Computer Organization and Architecture

Name: Pallav Biyala
Roll No.: 24110234

Course: Computer Organization and Architecture
Date: 17th January, 2025

Assignment – 1

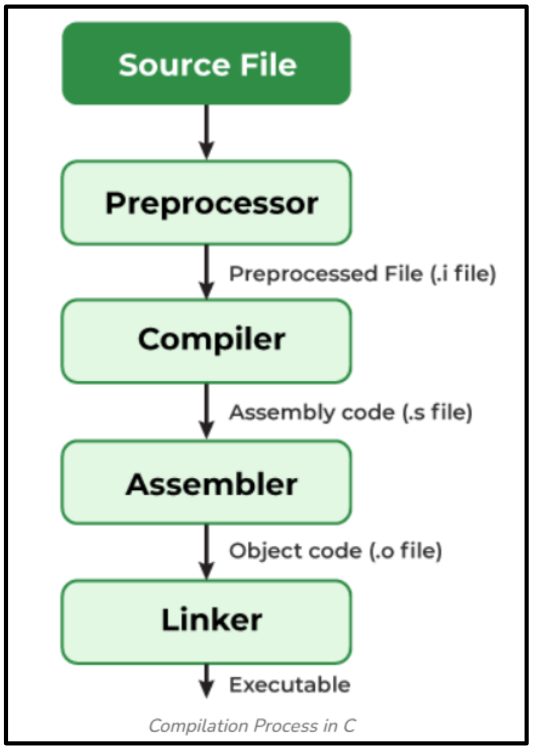
→ Brief Overview of Compilation in C and Python:

C Language:

The whole compilation is the process of converting the source code of the C language into an executable program. C is often referred to as a middle-level language as it provides both high-level abstractions and low-level hardware access. As C is a mid-level language, it needs a compiler to convert it into an executable code so that the program can be run on our machine.

When a C program is compiled, it undergoes multiple stages, each producing intermediate files before the execution file is created. These intermediate files are essential for understanding the compilation process.

So there are four main stages to compile a C Program into an executable file:



Reference: *Compiling a C Program: Behind the Scenes - GeeksforGeeks*

Let’s see all these four stages briefly:

1. **Pre – processing:**
   This is the first phase through which source code is passed. This phase includes:
   - Removal of comments
   - Expansion of Macros (Macros are instructions to preprocessor defined using #define)
   - Expansion of included files
   - Conditional Compilation

   This file helps in understanding:
   - The actual code seen by the compiler after preprocessing
   - The impact of header file inclusion on code size
   - How macros are expanded during compilation

The pre – processed output is stored in the name: code_file.i .

2. **Compilation:**
   The next step is to compile the pre – processed file and produce an intermediate compiled output file code_file.s. This file is now in **Assembly Level Instructions.**

   Hence, now High level language code is converted to Assembly Level language.

3. **Assembly:**
   The assembler takes the assembly code (code_file.s) and converts it into machine code, producing an object file with the .o extension. This file contains binary code but is not yet executable because it lacks linked libraries and dependencies.
   Hence, this file now contains machine level instructions. At this phase, only existing code is converted into machine language, and the function calls like printf () are not resolved.

4. **Linking:**
   This is the final phase in which all the linking of function calls with their definitions is done. Linker knows where all these functions are implemented. Linker does some extra work also: it adds some extra code to our program which is required when the program starts and ends.
   After successful linking, a complete executable file is generated.

Hence, the whole process of compilation in C can be thought as:

High Level Code → Assembly Level → Machine Level → Executable File

 Python:

Unlike C language, Python is an interpreted, high level language. They aren't directly compiled to Machine level executable files. Its compilation steps are different than that of C:

1. **Source Code:**
   The python program is firstly written in form of .py file.

2. **Compilation to Bytecode:**
   When a python program is executed, the Python Interpreter first compiles the source code into platform independent bytecode. This bytecode is stored in .pyc files inside the __pycache__ dictionary.

3. **Execution by Python Virtual Machine:**
   Byte code that is .pyc file is then sent to the Python Virtual Machine (PVM) which is the Python interpreter. PVM converts the Python byte code into machine-executable code and in this interpreter reads and executes the given file line by line. If an error occurs during this interpretation, then the conversion is halted with an error message.

   Within the PVM the bytecode is converted into machine code that is the binary language consisting of 0's and 1's. This binary language is only understandable by the CPU of the system as it is highly optimized for the machine code.

   In the last step, the final execution occurs where the CPU executes the machine code and the final desired output will come as according to your program.

Hence, in python the order of program to execution file is:

Source Code (.py) → Byte Code (.pyc) → PVM (Converts byte code to machine code)

So now we can understand what we are doing in the assignment.

Q1 Write a C program to multiply square matrices of size N x N where N is at-least 1024. Use either int or float data types. Compute the time needed to execute multiplication. You may use and refer timespec or timeval functions. As a part of compilation generate the intermediate. Observe and highlight the changes in .i and .s files. Execute the binary program with 'time'' command.

Sol. 1:

→ Code:

The code written in C programming language to perform the matrix multiplication is:

```
#include <stdio.h>
#include <stdlib.h> // to use malloc
#include <time.h> // to find time the whole program took to run

// Since matrix size is huge n*n, we must use dynamic memory allocation
int** create_matrix(int n){
    int **M = (int **)malloc(n*sizeof(int*));
    for (int i = 0; i<n; ++i){
        M[i] = (int*) malloc(n*sizeof(int));
    }
    return M;
}

// Freeing memory
void free_memory(int** M, int n){
    for (int i = 0; i<n; ++i){
        free(M[i]);
    }
    free(M);
}

int main()
{
    // Creating two square matrices of N*N where N is atleast 1024
    int n = 1024;

    // Matrix 1: M where mij = 2*i+j (example)
    int** M = create_matrix(n);
    // Adding values to the matrices
    for (int i = 0; i<n; ++i){
        for (int j = 0; j<n;++j){
            M[i][j] = 2*i + j; // each entry is 2*i+j
        }
    }

    // Matrix 2: N where nij = 3*j - 2*i + 5 (just example)
    int** N = create_matrix(n);
    for (int i = 0; i<n; ++i){
        for (int j = 0; j<n;++j){
            N[i][j] = 3*j - 2*i + 5;
        }
    }

    // Now finding product
    // Creating matrix that would contain product
    int** P = create_matrix(n);

    // Measuring time it took to run the code:
    struct timespec start, end;
    clock_gettime(CLOCK_REALTIME, &start);

    // Finding product of two matrix:
    for (int i = 0; i<n; ++i){ // loops through row of first matrix
        for (int j = 0; j<n;++j){ // loops through column if second matrix
            P[i][j] = 0; // as we didnt initialize it so to avoid garbage values
            for (int k = 0; k<n;++k){ // sum product of matrix row and column
                P[i][j] += M[i][k]*N[k][j];
            }
        }
    }

    // Ending the time
    clock_gettime(CLOCK_REALTIME, &end);

    // Calculating time it took to run the code
    double time_elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;


    // Printing the time
    printf("Matrix Multiplication of two N*N matrix took %.6f seconds where N = %d\n",
time_elapsed, n);

    // Freeing memory
    free_memory(M,n);
    free_memory(N,n);
    free_memory(P,n);
    return 0;
}
```

→ Intermediate Files:

1. **Pre – Processing File:**
   So the command that we used to pre – process the Matrix_multiplication.c is:

   gcc -E Matrix_multiplication.c > Matrix_multiplication.i

   Here, -E means that it instructs terminal to stop after pre – processing.

   So, if we see in the file, we can see it has thousands of lines. This is because, pre processing actually converted our code into **pure C code** by expanding all the three libraries we called:
   - Stdio.h
   - Stdlib.h

- Time.h

Hence, it expanded all the included libraries and replaced the macros at the places where we used it (if any) and this resulted in a single expanded source file that compiler would see.

So, Matrix_multiplication.i stores the whole pure C code.

2. **Compilation File:**

So the command that we used to compile the Matrix_Multiplication.c is:

```
gcc -S Matrix_multiplication.c
```

Here, -S means that compile the whole code but stop at assembly level.

So if we see inside the file, we can notice that it has been converted into functions like movl, leaq etc. This is because the code has been converted into Assembly level language and hence all our functions, loops are now translated to assembly level.
The assembly code appears complex and difficult to relate directly to the original C code because high-level constructs such as loops, array indexing, and function calls are broken down into low-level register operations and jumps by the compiler. Hence, it seems harder to read and understand as compared to source code we wrote and even pre processed file.

We can see an example of free_memory:
In C code we wrote, the function was:

```
// Freeing memory
void free_memory(int** M, int n){
    for (int i = 0; i<n; ++i){
        free(M[i]);
    }
    free(M);
}
```

But in .s file it became:

```
free_memory:
.LFB7:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movq    %rdi, -24(%rbp)
        movl    %esi, -28(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L6
.L7:
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -24(%rbp), %rax
```

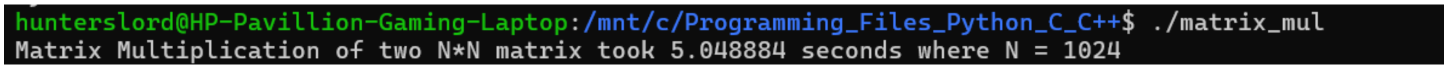
```
		addq	%rdx, %rax
		movq	(%rax), %rax
		movq	%rax, %rdi
		call	free@PLT
		addl	$1, -4(%rbp)
.L6:
		movl	-4(%rbp), %eax
		cmpl	-28(%rbp), %eax
		jl	.L7
		movq	-24(%rbp), %rax
		movq	%rax, %rdi
		call	free@PLT
		nop
		leave
		.cfi_def_cfa 7, 8
		ret
		.cfi_endproc
.LFE7:
		.size	free_memory, .-free_memory
		.section .rodata
		.align 8
.LC1:
		.string	"Matrix Multiplication of two N*N matrix took %.6f seconds where N = %d\n"
		.text
		.globl	main
		.type	main, @function
```

→ Running the Code:

So we can run the C code by using the following set of commands:

```
gcc Matrix_multiplication.c -o matrix_mul
./matrix_mul
```

The result that we got after running the command was:

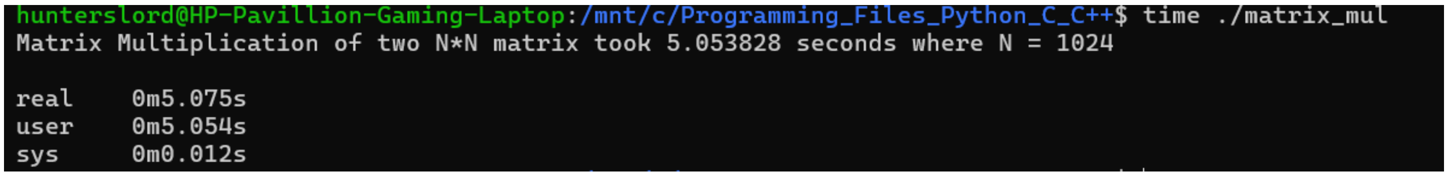


```
hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Programming_Files_Python_C_C++$ ./matrix_mul
Matrix Multiplication of two N*N matrix took 5.048884 seconds where N = 1024
```

We can also see the CPU Time, Elapsed time etc. using the following command:

```
time ./matrix_mul
```

The result we would obtain is:



```
hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Programming_Files_Python_C_C++$ time ./matrix_mul
Matrix Multiplication of two N*N matrix took 5.053828 seconds where N = 1024

real    0m5.075s
user    0m5.054s
sys     0m0.012s
```

Here, we can see time reported three values. These are:

1. Real time is the total elapsed time including CPU execution time, memory access, OS overhead etc.
2. User time is the User CPU time utilized for executing user's program, excluding I/O, system calls.
3. Sys time is the System CPU time spent inside the kernel (OS) on behalf of our program.

(We know CPU time = User CPU Time + System CPU Time)

Q2 Similarly write your own Python program using nested loops for matrix multiplication of same size N x N. Use time module to compute the execution time of the multiplication. Execute the python code with ‘time’’ command.

Sol. 2:

→ Code:

The code written in Python programming language to perform the matrix multiplication is:

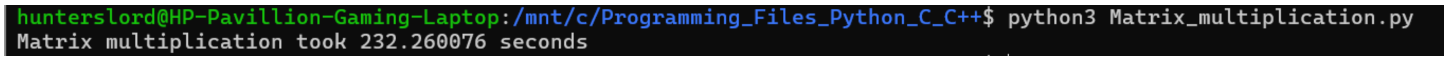
```python
import timeit

# Creating the matrices
n = 1024

# Creating first matrix
A = [[0 for _ in range(n)] for _ in range(n)]
for i in range(0,n):
    for j in range(0,n):
        A[i][j] = 2*i+j

# Creating second matrix
B = [[0 for _ in range(n)] for _ in range(n)]
for i in range(0,n):
    for j in range(0,n):
        B[i][j] = 3*j - 2*i + 5

# Finding product matrix
P = [[0 for _ in range(n)] for _ in range(n)]

# Starting counting time
start = timeit.default_timer()

for i in range(n):
    for j in range(n):
        for k in range(n):
            P[i][j] += A[i][k]*B[k][j]

# Ending time count
end = timeit.default_timer()

# Printing time
print(f"Matrix multiplication took {end - start:.6f} seconds")
```

→ Running the Code:

So, we can run the python file in Linux using the following command:

python3 Matrix_multiplication.py

The result that we got after running this command is:

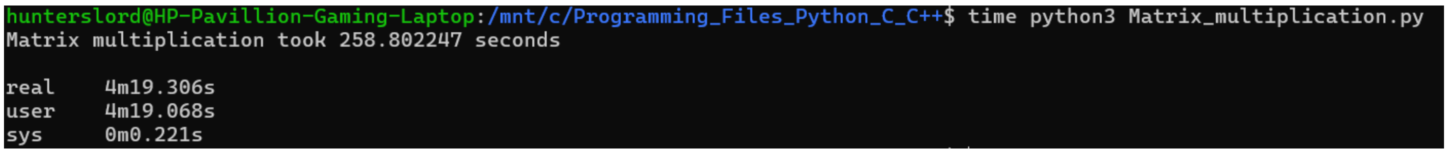


```
hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Programming_Files_Python_C_C++$ python3 Matrix_multiplication.py
Matrix multiplication took 232.260076 seconds
```

We can also see the CPU Time, Elapsed time etc. using the following command:

time python3 Matrix_multiplication.py

The result we would obtain is:



```
hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Programming_Files_Python_C_C++$ time python3 Matrix_multiplication.py
Matrix multiplication took 258.802247 seconds

real    4m19.306s
user    4m19.068s
sys     0m0.221s
```

So clearly we can see it took so much time. Let's compare it next Question:

Q3 Now, compare the execution time of the C & Python programs. Reason the outputs of time command, and the execution time observed by you in each of the case for multiplication.

Sol. 3: The execution time of C we found on running the code was 5.048884s while in case of Python, the execution time of same code was 232.260076s.

Clearly, Python took a very long time as compared to C on doing same type of problem for same value of N.

Now, we know the time command shows three values:

1. Real time is the total elapsed time including CPU execution time, memory access, OS overhead etc.
2. User time is the User CPU time utilized for executing user's program, excluding I/O, system calls.
3. Sys time is the System CPU time spent inside the kernel (OS) on behalf of our program.

So we can decipher few things from the data we saw:

1. The system (kernel) time in both C and Python was nearly negligible. This means the OS wasn't accessed a lot.
2. The total elapsed and CPU Execution time in both C and Python was almost same. This means that CPU was functioning almost the whole time of execution of program. Hence the other time like I/O, OS etc was negligible. Hence matrix multiplication we coded was CPU bound.
3. The Python code took approximately 46 times longer than what C took for same N. This means C was significantly a lot faster than Python.

 Why C was faster than Python:

The C language was faster than Python. This can be because of:

1. **Compilation (C) vs Interpretation (Python)**:
   C is a compiled language which means:
   - C code is translated directly into machine code by a compiler before execution.
   - The CPU executes this machine code directly, with no extra translation step at runtime.

   But Python is an interpreted language:
   - Python code is first compiled into bytecode, then executed by the Python Virtual Machine (PVM).
   - This adds an extra layer of interpretation for every operation, slowing execution.

2. **Memory Representation:**
   In C:
   - We control memory allocation and deallocation (malloc, free).
   - No garbage collector overhead unless we implement one.

   But Python:
   - Uses automatic garbage collection.

- This adds periodic pauses and extra CPU work.

3. **Function Call Overhead:**
   C's functions work on simple pointer arithmetic.
   But Python's function calls are more expensive due to dynamic typing and recursion depth checks.

□ **Summary:**

Hence, we can conclude that C is a lot faster than python because:

1. It compiles directly into Machine code which can be directly understood by CPU
2. Python has interpreted execution which takes time.
3. Function calls in python are more expensive while in C, they can be quick due to pointers.

Hence, C is far more suitable for compute-intensive tasks such as matrix multiplication when implemented using explicit loops.

Q4 You may now use the NumPy's built-in matrix multiplication function and measure the execution time. Compare the results and reason your findings.

Sol. 4:

→ **Code:**

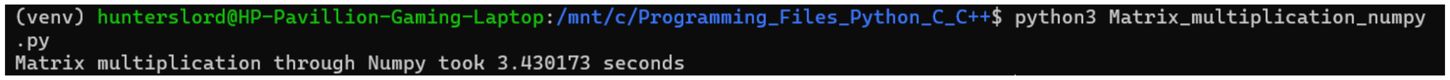
```python
import timeit
import numpy as np

# Creating the matrices
n = 1024

# Creating first matrix
A = np.fromfunction(lambda i, j: 2*i + j, (n, n), dtype=int)

# Creating Second Matrix
B = np.fromfunction(lambda i, j: 3*j - 2*i + 5, (n, n), dtype=int)

# Starting counting time
start = timeit.default_timer()

# Finding product matrix
P = np.dot(A, B)

# Ending time count
end = timeit.default_timer()

# Printing time
print(f"Matrix multiplication throuhg Numpy took {end - start:.6f} seconds")
```

→ **Running the Code:**

So, we can run the python file in Linux using the following command:

```
python3 Matrix_multiplication_numpy.py
```

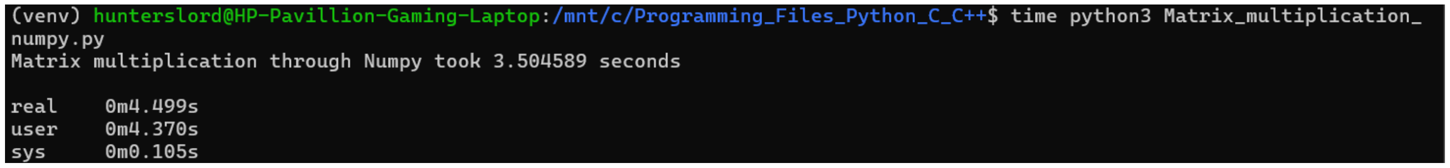
The result that we got after running this command is:


```
(venv) hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Programming_Files_Python_C_C++$ python3 Matrix_multiplication_numpy
.py
Matrix multiplication through Numpy took 3.430173 seconds
```

We can also see the CPU Time, Elapsed time etc. using the following command:

time python3 Matrix_multiplication_numpy.py

The result we obtained was:


```
(venv) hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Programming_Files_Python_C_C++$ time python3 Matrix_multiplication_
numpy.py
Matrix multiplication through Numpy took 3.504589 seconds

real    0m4.499s
user    0m4.370s
sys     0m0.105s
```

 Python vs C vs Numpy:

From the experimental results obtained for matrix multiplication of size N = 1024, the following execution times were observed:

- Python (nested loops): ~258 seconds
- C (compiled, nested loops): ~5 seconds
- NumPy (np.dot): ~3–4 seconds

Thus, NumPy performed the fastest among the three, even slightly faster than the C implementation. But how and why?

This is because:

1. **Backend:**
   Numpy's backend is not pure Python language. Its core numerical operations are implemented in highly optimized C and Fortran libraries.
   These libraries are specifically designed for linear algebra and matrix operations

2. **Vectorization:**
   Numpy do not use Python loops. Operations like np.dot() are vectorized, which allows CPU to process multiple data elements in parallel.

3. **Cache Friendly Memory Layout:**
   Numpy arrays are stored in contiguous memory blocks which improves cache efficiency.

4. **Highly Tuned Algorithms:**
   Numpy uses blocked matrix multiplication algorithms which reduces cache misses and improve throughput.

NOTE:

It seems that Numpy here worked faster than C but overall Numpy is not faster than C language. Here, in C we used the brute force algorithm to find the matrix multiplication while Numpy used different optimized algorithm to find the product. So, it is not wise to say that Numpy is faster than C as both codes had different algorithms.

The speed of C language can be appreciated by the fact that even through brute force, it took just 5s.

Q5 We will now use compiler optimization flags, say using gcc, we can recompile the C code with different optimization levels (-O2 and -O3) and measure the execution time. You might also want to look at the intermediate assembly code (.s file) and compare the differences.

Sol. 5: So for optimization levels –O2 and –O3, we can use the following commands:

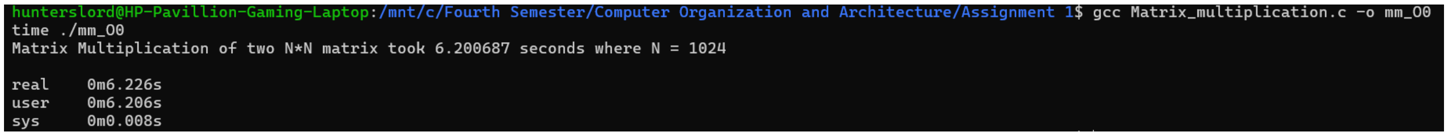
Baseline: gcc Matrix_multiplication.c -o mm_O0

Optimization level O2: gcc -O2 Matrix_multiplication.c -o mm_O2

Optimization level O3: gcc -O3 Matrix_multiplication.c -o mm_O3

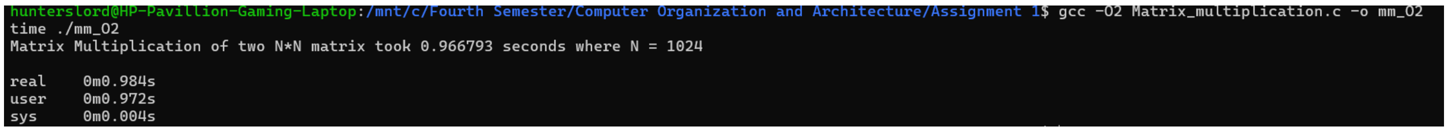
If we run the C code with these three operations the output we get is:

1. Baseline:



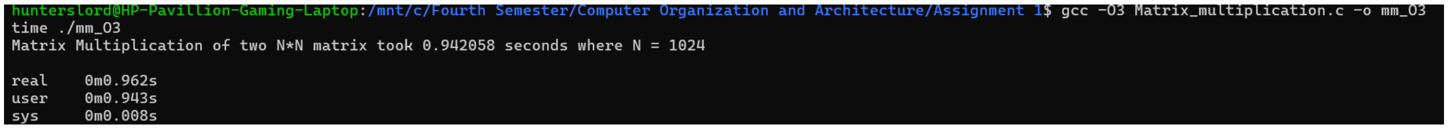
```
hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Fourth Semester/Computer Organization and Architecture/Assignment 1$ gcc Matrix_multiplication.c -o mm_O0
time ./mm_O0
Matrix Multiplication of two N*N matrix took 6.200687 seconds where N = 1024

real    0m6.226s
user    0m6.206s
sys     0m0.008s
```

2. Optimization level O2:



```
hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Fourth Semester/Computer Organization and Architecture/Assignment 1$ gcc -O2 Matrix_multiplication.c -o mm_O2
time ./mm_O2
Matrix Multiplication of two N*N matrix took 0.966793 seconds where N = 1024

real    0m0.984s
user    0m0.972s
sys     0m0.004s
```

3. Optimization level O3:



```
hunterslord@HP-Pavillion-Gaming-Laptop:/mnt/c/Fourth Semester/Computer Organization and Architecture/Assignment 1$ gcc -O3 Matrix_multiplication.c -o mm_O3
time ./mm_O3
Matrix Multiplication of two N*N matrix took 0.942058 seconds where N = 1024

real    0m0.962s
user    0m0.943s
sys     0m0.008s
```

 Comparison:

So we can see that when the same C code was compiled using different optimization levels, the running time of each level changed. For example:

1. Baseline: It took 6.200687s
2. Optimization level O2: It took 0.966793s
3. Optimization level O3: It took 0.942058s

So clearly:

- The unoptimized version (Baseline) had highest execution time.
- The Optimization level O2 showed a significantly reduced execution time
- While the third O3 level was slightly faster or we can say comparable to O2 level

 Why?

This is because compiler instruction flags instruct the compiler to improve run time performance by applying transformations like:

1. **Loop Optimization:** It reduces loop overhead and improves instruction scheduling
2. **Register Allocation:** Frequently used variables are stored in registers rather than memory
3. **Dead Code elimination:** Removes unnecessary instructions.
4. **Loop Unenrolling:** Reduces loop control overhead and improves instruction-level parallelism.

 Assembly Code Comparison:

The assembly file (.s) for each optimization can be generated using the following commands:

gcc -S Matrix_multiplication.c -o mm_O0.s
gcc -O2 -S Matrix_multiplication.c -o mm_O2.s
gcc -O3 -S Matrix_multiplication.c -o mm_O3.s

On comparing the .s files, it was observed that:

- The optimized versions contained fewer instructions.
- Loop control and memory access instructions were reduced.

- The generated assembly was more compact and efficient at higher optimization levels.

Hence, compiler optimizations significantly reduce execution time without any change in source code. Higher optimization levels generate more efficient machine code, making C programs much faster for compute-intensive tasks like matrix multiplication.

Bonus Question

Q1 Try to estimate the Instruction Count and CPI for each of the Q1, Q2, Q4 and Q5 approaches.

Sol 1: We know:

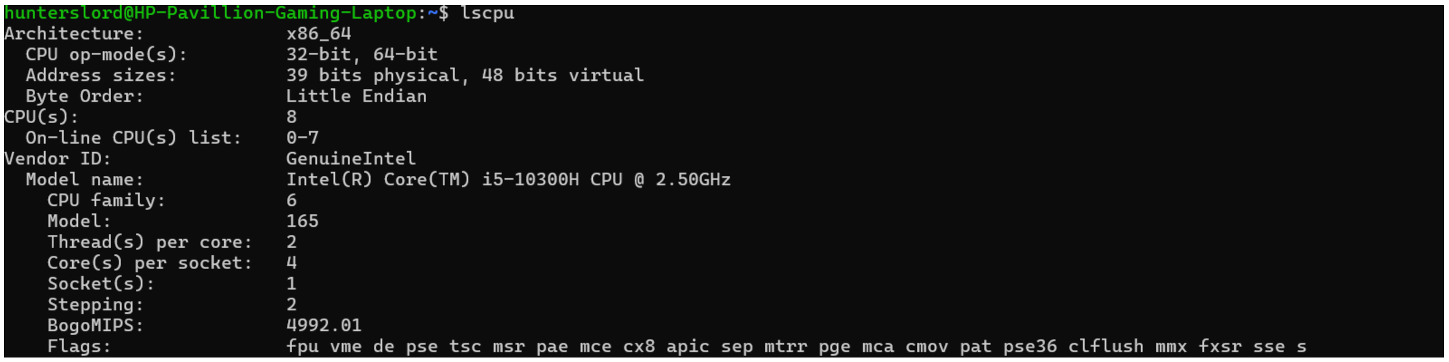
CPU Time = User time + System Time

Hence, we can easily find the execution time for each problem.

Now, we know:

$$\text{CPU TIME} = \frac{\text{IC } \times \text{CPI}}{\text{Clock Rate}}$$

To know clock rate, we can use the following command: lscpu
Then we can get a big result as:



```
hunterslord@HP-Pavillion-Gaming-Laptop:~$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   8
  On-line CPU(s) list:    0-7
Vendor ID:                GenuineIntel
  Model name:             Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
    CPU family:           6
    Model:                165
    Thread(s) per core:   2
    Core(s) per socket:   4
    Socket(s):            1
    Stepping:             2
    BogoMIPS:             4992.01
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse s
```

Clearly, here:

Clock rate = 2.5 GHz

So we also got Clock rate. But still we need to find either IC or CPI.

How to find Instruction Count (IC) in C:

So, we can find the Instruction count for C programs by using the Linux Tools: callgrind. So the command we would use is:

```
valgrind --tool=callgrind ./mm_O0
valgrind --tool=callgrind ./mm_O2
valgrind --tool=callgrind ./mm_O3
```

This will generate a huge file of ICs count with name “callgrind.out.<pid>” where pid is a number.

So to convert it into human readable form, we can use following commands:

```
callgrind_annotate callgrind.out.<pid> > mm_O0_callgrind.txt

callgrind_annotate callgrind.out.<pid> > mm_O2_callgrind.txt

callgrind_annotate callgrind.out.<pid> > mm_O3_callgrind.txt
```

This will generate a text file which at the end shows Total Programs and that’s our Instruction Count!

□ **How to find Instruction Count (IC) in Python:**

Similarly, like C, we can find the IC using same command in linux as we used for C:

valgrind --tool=callgrind python3 Matrix_multiplication.py

This will generate a huge file of ICs count with name “callgrind.out.<pid>” where pid is a number.

So to convert it into human readable form, we can use following commands:

callgrind_annotate callgrind.out.<pid> > Matrix_mulitplication_py_callgrind.txt

This will generate a text file which at the end shows Total Programs and that’s our Instruction Count!

→ C Code: (Q1)

We have:

$$\text{User time} = 5.054\text{s}$$

$$\text{Sys Time} = 0.012\text{s}$$

Hence,

$$\text{CPU Time} = 5054 + 0.012$$

$$= 5.066\text{s}$$

Hence,

$$\text{IC} \times \text{CPI} = \text{CPU Time} \times \text{Clock Rate} = \text{Number of cycles}$$

$$= 5.066 \times 2.5 \times 10^{9}$$

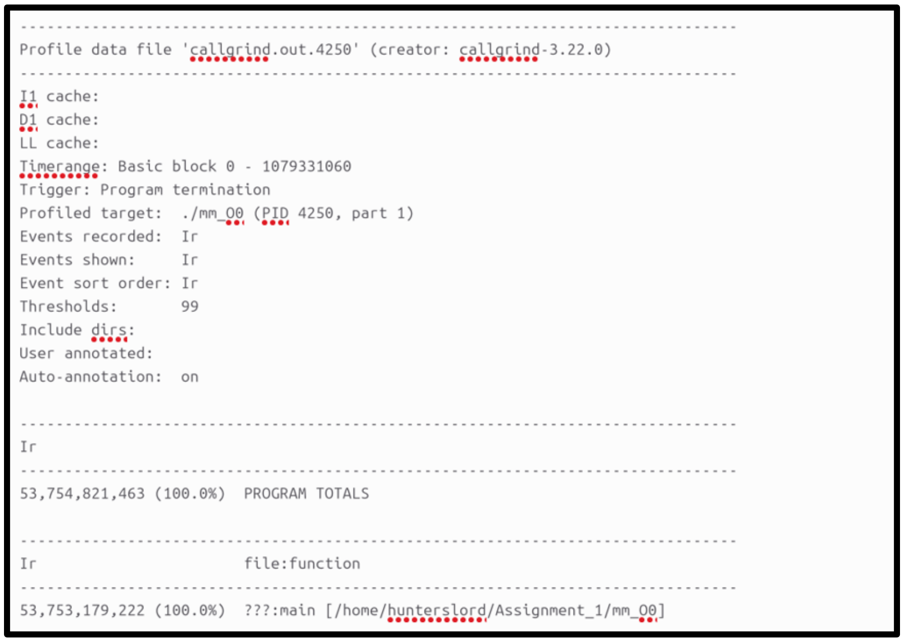
$$= 12.665 \times 10^{9}\ \text{cycles}$$

Hence, the total number of cycles is 1.2665×10^{10} cycles.

Now, we know:

$$\text{Total number of cycles} = \text{IC} \times \text{CPI}$$

From call grind tool, we got IC as:



```
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.4250' (creator: callgrind-3.22.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 1079331060
Trigger: Program termination
Profiled target:  ./mm_O0 (PID 4250, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  on

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
53,754,821,463 (100.0%)  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir                       file:function
--------------------------------------------------------------------------------
53,753,179,222 (100.0%)  ???:main [/home/hunterslord/Assignment_1/mm_O0]
```

Hence,

$$IC = 53{,}754{,}821{,}463$$

So,

$$\text{CPI} = \frac{1.2665 \times 10^{10}}{53{,}754{,}821{,}463}$$

Which gives:

$$\text{CPI} = 0.235 \text{ (approx.)}$$

Hence, for C program with no optimization (Baseline O0):

$$IC = 53{,}754{,}821{,}463$$

$$\text{CPI} = 0.235$$

→ Python Code (Q2):

We have:

$$\text{User time} = 4 \text{ min } 19.068\text{s} = 259.068\text{s}$$

$$\text{Sys Time} = 0.221\text{s}$$

Hence,

$$\text{CPU Time} = 259.068 + 0.221$$

$$= 259.289\text{s}$$

Hence,

$$IC \times CPI = \text{CPU Time} \times \text{Clock Rate} = \text{Number of cycles}$$

$$= 259.289 \times 2.5 \times 10^{9}$$

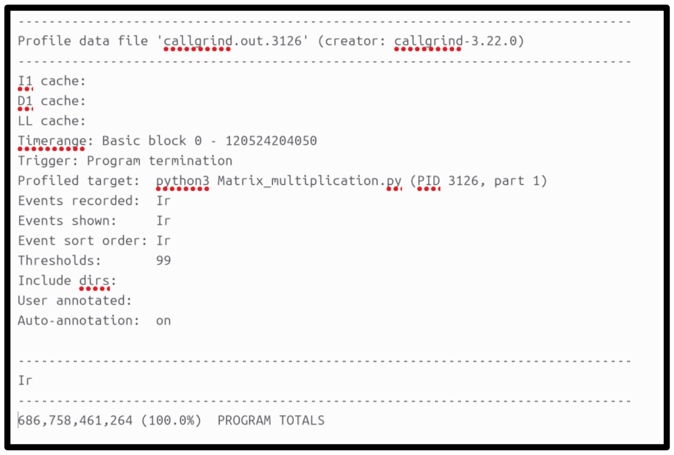
$$= 648.2 \times 10^{9} \text{ cycles}$$

Hence, the total number of cycles is 6.48×10^{11} cycles.

Now, we know:

$$\text{Total number of cycles} = IC \times CPI$$

From call grind tool, we got IC as:



```
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.3126' (creator: callgrind-3.22.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 120524204050
Trigger: Program termination
Profiled target:  python3 Matrix_multiplication.py (PID 3126, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  on

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
686,758,461,264 (100.0%)  PROGRAM TOTALS
```

Hence,

IC = 686,758,461,264

So,

$$\mathrm{CPI} = \frac{6.48 \times 10^{11}}{686{,}758{,}461{,}264}$$

Which gives:

CPI = 0.94 (approx.)

Hence, for C program with no optimization (Baseline O0):

IC = 686,758,461,264

CPI = 0.94

Note:

Since Python is an interpreted language with significant execution overhead, and the input size N was large (N=1024), the total CPU execution time for the brute-force matrix multiplication was approximately 259 seconds. The Call grind tool performs instruction-level simulation, counting each executed instruction, which introduces a substantial slowdown compared to native execution (typically around 10×–30×).

As a result, profiling the full program using Call grind becomes prohibitively time-consuming and can take several hours. Therefore, the Call grind execution was terminated after approximately 6 hours. Consequently, the reported instruction count (IC) and cycles per instruction (CPI) correspond to a partial execution of the program rather than the complete run.

→ Python (Numpy) Code (Q4):

We have:

User time = 4.370 s

Sys Time = 0.105s

Hence,

CPU Time = 4.370 + 0.105

= 4.475s

Hence,

IC × CPI = CPU Time × Clock Rate = Number of cycles

$= 4.475 \times 2.5 \times 10^9$

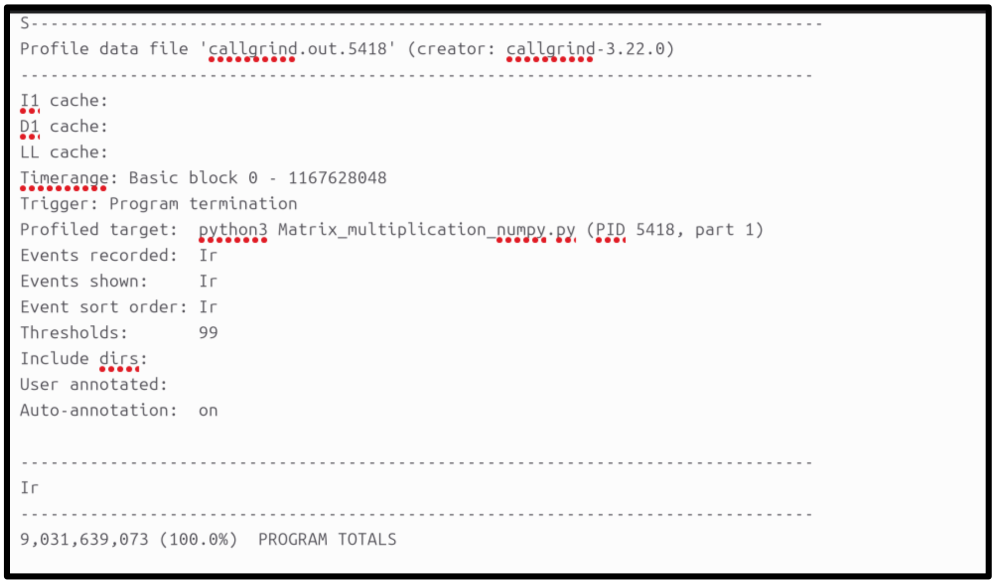
$= 11.18\times 10^9$ cycles

Hence, the total number of cycles is 1.11×10^{10} cycles.

Now, we know:

Total number of cycles = IC × CPI

From call grind tool, we got IC as:



```
S--------------------------------------------------------------------------------
Profile data file 'callgrind.out.5418' (creator: callgrind-3.22.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 1167628048
Trigger: Program termination
Profiled target:  python3 Matrix_multiplication_numpy.py (PID 5418, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  on

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
9,031,639,073 (100.0%)  PROGRAM TOTALS
```

Hence,

$$IC = 9{,}031{,}639{,}073$$

So,

$$CPI = \frac{1.11 \times 10^{10}}{9{,}031{,}639{,}073}$$

Which gives:

$$CPI = 1.23 \text{ (approx.)}$$

Hence, for python program in Numpy:

$$IC = 9{,}031{,}639{,}073$$

$$CPI = 1.23$$

→ Optimization Levels (Q5):

□ **O2:**

We have:

$$\text{User time} = 0.972 \text{ s}$$

$$\text{Sys Time} = 0.004\text{s}$$

Hence,

$$\text{CPU Time} = 0.972 + 0.004$$

$$= 0.976\text{s}$$

Hence,

$$IC \times CPI = \text{CPU Time} \times \text{Clock Rate} = \text{Number of cycles}$$

$$= 0.976 \times 2.5 \times 10^9$$

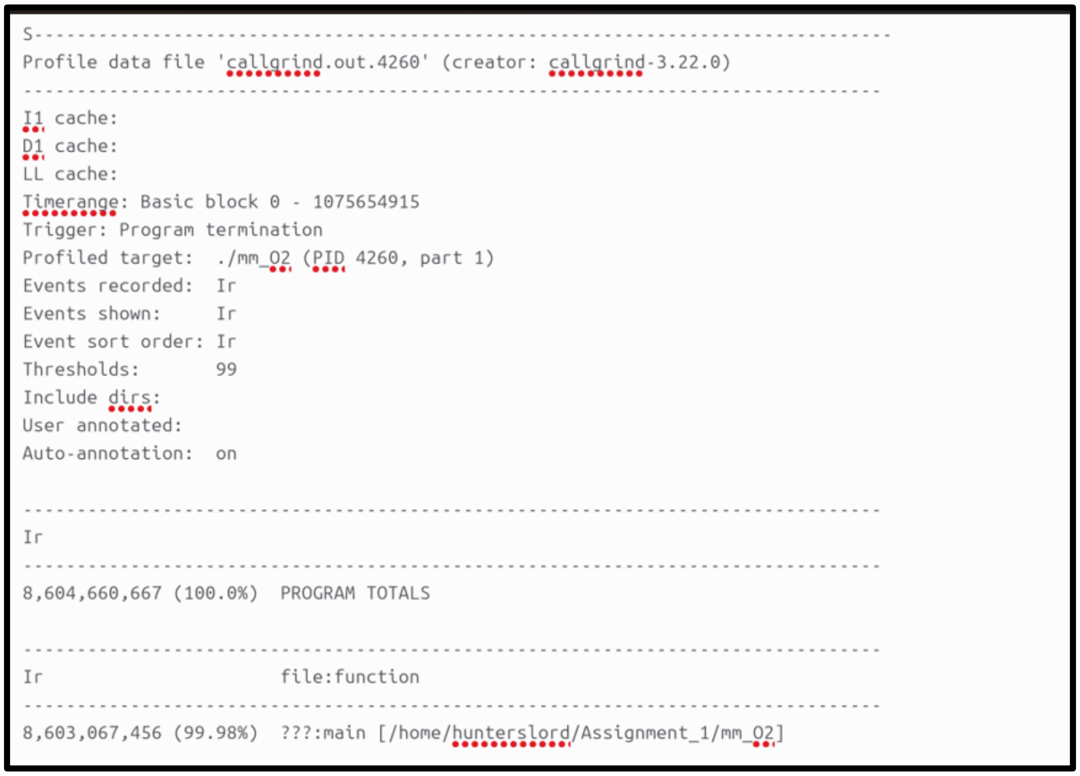
$$= 2.44 \times 10^9 \text{ cycles}$$

Hence, the total number of cycles is 2.44×10^9 cycles.

Now, we know:

$$\text{Total number of cycles} = IC \times CPI$$

From call grind tool, we got IC as:



```
S------------------------------------------------------------------------------
Profile data file 'callgrind.out.4260' (creator: callgrind-3.22.0)
-------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 1075654915
Trigger: Program termination
Profiled target:  ./mm_O2 (PID 4260, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  on

-------------------------------------------------------------------------------
Ir
-------------------------------------------------------------------------------
8,604,660,667 (100.0%)  PROGRAM TOTALS

-------------------------------------------------------------------------------
Ir                      file:function
-------------------------------------------------------------------------------
8,603,067,456 (99.98%)  ???:main [/home/hunterslord/Assignment_1/mm_O2]
```

Hence,

$$IC = 8{,}604{,}660{,}667$$

So,

$$CPI = \frac{2.44 \times 10^9}{8{,}604{,}660{,}667}$$

Which gives:

$$CPI = 0.284 \text{ (approx.)}$$

Hence, for C program with Optimization Level O2:

$$IC = 8{,}604{,}660{,}667$$

$$CPI = 0.284$$

 O3:

We have:

User time = 0.943 s

Sys Time = 0.008s

Hence,

CPU Time = 0.943 + 0.008

$$= 0.951s$$

Hence,

$$IC \times CPI = CPU\ Time \times Clock\ Rate = Number\ of\ cycles$$

$$= 0.951 \times 2.5 \times 10^9$$

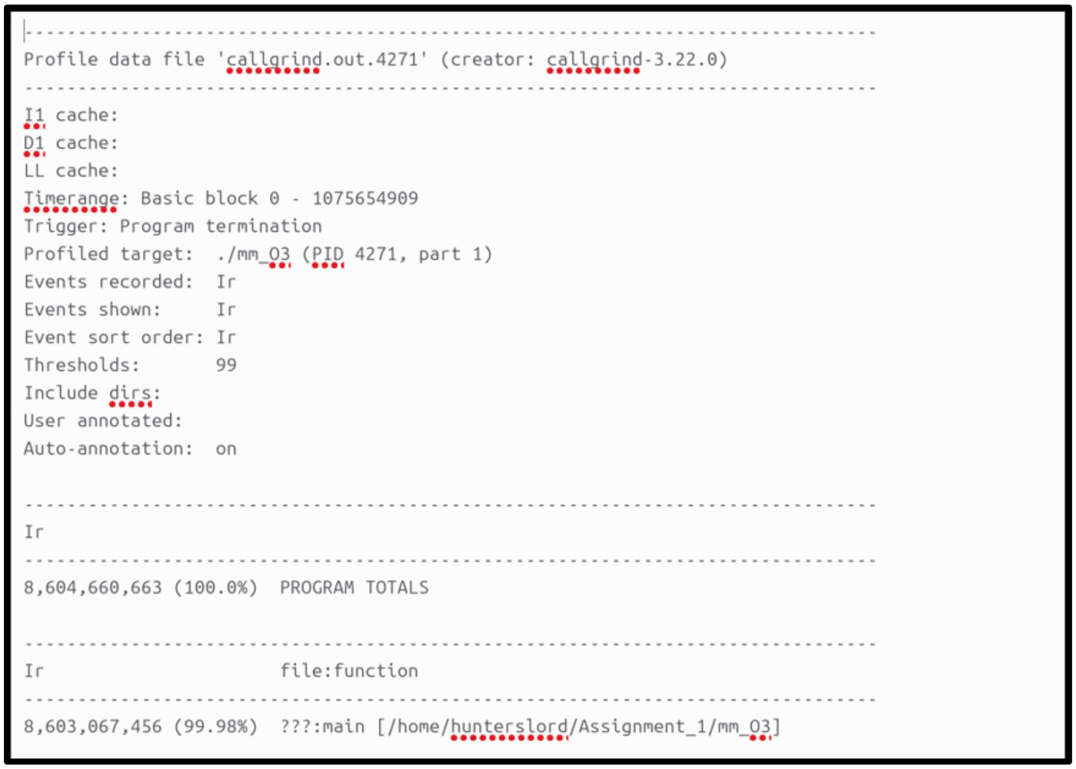
$$= 2.37 \times 10^9\ cycles$$

Hence, the total number of cycles is 2.37×10^9 cycles.

Now, we know:

$$Total\ number\ of\ cycles = IC \times CPI$$

From call grind tool, we got IC as:



```
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.4271' (creator: callgrind-3.22.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 1075654909
Trigger: Program termination
Profiled target:  ./mm_O3 (PID 4271, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  on

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
8,604,660,663 (100.0%)  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir                      file:function
--------------------------------------------------------------------------------
8,603,067,456 (99.98%)  ???:main [/home/hunterslord/Assignment_1/mm_O3]
```

Hence,

$$IC = 8{,}604{,}660{,}663$$

So,

$$CPI = \frac{2.37 \times 10^9}{8{,}604{,}660{,}663}$$

Which gives:

$$CPI = 0.275\ (approx.)$$

Hence, for C program with Optimization Level O3:

$$IC = 8{,}604{,}660{,}663$$

$$CPI = 0.275$$

→ Conclusion:

Hence, we can conclude that as the execution time of program reduced depending on optimization level, type of algorithm, or programming languages, the total number of cycles also reduced with it.

Also we can notice that how drastically the IC decreased for C program when optimized, i.e., from O0 to O2. Such a huge difference!

Q2 Learn to cross-compile your c-code for MIPS architecture.

Sol. 2: Cross Compilation refers to compiling a program one architecture (host) to generate an executable for a different architecture (target). In our case,

Host → My laptop (x86-64) machine
Target Architecture → MIPS

To cross compile the C code: Matrix_multiplication.c for MIPS, a MIPS cross compiler such as mips-linux-gnu-gcc can be used.
For eg:

```
mips-linux-gnu-gcc Matrix_multiplication.c -o mm_mips
```

This generates a MIPS compatible executable (mm_mips) file that can run on MIPS system.

We can also generate MIPS assembly code using the following command:

```
mips-linux-gnu-gcc -S Matrix_multiplication.c -o mm_mips.s
```

The generated assembly file contains MIPS instructions (e.g., `lw`, `sw`, `add`, `mul`) instead of x86-64 instructions, demonstrating successful cross-compilation.

Hence, by this way we can cross compile our code into MIPS Architecture and even see the assembly file.

֍֍֍֍֍