**Assignment 1: Optimizing Matrix-Vector Multiplication**

**Roll num: 2023121006**

**System information**

Using stream benchmark:

Memory Bandwidth(RAM)

Function Best Rate MB/s Avg time Min time Max time

Copy: 14686.9 0.011394 0.010894 0.014638

Scale: 14731.7 0.011507 0.010861 0.014544

Add: 16858.7 0.014702 0.014236 0.016994

Triad: 16813.9 0.014405 0.014274 0.014917

Processor(CPU)

* 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: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
* CPU family: 6
* Model: 140
* Thread(s) per core: 2
* Core(s) per socket: 4
* Socket(s): 1
* Stepping: 1
* CPU max MHz: 4200.0000
* CPU min MHz: 400.0000
* BogoMIPS: 4838.40
* Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant\_tsc art arch\_perfmon pebs bts rep\_good nopl xtopology nonstop\_tsc cpuid aperfmperf tsc\_known\_freq pni pclmulqdq dtes64 monitor ds\_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4\_1 sse4\_2 x2apic movbe popcnt tsc\_deadline\_timer aes xsave avx f16c rdrand lahf\_lm abm 3dnowprefetch cpuid\_fault epb cat\_l2 cdp\_l2 ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow flexpriority ept vpid ept\_ad fsgsbase tsc\_adjust bmi1 avx2 smep bmi2 erms invpcid rdt\_a avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb intel\_pt avx512cd sha\_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves split\_lock\_detect user\_shstk dtherm ida arat pln pts hwp hwp\_notify hwp\_act\_window hwp\_epp hwp\_pkg\_req vnmi avx512vbmi umip pku ospke avx512\_vbmi2 gfni vaes vpclmulqdq avx512\_vnni avx512\_bitalg avx512\_vpopcntdq rdpid movdiri movdir64b fsrm avx512\_vp2intersect md\_clear ibt flush\_l1d arch\_capabilities
* Virtualization: VT-x
* L1d cache: 192 KiB (4 instances)
* L1i cache: 128 KiB (4 instances)
* L2 cache: 5 MiB (4 instances)
* L3 cache: 8 MiB (1 instance)
* NUMA node(s): 1
* NUMA node0 CPU(s): 0-7
* Vulnerability Gather data sampling: Mitigation; Microcode
* Vulnerability Itlb multihit: Not affected
* Vulnerability L1tf: Not affected
* Vulnerability Mds: Not affected
* Vulnerability Meltdown: Not affected
* Vulnerability Mmio stale data: Not affected
* Vulnerability Retbleed: Not affected
* Vulnerability Spec rstack overflow: Not affected
* Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl
* Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and \_\_user pointer sanitization
* Vulnerability Spectre v2: Mitigation; Enhanced / Automatic IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS SW sequence
* Vulnerability Srbds: Not affected
* Vulnerability Tsx async abort: Not affected

CPU supports AVX512 family instruction set

1. Performance without compiler optimization -O0

Time taken: 0.002450 seconds

GFLOPS: 0.816327

All Speedup: is compared with baseline -O0

2. Performance with compiler optimization -O1, -O2, -O3

* -O1 : gcc -O1 mat-vec.c -o matp\_1
* Time taken: 0.001040 seconds
* GFLOPS: 1.923077
* Speedup=2.3557
* -O2 : gcc -O2 mat-vec.c -o matp\_2
* Time taken: 0.000957 seconds
* GFLOPS: 2.089864
* Speedup=2.56
* -O3 : gcc -O3 mat-vec.c -o matp\_3
* Time taken: 0.000981 seconds
* GFLOPS: 2.038736
* Speedup=2.4974

Performance with flag: O3 ~ O2 > O1 >> O0

The performance gap between -O1 and -O0 is notably large, highlighting the effectiveness of compiler optimization in improving program execution speed and efficiency.

Additionally, the speedup achieved with -O3 and -O2 optimizations over -O1 and -O0 is substantial.

3. Use any other interesting compiler flag combination and report the performance:

* gcc -march=native -O2 mat-vec.c
* Time taken: 0.001082 seconds
* GFLOPS: 1.848429
* Speedup:2.2643
* gcc -march=native -O3 mat-vec.c
* Time taken: 0.000996 seconds
* GFLOPS: 2.008032
* Speedup:2.4597
* gcc -O3 -ftree-vectorize mat-vec.c
* Time taken: 0.000944 seconds
* GFLOPS: 2.118644
* Speedup:2.5952
* gcc -funroll-loops -O3 mat-vec.c
* Time taken: 0.000935 seconds
* GFLOPS: 2.139037
* Speedup:2.6202
* (Using all optimization flags at once may not always provide the best results)
* gcc -march=native -O3 -funroll-loops mat-vec.c
* Time taken: 0.001095 seconds
* GFLOPS: 1.826484
* Speedup:2.2374
* gcc -O3 -march=native -flto -ftree-vectorize -funroll-loops -fprofile-generate -o matrix\_vector\_multiply\_optimized mat-vec.c
* Time taken: 0.000936 seconds
* GFLOPS: 2.136752
* Speedup:2.6174

**Analysis:**

* The flag -O3 consistently provides better performance compared to -O2, indicating that higher levels of optimization lead to improved execution speed and efficiency.
* Flags like -ftree-vectorize and -funroll-loops further enhance performance by enabling vectorization and loop unrolling optimizations, respectively.
* The combination of multiple optimization flags (-march=native -O3 -flto -ftree-vectorize -funroll-loops -fprofile-generate) yields the best performance with the lowest execution time and highest GFLOPS, indicating a significant speedup compared to the baseline.
* However, it's worth noting that using all optimization flags at once may not always provide the best results, as seen in the case of (-march=native -O3 -funroll-loops), where the performance slightly decreased compared to -O3 alone.

4. Performance with manual code vectorization using intrinsics:

See vectorized\_intrinsic\_mat\_vec.c

* gcc -mavx512f -o matrix\_vector\_multiply\_avx512 vectorized\_intrinsic\_mat\_vec.c
* Time taken: 0.000737 seconds
* GFLOPS: 2.713704
* Speedup:3.3242
* gcc -mavx512f -O3 -o matrix\_vector\_multiply\_avx512 vectorized\_intrinsic\_mat\_vec.c
* Time taken: 0.000728 seconds
* GFLOPS: 2.747253
* Speedup:3.3653

**Analysis:**

* This demonstrates the effectiveness of utilizing hardware-specific features, such as AVX-512 SIMD instructions.
* Manual code vectorization using AVX-512 intrinsics (**-mavx512f**) significantly improves performance.
* resulting in a notable speedup compared to non-vectorized implementations and even optimized versions(having flags also; vectorize) without manual vectorization
* The optimization level **-O3** further enhances the performance.

5. Performance with only parallelization using OpenMP

See ma\_vec\_omp.c

* gcc -fopenmp ma\_vec\_omp.c
* Number of threads: 8
* Time taken: 0.000832 seconds
* GFLOPS: 2.403901
* Speedup:2.9447
* gcc -fopenmp -O3 ma\_vec\_omp.c
* Number of threads: 8
* Time taken: 0.000614 seconds
* GFLOPS: 3.259288
* Speedup:3.9925

Analysis:

* With OpenMP parallelization alone, the time taken is reduced significantly compared to the baseline (-O0).
* the GFLOPS increases substantially compared to the baseline.
* OpenMP parallelization alone, the speedup is approximately 3x, indicating a significant performance improvement.
* When combining OpenMP parallelization with compiler optimization (-O3), the speedup further increases to around 4x compared to the baseline.

6. Performance with vectorization and OpenMP:

* gcc -mavx512f -fopenmp vectorization\_intrinsic\_OMP.c
* Time taken: 0.000595 seconds
* GFLOPS: 3.361345
* Speedup:4.1175
* gcc -mavx512f -O3 -fopenmp vectorization\_intrinsic\_OMP.c
* Time taken: 0.000560 seconds
* GFLOPS: 3.571429
* Speedup:4.3791

See vectorization\_intrinsic\_OMP.c

Analysis:

* Combining vectorization with OpenMP parallelization results in further performance improvement compared to the baseline.
* With vectorization and OpenMP parallelization, the GFLOPS increases significantly compared to only vectorization or only parallelization .
* The speedup is approximately 4.11x without compiler optimization and approximately 4.38x with compiler optimization.
* This improvement suggests that the code is efficiently utilizing the hardware capabilities for parallel processing and vectorization.
* The GFLOPS further improves when combined with compiler optimization -O3
* This highlights the importance of leveraging both hardware-specific optimizations (like vectorization) and software optimizations (like OpenMP parallelization and compiler optimization) to maximize the performance of computational tasks.