1. In this problem, you are to select a set of 3 single-threaded benchmark programs. Try to provide some diversity in the set of workloads you have chosen (e.g., floating point, integer, memory intensive, sparse). Then complete the following experiments using these benchmarks. Make sure to provide as many details as possible about the systems you are using, and where you obtained the source code for these benchmarks.
    a. Compile and run these 3 benchmarks on two different Linux-based system of your choosing (you can also use either the COE systems or the Explorer systems). Provide detailed information about the platforms you chose, including the model and frequency of the CPU, number of cores, the memory size, and the operating system version. You should record the execution time, averaged over 10 runs of the program. Is any run of the program faster than another? If so, comment on any difference you observe in your write-up, providing some justification for the differences.
    b. Comment on the differences observed in the timings on the two systems and try to explain what is responsible for these differences.
    c. Next, explore the compiler optimizations available with the compiler on one of your systems (e.g., gcc or g++), and report on the performance improvements found for the 3 workloads. Describe the optimization you applied and provide insight why each of your benchmarks benefitted from the specific compiler optimization applied.
    d. Summarizing benchmark performance with a single metric can be understood by a wider audience. Performance metrics such as FLOPS and MIPS have been used to report the performance of different systems. For one of your workloads, devise your own metric, possibly following how SPEC reports on performance. Generate a plot using this metric, while running the workload on 2 different CPUs platforms.
    e. Assume that you were going to rewrite these applications using pthreads. Describe how you would use pthreads to obtain additional speedup by running your benchmark on multiple cores.

Answer:

1 (a):
System: COE Linux System
Model CPU: Intel ® Xeon ® CPU E5-2698 v4 @ 2.20GHz
Cores: 2 sockets, 20 core/socket, 2 thread/socket
Operating System: Rocky Linux Release 8.9 (Green Obsidian)

Execution time in COE Linux system:

CPU Benchmark (Only floating point intensive output is considered):

```
CPU Benchmark timings (seconds), 10 runs:
  Run 1: 0.001973 sec
  Run 2: 0.001956 sec
  Run 3: 0.001958 sec
  Run 4: 0.001930 sec
  Run 5: 0.001941 sec
  Run 6: 0.001852 sec
  Run 7: 0.001868 sec
  Run 8: 0.001934 sec
  Run 9: 0.001881 sec
  Run 10: 0.001935 sec
Average CPU Benchmark FLOPS time: 0.001923 sec
```

GPU Benchmark (Floating point intensive):

```
GPU Benchmark Times (sec), 10 runs:
  Run 1: 0.000003 sec
  Run 2: 0.000000 sec
  Run 3: 0.000000 sec
  Run 4: 0.000000 sec
  Run 5: 0.000000 sec
  Run 6: 0.000000 sec
  Run 7: 0.000000 sec
  Run 8: 0.000000 sec
  Run 9: 0.000000 sec
  Run 10: 0.000000 sec
Avg GPU Benchmark FLOPS time: 0.000000 sec
```

DGEMM Benchmark (Floating point intensive):

```
FGEMM  Benchmark times (sec), 10 runs
Run 1: 0.011176 s
Run 2: 0.008573 s
Run 3: 0.008576 s
Run 4: 0.008567 s
Run 5: 0.008574 s
Run 6: 0.008576 s
Run 7: 0.008566 s
Run 8: 0.008577 s
Run 9: 0.008568 s
Run 10: 0.008577 s
Average time: 0.008833 s
```

==========

System: Discovery Explorer
Model CPU: Intel ® Xeon ® Gold 5318Y CPU @2.10 GHz
Cores: 2 socket, 24 core/socket, 2 thread/socket
Operating system: Rocky Linux 9.3 (Blue Onyx)

Execution time in Discovery Explorer:

CPU Benchmark (Floating point intensive):

```
FLOPS timings (seconds), 10 runs:
  Run 1: 0.003118 sec
  Run 2: 0.002313 sec
  Run 3: 0.002096 sec
  Run 4: 0.001709 sec
  Run 5: 0.002064 sec
  Run 6: 0.001812 sec
  Run 7: 0.002379 sec
  Run 8: 0.002739 sec
  Run 9: 0.002270 sec
  Run 10: 0.002756 sec
Average FLOPS time: 0.002326 sec
```

GPU Benchmark (Floating point intensive):

```
GPU Benchmark Times (sec), 10 runs:
  Run 1: 0.000001 sec
  Run 2: 0.000000 sec
  Run 3: 0.000000 sec
  Run 4: 0.000000 sec
  Run 5: 0.000000 sec
  Run 6: 0.000000 sec
  Run 7: 0.000001 sec
  Run 8: 0.000000 sec
  Run 9: 0.000000 sec
  Run 10: 0.000000 sec
Avg time: 0.000000 sec
```

DGEMM Benchmark (Floating point intensive):

```
FGEMM FLOPS Benchmark
Run 1: 0.004983 s
Run 2: 0.004064 s
Run 3: 0.004059 s
Run 4: 0.004060 s
Run 5: 0.004059 s
Run 6: 0.004059 s
Run 7: 0.004060 s
Run 8: 0.004059 s
Run 9: 0.004057 s
Run 10: 0.004057 s
Average time: 0.004152 s
```

Yes, there is a difference in the running time. The GPU running time in both the linux systems is 0.000 seconds which indicates that GPU completes the task too quickly and hence cannot measure the execution time accurately. The CPU Benchmark has slight variations in their running time with values ranging from 0.001868 to 0.003118 (in both COE Linux and Discovery cluster). The average time of execution is 0.001923 seconds in COE Linux and 0.002326 seconds in discovery cluster. The small changes in execution time is attributed to system interruptions and thermal throttling which slightly affects the performance. DGEMM Benchmark has slight variation in running time (although less than CPU Benchmark) in both COE Linux and Discovery cluster, since there is relatively more consistency in running time in comparison with other 2 benchmarks, it runs efficiently and well-optimized way in the setup.

1(b): (i) While examining about the difference between the running times of CPU Benchmark it is observed that the execution time for the benchmark is less in COE Linux system as compared to Discovery Linux which suggests that COE Linux system is generally faster. The variations in run time is marginally more in Discovery Linux as compared to COE Linux which shows that Discovery Cluster is less efficient for CPU benchmark running.

But while examining the difference between the running times of FGEMM Benchmark in both the Linux systems, the exact opposite phenomenon is observed i.e FGEMM Benchmark runs efficiently in Discovery cluster than COE Linux.

(ii) This phenomenon is observed because COE Linux System is configured for single-core performance which has better optimization for single threaded tasks making them well suited for CPU Benchmark execution but whereas FGEMM Benchmark runs efficiently in the system which has ability to perform complex mathematical computations especially the one with better memory band-width, parallel processing capability. Discovery cluster is mostly configured for parallel processing and high throughput computations as they have high memory bandwidth, parallel working of codes and nodes, advanced vector extensions (like AVX and AVX512).

1 (c): The compiler optimizations present in the COE linux systems are: -O1, -O2, -O3, -Ofast, -Os, -Og. Here, I will be using 3 different workloads: FLOPS (Floating-Point Operations Per Second), IOPS (Integer Operations Per Second) and SPARSE.

From the CPU Benchmark, I will be adding even IOPS and get the final execution time using the compiler optimization. STREAM code will be taken from here (https://github.com/intel/memory-bandwidth-benchmarks/blob/master/stream.c).

FLOPS and IOPS when optimization is not applied:

```
[sridhar.pray@explorer-02 ~]$ gcc -o cpu_benchmark cpu_benchmark.c
[sridhar.pray@explorer-02 ~]$ ./cpu_benchmark
Incorrect number of parameters.
Usage: ./cpu_benchmark [operation count] [thread count]
[sridhar.pray@explorer-02 ~]$ ./cpu_benchmark 1000000 1

Starting CPU Benchmark...
  Operation Count: 1000000
  Threads Implemented: 1

FLOPS timings (seconds), 10 runs:
  Run 1: 0.002916 sec
  Run 2: 0.001600 sec
  Run 3: 0.002004 sec
  Run 4: 0.002953 sec
  Run 5: 0.002506 sec
  Run 6: 0.001565 sec
  Run 7: 0.002004 sec
  Run 8: 0.002992 sec
  Run 9: 0.001665 sec
  Run 10: 0.002765 sec
Average FLOPS time: 0.002297 sec
Approx. FLOPS throughput: 13.061 G-FLOPs
```

```
IOPS timings (seconds), 10 runs:
  Run 1: 0.002199 sec
  Run 2: 0.001585 sec
  Run 3: 0.002082 sec
  Run 4: 0.001567 sec
  Run 5: 0.001662 sec
  Run 6: 0.001815 sec
  Run 7: 0.001565 sec
  Run 8: 0.001580 sec
  Run 9: 0.001394 sec
  Run 10: 0.001660 sec
Average IOPS time: 0.001711 sec
Approx. IOPS throughput: 17.535 G-IOPs
```

FLOPS and IOPS when optimization is applied:

```
[sridhar.pray@explorer-02 ~]$ gcc -O3 -o cpu_benchmark cpu_benchmark.c
[sridhar.pray@explorer-02 ~]$ ./cpu_benchmark 1000000 1

Starting CPU Benchmark...
  Operation Count: 1000000
  Threads Implemented: 1

FLOPS timings (seconds), 10 runs:
  Run 1: 0.000238 sec
  Run 2: 0.000056 sec
  Run 3: 0.000028 sec
  Run 4: 0.000027 sec
  Run 5: 0.000027 sec
  Run 6: 0.000028 sec
  Run 7: 0.000032 sec
  Run 8: 0.000028 sec
  Run 9: 0.000029 sec
  Run 10: 0.000030 sec
Average FLOPS time: 0.000052 sec
Approx. FLOPS throughput: 573.515 G-FLOPs
```

```
IOPS timings (seconds), 10 runs:
  Run 1: 0.000046 sec
  Run 2: 0.000024 sec
  Run 3: 0.000023 sec
  Run 4: 0.000022 sec
  Run 5: 0.000023 sec
  Run 6: 0.000023 sec
  Run 7: 0.000023 sec
  Run 8: 0.000026 sec
  Run 9: 0.000027 sec
  Run 10: 0.000022 sec
Average IOPS time: 0.000026 sec
Approx. IOPS throughput: 1157.582 G-IOPs
```

STREAM output when optimization is not applied:

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:            5442.2     0.029440     0.029400     0.029457
Scale:           4735.2     0.033825     0.033789     0.033884
Add:             8147.2     0.029477     0.029458     0.029491
Triad:           8341.5     0.028791     0.028772     0.028837
```

STREAM output when optimization is applied:

gcc -O2 sparse.c -o sparse

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           26359.6     0.006087     0.006070     0.006125
Scale:          14379.4     0.011143     0.011127     0.011159
Add:            16117.2     0.014905     0.014891     0.014937
Triad:          16145.1     0.014888     0.014865     0.014919
```

The compiler optimization I have used is -O3 in FLOPS and IOPS and -O2 is used in STREAM workload:

- For STREAM workload, -O2 optimization is used because it has vectorization function so that multiple data points can be processed with single instruction which can increase the data rate and it has **'-march=native'** flag which helps in utilizing specific parts of processor more efficiently, hence improving memory handling operations.
- For FLOPS and IOPS, -O3 optimization is used for enhanced instruction scheduling and efficient floating-point calculations.

1 (d): Here I would use **Normalized Performance Efficiency (NPE)** as one of my choice of metric:

Formula for calculating **G-FLOPS**:

GFLOPS= (Total Operations/Average Execution Time) ÷ 10^9

Given information:

Operation count (loop count): 1000000
Floating point operations per second: 30

Total operations: Operation count x Floating point operations per second = 1000000 x 30 = 30000000 operations.

Normalized Performance Efficiency: NPE= (GFLOPS / (Average Execution Time X Normalization Factor)

Normalization = Max (Intel Core I7 GFLOPS, AMD Ryzen 5 3600) = 16.269 G-FLOPS

From the 1st iteration, average FLOPS time: 0.002193 seconds

Calculate the normalized performance efficiency (NPE):

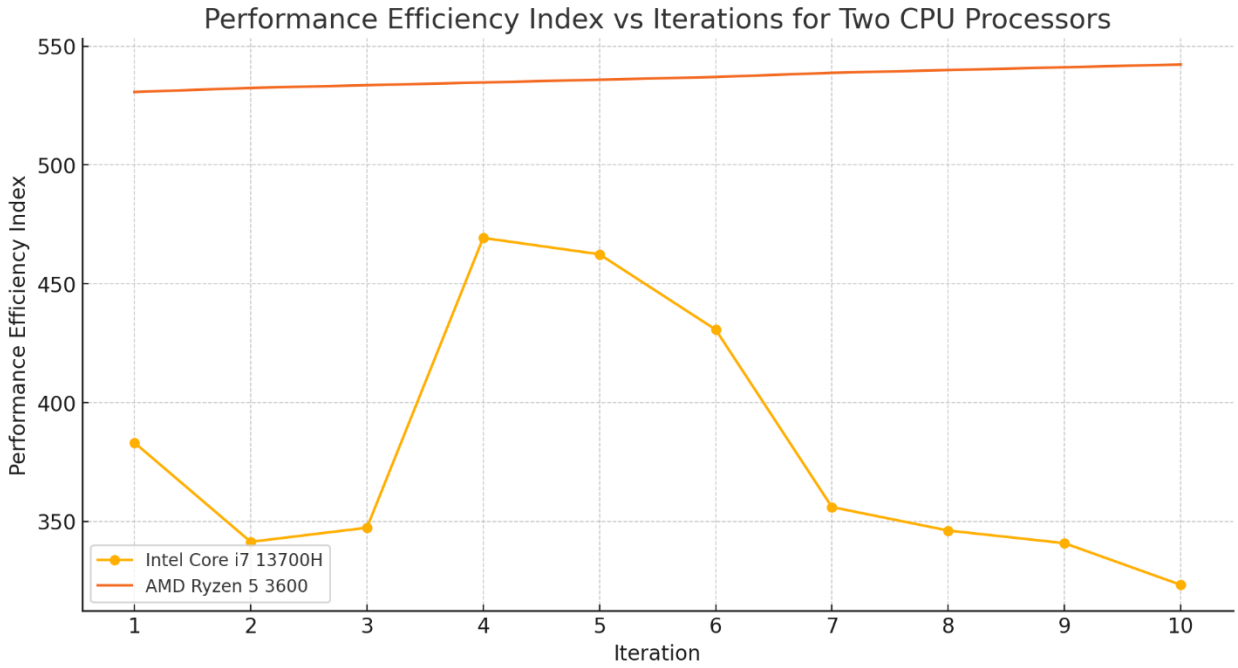
Similarly, other 9 iterations are done and Normalized performance Efficiency are calculated in Intel core I7 13700H:

Iterations	Average FLOPS time (in seconds)	G-FLOPS (in 10^9 seconds)	Normalized Performance Efficiency
1	0.002193	13.67	383.15
2	0.002324	12.908	341.40
3	0.002304	13.020	347.35
4	0.001982	15.133	469.31
5	0.001997	15.024	462.43
6	0.002069	14.498	430.71
7	0.002276	13.183	356.03
8	0.002308	12.998	346.16
9	0.002326	12.898	340.84
10	0.002388	12.561	323.32

We will calculate the same using AMD Ryen 5 3600:

Iterations	Average FLOPS time (in seconds)	G-FLOPS (in 10^9 seconds)	Normalized Performance Efficiency
1	0.001864	16.0944	530.72
2	0.001861	16.1204	532.44
3	0.001859	16.1377	533.58
4	0.001857	16.1551	534.73
5	0.001855	16.1725	535.89
6	0.001853	16.1900	537.04
7	0.001850	16.2162	538.79
8	0.001848	16.2338	539.95
9	0.001846	16.2514	541.13
10	0.001844	16.2690	542.30

This is the graph plotted from the obtained data:



1 (e): The benchmark is optimized using pthreads and achieved additional speedup by utilizing multiple cores by efficiently parallelizing the workload and minimizing synchronization overhead. The below are the steps that would be taken to optimize it using -pthreads:

- **Identifying the parallelable components**: Identifying the parts of benchmark that can be parallelized such as loops (performs independent calculations in different data elements) and tasks (which can be executed parallelly without depending on the results of one another).
- **Design the thread workload**: The workload should be divided evenly among the thread to prevent any single thread from being a bottleneck. Data dependencies need to be looked after so that threads do not write the same data simultaneously.
- **Create a management thread strategy**: The number of threads shouldn't exceed the number of cores as it could lead to excessive context switching and reduced efficiency. The dynamic approach can be used so that the number of threads can be adjusted based on the workload or system's current load.
- **Implementing thread synchronization**: When threads depend on each other's results, the synchronization mechanisms is implemented such as:
    - **Mutexes**: Use mutex locks to protect data structures that multiple threads might access simultaneously.
    - **Condition Variables**: Use these for signaling between threads when one thread's progress depends on another's.
    - **Barriers**: Use -pthread barriers to synchronize all threads at a certain point before any can proceed.
- **Initializing threads**: Utilize the pthread_create() method to instantiate threads. Each thread will execute a function that you specify, encompassing the segment of your program that has been parallelized. Transmit any requisite data to each thread through a struct or directly as parameters.

- **Managing Thread Execution**: Thread management may vary depending on the application.
    - **Static allocation of tasks**: Each thread handles a designated segment of the data. This approach is straightforward, although it may lack efficiency if the task is unevenly allocated.
    - **Dynamic scheduling**: Tasks are segmented into smaller pieces, and threads acquire new tasks upon the completion of their present ones. This may result in improved load distribution.
- **Finalizing of threads:** Upon the completion of thread execution, utilize pthread_join() to guarantee that the main program awaits the termination of all threads prior to proceeding. This is essential for accuracy, particularly when the primary application requires the results calculated by the threads.
- **Optimizing the application based on profiling**: Once the program has been implemented initially, profile it to find any inefficiencies or bottlenecks. Look for:
    - **Contention**: Excessive threads vying for identical resources.
    - **Idle time**: Threads awaiting without necessity.
    - **Disparate workload**: Certain threads conclude significantly faster than others.