Designing for Optimisation

Måns Rullgård

Why optimise?

- Increase throughput
- Reduce latency
- Reduce power consumption
- Meet deadlines

How to optimise

- Help the compiler
- Write key functions in assembly

Modular code

- Isolate basic operations
- Use simple interfaces
- Simplifies use of libraries and hardware

Simple interfaces

- Never use struct arguments
- Prefer multiple functions over conditionals
- Split out common cases

SIMD-friendly data

- Align arrays
- Pad to multiple of SIMD size
- Arrange for linear access

SIMD-friendly algorithms

- Allow 16-bit arithmetic (integer / fixed-point)
- Allow single precision floating-point
- Avoid division

SIMD-friendly algorithms

- Consider available rounding modes
- Can slight variations be tolerated?

Compiler-friendly code

- Use constant loop iterations
- Use const and restrict qualifiers
- Don't be too clever
- Avoid aliasing, use local variables

memcpy is murder

- Careless copying costs cycles
- Avoid in-place transformations
- Support user-supplied buffers

Integration

- Allow optimising individual functions
- Activate optimised versions at runtime
- Use function pointers

Tools

- Benchmarking
- Profiling
- Code analysis

Benchmarking

- Full application
- Micro-benchmarks

Benchmarking

- Disable power management
- Pin to single CPU (taskset)
- Run at full speed

Profiling

- Disable PM, pin to single CPU
- Linux perf

Linux perf

- Non-invasive profiling framework
- perf stat: global statistics
- perf record: collect and save profile
- perf report: display recorded profile
- perf annotate: display annotated code

Code analysis

- Assembly: objdump, gcc -S
- readelf
- dwarves

Questions?