Report of Rechnersysteme II CGRA-task

By Abdel al Amany and Mitja Stachowiak
Report - 26th of February 2018

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Introduction

The task was to program a GPS-Acquisition for the Amidar CGRA-simulator.

There is only one program version for minimum energy and shortest time, because for optimizations on a level, where a difference between energy and time efficiency can come up, was not nearly enough time.

Testing other CGRA-configurations then the default one wasn't possible as well.

Contents

1 Program structure

1.1 The class Complex

For working with complex numbers, a small class Complex was introduced. It supports all required complex arithmetical operations. Due to Java doesn't support operator overloading, a code like

```
complex3 = complex1 + complex2;
```

is expressed by

```
complex3 = complex1.add(complex2);
```

Because it is unclear, how efficient Java/Amidar can deal with such objects on the heap (There is no stack object in Java like the record/object in Pascal or the on-stack initialization of classes in C++!), the allocation of such complex numbers is avoided where ever possible. Therefore, there are operations like incBy or mulBy, which mutate the current object in-place.

1.2 The class ComplexVec

This class is responsible for storing complex vectors and processing the FFT-Transform on them.

1.3 The class Max

This class is responsible for searching and storing the maximum of rfd-vectors

1.4 The startAcquisition-function

Some optimizations made to the general structure of the acquisition were:

- to transform the code-vector C only one time at the beginning
- to search the maximum of R in-place without storing R (because it is not needed after searching the maximum)
- to modify the computation of gamma, for not dividing all vector elements by N

The computeXfd-function is separate just because nested loops cannot be synthesized:

```
private void computeXfd (int N, float fd) {
  for (int n = 0; n < N; n++) {
    Xfd.vec[n].takeFrom(Xin.vec[n]);
    Xfd.vec[n].mulBy(Complex.exp(new Complex(0, -2*(float)Math.PI*fd*n/fs)));
  }
}

public boolean startAcquisition() {
  int m = (int)((fmax-fmin)/fstep) + 1;
  int N = Xin.vec.length;
  // compute the transformation of code-vector
  C.fft();
  C.adj();
  // search maximum in R (compute R on the fly)
  Max max = new Max();
  for (int nf = 0; nf < m; nf++) {
    float fd = fmin + nf*fstep;
    // compute Xfd
    computeXfd(N, fd);
    // compute rfd*N
    Xfd.fft();
    ComplexVec rfd = Xfd;
    rfd.mul(C);
    rfd.invfft();
    // search for maximum
    max.searchIn(rfd, nf);
  }
  dopplerShift = (max.nf - (int)(m/2)) * (int)fstep;
  codeShift = max.i;
  // compute gamma = signal to noise
  double Pin = 0;
  for (int k = 0; k < N; k++) Pin += Xin.vec[k].abs2();
  double gamma = max.m / Pin; // the original formular would be max / (Pin/N),
                              but rfd is *N here.
  return gamma > sn_threshold;
}
```

2 Discrete Fourier Transform Algorithm

Usually, the most significant speedup comes from algorithmic improvement. The simple DFT-Algorithm consists just of a few lines:

```
public ComplexVec dft () {
  int N = this.vec.length;
  ComplexVec f = new ComplexVec(N);
  for (int k = 0; k < N; k++) {
    f.vec[k] = new Complex(0, 0);
    for (int i = 0; i < N; i++) {
      f.vec[k].incBy(Complex.exp(new Complex(0,
                                             -2*Math.PI*i*k/N)).mul(this.vec[i]));
    }
  }
  return f;
}
```

This algorithm was used in the first test application, which should improve correct understanding of the problem. The plan was, to make this version running in the Amidar-simulator first, before starting with the optimization.

But after hours of work, it turned out, that this algorithm is too slow for being executed in Amidar. The simulation always failed with some error, after a long run time. So the implementation of a FFT-algorithm was focused.

The problem is, that there isn't "the" FFT-algorithm. The most often used radix 2 algorithm can only handle input vectors of a length being a potency of two. First of all, this radix 2 algorithm was implemented for the next two-potency following the vector's length. The empty space behind the vector was filled with zeros. The frequency spectrum of the transformation gets blurred by this zero padding, and the result was wrong. An other idea is, to scale-up the vector by using bilinear interpolation. This has the same effect like the zero padding: Especially the high frequencies were blurred too much.

It is necessary, to use a variant of the FFT, which can handle arbitrary sized vectors. There are several approaches, none of them is easy, and working pseudo code is rare.

One simple solution is, to use the radix 2 FFT as long as the vector parts can be divided by two. The remaining vector parts then can be transformed using for example the simple DFT-algorithm. The given input vectors of length 400 can be divided by two for 4 times. So the remaining vectors for the DFT just got 25 samples.

The principle of this algorithm would be:

```
public void fft(int n, int offset) {
  if (n < 2) return;
  if (n % 2 == 0) {
    separate(n, offset);
    fft(n/2, offset);
    fft(n/2, offset+n/2);
    combine(n, offset);
  } else dft(n, offset);
}
```

Here, the FFT and the DFT transform the given vector in-place. This is also more efficient for the acquisition algorithm, because the original one is not longer needed after the transformation.

The execution of this algorithm would lead to a call tree like:

```
fft(n, 0);
  separate(n, 0);
  fft(n/2, 0);
    separate(n/2, 0);
    fft(n/4, 0);
      dft(n/4, 0);
    fft(n/4, n/4);
      dft(n/4, n/4);
    combine(n/2, 0);
  fft(n/2, n/2);
    separate(n/2, n/2);
    fft(n/4, n/2);
      dft(n/4, n/2);
    fft(n/4, 3*n/4);
      dft(n/4, 3*n/4);
    combine(n/2, n/2);
  combine(n, 0);
```

As Amidar is unable to synthesize recursive functions, the algorithm has to be re-written. It is also possible, to do all separations first, then all DFTs and finally all combines:

```
separate(n, 0);
separate(n/2, 0);
separate(n/2, n/2);
dft(n/4, 0);
dft(n/4, n/4);
dft(n/4, n/2);
dft(n/4, 3*n/4);
combine(n/2, 0);
combine(n/2, n/2);
combine(n, 0);
```

This is, what the used FFT-algorithm does:

```
public void fft () {
  int n = this.vec.length;
  int nMin = n;
  while (nMin % 2 == 0) nMin /= 2;
  // go town till nMin
  for (n = n/2; n >= nMin; n/=2)
    for (int offset = 0; offset < this.vec.length; offset += n + n)
      separate(n, offset);
  // do dft for all unsplittable parts
  if (nMin > 1)
    for (int offset = this.vec.length - nMin; offset >= 0; offset -= nMin)
      dft(nMin, offset);
  // go up till this.vec.length
  for (n = nMin; n < this.vec.length; n *= 2)
    for (int offset = 0; offset < this.vec.length; offset += n + n)
      merge(n, offset);
}
```

The final implementation is slightly different, especially puts the inner loops into separate functions as Amidar cannot synthesize nested loops.

This Algorithm was fast enough, to transform ~70% of the vectors, before the simulation failed. The DFT is still the most time consuming task here. One problem here is, that the computation of the difficult exponential function has to be done for each addition. The sequence of exponents in each DFT is equal for all DFT-calls. So a cache can be used to compute the exponents only one in each FFT. To share the cache between several FFT-calls would be the next step.

2.1 Optimizations for future work

There are a lot of significant optimizations left, which could be done to the transformation algorithm. For example:

- The computation of exponential functions not only occurs in the DFT, but also in the combine-phase. The usage of so called *twiddle factors* could reduce the computational effort here.
- The remaining vectors for DFT have a size of 25. This could be split into the prime factors 3*5. Implementing a radix-3-FFT could yield one more layer of FFT-mixed-radix.
- The sequence of separations may be replaced with a special one-step-separation more efficiently. There is a trick by swapping the bits in the indexes.
- Due to all transformations were done for the same vector size, the twiddle-factors and DFT-caches could be shared between all calls of the transformation.
- Functions having a return value must not contain loops.

3 Results

The simulation-result for `01_inputdata.dat`:

```
Ticks:                406468665
Bytecodes:            40641702
Energy consumption:   578655514.967
Execution Time:       335407 ms
```

3.1 Amidar Synthesis Bugs

Trying alternative implementations for working around the synthesizer errors was the most time-intensive part of work. Here some hints for working with Amidar:

- Nested loops are not possible. Put the code of inner loop into a separate function.
- Recursive functions are not possible.
- Java Garbage collector seems to have some issues with destroying objects kicked out of an array. Don't replace objects in an array, better replace the content of this objects (Done with Complex.takeFrom() in this program)
- Allocation of arrays in loops is not possible. The transformation helper array is therefore initialized in ComplexVec.initF() in this program.