Computer Architecture

LAB Assignment01

Ritik Tiwari
B21CS098

Aim: Performance Analysis Using Perf

Objective:
In this assignment, you will use the perf tool to analyze the performance of a program/commands. You will capture various performance metrics, interpret the data, and understand how these metrics reflect the program's behavior on the hardware.

Assignment :

Notes:
You need to adjust the security and access control settings for performance monitoring on a Linux system. Clearly explain the variables involved, the modifications made to the kernel, and the reasons for these changes in the context of performance evaluation.

Create or find a Python, C, or C++ script that performs a CPU-intensive task. For example, the script could handle matrix multiplication, sort a large list, or compute the Fibonacci sequence recursively.

You need to compare the performance of the program/commands in two or more different linux machines.

Key metrics to analyze:
- Task Clock:

● Context Switches:
● CPU Migrations:
● Page Faults:
● Cycles:
● Stalled Cycles:
● Instructions:
● Branches:
● Branch Misses:

Our Task:
Use perf to capture performance metrics by executing your script and saving thec performance statistics to a text file. Make sure your script runs for more than 1 second. Then, run the same program on different Linux machines and compare the performance results using the metrics mentioned above.

Identify potential bottlenecks and areas for optimization, providing a clear explanation of how and where optimization is needed. Specify which computational aspect in function is consuming the most resources and specify the reasons behind it.
Analyze the resource usage(using above metrics) of ‘linux commands’ (ls, pwd, cd, etc.) you can choose any 2 commands for performance evaluation using perf.

Introduction:

In this assignment I have to analyses the performance of the program/ commands. I have done it in Linux Dual boot system where I switch from windows to Ubuntu and performs the comparison. I have create three different linux container or we say ubuntu container which is version 20.04 , 22.04 and 23.04 and compare the stats given by the perf tool. This is teh brief of what I have done I will explain this in more detail in the further discussion.

Methodology:

1. Environment Setup: So I have as described above I used the docker container to create the different linux containers so that I can try to compare the performance between different linux machines of the program execution factors.

2. C++ Program Description: So I have used two cpp codes to analyse the different stats between the linux machines. The first program is the matrix multiplication while the second one is of recursive code which involves floating point numbers which makes the complexity of the algorithm difficult. I submited the code also in the GC. To setu C++ environment in each machine or formally we say in container you have to install g++ to run c++ code.

Matrix Multiplication Code and Explanation:

```
#include <iostream>
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main() {
    const int N = 500;
    vector<vector<int>> A(N, vector<int>(N, 1));
    vector<vector<int>> B(N, vector<int>(N, 2));
    vector<vector<int>> C(N, vector<int>(N, 0));

    auto start = high_resolution_clock::now();

    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start);

    cout << "Time taken: " << duration.count() << " ms" << endl;

    return 0;
}
```

This C++ code performs matrix multiplication on two 500x500 matrices. It multiplies matrix A, where all elements are 1, by matrix B, where all elements are 2, and stores the result in matrix C. The code also measures and outputs the time it takes to complete this matrix multiplication operation.

<u>Recursion code and explanation:</u>

```
#include <iostream>
#include <vector>
#include <chrono>
#include <complex>

using namespace std;
using namespace std::chrono;

const int WIDTH = 800;
const int HEIGHT = 600;
const int MAX_ITER = 1000;

void mandelbrot(const int width, const int height, vector<vector<int>>& output) {
    for (int x = 0; x < width; ++x) {
                 for (int y = 0; y < height; ++y) {
            complex<double> c((x - width / 2.0) * 4.0 / width, (y - height / 2.0) * 4.0 /
height);
            complex<double> z(0, 0);
            int iter = 0;
            while (abs(z) < 2.0 && iter < MAX_ITER) {
                z = z * z + c;
                ++iter;
            }
            output[x][y] = iter;
```

```
        }
    }
}

int main() {
    vector<vector<int>> output(WIDTH, vector<int>(HEIGHT, 0));

    auto start = high_resolution_clock::now();

    mandelbrot(WIDTH, HEIGHT, output);

    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start);

    cout << "Time taken: " << duration.count() << " ms" << endl;

    return 0;
}
```

This C++ code generates the Mandelbrot set, a famous fractal pattern, for an 800x600 grid of pixels. It calculates how many iterations each point in the grid takes to determine whether it belongs to the Mandelbrot set. The result is stored in a 2D array output, where each element represents the number of iterations for a specific point. The code also measures and outputs the time it takes to compute the Mandelbrot set.

```
sudo apt-get install g++
```

To save the C++ script into your ubuntu you have few commands. This nano command will open the editor where you can save your code.

```
nano matrix_multiplication.cpp
```

To run the c++ code into your linux machine there is one command which is this:

```
g++ -o matrix_multiplication matrix_multiplication.cpp
```

3. Docker Containers: As described above also I have used the docker containers to create linux machines and here are the few commands for docker to pull images , run containers and also to stop the containers and how you can run the perf tool in each machines.

   These commands is to install the docker.

```
sudo apt-get update
sudo apt-get install docker.io
sudo systemctl start docker
sudo systemctl enable docker
```

   These commands to download the docker images.

```
sudo docker pull ubuntu:22.04
sudo docker pull ubuntu:20.04
```

This command is to attach the linux containers to your terminal

```
sudo docker attach <container_name_or_id>
```

To run the docker container or we say a specific linux machine the command is given below:

```
sudo docker run privilege -it ubuntu:20.04
/bin/bash
```

Now for each container or we say linux machine you have to make available the code so for that there is command which is this. This command create a copy of your code in that machine in the root directory.

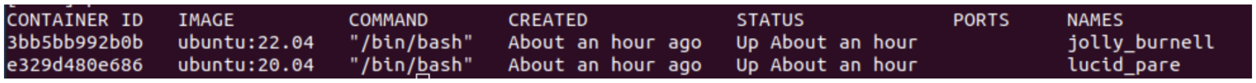
```
sudo docker cp matrix_multiplication.cpp <container_id>:/root/
```

Perf Tool :

Step 1: Installing `perf` Tool

To install the `perf` tool inside the Docker container, start by updating the package list to ensure all repositories are current. Then, install the necessary packages by running the command to install `linux-tools-common`, `linux-tools-generic`, and the specific `linux-tools` version corresponding to your kernel. This setup equips the Docker container with the `perf` tool, enabling you to perform performance analysis and capture detailed metrics for further evaluation.

```
apt-get update
apt-get install linux-tools-common linux-tools-generic
linux-tools-$(uname -r)
```



```
CONTAINER ID   IMAGE          COMMAND       CREATED              STATUS              PORTS     NAMES
3bb5bb992b0b   ubuntu:22.04   "/bin/bash"   About an hour ago    Up About an hour              jolly_burnell
e329d480e686   ubuntu:20.04   "/bin/bash"   About an hour ago    Up About an hour              lucid_pare
```

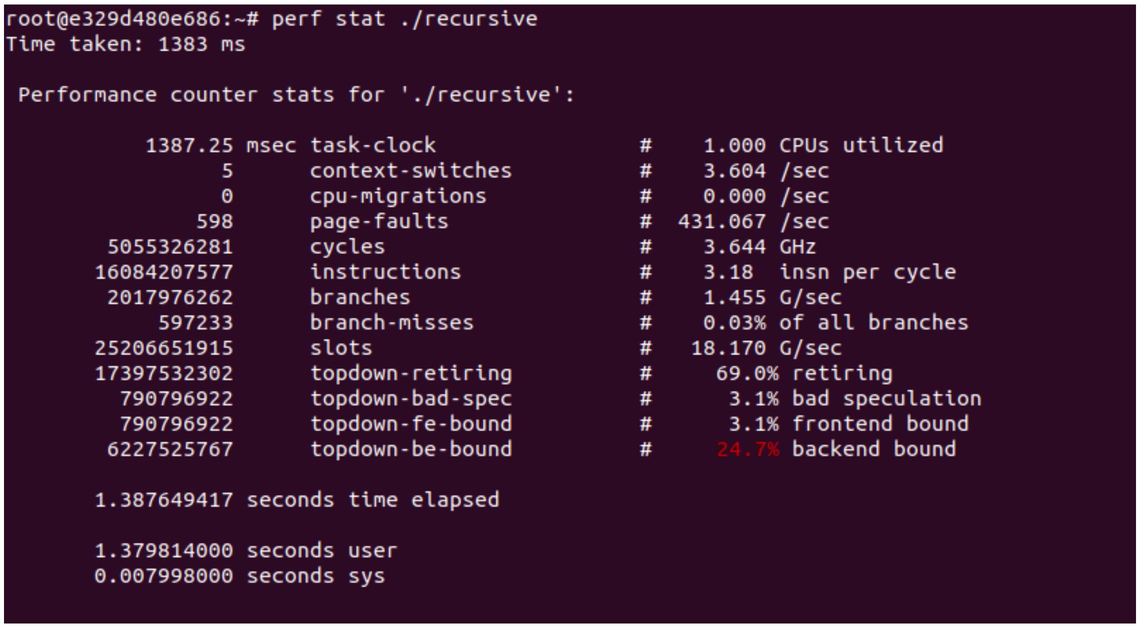
Step 2: Running Your Program with perf

To analyze your C++ program using perf, navigate to the directory where your compiled program is located (e.g., /tmp/user_workspace). Then, run your program with perf by executing the command perf stat ./matrix_multiplication. This command initiates the performance monitoring and captures key metrics such as CPU cycles, task clock, context switches, and more, providing valuable insights into how efficiently your program runs and where bottlenecks may exist.

```
perf stat ./matrix_multiplication
perf stat ./recursive
```

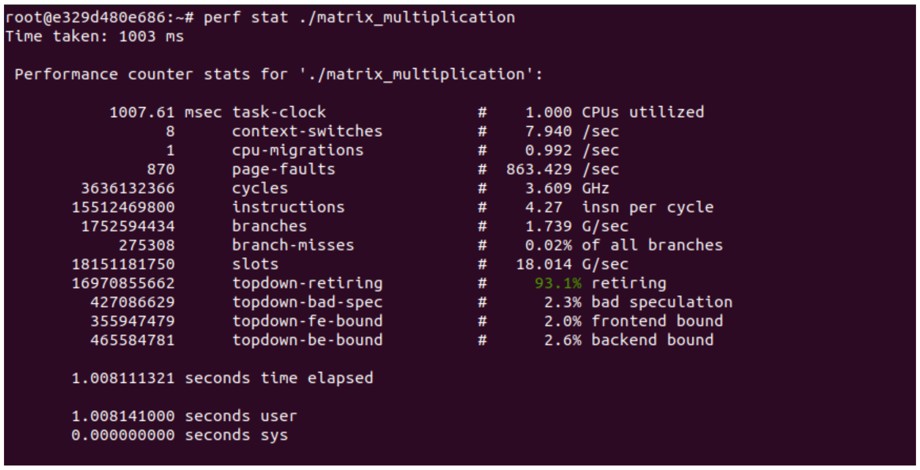
Here are the report of the two machines using the perf tool.

Ubuntu 20.04 Recursive



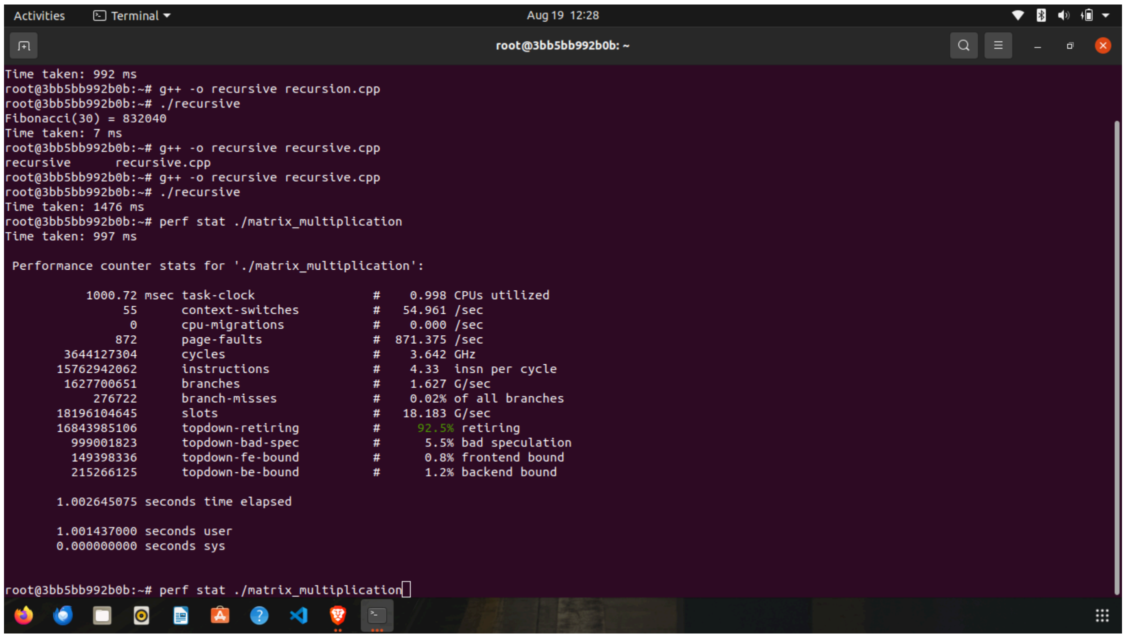
```
root@e329d480e686:~# perf stat ./recursive
Time taken: 1383 ms

 Performance counter stats for './recursive':

          1387.25 msec task-clock                #    1.000 CPUs utilized
                5      context-switches          #    3.604 /sec
                0      cpu-migrations            #    0.000 /sec
              598      page-faults               #  431.067 /sec
       5055326281      cycles                    #    3.644 GHz
      16084207577      instructions              #    3.18  insn per cycle
       2017976262      branches                  #    1.455 G/sec
           597233      branch-misses             #    0.03% of all branches
      25206651915      slots                     #   18.170 G/sec
      17397532302      topdown-retiring          #     69.0% retiring
        790796922      topdown-bad-spec          #      3.1% bad speculation
        790796922      topdown-fe-bound          #      3.1% frontend bound
       6227525767      topdown-be-bound          #     24.7% backend bound

      1.387649417 seconds time elapsed

      1.379814000 seconds user
      0.007998000 seconds sys
```

Ubuntu 20.04 Matrix Multiplication



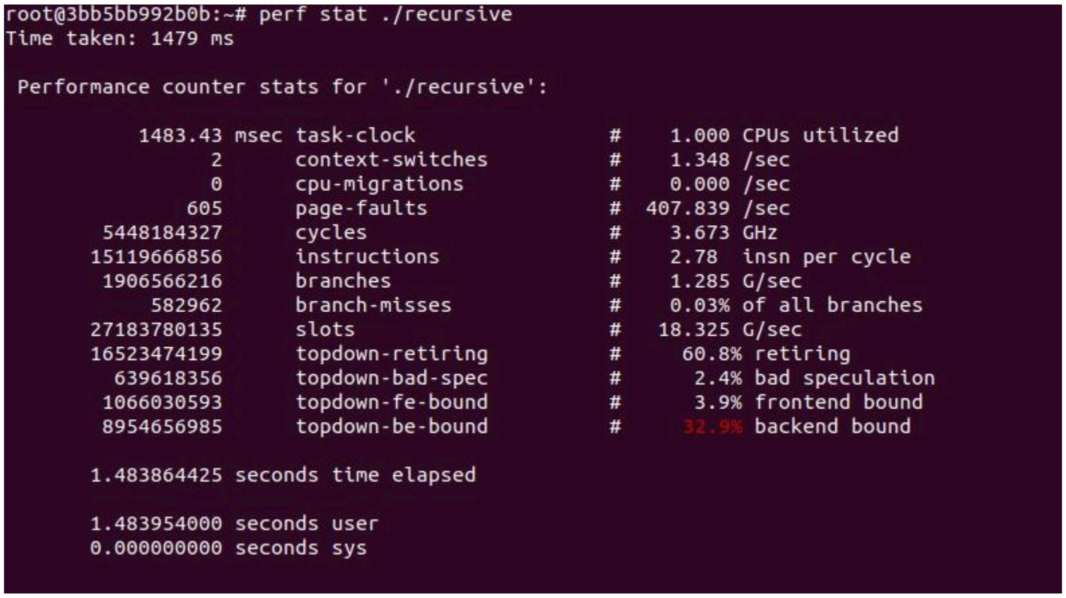
```
root@e329d480e686:~# perf stat ./matrix_multiplication
Time taken: 1003 ms

 Performance counter stats for './matrix_multiplication':

          1007.61 msec task-clock                #    1.000 CPUs utilized
                8      context-switches          #    7.940 /sec
                1      cpu-migrations            #    0.992 /sec
              870      page-faults               #  863.429 /sec
       3636132366      cycles                    #    3.609 GHz
      15512469800      instructions              #    4.27  insn per cycle
       1752594434      branches                  #    1.739 G/sec
           275308      branch-misses             #    0.02% of all branches
      18151181750      slots                     #   18.014 G/sec
      16970855662      topdown-retiring          #     93.1% retiring
        427086629      topdown-bad-spec          #      2.3% bad speculation
        355947479      topdown-fe-bound          #      2.0% frontend bound
        465584781      topdown-be-bound          #      2.6% backend bound

      1.008111321 seconds time elapsed

      1.008141000 seconds user
      0.000000000 seconds sys
```

Ubuntu 22.04 matrix multiplication



```
Time taken: 992 ms
root@3bb5bb992b0b:~# g++ -o recursive recursion.cpp
root@3bb5bb992b0b:~# ./recursive
Fibonacci(30) = 832040
Time taken: 7 ms
root@3bb5bb992b0b:~# g++ -o recursive recursive.cpp
recursive      recursive.cpp
root@3bb5bb992b0b:~# g++ -o recursive recursive.cpp
root@3bb5bb992b0b:~# ./recursive
Time taken: 1476 ms
root@3bb5bb992b0b:~# perf stat ./matrix_multiplication
Time taken: 997 ms

 Performance counter stats for './matrix_multiplication':

          1000.72 msec task-clock               #    0.998 CPUs utilized
               55      context-switches         #   54.961 /sec
                0      cpu-migrations           #    0.000 /sec
              872      page-faults              #  871.375 /sec
       3644127304      cycles                   #    3.642 GHz
      15762942062      instructions             #    4.33  insn per cycle
       1627700651      branches                 #    1.627 G/sec
           276722      branch-misses            #    0.02% of all branches
      18196104645      slots                    #   18.183 G/sec
      16843985106      topdown-retiring         #     92.5% retiring
        999001823      topdown-bad-spec         #      5.5% bad speculation
        149398336      topdown-fe-bound         #      0.8% frontend bound
        215266125      topdown-be-bound         #      1.2% backend bound

      1.002645075 seconds time elapsed

      1.001437000 seconds user
      0.000000000 seconds sys


root@3bb5bb992b0b:~# perf stat ./matrix_multiplication
```

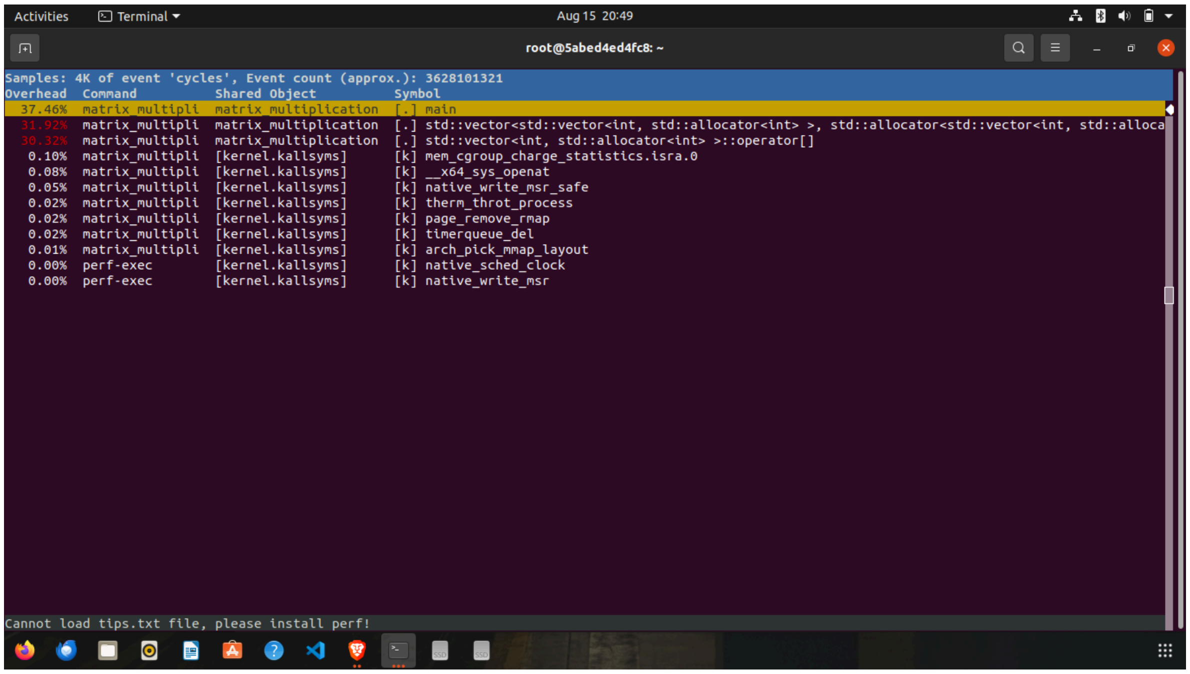
Ubuntu 22.04 Recursive stat



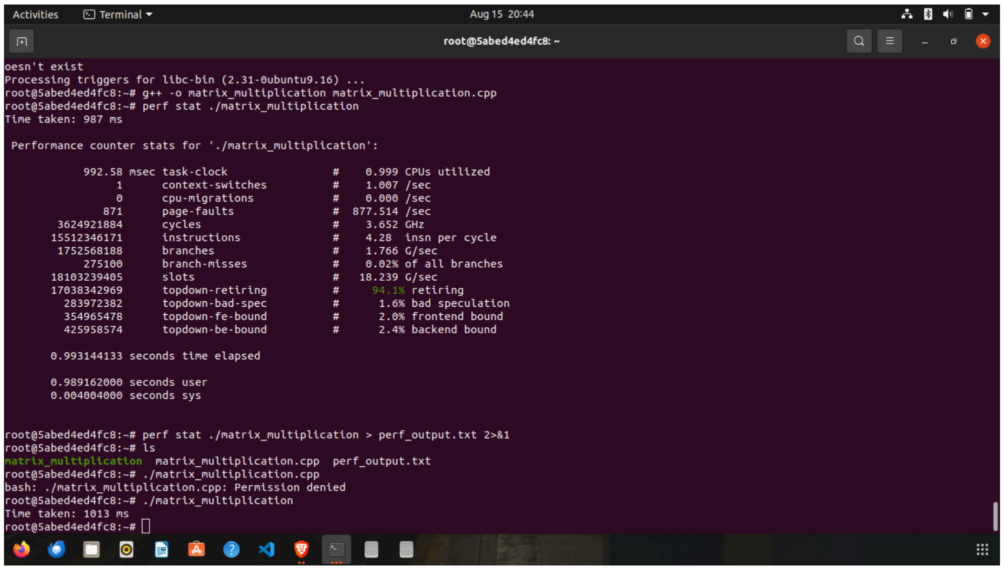
```
root@3bb5bb992b0b:~# perf stat ./recursive
Time taken: 1479 ms

 Performance counter stats for './recursive':

          1483.43 msec task-clock               #    1.000 CPUs utilized
                2      context-switches         #    1.348 /sec
                0      cpu-migrations           #    0.000 /sec
              605      page-faults              #  407.839 /sec
       5448184327      cycles                   #    3.673 GHz
      15119666856      instructions             #    2.78  insn per cycle
       1906566216      branches                 #    1.285 G/sec
           582962      branch-misses            #    0.03% of all branches
      27183780135      slots                    #   18.325 G/sec
      16523474199      topdown-retiring         #     60.8% retiring
        639618356      topdown-bad-spec         #      2.4% bad speculation
       1066030593      topdown-fe-bound         #      3.9% frontend bound
       8954656985      topdown-be-bound         #     32.9% backend bound

      1.483864425 seconds time elapsed

      1.483954000 seconds user
      0.000000000 seconds sys
```

So we can also analyze and get the results with the help of the some linux commands like (ls ,pwd). So to use these commands you can check what are the different resources are in use and in how much quantity like when any program is running what percentage of cpu it usage how many times there is I/O switch and with the effect of this how many times the page fault will occur. So these are the few things which makes the program sometimes faster or slower depending on the complexity of the program. Now when I analyze for both the recursive and matrix multiplication I found out that the recursive code used the memory stack a lot becuse in recursion it stores the each recursive result in a stack that lead to memory overflow while in the matrxi multiplication there is a problem of right address it many times possible that the memory address which is trying by the code does not found or it involve in some other resources so it will lead to index error. So these are my finding regarding the codes that I have used.

<u>Here is the screenshots of both the analysis:</u>



```
Samples: 4K of event 'cycles', Event count (approx.): 3628101321
Overhead  Command          Shared Object          Symbol
  37.46%  matrix_multipli  matrix_multiplication  [.] main
  31.92%  matrix_multipli  matrix_multiplication  [.] std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::alloca
  30.32%  matrix_multipli  matrix_multiplication  [.] std::vector<int, std::allocator<int> >::operator[]
   0.10%  matrix_multipli  [kernel.kallsyms]      [k] mem_cgroup_charge_statistics.isra.0
   0.08%  matrix_multipli  [kernel.kallsyms]      [k] __x64_sys_openat
   0.05%  matrix_multipli  [kernel.kallsyms]      [k] native_write_msr_safe
   0.02%  matrix_multipli  [kernel.kallsyms]      [k] therm_throt_process
   0.02%  matrix_multipli  [kernel.kallsyms]      [k] page_remove_rmap
   0.02%  matrix_multipli  [kernel.kallsyms]      [k] timerqueue_del
   0.01%  matrix_multipli  [kernel.kallsyms]      [k] arch_pick_mmap_layout
   0.00%  perf-exec        [kernel.kallsyms]      [k] native_sched_clock
   0.00%  perf-exec        [kernel.kallsyms]      [k] native_write_msr

Cannot load tips.txt file, please install perf!
```



```
oesn't exist
Processing triggers for libc-bin (2.31-0ubuntu9.16) ...
root@5abed4ed4fc8:~# g++ -o matrix_multiplication matrix_multiplication.cpp
root@5abed4ed4fc8:~# perf stat ./matrix_multiplication
Time taken: 987 ms

 Performance counter stats for './matrix_multiplication':

            992.58 msec task-clock                #    0.999 CPUs utilized
                 1      context-switches          #    1.007 /sec
                 0      cpu-migrations            #    0.000 /sec
               871      page-faults               #  877.514 /sec
        3624921884      cycles                    #    3.652 GHz
       15512346171      instructions              #    4.28  insn per cycle
        1752568188      branches                  #    1.766 G/sec
            275100      branch-misses             #    0.02% of all branches
       18103239405      slots                     #   18.239 G/sec
       17038342969      topdown-retiring          #     94.1% retiring
         283972382      topdown-bad-spec          #      1.6% bad speculation
         354965478      topdown-fe-bound          #      2.0% frontend bound
         425958574      topdown-be-bound          #      2.4% backend bound

       0.993144133 seconds time elapsed

       0.989162000 seconds user
       0.004004000 seconds sys


root@5abed4ed4fc8:~# perf stat ./matrix_multiplication > perf_output.txt 2>&1
root@5abed4ed4fc8:~# ls
matrix_multiplication  matrix_multiplication.cpp  perf_output.txt
root@5abed4ed4fc8:~# ./matrix_multiplication.cpp
bash: ./matrix_multiplication.cpp: Permission denied
root@5abed4ed4fc8:~# ./matrix_multiplication
Time taken: 1013 ms
root@5abed4ed4fc8:~# 
```

Identifying Potential Bottlenecks and Areas for Optimization

1. Matrix Multiplication Program

Bottlenecks:

1) Nested Loops: The matrix multiplication code involves three nested loops (each running N times where N = 500). The innermost loop is responsible for the bulk of the computation, specifically the multiplication of elements from matrices A and B and the accumulation of results into matrix C. This triple nested loop results in an overall time complexity of $O(N$^3), which is computationally expensive.
2) Memory Access Patterns: The program's memory access pattern may lead to cache misses. Since the elements of matrix A are accessed in a row-wise manner and matrix B in a column-wise manner, it is possible that the CPU's cache may not effectively store these values, leading to frequent cache misses and, subsequently, slower performance.

Areas for Optimization:

1) Loop Optimization: To optimize the loop structure, you can try loop unrolling or blocking (tiling). These techniques aim to improve cache utilization by working on smaller sub-matrices (blocks) that fit better into the cache, thereby reducing cache misses.
2) Matrix Transposition: Transposing matrix B before the multiplication can improve the cache performance since both matrices will then be accessed in a row-wise manner, aligning better with the CPU's cache architecture.
3) Parallelization: Given the independent nature of the operations within the loops, parallelizing the multiplication process using multi-threading (e.g., OpenMP) can significantly reduce the computation time.

Resource Consumption:

1) Computational Aspect: The most resource-intensive part is the innermost loop where the multiplication and accumulation happen. This part of the code is where most of the CPU cycles are spent due to the high number of floating-point operations and the large number of iterations.

<u>2. Mandelbrot Set Calculation (Recursive Code)</u>

Bottlenecks:

1) Complex Number Computations: The bottleneck in the Mandelbrot set calculation is the iterative computation of complex numbers in the mandelbrot function. For each pixel, the code iteratively squares a complex number and adds another complex number until a threshold is reached or the maximum number of iterations (MAX_ITER) is met. This process is computationally intensive due to the large number of iterations for each pixel.
2) Loop Iterations: The loops run over a large number of pixels (480,000 in total for an 800x600 image), and for each pixel, the code can run up to 1000 iterations (defined by MAX_ITER). The combination of these factors makes the program resource-intensive.

Areas for Optimization:

1) Escape Early: Currently, the code continues to calculate the Mandelbrot set for the maximum number of iterations unless the magnitude of z exceeds 2.0. If you can identify pixels that are likely to escape early based on the complex plane's properties, you can significantly reduce the number of iterations required for those pixels.
2) Optimization of Complex Number Operations: Optimizing the complex number operations could involve using approximations or optimized libraries that handle complex arithmetic more efficiently.
3) Parallelization: Similar to the matrix multiplication example, this problem can also benefit from parallelization. Since each pixel's calculation is independent of others, it can be divided among multiple

threads or even distributed across multiple cores or GPUs to achieve faster computation.

Resource Consumption:

1) Computational Aspect: The most resource-consuming aspect is the iterative calculation for each pixel. The high number of iterations per pixel and the extensive floating-point operations contribute to the significant CPU resource usage.

Conclusion

Both the matrix multiplication and Mandelbrot set (Recursive) programs have clear bottlenecks that can be optimized. In the matrix multiplication program, the primary issue lies in the inefficient memory access patterns and the high computational cost of the nested loops. Optimizing these through techniques like loop unrolling, matrix transposition, or parallelization can lead to substantial performance gains.

In the Mandelbrot program, the bottleneck is the intensive iterative computation of complex numbers for each pixel. Optimization strategies such as early escape detection, optimizing complex arithmetic, or leveraging parallel computing can significantly reduce execution time.

Results, Comparisons and Conclusion :

The observations indicate that Ubuntu 20.04 outperforms Ubuntu 22.04 in terms of computation speed for the same code. This suggests that Ubuntu 20.04 is more suitable for high-computation tasks. The recursive functions and matrix multiplication code, which are computationally intensive, take over one second to execute, highlighting their complexity. To efficiently run such code, it's crucial to ensure that the system's clock cycles are optimized for faster execution.

In this assignment, I successfully analyzed the performance of a C++ matrix multiplication program using Docker containers and the **perf** tool. Through this process, I identified key metrics such as CPU cycles, execution time, and cache misses, which provided valuable insights into the efficiency of our code. Despite facing challenges, such as compatibility issues with perf on certain Ubuntu versions, I overcame them by choosing the appropriate Docker environment and adapting our approach.

The analysis revealed areas where the program performed well, as well as potential bottlenecks that could be optimized further. This experience enhanced our understanding of system performance profiling and reinforced the importance of using tools like perf to fine-tune software for better efficiency. Overall, the assignment was a valuable learning experience, highlighting both the power and the limitations of performance analysis tools in software development.

Resources Used:

1. https://www.docker.com/get-started/
2. https://www.baeldung.com/ops/remove-docker-containers
3. https://www.brendangregg.com/perf.html