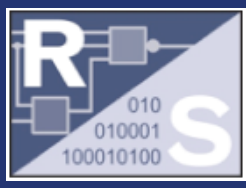
Optimization of a GPS Acquisition Algorithm for an AMIDAR Processor

Lab Report by Fallou Galass Coly and Julian Käuser
Supervisor: M.Sc. Johanna Rohde
Begin: 01/15/2018| Submission: 02/26/2018
Institute for Computer Engineering | Computer Systems Group | Prof. Dr.-Ing. Christian Hochberger

TECHNISCHE
UNIVERSITÄT
DARMSTADT



1 Introduction

In this lab, our task was to find an optimized combination of mathematical modification, code and hardware configuration of a GPS signal acquisition algorithm for the RS AMIDAR processor. The quality of the solution is measured in two metrics: required ticks for the computation in the AMIDAR simulato, further called *performance optimization* and consumed energy as measured by the simulator. For both goals, dedicated solutions in all aspects are allowed, but not required. All code must be written in Java 1.4 in order to obatin the compatibility with the current AMIDAR toolchain.

Our presented solutions include optimizations in the following fields:

1. **Mathematical Conversions**
   The algorithm represents a cross convolution between a vector of LA-1 codes and the samples of the GPS receiver. Any mathematical conversion which yields the same results can be applied. The ability to map the applied conversions efficiently in code and onto the hardware may be an aspect to watch while applying these conversions.

2. **Code Style Optimizations**
   Although the required programming language is only the Java 1.4 standard, many different ways to implement the mathematical description are possible. Additionally, the effects on the hardware mapping, orders of execution, parallelism and (eventually missing) compiler optimizations have to be taken into account.

3. **Hardware Configuration**
   The AMIDAR processor is parametrizable in some aspects and features a Coarse-Grained Reconfigurable Array (CGRA), which allows a high speedup if used and configured properly. This configuration, called *composition*, can particularly speedup the execution or affect the overall energy consumption.

In this report, an overview of our solutions in the presented fields is given in the particular order. The report closes with an evaluation of the results.

1.1 Original Algorithm

The subject of all optimizations is the following algorithm, which presents the extraction of a peak in the cross correlation between a sampled signal and a vector of LA-1 codes. The cross correlation is presented as the circular convolution of the inversed code vector with the sample vector, which can be converted to a DFT/IDFT pair:

eqns here

2 Mathematical Optimizations

In this chapter, the applied mathematical conversions are presented.

2.1 Frequency Shifting

For each of the target frequencies to examine, the sample vector $f_d(n)$ is multiplied by a complex factor $e^{-j2\pi n f_d/f_s}$. No mathematical expression, which improves this multiplication, is applied. In the following, the term f_d regards this multiplied vector for the sake of readability.

2.2 Cross-Correlation with Fast-Fourier Transforms

The kernel of the acquisition algorithm is the computation of the cross-correlation R between the vector $f_d(n)$ and the code vector $c(n)$:

$$R_{f_d,c}(n) = f_d(n) \star c(n) \tag{2.1}$$

The cross-correlation can be expressed in the terms of the *circular convolution* in the time domain:

$$R_{f_d,c}(n) = f_d(n) \circledast c(-n) \tag{2.2}$$

Note that the operation $c(-n)$ equals the order-reversal of the vector $c(n)$. In the frequency domain, the circular convolution resembles the element-wise product of the Fourier-transforms of the arguments:

$$f_d(n) \circledast c(n) \circ\!\!-\!\!- F_d(k) \cdot C(k) \tag{2.3}$$

If this conversion is used to compute the cross-correlation of the two vectors, both vectors have to be Fourier-transformed. Then, their element-wise product must be inverse-Fourier-transformed. In our case, all convolutions and transforms are realized discretely. Hence, the Fourier transform must be realized as DFT and IDFT, respectively.

For the discrete forward- and inverse Fourier transformation as well as the discrete convolution, the algorithmic complexity is in general $\mathcal{O}(N^2)$, where N is the length of the input vectors. This means that the number of basic operations (in our case, complex multiplications and additions of floating point values) scales quadratically in magnitude with the length of the signal and code vectors. If this complexity can be reduced, a lot of computations may be avoided, promising a runtime speedup.

In out solution, an approach to reduce the $\mathcal{O}(N^2)$ complexity at the cost of artificial extension of the input vectors is used. It is based on the characteristic of *Fast Fourier Transforms*(FFT), which allow a complexity of $\mathcal{O}(N \log N)$.

FFTs typically require input vectors of a length $N = 2^{k1}$, which is not given in general in this task. Therefore, a solution to be able to make use of these transforms for arbitrarily length input vectors is necessary.

The formula for the discrete circular convolution is given by 2.4:

$$a(n) \circledast b(n) = \sum_{k=0}^{N-1} a(k) \cdot b((n-k)\ mod\ N) \qquad \forall n \in N \tag{2.4}$$

As the formula shows, the resulting vector has a length of N. The *linear* convolution of two vectors $a(n) * b(n)$ generally results in a length $L_{lin} = L_a + L_b - 1$, where L_x is the length of vector x. If equations 2.5 and 2.6 are fulfilled,

$$a'(n) = \begin{cases} a(n) & n \in 0...L_a - 1 \\ 0 & n \in L_a...L_a + L_b - 1 \end{cases} \tag{2.5}$$

$$b'(n) = \begin{cases} b(n) & n \in 0...L_b - 1 \\ 0 & n \in L_b...L_a + L_b - 1 \end{cases} \tag{2.6}$$

the linear convolution of $a'(n)$ and $b'(n)$ and their circular convolution are equal:

$$a'(n) * b'(n) \stackrel{!}{=} a'(n) \circledast b'(n) \tag{2.7}$$

In that case, the transform relation $\mathscr{F}^{-1}\{\mathscr{F}\{a'(n)\} \cdot \mathscr{F}\{b'(n)\}\}$ equals the linear convolution.

The conversion applied in equations 2.5 and 2.6 are a *zero-padding* of the vectors in the time domain. The vector length is extended, whereas no information is added. This equals a over-sampling in the frequency domain: the resulting spectrum is interpolated from the original spectrum. Oversampling in one of the two domains can always be reverted by applying *aliasing* in the other domain. Aliasing is the process of wrapping components with an index $K + l$ higher than a threshold index K over and adding them up to the first l original samples:

$$\tilde{y}(n) = \sum_{r=-\infty}^{\infty} y(n + rN) \tag{2.8}$$

The ∞ expressions may be replaced with borders 0 and 1, because the values of $y(n)$ are 0 for negative indices and indices larger than $2N - 1$.

Therefore, the circular convolution of two vectors $a(n)$ and $b(n)$ can be expressed as the linear convolution of the zero-padded vectors $a'(n)$ and $b'(n)$, followed by an aliasing:

[1] There are implementations for basically every N, but these often require a knowledge about the length before the runtime or include recursion, which is badly suited in out case

$$a(n) \circledast b(n) \stackrel{!}{=} \text{alias}\left(a'(n) * b'(n)\right) \tag{2.9}$$

The only requirement for equation 2.9 is that the vectors $a'(n)$ and $b'(n)$ are padded to at least length $L_{lin} = L_a + L_b - 1$.

As one can see, the realtion given by equation 2.9 allows the use of vectors extended to a length N', which has only a lower bound, to compute the circular convolution of two vectors. Hence, the length N' can be chosen such that $N' = 2^k, k \in \mathbb{N}$. This allows the use of classical FFT algorithms to find the circular convolution of arbitrarily sized input vectors:

$$a(n) \circledast b(n) = \text{alias}\left(\text{IFFT}\left(\text{FFT}(a'(n)) \cdot \text{FFT}(b'(n))\right)\right) \tag{2.10}$$

, where

$$L_{a'(n)} = L_{b'(n)} = \min(2^k) \geq L_{a(n)} + L_{b(n)} - 1,\ k \in \mathbb{N} \tag{2.11}$$

Of course, this approach is of complexity $\mathcal{O}(N' \log N')$, but with N' at least twice as large as the original length for a $\mathcal{O}(N^2)$ circular convolution or IDFT-DFT approach. In the worst case, the original length is $N_{original} = 2^k + 1$, so that the the difference in length is maximal. Nevertheless, for larger lengths of the input vectors, the converted approach may save a significant amount of operations compared with the N^2 complexity convolution.

2.3 Chosing the FFT Implementation

For input lengths $N = 2^k$, the Cooley-Tukey-Algorithm provides a fast and proven implementation for FFTs. The original algorithm is recursive; it is available as iterative algorithm, which is favored for the AMIDAR processor, since loops can be accelerated on the CGRA. For input vector which force the length N' to be a even power of two, the radix-4 variant of the Cooley-Tukey-Algorithm can be used. A radix-4 implementation saves 25% of complex multiplications compared to the radix-2 variant. In uneven cases, the classical radix-2 variant must be used.

Both FFT variants include a bit-reversed index reordering of the input samples. This means that the new index of a samples, after the reordering, equals the bit-reversed old index:

$$\text{index}_{old} = \sum_{k=0}^{N} a_k 2^k$$
$$\text{index}_{new} = \sum_{k=0}^{N} a_{N-k} 2^k$$

A reordering in that manner requires non-sequential array accesses, which are usually not efficient regarding the cache performance. As the order of the output of the FFT is not relevant for the element-wise multiplication, the reordering is only be required due to the inverse FFT. Nevertheless, there are two other distinct FFT techniques, which differ in the order of the multiplication with the *e* terms: *Decimation-in-Time* (DIT) and *Decimation-in-Frequency* (DIF). DIT-FFTs require a bit-reverse index ordered input and produce natural-ordered output; DIF-FFTs require a natural-ordered input and produce bit-reverse index ordered output. If the FFT of codes and the samples are transformed into the frequency domain by a DIF-FFT and transformed back to the time domain by a DIT-FFT, no reordering is required at all. Hence, 4 different types of FFT are implemented:

1. DIF-Radix-2 forward FFT for uneven powers of two as input length, for the transform of codes and samples to the frequency domain

2. DIT-Radix-2 forward FFT for uneven powers of two as input length, for the transform of the element-wise product to the time domain

3. DIF-Radix-4 forward FFT for even powers of two as input length, for the transform of codes and samples to the frequency domain

4. DIT-Radix-4 forward FFT for even powers of two as input length, for the transform of the element-wise product to the time domain

The inverse FFTs can be implemented as a forward FFT, preceded and succeeded by a complex conjugation of the input and output. The second complex conugation can be omitted because it does not make a difference for the succeeding maximum value examination.

3 Code Style Optimizations

4 Composition Optimization

4.1 Performance Adaption of CGRA Compositions

4.2 Energy-Consumption Optimizations of CGRA Compositions

5 Evaluation

5.1 Performance Optimization

5.2 Energy Consumption Minimization

6 Bibliography