LABORATORY 5

3. Execute the my_program and measure the time of execution for different compiler optimization options (eg. O0 and O3)

- With **OPT = -O0**

```
ifranl00@ubuntu:~/Desktop/lab5/matrix$ ./my_program
start
end
Time: 0.387103
TEST
```

- With **OPT = -O3**

```
ifranl00@ubuntu:~/Desktop/lab5/matrix$ ./my_program
start
end
Time: 0.368025
TEST
```

4. Obtain the assembler code files for different levels of compiler optimization (eg. O0 and O3)

•gcc -S -O0 mat_vec.c

•cp mat_vec.s mat_vec.gcc_O0

```
ifranl00@ubuntu:~/Desktop/lab5/matrix$ gcc -S -O0 mat_vec.c
ifranl00@ubuntu:~/Desktop/lab5/matrix$ cp mat_vec.s mat_vec.gcc_O0
ifranl00@ubuntu:~/Desktop/lab5/matrix$ cat mat_vec.gcc_O0
	.file	"mat_vec.c"
	.text
	.globl	mat_vec
	.type	mat_vec, @function
mat_vec:
.LFB6:
	.cfi_startproc
	endbr64
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -24(%rbp)
	movq	%rsi, -32(%rbp)
	movq	%rdx, -40(%rbp)
	movl	%ecx, -44(%rbp)
	movl	$0, -8(%rbp)
	jmp	.L2
.L5:
	movl	-8(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-40(%rbp), %rax
	addq	%rdx, %rax
	pxor	%xmm0, %xmm0
	movsd	%xmm0, (%rax)
	movl	$0, -4(%rbp)
	jmp	.L3
.L4:
	movl	-8(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-40(%rbp), %rax
	addq	%rdx, %rax
	movsd	(%rax), %xmm1
	movl	-44(%rbp), %eax
	imull	-4(%rbp), %eax
	movl	%eax, %edx
	movl	-8(%rbp), %eax
	addl	%edx, %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-24(%rbp), %rax
	addq	%rdx, %rax
	movsd	(%rax), %xmm2
	movl	-4(%rbp), %eax
	cltq
	leaq	0(,%rax,8), %rdx
	movq	-32(%rbp), %rax
	addq	%rdx, %rax
	movsd	(%rax), %xmm0
	mulsd	%xmm2, %xmm0
	movl	-8(%rbp), %eax
```

```
ifranl00@ubuntu:~/Desktop/lab5/matrix$ gcc -S -O3 mat_vec.c
ifranl00@ubuntu:~/Desktop/lab5/matrix$ cp mat_vec.s mat_vec.gcc_O0
ifranl00@ubuntu:~/Desktop/lab5/matrix$ cat mat_vec.gcc_O0
        .file   "mat_vec.c"
        .text
        .p2align 4
        .globl  mat_vec
        .type   mat_vec, @function
mat_vec:
.LFB39:
        .cfi_startproc
        endbr64
        testl   %ecx, %ecx
        jle     .L1
        movq    %rdi, %r10
        leal    -1(%rcx), %eax
        movslq  %ecx, %rdi
        movq    %rdx, %r8
        salq    $3, %rdi
        leaq    8(%rsi,%rax,8), %r9
        pxor    %xmm2, %xmm2
        xorl    %r11d, %r11d
        .p2align 4,,10
        .p2align 3
.L4:
        movq    $0x000000000, (%r8)
        movq    %rsi, %rax
        movq    %r10, %rdx
        movapd  %xmm2, %xmm1
        .p2align 4,,10
        .p2align 3
.L3:
        movsd   (%rdx), %xmm0
        mulsd   (%rax), %xmm0
        addq    $8, %rax
        addq    %rdi, %rdx
        addsd   %xmm0, %xmm1
        movsd   %xmm1, (%r8)
        cmpq    %r9, %rax
        jne     .L3
        addl    $1, %r11d
        addq    $8, %r8
        addq    $8, %r10
        cmpl    %r11d, %ecx
        jne     .L4
.L1:
        ret
        .cfi_endproc
.LFE39:
        .size   mat_vec, .-mat_vec
        .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long    1f - 0f
        .long    4f - 1f
```

5. Analysis and the comparison of the obtained assembler codes – how many accesses to a variables are in each case?

Because depending on what memory hierarchy we are referring to by accesses to variables I am going to take as access to a variables for the first time, it means, that we are going to count access to memory that the stack do not already have.

When using 03: 4 variables are accessed to save its information in the involve registers so when this is done, the memory will not be accessed again because the registers have this information. Also, the values of the variables will be changed, so it is needed to access the memory to write the new information, so other 4 times: may be 8 accesses.

When using 00: The same situation occurs but it is going to take longer because 4 variables are saved to read but in this case, many times and written also, as we can see in the assemble file generated it is repeated 4 times so 4*8= may be 32 accesses.

6. Create a new procedure mat_vec_1 with a more optimal access, execute, measure the time and analyze the assembler for the fastest code obtained.

```
void mat_vec(double* a, double* x, double* y, int n)
{

  int i,j;

  for(i=0;i<n;++i){
    y[i]=0.0;
    for(j=0;j<n;++j){
      y[i]+=a[i+n*j]*x[j];
    }
  }
}
```

```
ifranl00@ubuntu:~/Desktop/lab5/matrix$ ./my_program
start
end
Time: 0.346923
TEST
```

So we can see that is faster than 0.36825 that was obtained in the exercise 3 without optimizing the code and using, for example, option 03 in Makefile.

The assembler code is the same as obtained for the no optimized program executed in exercise 3 for the option 03 in makefile because the variables that now are pre-incremented are not assigned to any variable so the compiler just take as relevant that the increment is done.

7. Measure the performance of the worst and best code and compare it with the theoretical values.

- ✔ **Best performance:(using pre-increment counters)**

```
void mat_vec(double* a, double* x, double* y, int n)
{

  int i,j;

  for(i=0;i<n;++i){
    y[i]=0.0;
    for(j=0;j<n;++j){
      y[i]+=a[i+n*j]*x[j];
    }
  }
}
```

```
ifranl00@ubuntu:~/Desktop/lab5/matrix$ ./my_program
start
end
Time: 0.346923
TEST
```

✗ **Worst performance: (using post-increment counters)**

```
void mat_vec(double* a, double* x, double* y, int n)
{

  int i,j;

  for(i=0;i<n;i++){
    y[i]=0.0;
    for(j=0;j<n;j++){
      y[i]+=a[i+n*j]*x[j];
    }
  }
}
```

```
ifranl00@ubuntu:~/Desktop/lab5/matrix$ ./my_program
start
end
Time: 0.368025
TEST
```

Using ++i and ++j that will provide us the same functionality, but sometimes it can be faster because i++ might involve to the need of making a copy of the object.

From the source https://betterprogramming.pub/stop-using-i-in-your-loops-1f906520d548, we can explain in more detail, that pre-incrementing our counters there is no need to create a copy to save the older value of the counter that i++ may have to create an object as copy and this may take more time.

Comparing with the theoretical values of my computer:

As we can see from the official page to search about my processor specifications, the memory is of the type DDR4-2666, which means that it could make 2666.67 mega-transfers per second. Also, theoretically, the maximum speed to read or write by the processor Intel Corei7-8750H in memory is 41.800 MB/s.

So we can see that it even the worst performance can process a nested loop with several operations in a bit more than a quarter of second, so less transfers than the theoretical possible ones made in a second occur and the program executes very quickly.