Lab1:
COL729: Compiler Optimizations

Priyanka Singla (2018ANZ8387)

February 4, 2019

1 common Things

In this part we present some points which are common to both optimization codes (-O0 and -O2) of LLVM IR and x86, for all the programs.

1.1 common LLVM IR

1. The function arguments and the local variables are stored as named variables, while the other temporary variables are stored as unknown variables.

2. The loops are translated to branch statements.

3. LLVM does not distinguish between the pointer variables or normal register variables or label variables. In particular, all the unnamed variables are assigned increasing numbers beginning from 1. From the rhs of an instructions, the type of the variable can be determined. The 32 bit variables are aligned at 4, while the unsigned integers (i.e., 64 bit variables) are aligned at 8.

4. To get the address of a sub-element of an aggregate data structure, getelementptr instruction is used which only performs the address calculations without accessing memory. This instruction can also be used to get vector of addresses. The index of the element is passed as an argument, and the index has to be 64 bit. If in some cases, the index is 32 bit, it is extended to 64 bit using zero extension (zext) or sext

5. The codes use phi instructions. A 'phi' instruction takes a list of pairs of arguments, with one pair for each predecessor block of the current block. At runtime, the variable in the phi instruction takes the value specified by the pair corresponding to the predecessor basic block that was executed just prior to the current block.

6. The hexadecimal values in the C code are converted into the decimal values and used.

1.1.1 common -O0 (LLVM IR)

There are some common things which are done for all the functions in O0. I will list them as follows:

1. For the function definitions, the arguments and the local variables are allocated space on stack using "alloca" instructions, which returns the pointers to the allocated locations, which are then used to load and store the values. This also reserves a location in stack for storing the return value.

2. Any library functions called in the C code have a corresponding call instruction to the same function, and this function is declared in the bit-code file (at the end).

1.1.2 common -O2 (LLVM IR)

1. There are no alloca and corresponding store instructions. The pointer arguments in the function definition are sometimes annotated with keywords like readonly (which indicates that the variables in the function are not written), and nocapture (which indicates that upon being called by any caller, the callee need not make any copies of it)

2. The last calls to the function are replaced by tail call instruction. This makes much sense when a function is called in a loop or recursively, as upon tail recursion optimization we get the last call. So if a function is not called in a loop, then also -O2 optimization will annotate it as tail call.

Similar to llvm IR, assembly code also has some common instructions, which generally exists in all the programs.

1.2 common x86

1. There are some CFI (Call Frame Information) directives in the assembly code, which describe the layout of the frame and how to unwind the stack upon the function call. They are used for debugging. These include .cfi_startproc, .cfi_endproc, .cfi_def_cfa_offset, .cfi_offset, .cfi_def_cfa_register, etc.

2. In comparison to LLVM IR bitcode, where we allocated space on stack just by alloca instruction and stored the corresponding pointers in different variables, here we have only stack pointer which points to the top of the stack, and the data is carefully stored contiguously by using proper offset and the addressing modes.

3. Corresponding to getelementptr, which returned the address of the element of an aggregate data structure (array), in this, we have 2 movl instructions, first to load the value of the base address of the array, then for accessing a particular index, the corresponding value is added to the base address. For e.g., for accessing the element at index 1, 4 is added to the pointer to get the base address retrieve the value.

4. Unlike LLVM IR where we had infinite number of variables possible, here in assembly we have a limited number of registers (depending on the architecture), so we need to spill the values to memory whenever required. Moreover, we also need to cautiously use the registers so as to avoid any overwrite of useful values.

5. Before making any call to a function the arguments of the call should be pushed onto the stack (caller's stack) which in case of O0 will be then copied to callee's stack.

6. Since we are considering 32 bit memory, any 64 bit number (for example unsigned integer) need to be stored in two registers, and while storing it in memory the order needs to be preserved (lower and upper half).

7. In LLVM IR the memory was only used for arguments and local variables in the C code, while all the intermediate results were stored in the unnamed register variables. In x86, for performing any computation, the data has to be brought into registers (which are available), and then to use the registers for other data, the previous data has to be spilled to memory.

1.2.1 common -O0, x86

1. Some instructions related to setup of the stack and changing the stack and base pointers are common to all the assembly codes (for a function call). These includes copying the arguments from the caller's activation record to callee's activation record via the help of data registers. Further the local variables are also moved to the stack (via the movl instruction).

2. As soon as we enter, we push the previous function's (caller's) base pointer to stack, which is popped just before the callee is to be returned. Stack pointer(esp) is assigned to base pointer(ebp), and then esp is decremented to allocate space on stack for all the required variables.

1.2.2 common -O2, x86

1. It directly decrements the stack pointer by the amount of memory required (program dependent), and at the end, just before returning the stack pointer's value is restored.

2. Directly use the arguments from the caller's stack frame.

3. Whenever any of the register like esp, ebp, esi, edi, etc., are pushed on to the stack, the assembler is informed that the particular register has been pushed on to the stack at a particular offset, using the .cfi_offset directive.

2 GCD

2.1 gcd1

gcd1 C code

1. It is a recursive code to compute the gcd (as an integer) of 2 integers: a and b.

2. The base case used is that b is zero, in which a is the gcd. Else, a recursive call is made in which new a is the old b, while new b is the modulo remainder of old a with old b. In particular, the b keeps on decreasing, and the process will continue, till the b becomes 0.

gcd1 -O0 LLVM IR

1. Signed remainder instruction is used to compute the result of a%b.

2. Before making a recursive call, the value of a is loaded twice, one to compute a%b, and other to pass as an argument.

gcd1 -O2 LLVM IR

1. It converts the recursive implementation to an iterative code, wherein the values of a and b changes in each iteration, and this is achieved by using a phi instruction for each variable.

2. Since in each recursive call in the C code, a%b is used as the new value of b, the iterative version directly compares the srem of a and b, and if it is 0, then the srem of the previous iteration is returned which will be equivalent to the gcd. This works because when srem a b returns 0 implies that b is a factor of a, and hence the gcd of the two numbers.

gcd1 -O0 x86

1. It performs the srem's equivalent in x86 as: it loads a in eax, and then use cltd instruction which converts the signed long in eax to a signed double long and store it in edx:eax by extending the most-significant bit (sign bit) of eax into all bits of edx. Later it calls idivl with b as argument, which takes the 64 bit number represented by %edx:%eax, divides it by the argument, and store the quotient in eax and remainder in edx.

2. The stack is then set up for the recursive call by pushing the corresponding new function arguments (a and b) on to the stack.

gcd1 -O2 x86

1. In this, the comparison if b is equal to zero or not is performed using the testl instruction on a register (Storing b's value) with itself, which is similar to and instruction. Thus it will return zero if the value is 0.

2. For performance gain, again the loop code is aligned at a 16 byte boundary.

2.2 gcd2

gcd2 C code

1. Iterative implementation for finding the gcd. It continue subtracting the smaller number from the larger till they become same. This is similar to doing a%b, where the remainder will be $\leq$ a, and then the arguments are reversed, i.e., the old b becomes new a, and the remainder becomes new b. The same is achieved in continuous subtraction of smaller from bigger, however multiple subtraction might be needed if the bigger number is multiple times larger than smaller.

gcd2 -O0 LLVM IR

1. For performing every comparison and computation, the values are load every time.

2. No separate location for storing the return value was allocated, as the return value was the modified value of one of the variable itself.

gcd2 -O2 LLVM IR

1. Multiple phi instructions have been used to choose the value of a and b, at different places.

gcd2 -O0 x86

1. In each iteration, the 'cmpl' instruction has been used twice, once to check if a and b are equal, the other for checking the less than flag. Also for each comparison, the corresponding variables were loaded from memory. All this resulted in executed a lot of redundant instructions.

gcd2 -O2 x86

1. Unlike -O0, this optimization avoids the redundant compares. Rather than checking which number is greater, it subtracts one number (after taking a backup) from the other and then check the carry flag. If the carry is set, then the subtraction is performed the other way round (using the backed up number). This happens in a loop till the numbers become equal.

2. The loop has been aligned to 16 byte boundary, using nops, for performance gains.

2.3 gcd3

gcd3 C code

1. It is similar to gcd1, i.e., it uses modulo remainder but in an iterative code, and uses a temporary variable for updating a by old value of b.

gcd3 -O0 LLVM IR

1. It uses srem instruction to compute the remainder i.e., a%b.

gcd3 -O2 LLVM IR

1. If srem results in 0, then rather than assigning a with the old b and then returning a, directly the old b (which resulted in 0 remainder) is returned.

2. Phi instruction is used to choose the returned value: either a (initially when b == 0) or b (when srem returned 0).

3. Rather than checking the exit condition at the beginning of each iteration, the check is done once before entering the loop for the first time, and then rather than in the beginning, the check is done at the end of each iteration.

gcd3 -O0 x86

1. cltd and idiv are used to perform modulo division (a%b).

2. For the temporary variable, an additional space is allocated on stack.

gcd3 -O2 x86

1. testl is used to check if b is 0.

2. The iterative loop is aligned at the 16 byte boundary.

3. No additional temporary memory location (i.e., stack) is used, instead a register is used.

4. As also mentioned in gcd3 -O2 LLVM IR, the check for each iteration is done at the end.

3 emptyloop

emptyloop C code

1. The file contains two macros, one defines a number which corresponds to the number of iterations for which for loop is run (in case the first argument to the function is < 2). If however, the first argument ≥ 2, then the maximum of the 2^{nd} element of the 2^{nd} argument (as the 2^{nd} argument is the pointer to character array) and the element defined by first macro is the number of iterations for which the loop is executed. This max function is defined as a macro. There are no instructions in the loop, and the function returns 0.

2. Iterator used i, numiter and MAGIC_NUMBER are the 2 numbers, the maximum of which decides the number of iterations.

emptyloop -O0

1. atoi function is called which returns a 32 bit number which has to be stored to the numiter which is a 64 bit number. So sign extension (sext) is performed on it and stored in numiter.

2. The comparison between numiter and MAGIC_NUMBER is also performed in every iteration which is redundant, as both numiter and MAGIC_NUMBER are constants and the result of their comparison does not change. However, -O0 does not perform any optimization and emits code exactly same as the C code. Also, if numiter > MAGIC_NUMBER, then the variable numiter is again loaded before comparing it with the iterator, which I believe could have been avoided, as no one is changing numiter in between and since we are using SSA we can directly access the previously loaded value.

3. Phi instruction is used to capture the maximum of numiter and MAGIC_NUMBER, which is then compared with i.

4. While incrementing i, it is loaded every time, which again seems redundant, as we already had the latest value in the previous step (while performing the comparison).

emptyloop -O2

1. Rather than checking if argc ≥ 2 as in O0, here it checks if argc >1.

2. The atoi call has been replaced by strtol() call, which returns a 64 bit value, thus we need not do any sign extension. Another probable reason for replacing atoi with strtol is that as mentioned in the man page of atoi, atoi does not detect errors (unlike strtol).

3. Since the loop did not do anything, it got removed in the bitcode.

4. As mentioned in the common part, the arguments in this are marked with nocapture, as the pointer is only dereferenced and not stored and hence no copies were made by the function.

emptyloop -O0 x86

1. The function call to atoi takes only one argument, so this argument is first stored on to stack, and then the function is called. The result of this is implicitly stored in register eax.

2. Since atoi returned 32 bit number, while the result was assigned to a 64 bit variable (numiter). This is achieved by performing arithmetic shift (sarl: which preserves the sign of the source operand) by 31. Then both the 32 bit registers (in combination sorting the 64 bit data) are copied to stack in proper order.

3. Also, the ebx, esi are pushed on to stack and the stack pointer is decremented by the memory which is used in the function. These operations are performed in reverse order when the callee is just about to return (i.e., esp is incremented by the same amount i.e., restored, esi and ebx are popped respectively).

4. When the iteration begins, each required variable is loaded into the registers, but after performing some computations, we need more registers to store the values. But as number of available registers are less,

then the values are spilled on to stack, which will be later reloaded. This results in a large number of additional instructions. This happens upon each iteration, and will impact the performance.

5. For finding the max between numiter and MAGIC_NUMBER, the assembly code performs a subtract instruction and correspondingly set a byte in a register (using setb), which is later checked by 'testl' instruction, i.e., the AND of the register value with itself is performed, and depending on, if the value is 0 or 1 gives information about the subtraction result and hence the max value. Please note that since the variables (in C code) were 64 bit, so the corresponding registers were subtracted to get the correct result.

6. The phi instruction was used in the LLVM IR code for choosing the max value. In assembly, depending upon the result in previous step, different branches are taken, and these branches store the value of the resulting max in the same locations. So since only one branch will be actually take, this will behave similar to the phi instruction.

7. Again for comparing the iterator's value (i) with the max value, the same logic of subtraction, set and test is used.

8. Since the return value is 0 (in the C code), this is achieved in assembly by performing a xor on eax with itself and it is considered as the default return register.

emptyloop -O2 x86

1. There is no code for the loop, also nothing regarding the macro MAGIC_NUMBER and the iterator variable i.

2. Before making a call to strtol, its required three parameters are pushed onto the stack.

3. By performing xor of eax with itself returns at the end, results in the function returning the value zero.

4. the return value of the function is not pushed on to the stack unlike in -O0, and register eax is implicitly assumed to be the return register.

4 print_arg

print_arg C code

1. This function has 2 arguments. The function checks the value of the first argument, and if it is not equal to 2, it returns -1, other wise it prints the second element of the second argument which is a string and has '%s' format specifier, and returns 0.

print_arg -O0 IR

1. The format specifier is stored as a global private constant vector of size 3, with unnamed address (here the address does not matter as we only need value), with the value as "%s", followed by '0' character.

2. printf's signature in LLVM is as follows: It returns i32 and takes atleast one i8* (i.e., pointer to a character). In the given code, the printf had 2 arguments: i) the format specifier "%s": which is retrieved using the getelementptr on the global private constant mentioned above, and ii)the element to be printed.

3. Depending upon the path taken, the value of -1 or 0 is stored in the stack location reserved for storing the result.

print_arg -O2 IR

1. tail call

2. Rather than storing the result value along different values (as in -O0), here the result value is chosen using the phi instruction. This results in reducing the number of instructions in comparison to -O0.

print_arg -O0 x86

1. The global variables are defined outside the function definition (i.e., outside .cfi_startproc and .cfi_endproc) using a label.

2. The constant string (format specifier) is stored in .rodata.str1.1 section and is of type object (as specified in the .type directive, which is used to tell the assembler the type of the symbol).

3. The format specifier is assembled using the .asciz (z because the string "%s" is followed by zero byte) assembler directive.

4. leal (loads the effective address) instruction is used to compute the effective address of the symbol (which has the format specifier value) in the eax register, and then the address of the specifier object is pushed onto the stack, along with the 2^{nd} argument of the print_arg function, before calling the printf function.

5. The .size directive has been used to tell the assembler that the object will be using 3 bytes.

6. Before returning, the value 0 is to be pushed on the stack and also stored in eax. So to do this, if any previous value is there in eax it has to be moved to the stack, i.e., the value has to spilled.

print_arg -O2 x86

1. Instead of first determining the effective address (using leal), and then pushing the corresponding value on stack as in -O0, -O2 directly pushes on the stack using the immediate addressing mode for the label (or symbol) defining the object.

2. Rather than explicitly putting 0 into the register eax, a an xor of eax with itself will perform the same action (and in various cases will prevent spilling, as it happened in O0 case)

5 fib

fib C code

1. The function returns the $(n)^{th}$ element of the Fibonacci series (where n is the argument to the function), by the use of recursion.

fib -O0 IR

1. Within the function definition with parameter n, 2 independent calls to the same function, with parameters n-1 and n-2, are made. The call to f(n-1) will in-turn call f(n-2) and f(n-3). Thus there will be multiple calls to f with same parameter value, and the number of calls become exponential as n decreases.

fib -O2 IR

1. This converts the code into an iterative form, i.e., which will iterate for approximately n/2 times (where n is the parameter in the function definition).

2. In each iteration, only one recursive call is made (each with different value of n). 2 types of phi instructions are used: One phi instruction is used to set the value of changing n. The other phi instruction is used to store the changing result of the Fibonacci series which will be initially 1.

fib -O0 x86

1. The result of the 1^{st} recursive call will be stored in the register eax, so before making the 2^{nd} recursive call, the contents of eax have to be spilled in memory.

fib -O2 x86

1. Alignment by 4 byte is used (2^4) by padding with NOPS i.e., 0x90.

6 fibo_iter

fibo_iter C code

- This function is an iterative version for computing the n^{th} number of the Fibonacci series. The argument and the return values are unsigned int and unsigned long, respectively.

- If n<3, the function returns 1.

- It uses 2 variables cur and prev (both are initially 1), and then in each iteration cur is updated to prev+cur, while the prev stores the old value of cur (i.e., before the updation)with the help of a temporary variable. The value of cur after is last iteration is the result.

fibo_iter -O0 LLVM IR

- In each iteration, the values of the iterator, i, and n are loaded from memory.
- Since the code use a mixture of unsigned long and unsigned int variables, so during corresponding assignments, trunc (to decrease size from 64 to 32 bits) and zext (increasing size from 32 to 64 bits by filling the higher order bits with 0) have been used.

fibo_iter -O2 LLVM IR

- It uses 3 phi instructions for the iterator, prev and cur variables, as these variables have some value before the loop begins, and the values get changed in each iteration. So, at runtime, depending upon the execution, the corresponding value can be chosen.
- This eliminates the use of temporary variable, and also corresponding trunc and zext instructions, by performing the logical AND of the 64 bit 'cur' with 4294967295 (which in binary will be 32 1's, i.e., in terms of 64 bits it will be 32 zeroes followed by 32 ones).

fibo_iter -O0 x86

1. The esi register is used to store the data value, so to prevent the old value (might have been set by the caller) from being trashed, the esi register contents are pushed onto the stack and are popped immediately before returning.
2. Since in the comparison of (n<3), n is an unsigned integer, the corresponding jae (Jump if Above or Equal) is used to check the carry flag.
3. Since the return value is of type long, this is done by the use of 2 registers, namely edx and eax. For example, if value 1 is to be returned, then 0 is stored in edx and 1 is stored in eax. Both of these combined return a 64 bit value of 1.
4. Similarly, the value 1 for 64 bit variables cur and prev are also accessed using 2 mov instructions. Even while adding cur to prev, it is done

in parts, i.e., initially the lower halves are added, and later the upper halves.

5. The truncation of cur to 32 bit value is done by storing only the lower 32 bits of cur to memory. Similarly, the corresponding zext is done by reading the 32 bit into lower half, and then storing 0 in upper.

6. Finally, when all the iterations have finished, vector move (vmovsd) instruction is used to read value to xmm register, and then copy it back to the proper location (the same location where 64 bit value of 1 is stored when $n < 3$) in the stack.

fibo_iter -O2 x86

1. The ebp, ebx, esi, edi, registers are pushed onto stack (will be later popped).

2. Rather than explicitly moving 0 to a register, it performs xor of the register with itself, as it is an optimized way for zeroing a register. []

3. For achieving good performance, aligning the branch targets, especially the loops, at a 16-byte boundary. This is followed in the code.

4. It does not use temp to store the cur value, rather it use temp to store the sum of prev and cur, and then prev and cur are updated from cur and prev values, respectively. Since temp in the C code is an int (rather than long), so only the lower 32 bits of sum (prev and cur) are stored in temp, while the upper 32 bits are directly stored in the destination register.

7 loops

7.1 is_sorted

is_sorted, c code

- It iteratively checks if an array 'a' of size 'n' is sorted in increasing order, and returns true if sorted, else returns false.

- An empty array is considered to be sorted by default.

is_sorted, -O0 LLVM IR

1. Returns a 1 bit zero extended (zext) value (i.e., a boolean variable).
2. During each iteration, for comparing the iterator i to n-1, n is loaded and 1 is then subtracted. Also, the computation of n-1 is done in each iteration which is redundant, as n is never modified in the entire code.
3. For loading an element of the array, multiple instructions are used: first load i, perform sign extension to it (using sext) and coverts it to 64 bit, use getelementptr to get the address of a[i] and finally load the value pointed by this address.

is_sorted, -O2 LLVM IR

1. As the array is never modified, the array pointer is annotated as nocapture readonly in the argument list.
2. Unlike -O0, -O2 performs sign extension to 'n-1'. This optimization is performed to avoid performing sign extension on i. In particular, now n-1 is converted to 64 bit, and i is by default considered to be 64 bits and all the comparisons and array accesses can be seamlessly performed.
3. There is no separate code for incrementing the iterator variable i in successive iterations (in -O0, i+1 was performed 2 times in each iteration). This is because, while comparing a[i] with a[i+1], i has been already incremented. Thus this value is reused via a phi instruction to update the iterator.
4. Also in -O0, the end results were stored in the result location via 2 different paths, while in -O2 they are stored only once using the phi instruction.

is_sorted, -O0 x86

1. To load the elements at different indexes of the array, base-indexed addressing mode is used, i.e., The address is of the form "displacement(base register, index register, scale factor)", which is computed as "[base register + displacement + index register * scale factor]"

2. The zero extension of the result is performed by using movzbl (which moves the zero extended byte to long, after padding the 8 bit value with zeros), on an 8 bit register.

is_sorted -O2 x86

1. Unlike -O0 which does not perform any extension to n-1, here the extension is performed using sarl (shift arithmetic) instruction. Similarly, i is stored as a 64 bit number by using 2 registers. So for comparing i and n-1, the corresponding lower halves and upper halves are compared separately.

2. For finding which one is greater than the other, initially setae instruction is used for the lower halves. This set the byte in the register if the CF = 0. However, for comparing the upper halves setge instruction is used, which sets the byte if sign flag is equal to overflow flag. This is because the most significant bit will be a sign bit, and hence checking only the carry flag won't suffice.

7.2 add_arrays

add_arrays: C code

1. This function iteratively adds 2 arrays and store the element-wise sum in the 3^{rd} array.

2. Input: Pointers to the arrays a,b,c (where result is to be stored), n: number of elements in each array.

add_arrays: -O0 LLVM IR

1. In each iteration, for accessing the array elements, the index is sign extended to 64 bits (using sext), and then getelemntptr is used to retrieve the pointer the array and the corresponding value is loaded. For accessing the element at the i^{th} index for each array, i is loaded 3 times and sign extension is performed on it, which is extremely redundant.

add_arrays: -O2 LLVM IR

1. In the function definition, the argument c is not annotated with read-only, as the sum of the array elements will be stored to it.

2. It computes some meta-data like the last index (of size 64 bit) of the array, and checks if n is less than 8, and performs vector addition if yes, else compute sum one element at a time as described below.

3. In particular, a number (say n') which is a multiple of 8 but less than n is computed by performing logical AND of n with 8589934584. 8589934584 in binary is equivalent to a representation where the 0-2 bits are zero, and the remaining are one.

4. If the new number n' is greater than 0, then the sum is performed for elements at 8 indexes in one iteration. In particular, using getelementptr, the pointer to a 32 bit element at an array index is obtained. Then using bitcast instruction, the type of this pointer is converted to $<$ 4 x i32$>$* (i.e., point it to a vector of 32 bit elements). Then the 4 consecutive elements are loaded, added to the 4 consecutive elements from the other array, and finally stored to the 4 consecutive indexes in array c. The index is then incremented by 4 by ORing it with 4, and the same procedure is followed to compute the elements wise some of the elements of the next four indexes. In other words loop unrolling is done for 2 iterations. The number of additions performed using vectors decrease substantially in contrast to number of scalar additions.

5. If the new number (n') is 0, it indicates that there are less than 8 elements in the arrays (i.e., $n < 8$). If the less than 8 elements are left (either because $n < 8$, or $n' < n$ and n-n' elements are left), it is checked if this number is odd or even by ANDing it with 1. If the number is even, then the sum of array entries is performed in an unrolled loop (i.e., 2 iterations are unrolled thus reducing the iterations by half). If however, the number is odd, then sum of only 1 index is performed fist, and then again the sum of elements at 2 indexes is computed and stored in each iteration.

6. The code does not blindly perform vector addition blindly, but performs some memory checks to see if the array c is overlapping with any of the two other arrays. In particular, it checks if array c either completely

overlaps with a, or some later portion of c overlaps with some beginning portion of a. If such a case happens, then rather than performing vector sum, the numbers are added one by one.

add_arrays -O0 x86

1. Register indexing is used to retrieve the element from the base address of any array. i.e., base address + (array_index *4). In comparison to LLVM's getelementptr and corresponding load is equivalent to accessing data by indexing.

2. In every iteration, i is loaded 3 times, one to compare with n (which inturn is also loaded in each iteration), then to access the elements of the array, and finally to increment i (after which it is stored back to memory).

add_arrays -O2 x86

1. To find n' (max value which is a multiple of 8 and less than n), n is logically ANDed with 2's complement of 8.

2. To perform the memory checks, initially the addresses of the arrays are computed using leal instruction, and the control flow happens accordingly.

3. SIMD instruction vmovdqu is used to move(load) the set of values into xmm registers. These are 128 bit registers, and can store 4 32 byte values. Once 4 values of an array are loaded, they are added to the same index values from other array (accessed from memory using register indirect addressing) using vpaddd instruction. The result is stored in the xmm register (which was one of the source), and is finally stored in the array 'c' using vmovdqu instruction.

4. To access the 2nd four bytes (unrolled loop), the index was incremented by 4 by ORing it with 4.

5. The loop which performs vector addition is 16 byte aligned. Similarly, the loop which performs the sum of an element (2 iterations have been unrolled) at a time is also 16 byte aligned.

6. To compute, if the remaining number of elements are odd or even, testb with 1 is performed on the register (Storing the number of elements)

7.3 sum

sum: C code

1. It iteratively computes the sum of first 'n' elements of an array 'a'. The array is passed as an unsigned character pointer along with the argument n. The computed sum value is then returned.

sum: -O0 LLVM IR Most of the steps are similar as in prior functions.

1. In each iteration, i is loaded 3 times: i) to compare with n, ii) to access the array element, iii) to increment for the next iteration.

2. Also, i is sign extended to 64 bits, for using it to get the address of the element via getelementptr.

3. Since, the array is of type unsigned character, but the return value is 32 bit, so the value loaded from the getelementptr address is zero extended from 8 bit to 32 bits, which is then added to the 'ret' variable.

sum: -O2 LLVM IR A lot of instructions and the procedure is similar to that of add_arrays LLVM IR -O2 code.

1. It checks if $n > 0$. If no, then code flows to the exit branch.

2. n' is computed by performing logical AND of n with 8589934584, and then n' is compared with 0. If they are equal, it indicates that n is a multiple of 8 and all the elements can be added via a single procedure. If however, they are unequal, then elements till n' are accessed using vectors, while the remaining are accessed as scalars.

3. While performing addition using vectors, 2 vectors each of size 4 (each element 32 byte) are initialized to 0 using LLVM's string zeroinitializer (which initializes a value to zero of any type). The address at a particular index is retrieved using getelementptr which is then bitcast from i8* to < 4 x $i8 >$*, and the group of elements is loaded. The same is repeated to load the next 4 elements (due to the unrolled loop).

4. Since the array elements are of unsigned char type, but the computed sum is an integer, so they are zero extended (zext) to i32 type before performing the addition. These loaded values are added to the zero initialized vector. Then the index is incremented by 8 to perform the next iteration, and the new vector elements will be added to the sum computed in this iteration. This happens till the iterator reaches the value n' (i.e., all the sets of size 8 have been processed). By the end of this, we will have 2 vectors each with 4 elements, and the elements are the sum of the elements of the input array a.

5. The 2 vectors are then added to obtain a single vector. To perform the addition of elements in this vector, shufflevector instruction is used, where in this vector is shuffled with another undef vector of the same size, and the shuffle mask operand is of the form <4 x $i32 > < i32$ 2, i32 3, i32 undef, i32 undef>, i.e., the resultant vector will be of size 4 and will have elements of type i32, and the elements at index 2 and 3 of our vector will be stored at the first 2 indexes of the result vector. Then our vector is added to the resulted shuffled vector, and we get a new vector whose first 2 elements are the sum of first-third and second-fourth elements of the original vector.

6. Similarly a shuffle is again performed to have the final vector where the first element is the sum of the entire original array. Then the llvm's extractelement instruction is used to extract the scalar element at index 0 (i.e. the sum)

7. On the other path, i.e., when the remaining elements are less than 8, then it is determined if the elements are are multiple of 4 by anding them with 3. If they are, then they are added in an iterative form, 4 elements in 1 iteration (due to loop unrolling by a factor of 4). Else they are iteratively added, one element at a time.

sum: -O0 x86

1. Elements of the array 'a' are accessed using the register indirect mode, and since the array is a character array, the elements are moved to only the lower half of the register (e.g. bl)

2. Then using "movzbl" instruction, the contents are converted to 32 bit number.

sum: -O2 x86

1. The initial steps are similar as in add_arrays, i.e., initially 'testl' instruction is used to see if $n > 0$. n' is computed by ANDing n with -8.

2. Analogous to LLVM IR's -O2 representation, where 'zeroinitializer' string was used, here 'vpbroadcastd' instruction is used. In particular, 'vpbroadcastd .LCPI2_0, %xmm2', this instruction broadcasts the double word integer at the label LCPI2_0 (which referred to 0xff) to four locations in xmm2.

3. To load the 4 consecutive elements of the array, 'vpmovzxbd' instruction is used (i.e., it performs move with zero extended operation for a vector of values), and the values are stored in a xmm register. These values are iteratively added (in a 16 byte aligned loop) using 'vpaddd' instruction.

4. When we are left with a single vector of 4 values, then 'vpshufd' instruction is used. In particular, 'vpshufd $78, %xmm0, %xmm1' is used, which shuffles the contents of xmm0 in such a way that xmm1= xmm0[2,3,0,1], i.e., the element at 2nd index in xmm0 is at the 0th index in xmm1.

5. After shuffling, 'vpaddd' instruction is used to add the elements. Then finally, 'vphaddd' instruction is used to add the two adjacent 32 bite elements horizontally and then the result is moved to a 32 bit register.

6. The remaining part is same as described in LLVM IR -O2, and there are no complex (or new) instructions. Briefly, the numbers are either added one at a time (in one iteration), or 4 in one iteration (i.e., loop unrolling by factor of 4).

7.4 sumn

sumn: C code

1. It iteratively computes the sum of first n integer numbers starting from 0 (i.e., n is excluded, so we have (0+1+2+...+(n-1)) using a for loop, and returns it.

2. Input: n, output: sum of the numbers, iterator: i

sumn: -O0 LLVM IR

1. In each iteration, it loads i three time: i) to compare i with n, ii) to add i to ret, iii) increment i for the next iteration.

2. Similarly, n is loaded in every iteration. This could have been avoided as n is never getting changed.

sumn: -O2 LLVM IR This formulation eliminates any kind of loop, and optimizes it to use the formula for sum of n natural numbers (i.e., $\frac{(n-1)*n}{2}$ since starting from 0). In particular, the following steps are taken:

1. Directly check if $n > 0$. (This check was not made in -O0)

2. if true, then 1 is subtracted from n and the result is zero extended from 32 bit to 33 bit (i.e., we get (n-1)). This extension is done because, later when multiplication of two 32 bit numbers will be performed, it might result in overflow. Similarly n-2 (extended to 33 bit) is computed and assigned to an unnamed variable. These two terms are then multiplied and the result is then left shifted by 1 using lshr instruction(to perform division by 2), and the result is truncated from 33 bit to 32 bit using trunc instruction. Thus, at this point we have computed $\frac{(n-1)*(n-2)}{2}$

3. We add n-1 to it, thus resulting in $\frac{(n-1)*n}{2}$, which is returned by the function.

sumn: -O0 x86

1. i is read into registers 2 times, one while comparing with n (directly from memory), and other while performing increment. The addition of i to ret variable is done directly done by accessing i from memory directly.

sumn: -O2 x86

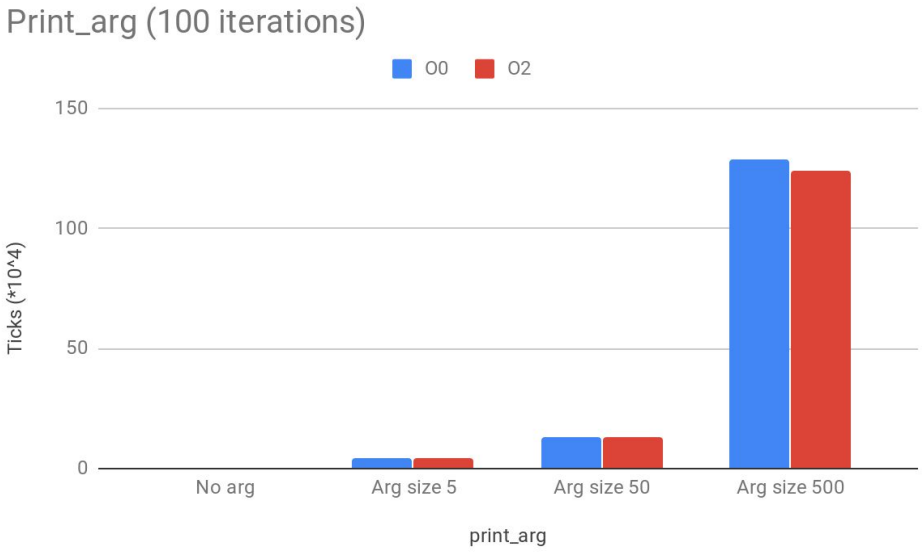
1. It initially checks if $n < 0$ by a testl instruction and exits if the condition is true.

2. It computes n-1 and n-2, using leal instruction. In particular, "leal -1(%ecx), %edx" instruction is used. Here, the register ecx stores n (the equivalent C variable).

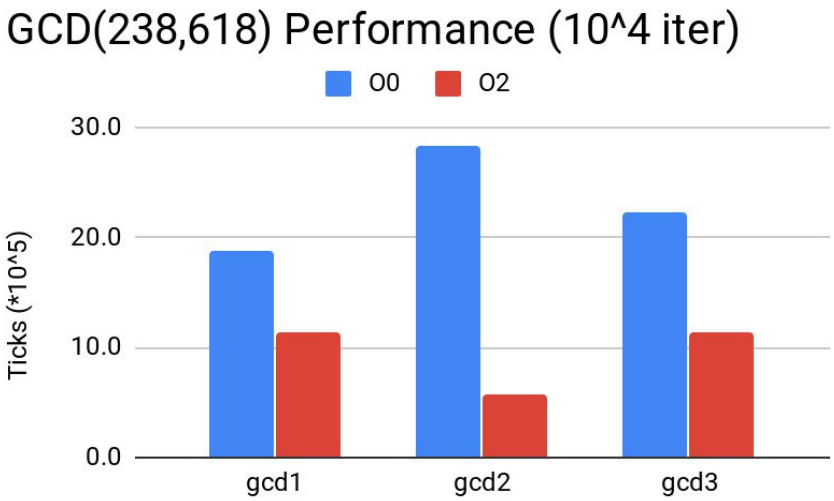
3. It then computes (n-1)*(n-2) using mulxl instruction, then shldl instruction (with shift of 31 bits) is used.

Following, I summarize the results of all the experiments. The **ticks** on y axis represent the **time** (in ms) taken.

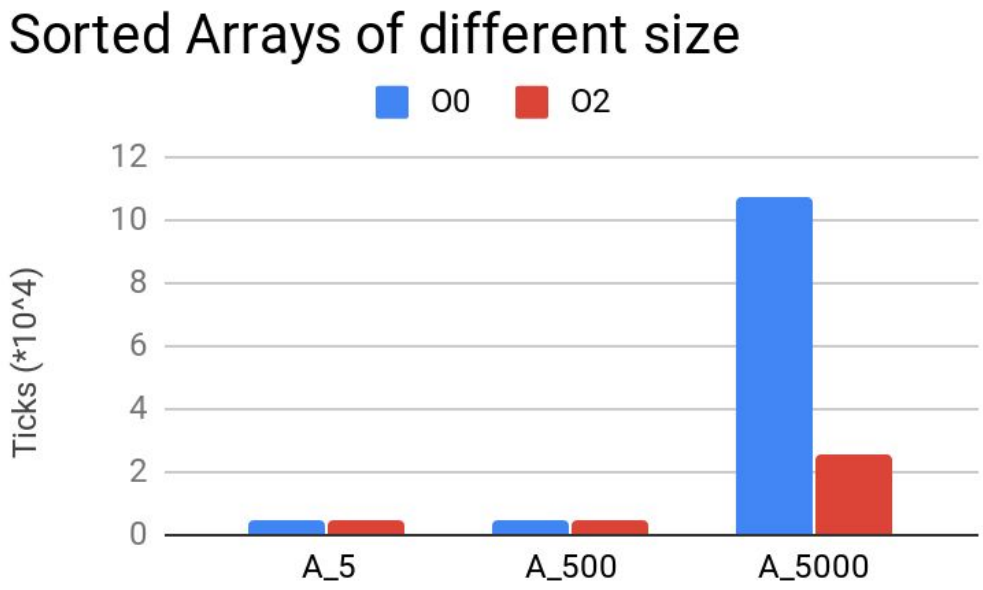
1. **Print_arg()**: I ran it for 100 iterations. Initially when no argument was passed, the functioned returned almost in 0 time. When an argument string was passed, initially the function took very less time, but it increased proportionally with increase in argument's length. However the results were **similar** in both the optimization levels. Similar results were observed for iterative version and loop unrolled with a factor of 100. I have shown only the iterative results.



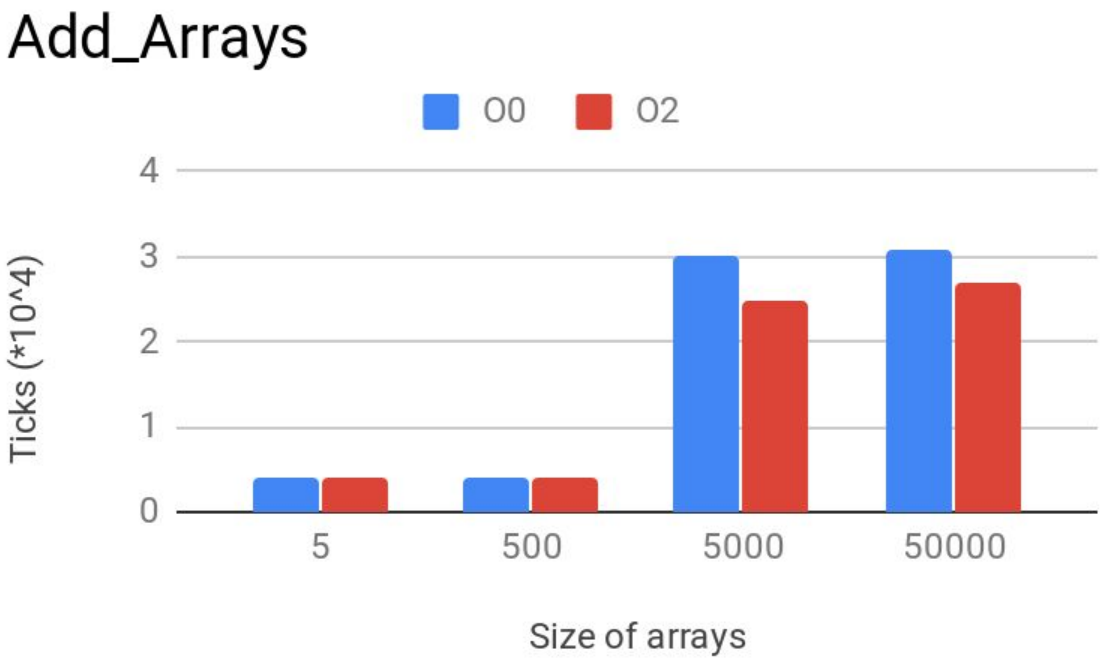
2. **gcd()**:The result will depend on the inputs a and b. If a is a multiple of b, then this will result in extremely fast computation (hence much better performance of gcd1 and gcd3).



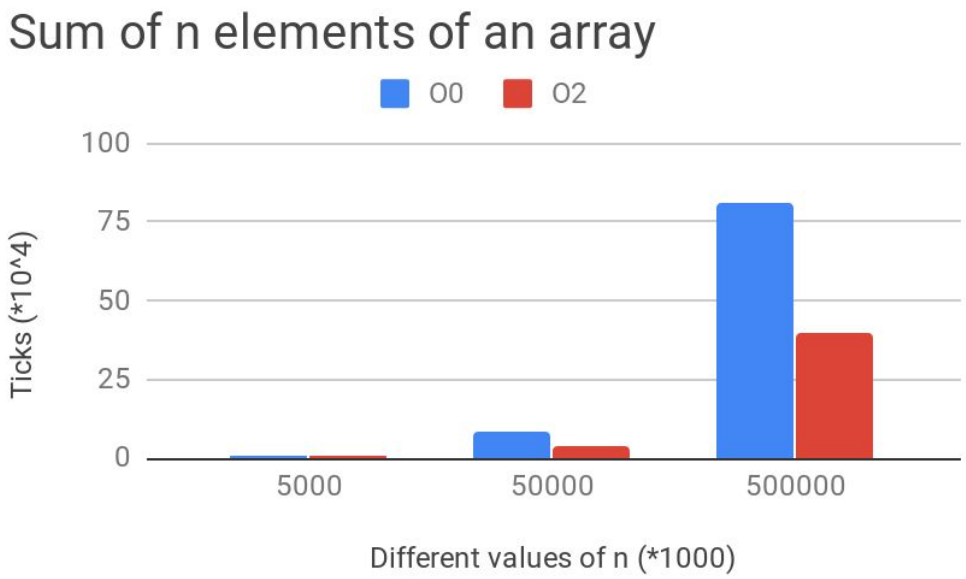
3. **Loops**
    a. **is_sorted():** The experiments were performed with different size of arrays. Initially when the size of the arrays were small, both O0 and O2 performed similarly. However, as the size increased, for sorted array, the O2's performed much better than O0. However, for unsorted array, the performance of both O0 and O2 were same irrespective of the size as the result was probably received during the first few comparisons, unlike in sorted array were the comparisons equal to the number of elements in the arrau were performed. The sorted array had numbers from 1 to its size, while the unsorted array has random numbers with the range [1,size], and the arrays were populated using a loop.



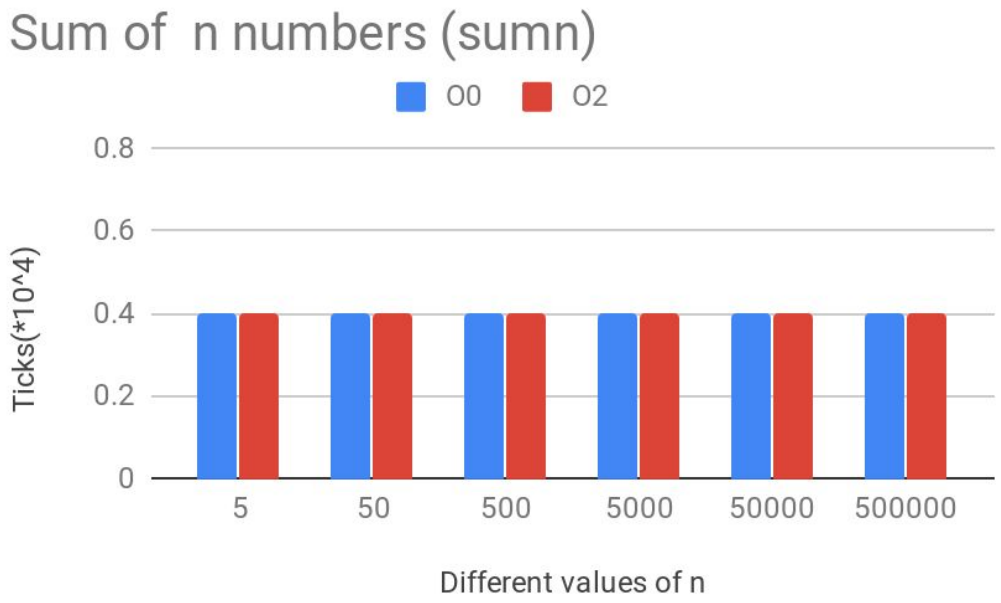
b. **Add_arrays():** The time for computation increased with size for both the optimization levels. However, at large size O2 showed better performance than O0. This performance gain might be due to the vector instructions used.



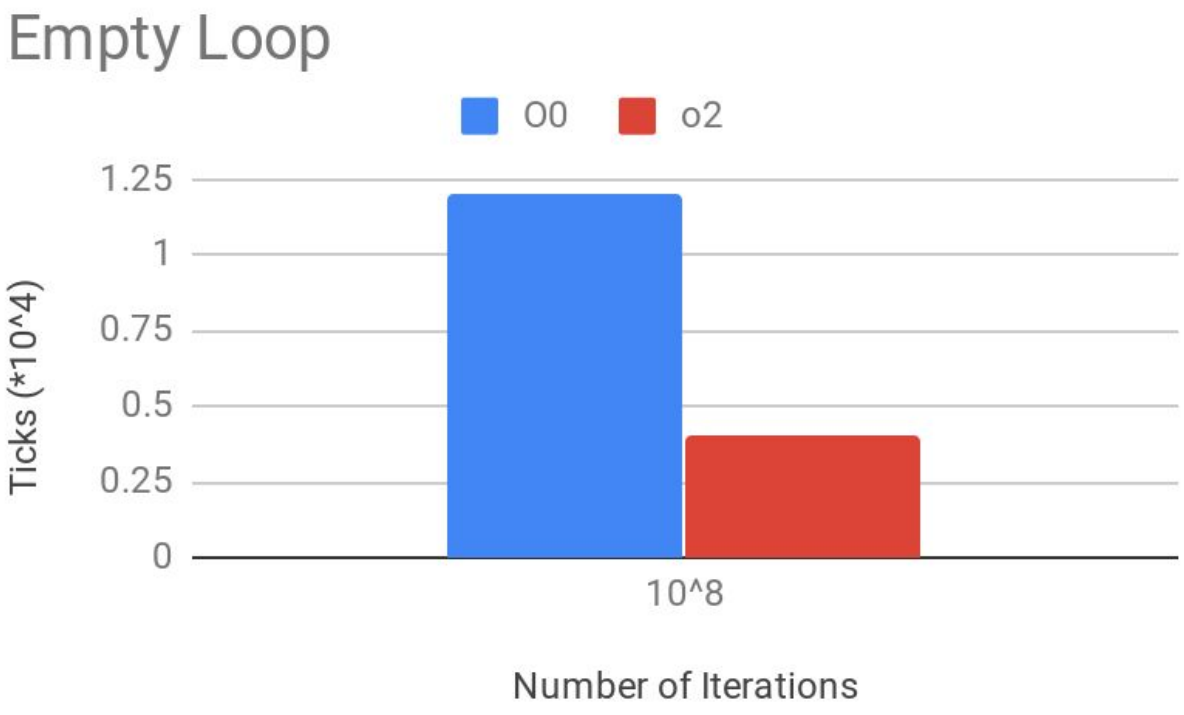
c. **Sum():**



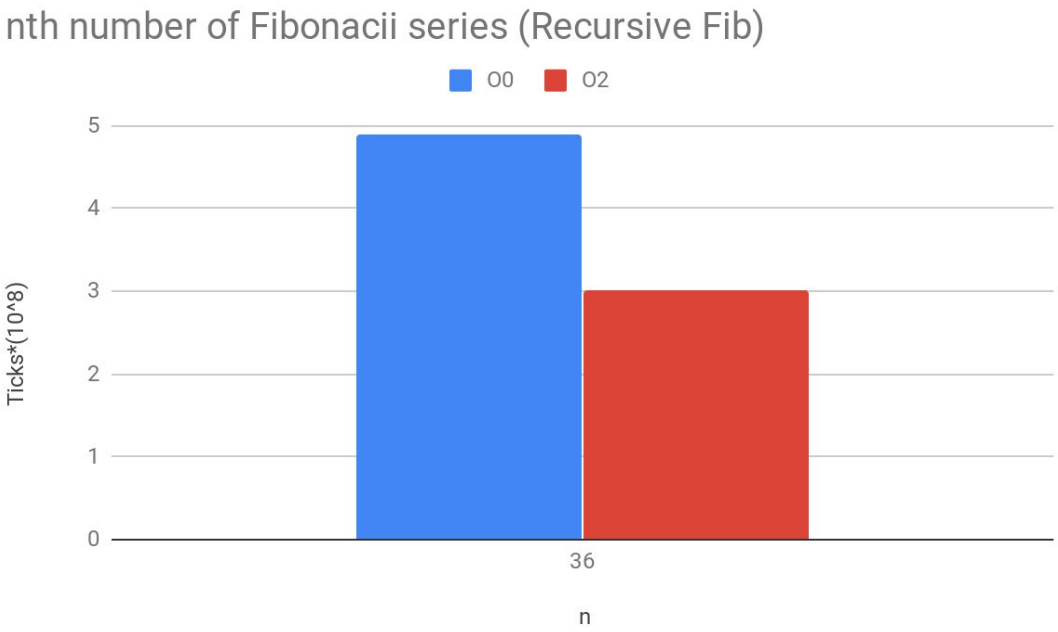
d. **Sumn():** The performance was almost similar in both the cases, irrespective of the numbers to be added. In the assembly code of O0, it can be seen that it performs an iterative addition, while O2 computes according to the formula.



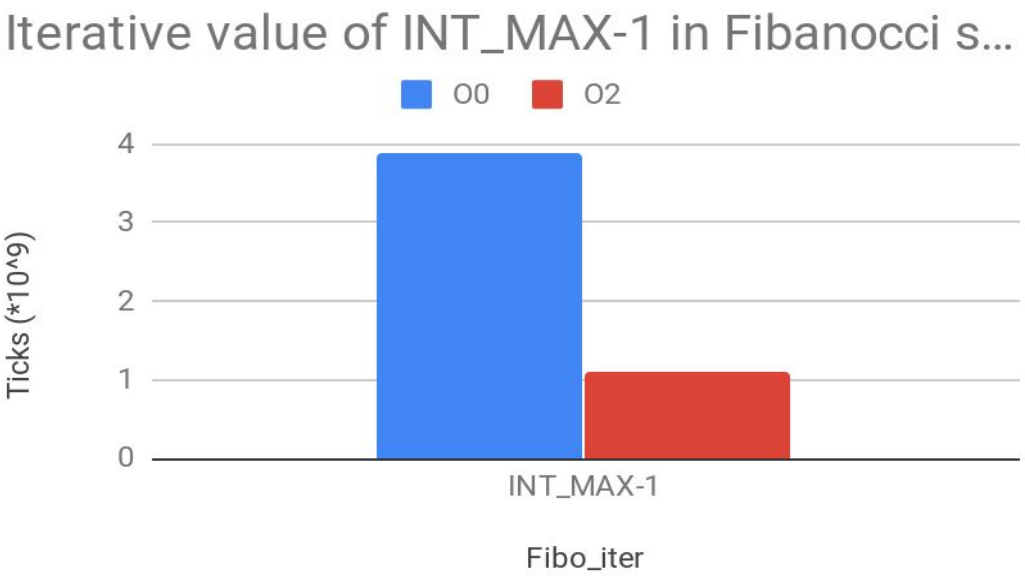
4. **Empty loop():** O2 optimizes the loop and performs better than O0.



5. **Fib():** Again O2 has better performance than O0. And the time increases drastically when n increases 36 from 35.



6. **Fibo_iter():** In the iterative version again, the O2 performs much better than O0. Moreover, in comparison to recursive implementation, the iterative implementation performs extremely better



7. **Size (in KB) of each file:** I also checked the size of the assembly files for all the files. The results show that the size of O2 is less than that of O0. However, for loops.c, the assembly file corresponding to O2 has almost double the size of that of O0. This is due to the complex logic of memory check, and the vector instructions in the O2 code.

Size of different files