

HPC - Lab 02

Aubry Mangold
ISCS, HEIG-VD
aubry.mangold@heig-vd.ch

Benchmarking

Introduction

The following report describes the benchmarking of the DTMF encoding and decoding program developed in the previous lab. The goal of this lab is to analyze the performance of the program using `likwid`, identify potential bottlenecks and implement or suggest fixes.

System

The specifications of the system used throughout this practical work are presented in Figure 1. features the following topology: Below is a concise summary of the processor and memory topology. An accompanying figure will illustrate further details.

- **CPU:** Intel Core i5-6200U @2.30 GHz (1 socket, 2 cores with 2 threads each)
- **Caches:** L1: 32 kB (per pair of threads), L2: 256 kB (per pair of threads), L3: 3 MB (shared)
- **Memory:** 2 x 8GB SODIMM DDR3 Synchronous RAM @1600 MHz in a single NUMA domain
- **OS:** Ubuntu x86_64 (kernel 5.19.0) with scaling governor set to performance and turbo features disabled

The topological specifications of the system are presented in Figure 1.

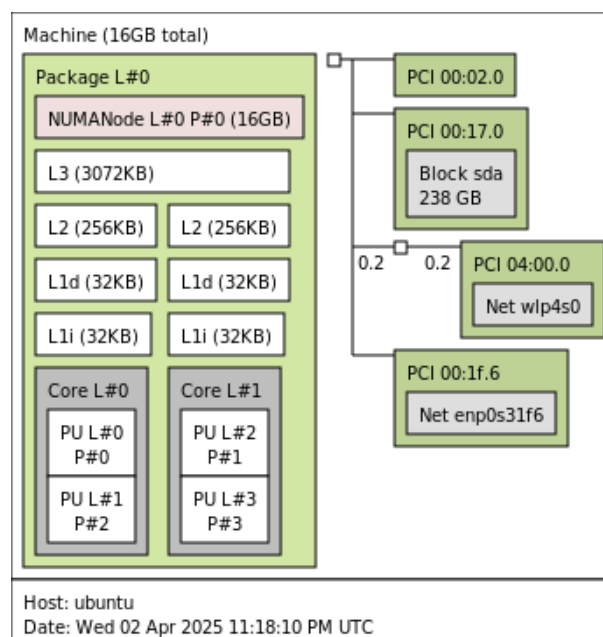


Figure 1: System topology

Roofline model

The roofline model that is devised for the performance monitoring of the *DTMF* application is based on the `likwid` FLOPS/s and MBytes/s metrics. These metrics were chosen for this lab because the program's internal structure is based on the Fast Fourier Transform and Goertzel algorithms, which are computationally intensive algorithm that requires a lot of floating-point operations. The roofline model is then used to plot the baseline performance of the program, to define the bounds (memory, CPU or I/O) of the program and to identify if some given refactoring improved the program's performance.

The `likwid-bench stream` mode was chosen to produce the roofline because the program reads and writes data to the memory (in contrast to the `read` mode which only measures data reads). Some trial and error were necessary to find the `likwid peakflops` and `stream` configurations that produced the best results. To minimized variance due to the OS' use of the CPU cores, all benchmarks done throughout this report were pinned to the second core of the CPU. The commands are as follows:

```
taskset -c 2 sudo likwid-bench -t peakflops -w S0:32kB:1  
taskset -c 2 sudo likwid-bench -t stream -w S0:3MB:1
```

Listing 1: Commands used to produce the data of the roofline model

The operational intensity metric is then computed by dividing FLOPS/s with MBytes/s to get the number of floating-point operations per byte transferred. After some research, it turns out that counting all instructions (including integer instructions, control instructions, etc.) would dilute the model's focus on the critical floating-point instructions and would not produce any meaningful results. Moreover, the functions that are benchmarked all make heavy use of floating-point calculations.

The produced roofline model is presented at Figure 2.

The result of the `peakflops` test using the size of the 32Kb L1 cache is **5568.76 MFLOPS/s**. The result of the `stream` test using the size of the L3 cache (3MB) as the workingset is **25732.69 MB/s**.

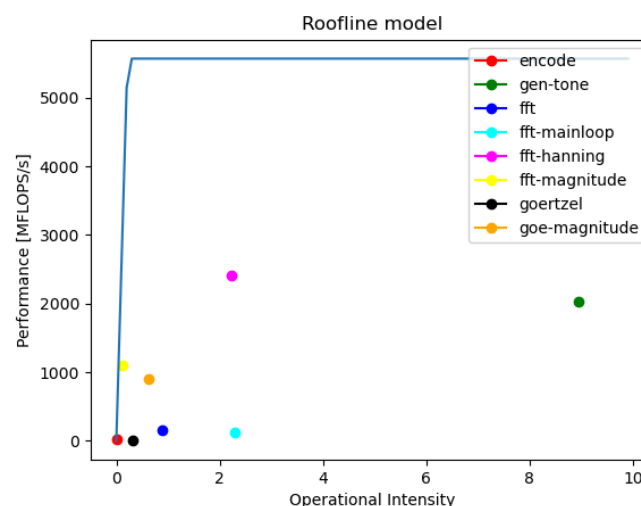


Figure 2: Roofline model

Performance group

A custom likwid group is devised to show only the metrics relevant to the application. The custom group is called HPC and includes FLOPS/s, MBytes/s and OpIntensity.

Source code markings

The library files relative to the encoder and the Goertzel/FFT decoders were modified to integrate the likwid profiling macros. Quite a few regions of the program were wrapped in likwid markers because we to be broad with the potential areas of interest while working on performance improvements. Having a rather large set of markers enables us to make more informed decisions when evaluating the impact of a tweak to the program.

For the sake of brevity, the markers are only listed once in Table 1.

Performance baseline

The baseline performance tests are performed with the program compiled in Release mode using the 03 flag. A sufficiently large test input of about 1290 bytes is provided in order to emulate real-world conditions.

The results of an initial run of likwid are presented in Table 1. These result form the baseline of the program's performance and are used to compare the performance of the program before and after the optimizations.

Region	FLOPS [MFLOPS/s]	Memory throughput [MBytes/s]	Operational Intensity [FLOP/Byte]
encode	28.51	3391.85	0.01
encode-tone-gen	2035.57	227.48	8.94
decode-preprocess	1624.96	5435.95	0.30
decode-fft	151.76	173.12	0.88
decode-fft-mainloop	117.65	51.22	2.30
decode-fft-hanning	2403.93	1034.54	2.23
decode-fft-magnitude	1099.19	9977.86	0.11
decode-goe	13.84	43.43	0.32
decode-goe-mainloop	12.22	37.99	0.32
decode-goe-magnitude	907.72	1463.54	0.62

Table 1: Baseline performance values

The results of the performance baseline of the groups are display in Figure 2.

Performance analysis

Encoding

In encode mode, there is a stark contrast between the main encode region (with low MFLOP/s but relatively high memory throughput) and the encode-tone-gen region (with much higher MFLOP/s but lower memory throughput). The extremely low overall operational intensity suggests that most of the time is spent moving data rather than performing arithmetic, making the region memory-bound. The encode-tone-gen in comparison exhibits a higher operational intensity due to code sections that carry out a larger amount of arithmetic operations per byte of data transferred.

FFT decoding

The decode-preprocess region has high memory bandwidth and moderate FLOP rate, hinting that data is moved frequently while performing a good amount of transformations. The FFT benchmarking shows that decode-fft-mainloop and decode-fft-hanning hit relatively high compute and memory throughput. The decode-fft-magnitude function mostly reads from the real and imaginary arrays and then writes back after a few math operations, making the region memory-bound and explaining why its bandwidth is high while its operational intensity stays low.

Goertzel decoding

The Goertzel sections show mostly low MFLOP/s and memory bandwidth values. Their operational intensities remain below 1 FLOP/Byte, indicating that they do not perform extensive calculations relative to data movement. This might be because the data is copied around quite a bit throughout the implementation. This might arise from poorly-sized loops and windows causing overhead. The decode-goe-magnitude region delivers high MFLOP/s and correspondingly high operational intensity due to the fact that the magnitude calculations are mostly cpu-bound.

Improvements

The correctness of the applied improvements was verified against the baseline performance values. End-to-end tests are implemented using GoogleTest ensure that the program still works as expected after each round of optimization. The performance improvements are also measured with likwid.

The following improvements were tried on the program:

Compiler optimizations

Aggressive compiler optimization produced a small improvement in operational intensity in the encode region (particularly the tone-gen part which saw its *OI* reach 12), but deteriorated overall performance of other regions. After some trial and error, the only optimization that was kept is the change from 03 to 0fast which produced a really small overall improvement. The options -march=native -mtune=native -flto -funroll-loops -fstrict-aliasing -fomit-frame-pointer -ffast-math -funsafe-math-optimizations -fno-math-errno were not kept. This doesn't mean that the compiler has given everything it has to offer and the issue should be explored further.

Further work

A few potential areas of improvement are:

- More aggressive platform-dependent compiler optimizations
- Usage of SIMD instructions to speed up the FFT and Goertzel algorithms by vectorizing the code
- Parallelization of the decoding process which can be done by splitting the input data into smaller chunks and processing them in a thread, leading to a speedup of the decoding process.
- Caching of the frequently seen frequencies while decoding. Might be possible because the number of frequencies that are used to encode the 12 DTMF tones is limited. This would need to be done in a way that does not introduce any additional overhead, and might not even be possible due to the imprecise nature of computerized floating-point calculations.

Conclusion

Overall, this work showcases the importance of the establishment of performance baselines and how it is not only used to measure performance improvement, but also to detect potential bottlenecks in

the code. The roofline model helps to visualize the performance of the program and to identify areas of improvement. The `likwid` library, although somewhat difficult to use, is a useful tool measure the performance of the program and to identify potential bottlenecks or other discrepancies. It also helps developers gauge if the behavior of the code somewhat matches what's expected.

The performance analysis of the program shows that there are areas of improvement in the encoding and Goertzel decoding sections. This is explained by that fact that these portions of the source code do not use any well-known libraries – `fftw3` for instance – and are hence less optimized because they did not benefit from enough scrutiny. Further work would be needed to improve the performance of the program and to make it really efficient.