Problem 1 (30 Points)

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 rite-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.

*Answers to this question should be included in your homework 1 write-up in pdf format.

Part (a)

Explorer Cluster, Node c0744: - CPU Model: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz - CPU Frequency: Min=1200.0000 MHz, Max=3300.0000 MHz, Current=3300.0000 MHz - CPU Cores: 28 - Memory Size: 251 GiB (263358376) - Cache Hierarchy: - L1d: 32K, 8-way set associative - L1i: 32K, 8-way set associative - L2: 256K, 8-way set associative - L3: 35M, 20-way set associative - Operating System: Rocky Linux 9.3 (Blue Onyx)

Vector: - CPU Model: Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz - CPU Frequency: Min=1200.0000 MHz, Max=3600.0000 MHz, Current=3600.0000 MHz - CPU Cores: 80 -

Memory Size: 755 GiB (792237220) - Cache Hierarchy: - L1d: 32K - L1i: 32K - L2: 256K - L3: 51M - Operating System: Rocky Linux 8.7 (Green Obsidian)

Floating Point Baseline Benchmark

Baseline Benchmark	MulBig Double	DivBig Double	MulSmallDouble	DivSmallDouble	AddBig Double	SubBig Double	AddSmallDouble	SubSmallDouble
Explorer Average Latency (ms)	6209	6220	7704	8672	6208	6156	8656	8639
Vector Average Latency (ms)	4969	5028	6880	6212	4945	5013	6125	6206

Integer Baseline Benchmark

Baseline Benchmark	MulBig Int	DivBigInt	MulSmallInt	DivSmallInt	AddBig Int	SubBig Int	AddSmallInt	SubSmallInt
Explorer Average Latency (ms)	5814	14251	8147	18040	5976	5794	7195	8278
Vector Average Latency (ms)	4688	11293	5857	13819	4709	4712	6579	6657

Memory Intensive Baseline Benchmark

Baseline Benchmark	Forward Sum	Reverse Sum	Random Sum
Explorer Average Latency (s)	24.088428	26.4901	195.464314
Vector Average Latency (s)	19.375647	21.372409	135.990249

Overall, there were no outlier runs of each benchmark (i.e. every benchmark run had consistently the same latency values with very little variance).

Part (b)

The Explorer Cluster node had worse performance compared to the Vector system for all benchmarks. Specifically, the Explorer system had a higher average latency for the floating-point, integer, and memory-intensive benchmarks compared to the Vector system. The most likely cause of this, since these were single-threaded programs, must be to do with the CPU model and frequency. The Explorer system had an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, while the Vector system had an Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz. The Vector system had a higher current CPU frequency (3600 MHz), which could explain the better performance of the benchmarks on the Vector system as opposed to the Explorer system (3300 MHz).

The Vector system also had more cores than the Explorer system, which could have also contributed to the better performance of the benchmarks on the Vector system. The memory size of the Vector system was also larger than the Explorer system, which may have also contributed to the better performance of the memory-intensive benchmark

(however, this is highly unlikely since the cache hierarchies between the two CPUs were nearly identical up until the L3 cache).

If we also compare the raw benchmarks of the two CPUs, we can see that overall, the Vector system CPU outperforms the Explorer system CPU (see https://www.cpu-world.com/Compare/424/Intel_Xeon_E5-2680_v4_vs_Intel_Xeon_E5-2698_v4.html). Notably, the Vector system CPU can run at a higher power (wattage) which is known to lead to direct performance improvements since it drives the clockspeed at higher rates.

Part (c)

For each benchmark, the following compiler optimizations were applied: - **O3:** Enable more aggressive optimizations, including those that may increase the size of the binary. - **funroll-loops:** Unroll loops whose trip counts can be determined at compile time. - **march=native:** Generate code optimized for the host machine's CPU.

Floating Point Optimized Benchmark

Optimized Benchmark	MulBig Double	DivBig Double	MulSmallDouble	DivSmallDouble	AddBig Double	SubBig Double	AddSmallDouble	SubSmallDouble
Explorer Average Latency (ms)	29	74	35	83	29	29	35	35
Vector Average Latency (ms)	25	59	26	73	22	22	25	25

Integer Optimized Benchmark

Optimized Benchmark	MulBig Int	DivBig Int	MulSmallInt	DivSmallInt	AddBigInt	SubBig Int	AddSmallInt	SubSmallInt
Explorer Average Latency (ms)	41	212	43	202	27	31	31	31
Vector Average Latency (ms)	34	169	33	171	22	23	25	25

Memory Intensive Optimized Benchmark

Optimized Benchmark	Forward Sum	Reverse Sum	Random Sum
Explorer Average Latency (s)	0.389829	0.389937	173.787937
Vector Average Latency (s)	0.316233	0.319048	106.332679

The reason why each benchmark benefitted from the specific compiler optimization applied is because the benchmarks were highly looped. Every operation in every benchmark was able to be fully unrolled (`-funroll-loops`) since the looping mechanism had no loop dependencies, except for the Random Sum operation in the Memory-Intensive benchmark. The Random Sum operation had a loop dependency because the loop was

dependent on the random number generator. The compiler optimizations that were able to fully unroll the loops resulted in a significant performance improvement for each benchmark. The `-O3` and `-march=native` compiler flags were general optimizations that also gave an overall speedup to the benchmarks.

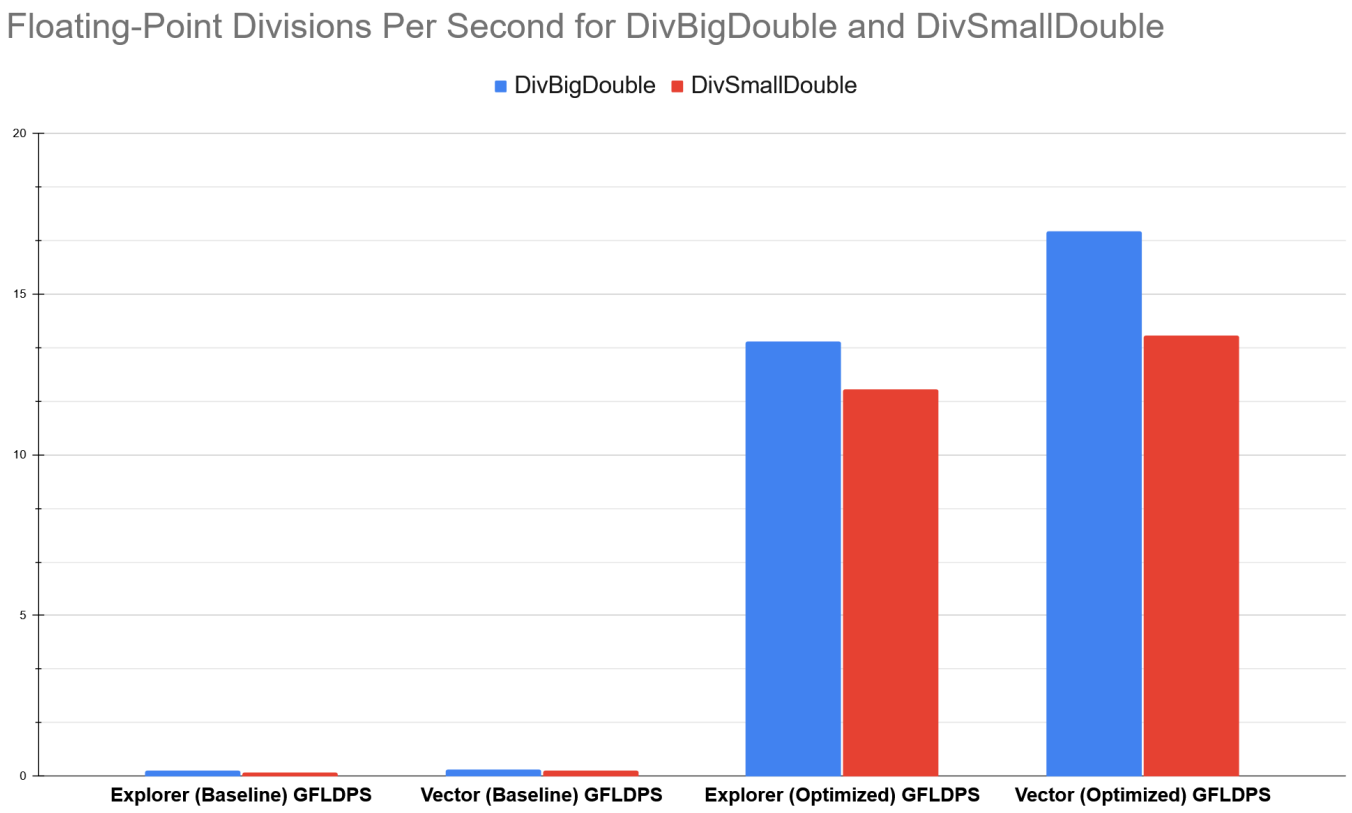
Part (d)

Metric Devised: Floating Point Divisions Per Second (FLDPS)

Taking the average latency of the baseline and optimized floating point benchmarks, the FLDPS metric was calculated as follows:

```
num_divisions = 1000000000
fldps = num_divisions / average_latency
```

The FLDPS metric was plotted for the baseline and optimized benchmarks on the Explorer and Vector systems in Giga-FLDPS (GFLDPS).



Part (e)

Assuming I was going to rewrite these benchmarks with pthreads to obtain additional speedup by running the benchmarks on multiple cores, I would split up the major parts of each benchmark and allocate the workload to individual threads. For example, the floating point and integer benchmarks loop a billion times for each operation type (addition, subtraction, division, etc…). A simple thread-workload allocation could be allocating a thread to the additions, another thread to the subtractions, and so on. This way, the workload is split up evenly among the threads, and the threads can run in parallel to complete the benchmark faster. Even further than that, more threads could be spawned to handle a certain amount of a loop since the loops carry no data dependencies. This means that the threads can run in parallel without any synchronization overhead. Similarly with the memory benchmark, the looping mechanism works in the same way, and thus the threading would be highly similar to the floating point and integer benchmarks.

Miscellaneous

- The program was compiled and run using the following command within each benchmark subdirectory: `make`