Part D: Weaknesses of Polly Framework

One limitation of Polly is that, Polly does not model the problem of integer overflow. But, in any program, the most interesting parts consist of loops with several iteration variables which might be of different data types with different storage sizes. In order to be correct, Polly takes a conservative decision. It models them with 64-bit loop counters. Though in the original source code, a loop-iteration variable is of data type ‘int’, Polly would treat it as a ‘long’ data type which consumes 64-bit storage length.

While analyzing the bit-code(and x86 assembly code) generated by clang -O3 polly from gemm.c), it is observed that the polly takes some really conservative and limited decisions regarding Vectorization width or Tile sizes. For example, it used a vectorization width of two in case of vectorizing the loop associated with the statement, C[i , j] += alpha*A[i , k] * B[k , j]. But, a vectorization width of four can easily be used in that scenario. It is observed that a 4-width vectorization there would improve the runtime of the gemm program. But, when the vectorization width is further increased, the runtime starts to increase after a certain width. Surely, polly did not take the optimal vector width in the gemm program. The reason for this is that, Polly lacks a proper model of memory hierarchy and normally uses some hard-coded numbers for vectorization width or tiling size. Polly uses some hard-coded settings which may or may not match the actual processor settings. Polly requires the modelling of underlying processor architecture and memory hierarchy in more detailed manner. Polly cannot model ‘memmove’, ‘memset’ instructions.

Polly also cannot modify the structure of any basic block in any source program. Polly treats a basic block as a statement in its polyhedral representation. If more finer granularity is expected, decomposing basic blocks into its constituent statements is needed to exploit more aggressive parallelism. Polly also might not be promising in terms of changing the data layout in order to enforce good locality.

Vector load/store operations are implemented in terms of multiple scalar loads/stores. Polly cannot optimize for complex load/store which is neither stride-one, or stride-zero.

Polly also induces some limitations in vectorization of loops in the source programs. It can only vectorize those loops which have constant, non-parametric number of loop iterations and doesn't contain conditional control flow or any further loops. Preparing optimization passes are necessary to expose these kinds of trivially vectorizable loops to Polly. Sometimes, optimizations might take much more conservative decisions and loop vectorization opportunity can be neglected.

When the memory access function is not affine, Polly can still capture the memory access but needs to take a conservative assumption. LLVM provides some alias analysis which can identify any aliasing as must-alias, may-alias or no-alias. In case of may-alias, LLVM is not certain and an SCoP(Static Control Parts) can be discarded by Polly. A more rigorous alias analysis might improve the accuracy.