Analysis of IR Code of CG (Conjugate Gradient) at different optimization levels:

Optimization Level	Total Instruction Count in Module	Number of Functions	Memory Access Instruction count in all Functions	Arthmetic Instruction Count in all Functions	Branching Instruction Count in all Functions
O0	1687	7	1096	152	201
O1	952	7	263	168	159
O2	1672	2	528	341	280
O3	1906	2	604	407	316
Ofast	2116	2	626	452	336
Os	783	2	211	134	150

Observartions :

Function Count: The number of functions decreased with higher optimization levels. From the above data we can see that the number of functions without optimization are seven and with optimization they decreased to 2. This can be attributed to inlining and other optimization attributes.

Memory accesses: The memory access instructions will increase the execution time of the program. Without any optimization, the O0 level has 1096 where as with O1 flag it decreased to 263 because of the use of Phi nodes in place of “alloca” instruction. But with O2 and O3, the number of memory accesses increased which can be attributed to other instruction optimizations that more expensive than memory access.

Instruction Count: The number of instructions increased with higher optimization levels like O3. Also with Ofast flag the instruction count has increased by nearly 30 percent. The compiler merges smaller instructions into a single instruction which causes the decrease in count.

Basic analysis of IR Code generated by llvm for CG (Conjugate Gradient) :

1. Global variables:
    1. All the Global varibales start with @. A module pass can be used to iterate over the list of all global variables.

2. Local variables:
    1. All the Local variables that are inside a function will have % at the beginning.
    2. Another observation is that main function tends to contain an extra local variable **%0** which is not any of the declared variables.

3. Store :
    1. writes the variable value to memory and will be called whenever an assignment statemnet exists in the source code.
    2. Even though we declare and initialize at the same time, the IR code tends to have all the declarations at one place and all the store operations related to initialzation at another place.

4. Load:
    1. This instruction loads the variable value from memory and will be called whenever a variable is used.
    2. The getelementptr instruction which calculates the address will always be accompanied by load because getelementptr cannot access memory.

5. For Loop:
    1. The for loop optimization observations at different levels are explained below. The most important being using Phi instruction to implement the loop from O1 onwards

    O0 - At this level, the compiler tends to represent the for loop with three explicit branches.

    O1 - Here llvm uses “Phi” instruction to implement the loop

    O2 - At this level the compiler tends to geneate all the statements in the loop explicitly until 50 iterations and the variables are also calculated that are inside the loop body. For Example - if the loop is set to run for 20 iterations with a “printf” statement, the compiler will generate 20 “printf” calls explicitly. After 50 iteration count, it uses “Phi” instruction to generate the loop.

    O3 - Simlar to O2 flag, it also explicitly writes all the statements.

    Ofast - This optimization is also simialr to O3 and O2.

    Os - But at this level the optimization is similar to O1 with three explicit branches.

6. Function :
    1. Each function is optimized with certain atrributes and these attributes are grouped under different sets. The attribute set used for the function is mentioned in the IR file and all the attribute sets are at the end of the file.

    2. Under O0 optimization flag noinline attribute is used for every function and for all other optimization flags, inlinine attribute is used, which tries to inline smaller functions.

7. getelementptr :

    This instruction calculates the address of an element in a data structure. But this function does not access the memory of that address and will always be follwed by load to access the calculated memory address.

8. icmp:

    This instruction compares two integers or integer vector and returns a boolean value or vector. It takes the condition code and two operands to perform the comparison.

9. alloca:

    This instruction allocates memory on stack of the currently executing function.

10. br:
    This is a branching instruction which tell the execution to break and start from a different place in the code similar to GOTO command in C

11. SSA:
    Static Single Assignment form is used in LLVM IR representation in which each variable is only used once.

Note:

For question 3, I have taken the following instructions as arthimetic, memory acess and branching instructions:

1. **Arthimetic Instrutions :**
    1. Combination of both arthimetic and bitwise operators are used.
    2. **Instruction Set :**add, sub, mul, fadd, fsub, fmul, udiv, urem, sdiv, srem, fdiv, frem, and, or, xor, shl, lshr, ashr
2. **Memory Access Instructions:**
    1. Combination of both memory access and memory addressing instructions are used.
    2. **Instruction Set** : load, store, alloca, getelementptr, cmpxchg, atomicrmw
3. **Branching Instructions:**
    1. **Instruction Set:** br, indirectbr, switch