Assignment 3

Runtimes

Run time with various versions of the compilers with O3 are:

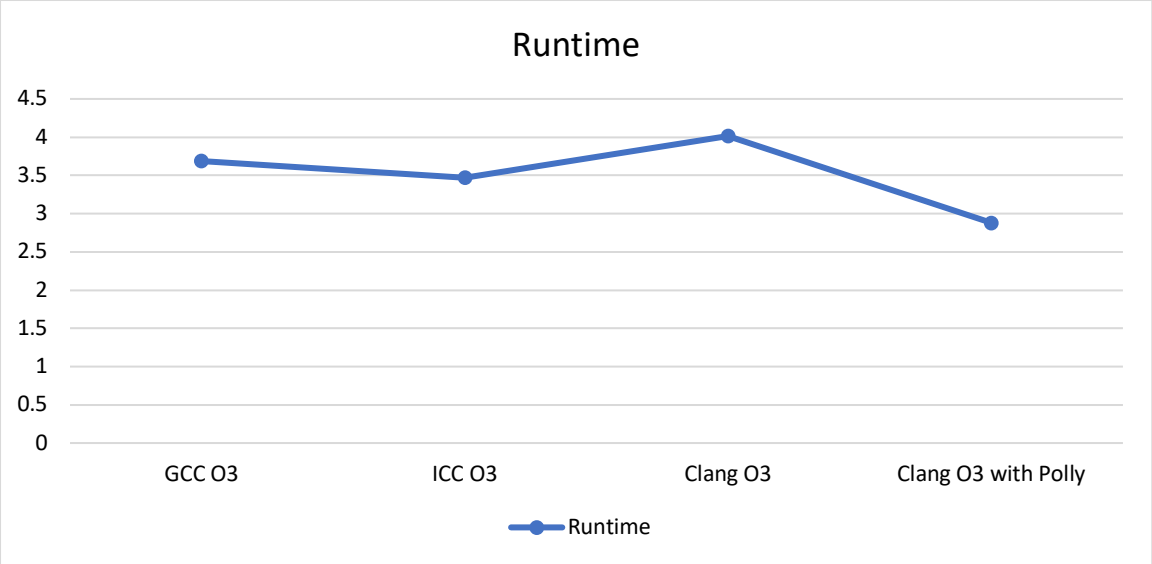

Figure 1
Runtime Comparison of various compilations

From the Runtime it is clear that ICC O3 gives the best performance out of all three vanilla compilers. Addition of **polly** to Clang gives a considerable improvement. Runtime is improved by 40% of vanilla Clang run time and 20% of vanilla ICC O3 run time.

Assembly Analysis

Assembly code generated by all 4 variants of the compilation has one thing common, i.e. vectorization and inlining. The major difference is in the degree of the vectorization and inlining of the loops. Another interesting observation is the binary sizes or number of assembly instructions generated by each of them. Other than ICC all other variants have number of assembly instructions in range of 1700 – 2100. The number of instructions generated by ICC is 3x more, i.e. around 6000 lines. The details for each compilation in Figure 2.

one thing that is apparent between GCC and variants of Clang is that performance is inversely proportional to number of instructions, with Clang O3 running slowest and having minimum number of instructions, following CGG O3 and Clang O3 with Polly.

ICC is an outlier here, and can be seem in its assembly that it is doing extra work for initializing some extra features specific to Intel CPUs in procedure: __intel_new_feature_proc_init.

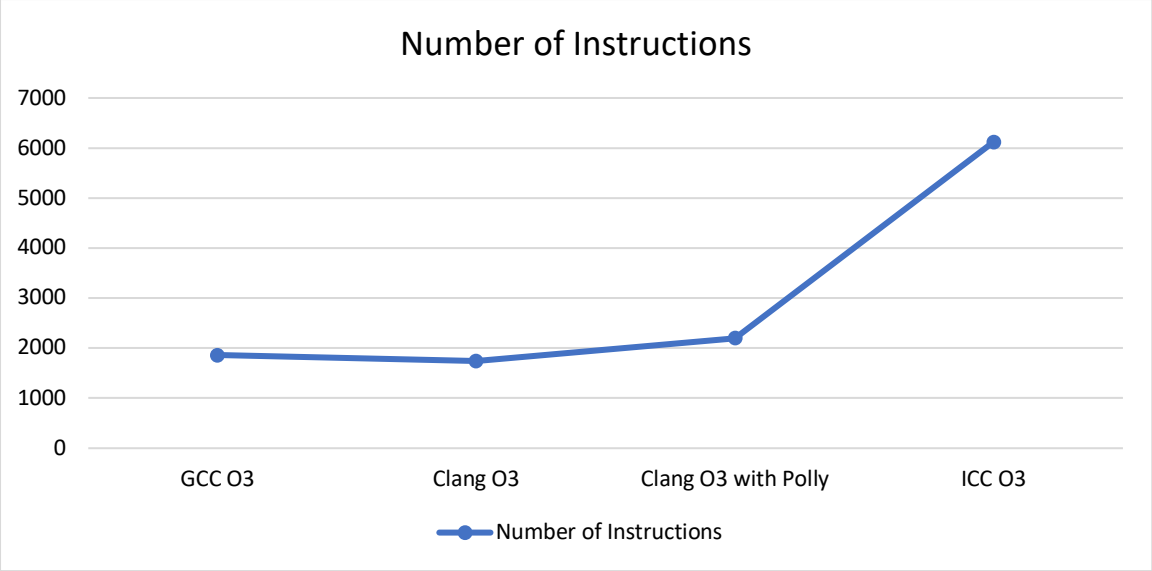

Figure 2
Number of Instructions of various compilations.

The major reason for the performance improvement is the degree of inlining/parallelization and usage of vectorizations instructions by the corresponding compilers. Clang with Polly has the maximum vectorizations and inlining. This variant uses a total of 8 set of vector instructions to perform the calculations and is performing maximum parallelization of the loops. GCC is using only 1 set of vectorizations and is exploiting minimum parallelization of the loops. ICC is also extracting a fair amount of parallelization in the loop by using a total of 8 vector instructions sets to perform the computations.
Here the outlier is the Clang O3 Vanilla version which looks like is extracting better parallelization and GCC and is using 2 set of vector instructions, but it is still giving a poorer performance than GCC. This could be because improper of usage of cache lines and sharing.

Polly Framework

Polly is a loop optimization framework based on Polyhedral abstractions. It works on LLVM IR to extract natural loops.
Traditionally polyhedral optimizations work on the Static Control Parts (SCoPs) of a function. SCoPs are parts of a program in which all control flow and memory accesses are known at compile time. As a result, they can be described in detail and a precise analysis is possible. Polly currently focuses on detecting and analyzing SCoPs.
Polly uses Region based analysis to merge loops and calculate block level transfer functions.

Clang Vs Polly

Comparing the Polly framework with clang for the modified Matrix Multiplication in Figure 3, which has been optimized for Cache reuse, as per studied in class. The runtime can comparisons are present in Figure 4. We can see that Polly runtime is almost same, although for block size of 10*10*10, the runtime for Polly and Clang are the same, and Clang has given best performance for block size of 100*100*100 and no block.

One thing that is apparent is that Polly is not taking advantage of the Block Level code break up. Also, for some block sizes like at 10, Polly is not doing any optimization at all.

```
int B1 = 1;
#pragma scop
  for (ii = 0; ii < _PB_NI; ii += B1) {
    for (kk = 0; kk < _PB_NK; kk += B1) {
      for (jj = 0; jj < _PB_NJ; jj += B1) {
        for (i = ii; i < ii + B1; i++) {
          for (k = kk; k < kk + B1; k++) {
            for (j = jj; j < jj + B1; j++) {
              C[i][j] += alpha * A[i][k] * B[k][j];
          }
        }
      }
    }
  }
 }
#pragma endscop
```

Figure 3
Matrix Multiplication optimized for Cache reuse

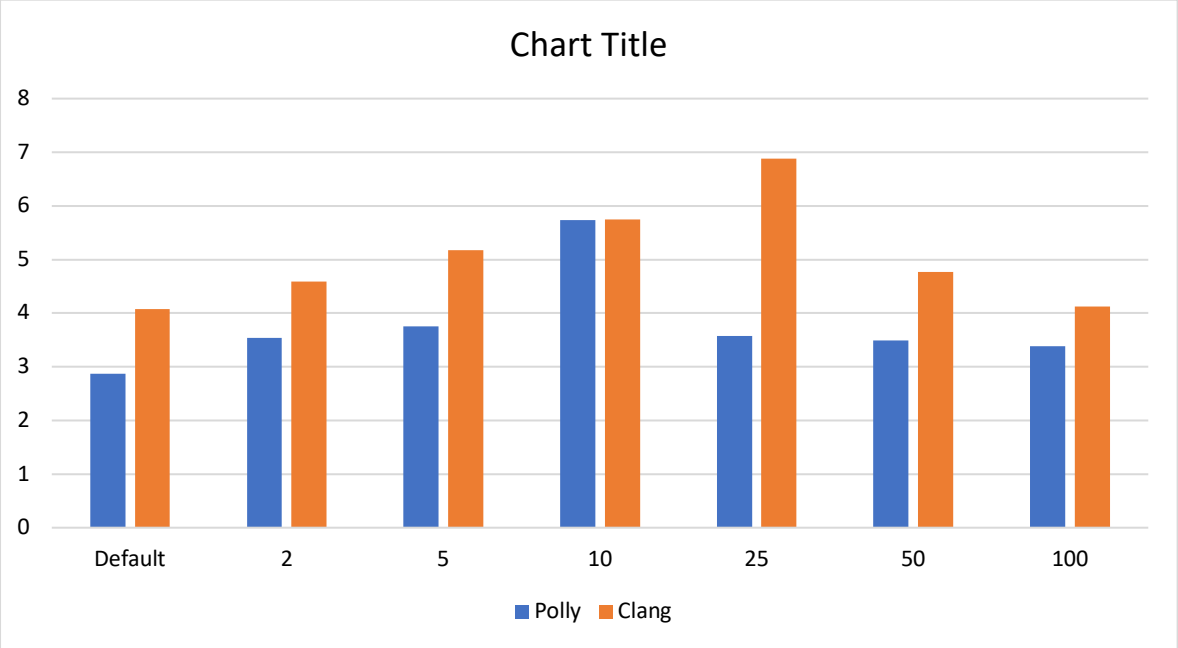

Figure 4
Run Time comparison between Clang and Polly