

Flake Encoder

Performance Tuning Project

By

Ami Schwartzman

Supervisor: Liat Atsmon

Project Background

- Choose an open source application
- Tune it's performance by exploiting architecture specific advantages such as multicores/multithreads, SIMD instructions, etc.
- Improve algorithms efficiency
- Return SW to the community



Tools

- All work was done in Linux
- Compilers: GCC & ICC
- Debuggers: GDB & Valgrind
- Threading library: OpenMP

- Intel VTune
- Intel PTU – Performance Tuning Utility
- Intel Thread Checker



Flake – FLAC Encoder

- Open source FLAC encoder
- FLAC – Free Lossless Audio Codec
- Flake was developed as an alternative to FLAC
- Achieves about the same compression rate as FLAC, but has better timing
- Original version has no machine specific optimization
- Comes with a frontend tool



Benchmarking method

- Benchmarking is time based
- Results are averaged over 10 iterations*
- Benchmarks are divided into raw processing time and IO time
- Several large files used, 300-500 MBs
- Two types of benchmark: default / hard



Original program

- No threads
- No SIMD operations
- Flow:
 - Read a frame from DISK
 - Encode it
 - Write back to DISK
- Processing time: 18 seconds / 198 seconds
- Good IO performance: 0.2 seconds



First run: Thread Optimizations

- New flow:
 - Read entire file into memory
 - Divide the file into frames
 - Divide the frames among threads via OpenMP using dynamic scheduling
 - Keep results in memory
 - Write back file to DISK
- High memory requirements



First run: Thread Optimizations

- Results:
 - Processing time reduced almost by number of threads:
 - 1 thread: 18 seconds / 200 seconds
 - 2 threads: 9.5 seconds / 103 seconds
 - 4 threads: 5.2 seconds / 53.4 seconds
 - 8 threads: 4.6 seconds / 48 seconds
 - IO time increased dramatically: 5.5 seconds



Second run: Thread Optimizations

- New flow:
 - Read a chunk of file into memory (2MB)
 - Divide the chunk into frames
 - Divide the frames among threads via OpenMP using dynamic scheduling
 - Keep chunk results in memory
 - Write results to DISK
- Memory requirements reduced significantly



Second run: Thread Optimizations

- Results:
 - Processing time for 1 & 2 threads didn't change.
 - Processing time:
 - 4 threads: 6.8 seconds / 55 seconds (increased)
 - 8 threads: 5.5 seconds / 50.4 seconds
 - IO time reduced to 0.7 seconds – almost same as original program
 - ~10% - 40% improvement in 4 threads over previous version in total run time



Third run: Architectural Optimizations

- Program flow same as before
- Architectural optimizations such as using aligned mallocs and rearranging data structures to align arrays of frequent use. All alignments are made to 16-Kbyte boundary.
- Using VTune to find hot spots



Third run: SIMD Optimizations

- First function found by VTune was rewritten using inline Assembly & SSE instructions.
 - Results weren't better than compiler's own optimizations
- Second function found by VTune was rewritten using SSE intrinsics
 - Highly compatible between GCC & ICC
 - Loop unrolling



Third run: SIMD Optimizations

- Results:
 - Processing time reduced:
 - 1 thread: 17.1 seconds / 181.5 seconds
 - 2 threads: 9 seconds / 92.5 seconds
 - 4 threads: 6.4 seconds / 48.5 seconds
 - 8 threads: 5.2 seconds / 42.7 seconds
 - IO performance didn't change
- ~5% – 13.5% improvement over previous version



Forth run: Intel Compiler

- Mixed results: couldn't benchmark in "hard" mode due to extremely long run times
 - Processing time:
 - 1 thread: 14.9 seconds
 - 2 threads: 8.3 seconds
 - 4 threads: 5.6 seconds
 - 8 threads: 4.8 seconds (increased IO time)
 - IO performance didn't change
- When there was improvement over previous version it was $\sim 8.5\% - 14.7\%$



Overall Improvements

