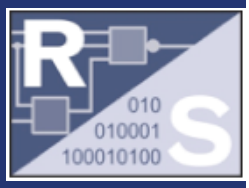
Optimization of a GPS Acquisition Algorithm for an AMIDAR Processor

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



Contents

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 simulator, 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 obtain 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 L1 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.

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.

Algorithmic Complexity

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 our 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.

Circular Convolution by FFTs

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:

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

[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 our case

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:

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

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 relation 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 vectors 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 either of the input or of the output samples. This means that the new index of a sample, after the reordering, equals the bit-reversed old index:

$$\text{index}_{old} = \sum_{k=0}^{N} a_k 2^k$$
$$\text{index}_{new} = \sum_{k=0}^{N} a_{N-k} 2^k$$
$$a_n \in \mathbb{N}, 0 \leq a_n \leq 1$$

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 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 conjugation can be omitted because it does not make a difference for the succeeding maximum value examination.

3 Code Style Optimizations

In this chapter, certain aspects on the way the solution is implemented in Java are presented.

3.1 General Code Structure

The class `Acquisition`, as given by the framework, is once instantiated and performs all computations. The samples and codes are entered by calls to the methods `enterSample` and `enterCode`.

Static Assignments

Constants such as π, 2π, $-\pi$, -2π or precomputed ratios of the sampling rate are defined as `final static` fields of the class. The values are explicitly written out because the Java compiler is not as powerful as other compilers. As we were not sure how far the Java 1.4 compilation standard works, this was a safety decision.

Constructor

The constructor is called with the number of samples as argument. At that time, the decision between a Radix-2 and -4 FFT is made. Arrays for the samples and codes of a length N', as depicted in chapter 2, are allocated.

Entering Samples

Each sample's real and imaginary part is inserted at the current sample index in the samples' real and imaginary array. Then, the index is incremented by 1. Additionally, a global variable holding the total signal power is incremented by the sum of squares of the input sample. It is required later for the acquisition.

Entering Codes

The codes have to be inserted in descending order into their respective arrays, since the relation between the correlation and circular convolution is $\text{samples}(n) \star \text{codes}(n) = \text{samples}(n) \circledast \text{codes}(-n)$. Therefore, the index to insert starts at `nrOfSamples`-1 and is decremented by 1 at the end of the method. All higher indices than `nrOfSamples` are never written and implicitly zero.

3.1.1 Acquisition Kernel

The kernel of the acquisition is started by a call of `startAcquisition`, which returns the acquisition result as boolean.

Algorithm 1 describes the computation for each of the 11 target frequencies. The call of a DIF-FFT for the codes, before the first loop, is actually implemented as method call; all other occurrences of FFTs are inserted at this place, because the CGRA cannot perform method calls.

Data: codes = cReal, cImag; samples
radix = power of 2 even ? 4 : 2;
cReal, cImag = DIF-FFT(radix, codes);
foreach *frequency* $f_d \in$ *targetFrequencies* **do**
    sReal, sImag = new arrays of length N'
    for *n = 0; n<nrOfSamples; n++* **do**
        sReal, sImag = multiply(samples(n), $e^{-2\pi f_{step}/f_s - 2\pi minRate/f_s}$);
    end
    sReal, sImag = DIF-FFT(radix, sReal, sImag);
    for *n = 0; n<N'; n++* **do**
        sReal(n), sImag(n) = multiply(sReal(n), sImag(n), codes(n));
        sImag = -1 · sImag;
    end
    sReal, sImag = DIT-FFT(radix, sReal, sImag);
    maxPower(f_d) = 0;
    maxIndex(f_d) = 0;
    foreach *n = 0; n<nrOfSamples; n++* **do**
        sReal(n) = sReal(n+N')/N';
        sImag(n) = sImag(n+N')/N';
        if *sReal(n)*^{2}+*sImag(n)*^{2} $\geq$ *maxPower*(f_d) **then**
            maxPower(f_d) = (sReal(n)^2+sImag(n)^2);
            maxIndex(f_d) = n;
        end
    end
end
totalMaxPower = 0;
dopplerShift = 0;
codeShift = 0;
foreach *frequency* $f_d \in$ *targetFrequencies* **do**
    if *maxPower*(f_d) $\geq$ *totalMaxPower* **then**
        maxTotalPower = maxPower(f_d);
        dopplerShift = f_d;
        codeShift = maxIndex(f_d);
    end
end
acquisition = totalMaxPower $\geq$ totalPower · 0.015;
return *acquisition*

Algorithm 1: The kernel of the acquisition.

Although the CGRA would have been capable of performing even the outermost loop, the synthesis produced an error, so that this loop was not accelerated. This is the reason why we decided to allocate a new array for the samples *inside* the outermost loop - it is not possible to be accelerated anyway, and by this allocation, we made use of the zero-initialization of simple data type arrays in Java. Like this, $2N' - 2 \cdot$nrOfSamples assignments with zero, which would have been necessary in each iteration, could be avoided.

3.1.2 FFT Implementations

All four types of FFT have been developed by rapid prototyping in octave, which is easier to find errors and eases the handling with complex numbers. Moreover, the octave function `fft` provides a easy accessible reference output.

Common to all four types of FFT is that they consist of three nested loops. The outermost loop is the level of recursion and is executed $\log N'$ times. Within the second loop, blocks of values which have to be computed equally are iterated; the innermost loop holds the computation of the so-called *twiddle factors*, reverses the order of the values from the block, multiplies the values with the twiddle factors and re-assigns them to the memory. In the Radix-4 immplementations, four values are handled in the innermost loop at once. The Radix-2 FFTs only handle 2 values at once.

3.1.3 Complex Algebra

In general, all FFT operations have to be performed as complex-valued. This can be either done in the polar or cartesian representation. Since most of the steps include additions and multiplication, it is easier to perform all oeprations in the cartesian representation. That way, no conversions, which involve square roots and trigonometric functions, are required; a multiplication in the cartesian form is less difficult with four floating point multiplications and two additions.

3.2 Other Coding Remarks

Debugging

No debugging was done in the AMIDAR simulator; functional debugging has only been performed by running the `AcquisitionTest` main method in Eclipse and using the native Eclipse debugger. As stated before, mathematical optimizations have been prototyped in octave.

Object-Oriented Programming

Whereever it is possible, no objects are used. There is no need for object-oriented programming, since the complexity of this task is manageable with only simple data types. Nevertheless, the use of objects would have resulted in a less messy code, which was not among the optimization goals.

Performance and Energy

In the code, no differentiation between performance-optimized and energy-optimized variants has been made. Both proposed solutions use the same code.

4 Composition Optimization

The last field in which changes are allowed in order to enhance the solution quality is the hardware configuration of the AMIDAR processor. Our approach towards these optimization mainly targeted the utilized CGRA, which is highly adaptable. The proposed solutions only differ in the utilized CGRA composition.

In the following, the approaches for both targets are described.

4.1 Performance Adaption of CGRA Compositions

As an obvious first approach, the largest available composition, homogenously operator-equipped and with 16 PEs, is chosen. Since irregularly connected and inhomogneously equipped compositions can yield a significant acceleration (mostly due to characteristics of the used scheduler), we targeted more different compositions.

Finding a good composition in terms of the speedup can be done in different ways. Manual optimizations are time-inefficient and often do not yield much gain. Optimal solutions, defined by algorithms, are not targetable, due to the huge number of combinations. Heuristics either need a lot of knowledge of the problem or require a lot of time to tailor the heuristics to the implementation details. Metaheuristics are often easier to use. In our case, we used a very simple kind of Simulated Annealing, which only differed slightly from a random search of compositions.

4.1.1 Simulated Annealing

Simulated Annealing is a well-known metaheuristic which is able to optimize a configuration by the use of a cost function and a mutation function. The cost function takes a configuration and computes the quality of it, depending on the optimization goal. The mutation function takes a configuration, changes some parameters randomly, and returns the altered configuration. Within two loops, the main algorithm of Simulated Annealing continously generates new configurations with the mutation function, and evaluates the quality. If the quality is better than the previous configuration, the new configuration is always accepted as new optimum. If it is not better, the configuration is only accepted with a probability, which depends on the number of iterations and acceptance.

The Simulated Annealing used here is not thoroughly parametrized. Only the reference implementation (taken from the High Level Synthesis lecture) is used. This evaluates to ca. 8,000 to 10,000 iterations in total. This choice was made due to the lack of time and because even the first tests showed great improvements with this approach.

As a cost function, a schedule produced by the AMIDAR list scheduler is used. It is computed for the currently examined composition; the shorter the schedule and especially the innermost loop, which is most often executed, the better is the composition quality, and the smaller is the cost. Criteria like size of the CGRA and reachable clock frequency are not targeted, because they do not contribute to the optimization goal (although this must be done in practice!).

The mutation function applies changes in two criteria: Processing elements (PEs) are added or deleted, and connections between them are added or removed. This is done up to a maximum size of 16 PEs and a minimum size of three PEs. In any case, the maximum number of memory access PEs is secured by adding as many memory operators as have been removed to the remaining PEs. No more changes are applied to the set of available operations in the PEs, because other kernels in the test environment rely on some operations, which makes the outcome of the Simulated Annealing unpredictable in its overall quality. Therefore, not all potential gain of the Simulated Annealing can be achieved.

4.1.2 Encountered Problems

While the scheduling results for the found compositions by the Simulated Annealing approach promised good solutions, we encountered some problems which are not easy to fix.

First of all, the quality of a composition could only be found for one synthesized method. No combination of kernels could be examined, since the framework only gave access to one CDFG object at a time. Serialising the kernel data for the scheduler was not possible due to inconsistencies in the JSON I/O parts of the framework.

This problem not only led to f̈alse positiveq̈uality compositions. Other kernels, which are part of the test environment for the `Acquisition` class, are also typically executed on the CGRA. These kernels could not be included in the cost evaluation of the composition and could possibly slow down the execution massively.

For unknown reasons, compositions which were not reported as invalid and which did not cause logs about unsynthesizable kernels created non-terminating behaviour of the simulator. While, on the test system, typically a full simulation finished after maximum two minutes, no termination of the program happened even after two hours. The bug could not be resolved.

4.1.3 Final Composition

Due to the encountered problems with the Metaheuristics composition optimization approach, the best found solution which actually worked remained the standard homogenous 16-PE composition. There was too less time and too many problems to solve to optimize the composition in a reasonable way, although the schedule length improvements by the Simulated Annealing showed around 30% shorter schedules (for the innermost loop) even after short runtimes.

4.2 Energy-Consumption Optimizations of CGRA Compositions

In order to find a good CGRA composition in terms of the total energy consumption, a compromise between the composition size (number of PEs and operators within the PEs) and the overall runtime must be found, i.e. the product of runtime and energy consumption by the circuit must be minimized.

For the best results regarding the energy consumption, a CGRA with four PEs, one of them with memory access, has been used. To our surprise, the runtime did not extend more than 1 million cycles over the performance-optimized composition in the most cases; Still, the energy consumed by the CGRA is significantly lower than with larger compositions. The choice of the small composition did hence result in a not much slower, but more energy efficient system.

5 Evaluation

5.1 Performance Optimization

5.2 Energy Consumption Minimization

6 Bibliography