Haskell FFT 11: Optimisation Part 1

January 10, 2014

Based on what we saw concerning the performance of our FFT code in the last article, we have a number of avenues of optimisation to explore. Now that we've got reasonable looking $O(N \log N)$ scaling for all input sizes, we're going to try to make the basic Danielson-Lanczos step part of the algorithm faster, since this will provide benefits for all input sizes. We can do this by looking at the performance for a single input length (we'll use $N = 256$). We'll follow the usual approach of profiling to find parts of the code to concentrate on, modifying the code, then looking at benchmarks to see if we've made a positive difference.

Once we've got some way with this "normal" kind of optimisation, there are some algorithm-specific things we can do: we can include more hard-coded base transforms for one thing, but we can also try to determine empirically what the best decomposition of our input vector length is – for example, for $N = 256$, we could decompose as $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$, using length-2 base transforms and seven Danielson-Lanczos steps to form the final transform, or as 16×16, using a length-16 base transform and a single Danielson-Lanczos step to form the final result.

Basic optimisation

The first thing we need to do is set up some basic benchmarking code to run our $N = 256$ test case, which we're going to use to look at optimisations of the basic Danielson-Lanczos steps in our algorithm. The code to do this benchmarking is more or less identical to the earlier benchmarking code, but we're also going to use this program:

```
module Main where

import Criterion.Main
import Data.Complex
import Data.Vector
import qualified Numeric.FFT as FFT

tstvec :: Int -> Vector (Complex Double)
tstvec sz = generate sz (\i -> let ii = fromIntegral i
                               in sin (2*pi*ii/1024) + sin (2*pi*ii/511))

main :: IO ()
main = run (nf (FFT.fftWith $ FFT.plan 256) $ tstvec 256) 1000
```

This does nothing but run the $N = 256$ FFT calculation 1000 times – we use the `run` and `nf` functions from the Criterion package to make sure our test function really does get invoked 1000 times. This allows us to get memory usage information without any of the overhead associated with benchmarking. We'll also use this code for profiling.

The first issue we want to look at is allocation. By running our benchmarking program as `./profile-256 +RTS -s`, we can get a report on total memory allocation and garbage collection statistics:

```
   5,945,672,488 bytes allocated in the heap
   7,590,713,336 bytes copied during GC
       2,011,440 bytes maximum residency (2002 sample(s))
          97,272 bytes maximum slop
               6 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      9232 colls,     0 par    2.94s    2.95s     0.0003s    0.0010s
  Gen  1      2002 colls,     0 par    1.98s    1.98s     0.0010s    0.0026s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    2.14s  (  2.14s elapsed)
  GC      time    4.92s  (  4.92s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    7.07s  (  7.07s elapsed)

  %GC     time      69.7%  (69.7% elapsed)

  Alloc rate    2,774,120,352 bytes per MUT second

  Productivity  30.3% of total user, 30.3% of total elapsed
```

For 1000 $N = 256$ FFTs, this seems like a lot of allocation. The ideal would be to allocate only the amount of space needed for the output vector. In this case, for a `Vector (Complex Double)` of length 256, this is 16,448 bytes, as reported by the `recursiveSize` function from the `ghc-datasize` package. So for 1000 samples, we'd hope to have only about 16,448,000 bytes of allocation – that's actually pretty unrealistic since we're almost certainly going to have to do *some* copying somewhere and there will be other overhead, but the numbers here give us a baseline to work from.

Profiling

To get some sort of idea where to focus our optimisation efforts, we need a bit more information, which we can obtain from building a profiling version of our library and test program. This can end up being a bit inconvenient because of the way that GHC and Cabal manage profiled libraries, since you need to have installed profiling versions of *all* the libraries in the transitive dependencies of your code in order to profile. The easiest way to deal with this issue is to use sandboxes. I use `hsenv` for this, but you could use the built-in sandboxes recently added to Cabal instead. Basically, you do something like this:

```
hsenv --name=profiling
source .hsenv_profiling/bin/activate
cabal install --only-dependencies -p --enable-executable-profiling
```

This will give you a `hsenv` sandbox with profiling versions of all dependent libraries installed.

To build our own library with profiling enabled, we add a `ghc-prof-options` line to the Cabal file, setting a couple of profiling options (`prof-all` and `caf-all`). Then, if we build our library with the `-p` option to Cabal, we'll get a profiling version of the library built with the appropriate options as well as the "vanilla" library.

A second minor problem is that we really want to profile our library code, not just the code in the test program that we're going to use. The simplest way to do this is to add the test program into our Cabal project. We add an `Executable` section for the test program and this then gets built when we build the library. This isn't very pretty, but it saves some messing around.

Once we have all this set up, we can build a profiling version of the library and test program (in the sandbox with the profiling versions of the dependencies installed) by doing:

```
cabal install -p --enable-executable-profiling
```

and we can then run our profiling test program as:

```
./dist_profiling/build/profile-256/profile-256 +RTS -p
```

The result of this is a file called `profile-256.prof` that contains information about the amount of run time and memory allocation ascribed to different “cost centres” in our code (based on the profiling options we put in the Cabal file, there’s one cost centre for each top level function or constant definition, plus some others).

Aside from some header information, the contents of the profile file come in two parts – a kind of flat “Top Ten” of cost centres ordered by time spent in them, and a hierarchical call graph breakdown of runtime and allocation for each cost centre. For our `profile-256` test program, the flat part of the profiling report looks like this:

```
COST CENTRE      MODULE              %time %alloc

dl.doone.mult    Numeric.FFT.Execute  26.2   23.8
dl.ds            Numeric.FFT.Execute  21.6   21.6
dl.d             Numeric.FFT.Execute  19.7   24.7
dl.doone.single  Numeric.FFT.Execute  14.0   13.9
dl.doone         Numeric.FFT.Execute   5.7    5.7
slicevecs        Numeric.FFT.Utils     2.7    3.3
dl               Numeric.FFT.Execute   2.1    1.9
special2.\       Numeric.FFT.Special   1.5    0.8
special2         Numeric.FFT.Special   1.4    2.4
slicevecs.\      Numeric.FFT.Utils     1.2    0.9
execute.multBase Numeric.FFT.Execute   1.1    0.6
```

and the first half or so of the hierarchical report like this:

```
                                                    individual    inherited
COST CENTRE             MODULE               no.  entries  %time %alloc  %time %alloc

MAIN                    MAIN                 235        0    0.3    0.0  100.0  100.0
 fftWith                Numeric.FFT          471        0    0.0    0.0   99.7  100.0
  execute               Numeric.FFT.Execute  473     1000    0.0    0.0   99.7  100.0
   execute.sign         Numeric.FFT.Execute  528     1000    0.0    0.0    0.0    0.0
   execute.bsize        Numeric.FFT.Execute  503     1000    0.0    0.0    0.0    0.0
    baseSize            Numeric.FFT.Types    504     1000    0.0    0.0    0.0    0.0
   execute.fullfft      Numeric.FFT.Execute  489     1000    0.4    0.1   99.7   99.9
    execute.recomb      Numeric.FFT.Execute  490        0    0.0    0.0   99.4   99.8
     dl                 Numeric.FFT.Execute  514     7000    2.1    1.9   93.5   95.3
      dl.ws             Numeric.FFT.Execute  527     7000    0.1    0.0    0.1    0.0
      dl.ns             Numeric.FFT.Execute  522     7000    0.0    0.0    0.0    0.0
      dl.doone          Numeric.FFT.Execute  516   127000    5.7    5.7   90.8   92.6
       dl.doone.vs      Numeric.FFT.Execute  523   127000    0.8    0.2    3.6    2.9
        slicevecs       Numeric.FFT.Utils    525   127000    1.9    2.3    2.8    2.7
         slicevecs.\    Numeric.FFT.Utils    526   254000    0.8    0.4    0.8    0.4
       dl.doone.single  Numeric.FFT.Execute  517   254000   14.0   13.9   81.5   84.0
        dl.doone.mult   Numeric.FFT.Execute  518   508000   26.2   23.8   67.5   70.1
         dl.d           Numeric.FFT.Execute  520   508000   19.7   24.7   41.3   46.3
          dl.ds         Numeric.FFT.Execute  521        0   21.6   21.6   21.6   21.6
         dl.ds          Numeric.FFT.Execute  519   508000    0.0    0.0    0.0    0.0
      slicevecs         Numeric.FFT.Utils    515     7000    0.4    0.6    0.5    0.8
       slicevecs.\      Numeric.FFT.Utils    524   127000    0.1    0.2    0.1    0.2
     execute.multBase   Numeric.FFT.Execute  502     1000    1.1    0.6    5.8    4.5
      applyBase         Numeric.FFT.Execute  512   128000    0.6    0.1    4.1    3.2
       special2         Numeric.FFT.Special  513   128000    1.4    2.4    3.5    3.1
        special2.b      Numeric.FFT.Special  536   128000    0.4    0.0    0.4    0.0
        special2.a      Numeric.FFT.Special  534   128000    0.3    0.0    0.3    0.0
        special2.\      Numeric.FFT.Special  533   256000    1.5    0.8    1.5    0.8
      slicevecs         Numeric.FFT.Utils    511     1000    0.4    0.5    0.6    0.7
       slicevecs.\      Numeric.FFT.Utils    535   128000    0.3    0.2    0.3    0.2
   execute.recomb       Numeric.FFT.Execute  488     1000    0.0    0.0    0.0    0.0
   execute.rescale      Numeric.FFT.Execute  476     1000    0.0    0.0    0.0    0.0
   execute.(...)        Numeric.FFT.Execute  475     1000    0.0    0.0    0.0    0.0
   execute.n            Numeric.FFT.Execute  474     1000    0.0    0.0    0.0    0.0
```

It’s clear from this that the vast majority of the allocation (89.7%) and time (87.2%) is spent in the Danielson-Lanczos step function `dl` in `Numeric.FFT.Execute`. Here’s the code for this function from

version `pre-release-1` of the repository:

```
-- | Single Danielson-Lanczos step: process all duplicates and
-- concatenate into a single vector.
dl :: WMap -> Int -> (Int, Int) -> VCD -> VCD
dl wmap sign (wfac, split) h = concatMap doone $ slicevecs wfac h
  where
    -- Size of each diagonal sub-matrix.
    ns = wfac `div` split

    -- Roots of unity for the size of diagonal matrix we need.
    ws = wmap IM.! (sign * wfac)

    -- Basic diagonal entries for a given column.
    ds c = map ((ws !) . (`mod` wfac) . (c *)) $ enumFromN 0 ns

    -- Twiddled diagonal entries in row r, column c (both
    -- zero-indexed), where each row and column if a wfac x wfac
    -- matrix.
    d r c = map ((ws ! ((ns * r * c) `mod` wfac)) *) (ds c)

    -- Process one duplicate by processing all rows and concatenating
    -- the results into a single vector.
    doone v = concatMap single $ enumFromN 0 split
      where vs :: VVCD
            vs = slicevecs ns v
            -- Multiply a single block by its appropriate diagonal
            -- elements.
            mult :: Int -> Int -> VCD
            mult r c = zipWith (*) (d r c) (vs!c)
            -- Multiply all blocks by the corresponding diagonal
            -- elements in a single row.
            single :: Int -> VCD
            single r = foldl1' (zipWith (+)) $ map (mult r) $ enumFromN 0 split
```

In particular, the `mult`, `ds`, `d` and `single` local functions defined within `dl` take most of the time, and are responsible for a most of the allocation. It's pretty clear what's happening here: although the `vector` package has rewrite rules to perform fusion to eliminate many intermediate vectors in chains of calls to vector transformation functions, we're still ending up allocating a lot of temporary intermediate values. Starting from the top of the `dl` function:

- We generate a list of slices of the input vector by calling the `slicevecs` utility function – this doesn't do much allocation and doesn't take much time because slices of vectors are handled just by recording an offset into the immutable array defining the vector to be sliced.
- We do a `concatMap` of the local `doone` function across this list of slices – besides any allocation that `doone` does, this will do more allocation for the final output vector. Moreover, `concatMap` cannot be fused.
- The `doone` function also calls `concatMap` (so again preventing fusion), evaluating the local function `single` once for each of the sub-vectors to be processed here.
- Finally, both `single` and the function `mult` that it calls allocate new vectors based on combinations of input vector elements and powers of the appropriate ω_N.

So, what can we do about this? There's no reason why we need to perform all this allocation. It seems as though it ought to be possible to either perform the Danielson-Lanczos steps in place on a single mutable vector or (more simply) to use two mutable vectors in a "ping-pong" fashion. We'll start with the latter approach, since it means we don't need to worry about the exact order in which the Danielson-Lanczos steps access input vector elements. If we do things this way, we should be able to get away with just two vector allocations, one for the initial permutation of the input vector elements, and one for the other "working" vector. Once we've finished the composition of the Danielson-Lanczos steps, we can freeze the final result as a return value (if we use `unsafeFreeze`, this is an $O(1)$ operation). Another thing we can do is to use *unboxed* vectors,

which both reduces the amount of allocation needed and speeds up access to vector elements. Below, I'll describe a little how mutable and unboxed vectors work, then I'll show the changes to our FFT algorithm needed to exploit these things. The code changes I'm going to describe here are included in the `pre-release-2` version of the `arb-fft` package on GitHub.

Mutable vectors

A "normal" `Vector`, as implemented by the `Data.Vector` class, is *immutable*. This means that you allocate it once, setting its values somehow, and thereafter it always has the same value. Calculations on immmutable vectors are done by functions with types that are something like `... -> Vector a -> Vector a` that create *new* vectors from old, allocating new storage each time. This sounds like it would be horribly inefficient, but the `vector` package uses GHC rewrite rules to implement *vector fusion*. This is a mechanism that allows intermediate vectors generated by chains of calls to vector transformation functions to be elided and for very efficient code to be produced.

In our case, the pattern of access to the input vectors to the `execute` and `dl` functions is not simple, and it's hard to see how we might structure things so that creation of intermediate vectors could be avoided. Instead, we can fall back on mutable vectors. A *mutable vector*, as defined in `Data.Vector.Mutable`, is a linear collection of elements providing a much reduced API compared to `Data.Vector`'s immutable vectors: you can create and initialise mutable vectors, read and write individual values, and that's about it. There are functions to convert between immutable and mutable vectors (`thaw` turns an immutable vector into a mutable vector, and `freeze` goes the other way).

Now, the bad news. As soon as we introduce mutability, our code is going to become less readable and harder to reason about. This is pretty much unavoidable, since we're going to be explicitly controlling the order of evaluation to control the sequence of mutations we perform on our working vectors. The `vector` package provides a clean monadic interface for doing this, but it's still going to be messier than pure functional code. This is something that often happens when we come to optimise Haskell code: in order to control allocation in the guts of an algorithm, we quite often need to switch over to writing mutating code that's harder to understand[1]. However, this is often less of a problem than you might think, because you almost never sit down to write that mutating code from nothing. It's almost invariably a good idea to write pure code to start with, code that's easier to understand and easier to reason about. Then you profile, figure out where in the code the slowdowns are, and attack those. You can use your pure code both as a template to guide the development of the mutating code, and as a comparison for testing the mutating code.

I worked this way for all of the changes I'm going to describe in this section: start from a known working code (tested using the QuickCheck tests I described above), make some changes, get the changes to compile, and retest. Writing code with mutation means that the compiler has fewer opportunities to protect you from yourself: the (slightly silly) Haskell adage that "if it compiles, it works" doesn't apply so much here, so it's important to test as you go.

Since we have to sequence modifications to mutable vectors explicitly, we need a monadic interface to control this sequencing. The `vector` package provides two options – you can either do everything in the `IO` monad, or you can use an `ST` monad instance. We'll take the second option, since this allows us to write functions that still have pure interfaces – the ST monad encapsulates the mutation and doesn't allow any of that impurity to escape into the surrounding code. If you use the `IO` monad, of course, all bets are off and you can do whatever you like, which makes things even harder to reason about.

Given these imports and type synonyms:

```
import Data.Vector
import qualified Data.Vector.Mutable as MV
import Data.Complex
```

```
type VCD = Vector (Complex Double)
type MVCD s = MV.MVector s (Complex Double)
type VVCD = Vector VCD
type VVVCD = Vector VVCD
```

a version of the `dl` function using mutable vectors will have a type signature something like

```
dl :: WMap -> Int -> (Int, Int, VVVCD, VVVCD) -> MVCD s -> MVCD s -> ST s ()
dl wmap sign (wfac, split, dmatp, dmatm) mhin mhout = ...
```

Here, the first two arguments are the collected powers of ω_N and the direction of the transform and the four-element tuple gives the sizing information for the Danielson-Lanczos step and the entries in the diagonal matrices in the "$I + D$" matrix[2]. The interesting types are those of `mhin`, `mhout` and the return type. Both `mhin` and `mhout` have type `MVCD s`, or `MV.Vector s (Complex Double)` showing them to be mutable vectors of complex values. The return type of `dl` is `ST s ()`, showing that it is a monadic computation in the `ST` monad that doesn't return a value. The type variable `s` that appears in both the types for `mhin`, `mhout` and the return type is a kind of tag that prevents internal state from leaking from instances of the `ST` monad. This type variable is never instantiated, but it serves to distinguish different invocations of `runST` (which is used to evaluate `ST s a` computations). This is the mechanism that allows the `ST` monad to cleanly encapsulate stateful computation while maintaining a pure interface.

When I made the changes need to use mutable vectors in the FFT algorithm, I started by just trying to rewrite the `dl` function to use mutable vectors. This allowed me to get some of the types right, to figure out that I needed to be able to slice up mutable vectors (the `slicemvecs` function) and to get some sort of feeling for how the `execute` driver function would need to interface with `dl` if everything was rewritten to use mutable vectors from the top. Once a mutable vector-based version of `dl` was working, I moved the allocation of vectors into `execute` function and set things up to bounce back and forth between just two vector buffers.

Here's part of the new `dl` function to give an impression of what code using mutable vectors looks like (if you're really interested in how this works, I'd recommend browsing through the `pre-release-2` version of the code on GitHub, although what you'll see there is a little different to the examples I'm showing here as it's a bit further along in the optimisation process):

```
1  -- Multiply a single block by its appropriate diagonal
2  -- elements and accumulate the result.
3  mult :: VMVCD s -> MVCD s -> Int -> Bool -> Int -> ST s ()
4  mult vins vo r first c = do
5    let vi = vins V.! c
6        dvals = d r c
7    forM_ (enumFromN 0 ns) $ \i -> do
8      xi <- MV.read vi i
9      xo <- if first then return 0 else MV.read vo i
10     MV.write vo i (xo + xi * dvals ! i)
```

This code performs a single multiplication of blocks of the current input vector by the appropriate diagonal matrix elements in the Danielson-Lanczos $I + D$ matrix. The return type of this function is `ST s ()`, i.e. a computation in the `ST` monad tagged by a type `s` with no return value. Take a look at lines 7-10 in this listing – these are the lines that do the actual computation. We loop over the number of entries we need to process (using the call to `forM_` in line 7). Values are read from an input vector `vi` (line 8) and accumulated into the output vector `vo` (lines 9 and 10) using the `read` and `write` functions from `Data.Vector.Mutable`. This looks (apart from syntax) exactly like code you might write in C to perform a similar task!

We certainly wouldn't want to write code like this all the time, but here, since we're using it down in the depths of our algorithm *and* we have clean pure code that we've already written that we can test it against if we need to, it's not such a problem.

Here is the most important part of the new `execute` function:

```
1  -- Apply Danielson-Lanczos steps and base transform to digit
2  -- reversal ordered input vector.
3  fullfft = runST $ do
4    mhin <- thaw $ case perm of
5          Nothing -> h
6          Just p -> backpermute h p
7    mhtmp <- MV.replicate n 0
8    multBase mhin mhtmp
9    mhr <- newSTRef (mhtmp, mhin)
10   V.forM_ dlinfo $ \dlstep -> do
11     (mh0, mh1) <- readSTRef mhr
12     dl wmap sign dlstep mh0 mh1
13     writeSTRef mhr (mh1, mh0)
14   mhs <- readSTRef mhr
15   unsafeFreeze $ fst mhs
16
17 -- Multiple base transform application for "bottom" of algorithm.
18 multBase :: MVCD s -> MVCD s -> ST s ()
19 multBase xmin xmout =
20   V.zipWithM_ (applyBase wmap base sign)
21               (slicemvecs bsize xmin) (slicemvecs bsize xmout)
```

These illustrate a couple more features of working with mutable vectors in the `ST` monad. First, in the `multBase` function (lines 18-21), which applies the "base transform" at the "bottom" of the FFT algorithm to subvectors of the permuted input vector, we see how we can use monadic versions of common list and vector manipulation functions (here `zipWithM_`, a monadic, void return value version of `zipWith`) to compose simpler functions. The code here splits each of the `xmin` and `xmout` mutable vectors into subvectors (using the `slicemvecs` function in `Numeric.FFT.Utils`), then uses a partial application of `applyBase` to zip the vectors of input and output subvectors together. The result is that `applyBase` is applied to each of the input and output subvector pairs in turn.

The calculation of `fullfft` (lines 3-15) demonstrates a couple more techniques:

- We see (line 3) how the overall `ST` monad computation is evaluated by calling `runST`: `fullfft` is a *pure* value, and the `ST` monad allows us to strictly bound the part of our code where order-dependent stateful computations occur.
- We see (lines 4-6) how the `thaw` function is used to convert an immutable input vector `h` into a mutable vector `mhin` for further processing.
- The final result of the computation is converted back to an immutable vector using the `unsafeFreeze` function (line 15). Using `unsafeFreeze` means that we promise that no further access will be made to the mutable version of our vector – this means that the already allocated storage for the mutable vector can be reused for its immutable counterpart, making `unsafeFreeze` an $O(1)$ operation. Here, we can see immediately that no further use is made of the mutable version of the vector since the result of the call to `unsafeFreeze` is returned as the overall result of the monadic computation we are in.
- The "ping pong" between two mutable vectors used to evaluate the Danielson-Lanczos steps is handled by using a mutable reference (created by a call to `newSTRef` at line 9). At each Danielson-Lanczos step, we read this references (line 11), call `dl` (line 12), passing one of the vectors as input (penultimate argument) and one as output (last argument), then write the pair of vectors back to the mutable reference in the opposite order (line 13), achieving the desired alternation between vector buffers. After each Danielson-Lanczos step, the result so far is held in the first vector of the pair, so this is what's extracted for the final return value in line 15.

We apply similar speedups to the Rader's algorithm code for prime-length FFTs, although I won't show that here since it's not all that interesting. (We also fold the index permutation involved in the first step of Rader's algorithm into the main input vector index permutation, since there's little point in permuting the input then permuting it again!) In fact, a more important issue for speeding up Rader's algorithm is to get transforms of lengths that are powers of two as fast as

possible (recall that we pad the input to a power of two length to make the convolution in the Rader transform efficient). We'll address this more directly in the next section.

Unboxing

Until now, all the vectors we've been dealing with have been *boxed*. This means that there's an extra level of indirection in each element in the vector allowing values of composite data types to be stored. However, all of our vectors are just plain `Vector (Complex Double)`, so we might hope to be able to get away without that extra level of indirection.

In fact, apart from one small issue, getting this to work is very easy: just replace imports of `Data.Vector` with `Data.Vector.Unboxed` (and the equivalent for mutable vectors). This simplicity is the result of some very clever stuff: `Data.Vector.Unboxed` is one of those pieces of the Haskell ecosystem that make you sit back and say "Wow!". There's some very smart type system stuff going on that allows the unboxed vector implementation to select an efficient specialised representation for every different element type. For our purposes, in particular, although `Complex Double` is a composite type (and so not, on the face of it, a good candidate for unboxing), `Data.Vector.Unboxed` has an instance of its `Unbox` type class for `Complex a` that leads to efficient memory layout and access of vectors of complex values. It's really very neat, and is one of those things where the Haskell type system (and the implementor's clever use of it!) allows you to write code at a high level while still maintaining an efficient low-level representation internally.

The "small issue" mentioned above is just that we sometimes want to have `Vector`s of `Vector`s, and a `Vector` is not itself unboxable (because you can't tell how much memory it occupies in advance without knowing the length of the vector). This just means that you need to use the "normal" boxed vectors in this case, by importing `Data.Vector` qualified, as shown here (compare with the other listing of type synonyms above, which is without unboxing):

```
import Data.Vector.Unboxed
import qualified Data.Vector as V
import qualified Data.Vector.Unboxed.Mutable as MV
import Data.Complex

type VCD = Vector (Complex Double)
type MVCD s = MV.MVector s (Complex Double)
type VVCD = V.Vector VCD
type VVVCD = V.Vector VVCD
type VMVCD a = V.Vector (MVCD a)
type VI = Vector Int
```

More hard-coded base transforms for prime lengths

Although they're very tedious to write, the specialised transforms for small prime input lengths make a *big* difference to performance. I've converted "codelets" for all prime lengths up to 13 from the FFTW code.

The more I think about it, the more attractive is the idea of writing something comparable to FFTW's `genfft` in order to generate these "straight line" transforms programmatically. There are a number of reasons for this:

1. Copying the FFTW codelets by hand is not a very classy thing to do. It's also very tedious involving boring editing (Emacs keyboard macros help, but it's still dull).

2. If we have a programmatic way of generating straight line codelets, we can use it for other input length sizes. Most of FFTW is implemented in C, which isn't an ideal language for the kind of algebraic metaprogramming that's needed for this task, so the FFTW folks wrote `genfft` in OCaml. In Haskell, we can do metaprogramming with Template Haskell, i.e. in Haskell itself, which is much nicer and means that, if we know our input length at compile time, we could provide a Template Haskell function to generate the optimum FFT

decomposition with any necessary straight line code at the bottom generated automatically, allowing us to handle awkward prime input length factors efficiently (within reason, of course – if the straight line code gets too big, it won't fit in the machine's cache any more and a lot of the benefit of doing this would be lost).

3. We've only been specialising the primitive transforms at the bottom of the FFT decomposition. Let's call those specialised bits of straight line code "bottomlets". It's also possible to write specialised "twiddlets" to do the Danielson-Lanczos $I + D$ matrix multiplication – there are often algebraic optimisations that could be done here to make things quicker. If we had a Template Haskell `genfft` equivalent, we could extend it to generate these "twiddlets" as well as the "bottomlets", giving us the capability to produce optimal code for all steps in the FFT decomposition.

This seems like it would be an interesting task, although probably quite a big piece of work. I'm going to leave it to one side for the moment, but will probably have a go at it at some point in the future.

Strictness/laziness

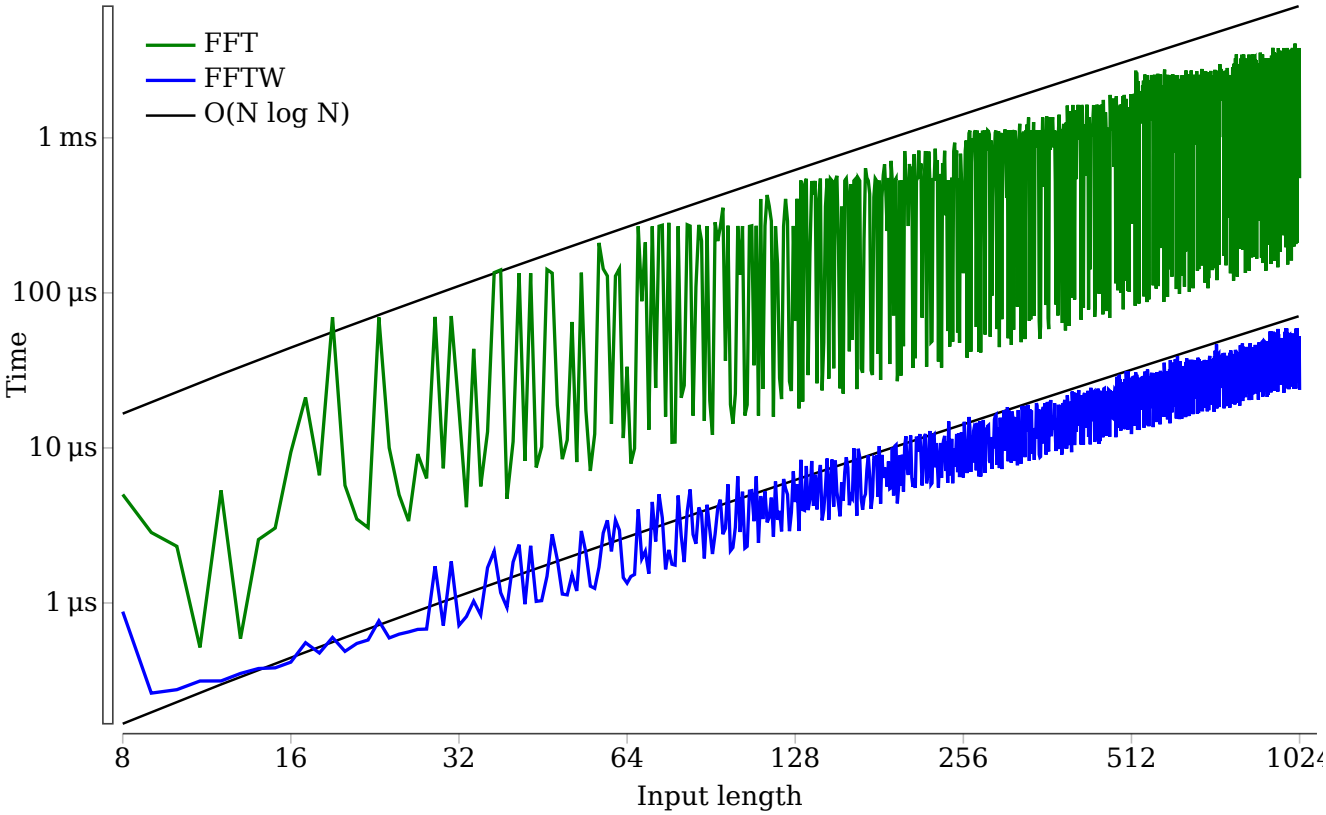
One topic we've not looked at which usually turns up in discussions of Haskell optimisation is strictness (or, if you prefer, laziness). We can tell just from thinking about the basic DFT algorithm that any FFT computation is strict in its input vector, and strict in all of the elements of that input vector. There shouldn't be any laziness going on anywhere once we get into executing the FFT. (It matters less at the planning stage, since we'll only do that once.)

To some extent, the behaviour of the `Vector` types helps us here. The data structures used to implement the various flavours of `Vector` are all strict in the arrays that are used to implement them, and since we're now using unboxed vectors, the arrays inside the vectors are necessarily strict in their contents.

I'm actually going to sidestep this issue for a little while, since we'll be coming back to this kind of question when we do a final "kicking the tyres" examination of the whole FFT code after we've done the empirical optimisation described in the next section.

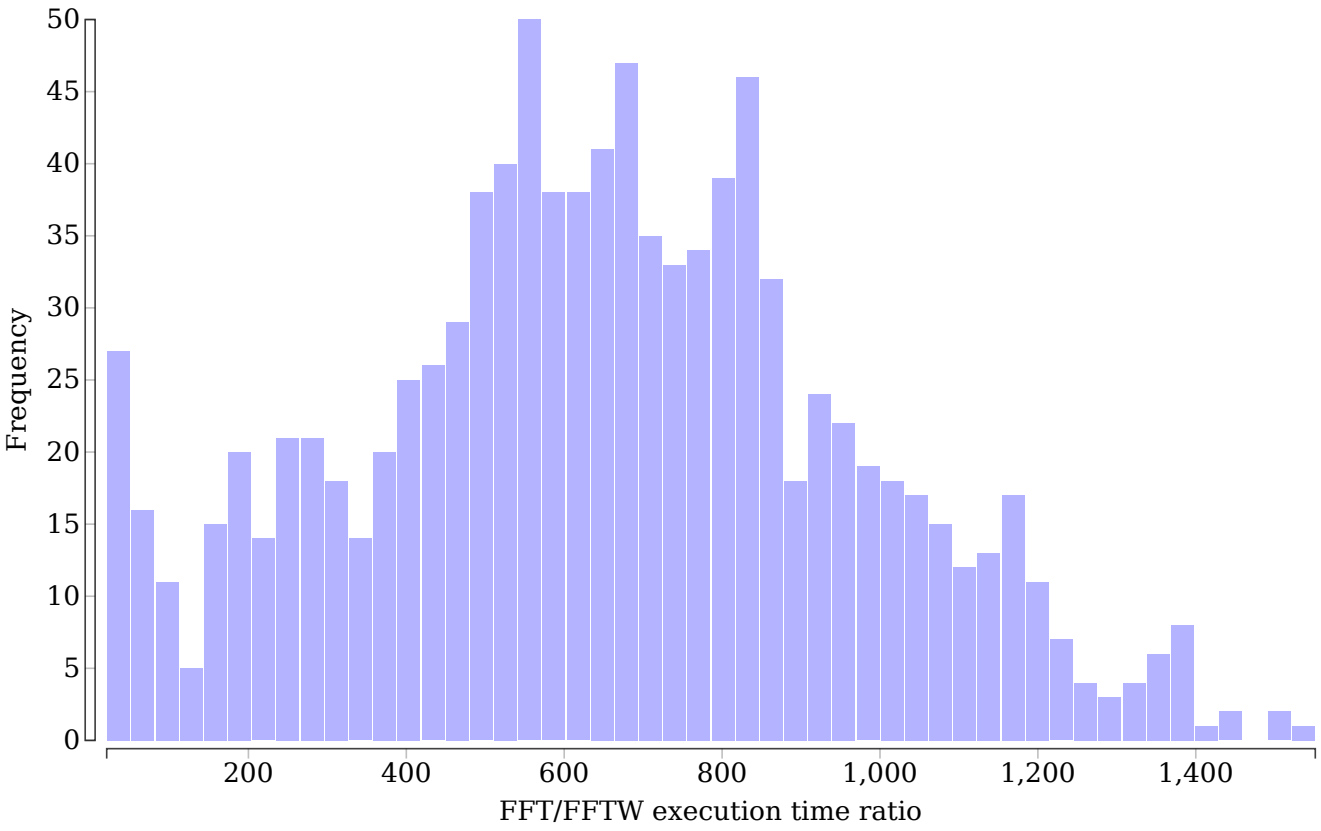
Profiling and benchmarking check

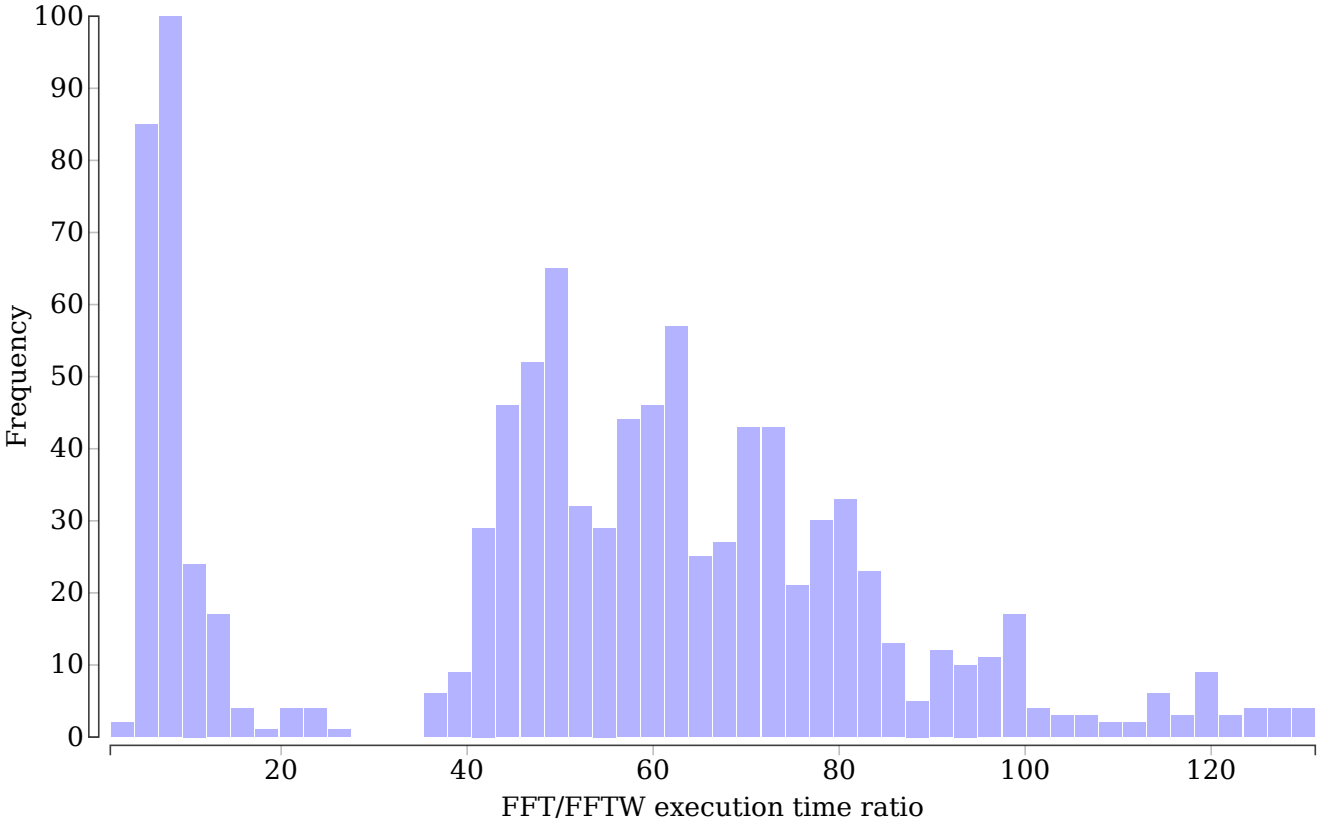
Let's see where we've got to in terms of performance. Here's a view of the performance of the current optimised code (which is tagged `pre-release-2` in the GitHub repository) in the same format as we've used previously:

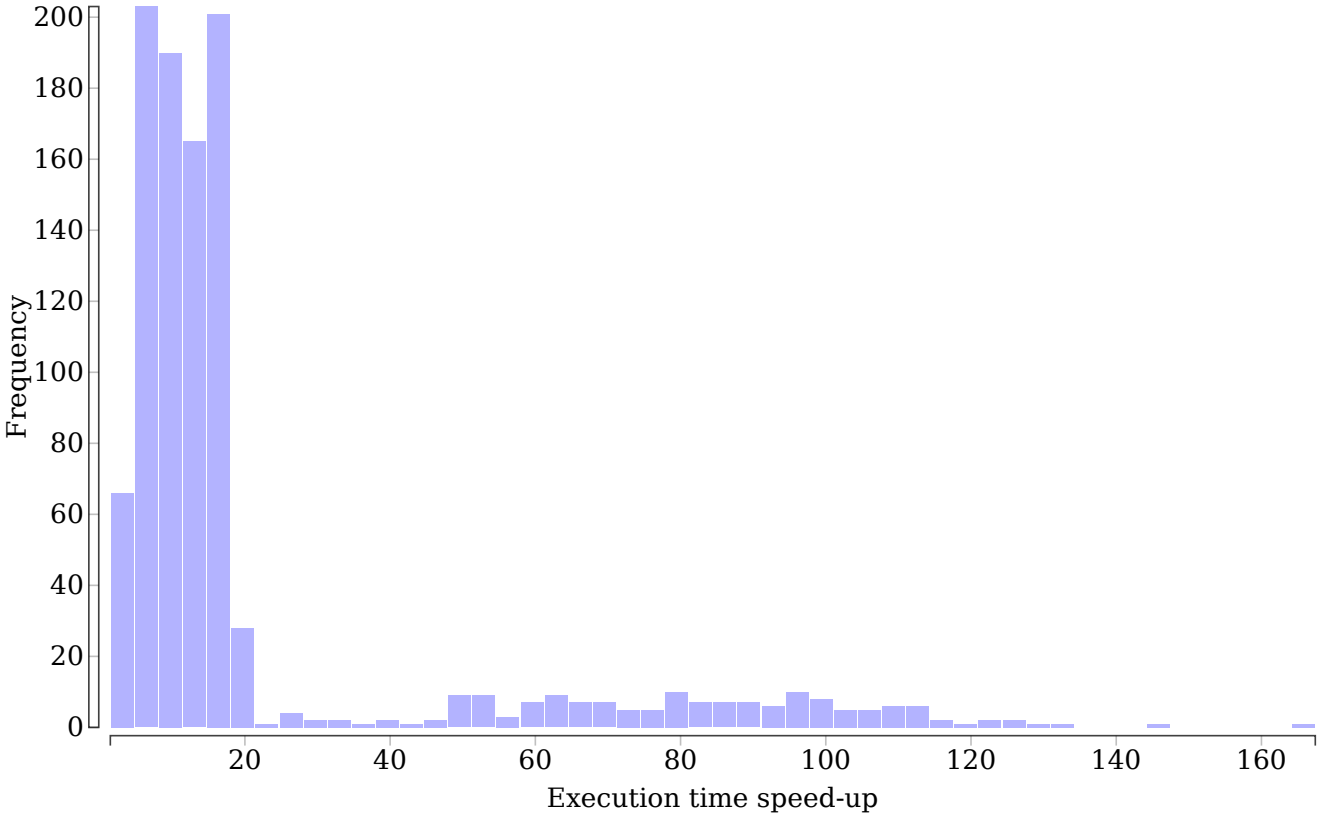


The overall performance and scaling behaviour of our code isn't all that different to what we saw before (last article), but everything is faster! We can get some idea of how much faster from the plot stack below. Each of the three panels in this plot is a histogram of execution time ratios for all problem sizes in the range $8 \leq N \leq 1024$. The tab labelled "pre-release-1" shows the execution time ratios between the `pre-release-1` version of our code and FFTW – larger values represent slower execution relative to FFTW. The tab labelled "pre-release-2" shows the same ratios for the `pre-release-2` version of our code – much better! – and finally, the tab labelled "Speed-up" shows execution time ratios between the `pre-release-1` and `pre-release-2` versions of our code, which gives a measure of the speed-up we've achieved with our optimisations.

- pre-release-1
- pre-release-2
- Speed-up


Frequency
0
5
10
15
20
25
30
35
40
45
50
200
400
600
800
1,000
1,200
1,400
FFT/FFTW execution time ratio


Frequency
0
10
20
30
40
50
60
70
80
90
100
20
40
60
80
100
120
FFT/FFTW execution time ratio



Concentrating on the “Speed-up” tab showing the speed-up between the unoptimised and optimised versions of our code, we can see that for most input vector lengths the optimised code is around 10-20 times faster. For some input vector lengths though, we get speed-up factors of 50-100 times compared to the unoptimised code. This is definitely good news. A lot of this improvement in performance comes from using specialised straight line code for small prime length transforms. In the next article, we’ll think about how to exploit this kind of specialised code for other input sizes (especially powers of two, which will also give us a speed-up for the convolution in Rader’s algorithm).

Looking at the “pre-release-2” tab, we appear to be getting to within a factor of 10 of the performance of FFTW for some input vector lengths, which is really not bad, considering the limited amount of optimisation that we’ve done, and my own relative lack of experience with optimising this kind of Haskell code!

---

1. Note that this is advice aimed at mortals. There are some people – I’m not one of them – who are skilled enough Haskell programmers that they can rewrite this sort of code to be more efficient without needing to drop into the kind of stateful computation using mutation that we’re going to use. For the moment, let’s mutate!↩

2. I moved the calculation of these out into the FFT planning stage on the basis of profiling information I collected, after I’d switched over to using mutable vectors, that indicated that a lot of time was being spent recalculating these diagonal matrix entries on each call to `dl`.↩

data-analysis haskell

 Powered by Hakyll