DSP Optimization Techniques*

Matt Kleffner
Douglas L. Jones

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 1.0†

Abstract

An overview of DSP optimization techniques, from the design stage, through high-level implementations, to assembly implementations.

1 Introduction to Code Optimization

Most practical DSP applications have a clock-cycle and/or memory budget. Initial implementations typically don't meet these budgets; therefore, the code must be optimized. Code development usually follows six steps:

1. Develop algorithm on paper
2. Simulate in MATLAB
3. Develop and simulate more efficient implementations
4. Implement algorithm in C
5. Use library routines when available
6. Use optimizing compiler
7. Manually write assembly routines for key routines

2 Develop algorithm on paper

The algorithm to be implemented should first be designed on paper. In addition to equations describing the algorithm, its design should include inputs and outputs of each routine and a flow chart of the operation of the complete program. **Never design an algorithm or a program at a computer.** While it may be possible to implement some basic algorithms and programs this way, compilers cannot overcome any design flaws that are likely to be introduced.

3 Simulate in Matlab

Before any C code is written, the algorithm should be developed and simulated in MATLAB since problems with the algorithm design can be found more easily. This is done by applying the algorithm to test vectors and inspecting and/or plotting the results. Once the algorithm is completely defined, C implementation can begin. Recall that in each of the previous labs a MATLAB simulation step was given before assembly implementation.

*Version 1.2: Sep 21, 2004 5:06 pm +0000
†http://creativecommons.org/licenses/by/1.0

4 Develop and simulate more efficient implementations

An efficient algorithm used as few multiplications and additions as possible. Applying DSP theory to "simplify" the algorithm is a common way of doing this. The autocorrelation function is a simple example: a simple-minded way of implementing this is to compute sums for each lag, but careful inspection of the autocorrelation function reveals that it is actually a function of the **absolute value** of the lag. Therefore each value at a negative lag is identical to the value at the corresponding positive lag. Approximately half the apparent multiplications and additions are therefore required. If the entire autocorrelation sequence is needed, the autocorrelation can be optimized even further if it is treated as a convolution of two sequences. An FFT and an inverse FFT can then be used to compute the autocorrelation.

5 C Implementation

After the algorithm has been designed and optimized, an initial implentation in C is done. Tips on writing code with efficiency in mind can be found *here*[**?**]. It is important, however, to get a working implementation of the algorithm in a reasonable amount of time as some optimizations cannot be anticipated by the programmer. Some of these coding techniques in the URL reference can be applied later on when it is clear which routines need the most optimization. This implementation can serve as a reference implementation to compare the correctness and speed of optimized versions.

6 Library routines

If the algorithm uses common mathematical operations, such as the cosine and FFT operations, it is usually wise to use existing library routines instead of "reinventing the wheel." As many library routines are readily available from DSP manufacturers and over the internet, the first factor to consider in using a library is its license: do you have permission to use it in your application? Libraries can often be used freely for educational and research purposes, but any other use requires inspection of the library license.

The second factor to consider is the design goal of the library: was it designed for speed or low memory usage? Typically speed can be bought with more memory and vice-versa, so when selecting a library it is important to decide which budget (speed or memory) is more important with respect to the routine.

7 Compiler Optimization

Recall that the basic operation of a C compiler is to translate C source code into assembly instructions and then into an executable.

> *"Compiler optimization is used to improve the efficiency (in terms of running time or resource usage) of the executables output by a compiler. These techniques allow programmers to write source code in a straightforward manner, expressing their intentions clearly, while allowing the computer to make choices about implementation details that lead to efficient execution. Contrary to what the term might imply, this rarely results in executables that are perfectly "optimal" by any measure, only executables that are much improved compared to direct translation of the programmer's original source."* - [**?**]

An optimizing compiler traditionally groups optimizations into **phases**. Each phase contains a series of optimizations (or transformations) that are performed in a fixed order. These phases are usually turned on with command-line flags such as `-O1`, `-O2`, etc. Each flag indicates an optimization "level" where the level includes all of the lower levels. At higher optimization levels bugs in the code are sometimes introduced, so it is important to check the behavior of a compiler-optimized program against the reference implementation. Keep the highest optimization level that produces accurate code.

At this point the compiled code should be checked against the budgetary constraints. Is it fast enough? Does it fit in available memory? Total memory usage is placed in a file produced by the compiler (sometimes a command-line flag is needed for this). Speed can be measured in a couple of ways. The most common

method is the use of a **profiler**. A profiler tracks the performance of the program, providing data on how many times each function is called, as well as how much time each function takes in terms of cycles and percentages of total program cycles. A simulator also allows clock cycles to be measured, typically by allowing the user to place breakpoints around sections of code to be measured. If the speed and memory properties of the compiled code fit the budget, optimization is finished. If not, some of the routines must be hand-written in assembly.

8 Write key assembly routines manually

Finally, if the budget cannot be met with the other optimization techniques some routines must be written in assembly. Manually-written assembly code is usually the most efficient, but it is labor-intensive and it is not portable to other architectures. Therefore it is done as a last resort and only on routines that absolutely require it for budget constraints to be met. This is done by rewriting the routine that consumes the largest portion of the budget, followed by the next largest budget-consuming routine and so on until the budget is met. It should be noted that this step is almost always required in embedded applications as current state-of-the-art C compilers and optimizers do not produce sufficiently fast and/or small code.

If meeting the budget is unexpectedly difficult, remember that **no compiler optimization or assembler can effectively overcome a poor algorithm design or implementation.** If you are confident that your implentation is fast and accurate, then the budget may be too tight for the application. Either some parts of the application must be removed (extra "features", for example) or an architecture with more resources must be used.

References

[1] Various authors. Compiler optimization. *Wikipedia*, 2002-Present. http://en.wikipedia.org/wiki/Compiler_optimization.