Haskell FFT 13: Optimisation Part 3

January 26, 2014

In this article, we’ll finish up with optimisation, tidying up a couple of little things and look at some “final” benchmarking results. The code described here is the `pre-release-4` version in the GitHub repository. I’m not going to show any code snippets in this section, for two reasons: first, if you’ve been following along, you should be familiar enough with how things work to find the sections of interest; and second, some of the optimisation means that the code is getting a little more unwieldy, and it’s getting harder to display what’s going on in small snippets. Look in the GitHub repository if you want to see the details.

Rader’s algorithm improvements

There are two things we can do to make our implementation of Rader’s algorithm go a bit more quickly.

First, so far we’ve always calculated the convolution involved in Rader’s algorithm using a transform for the next power-of-two size greater than the actual convolution size. This is based on the observation that powers of two usually have a pretty efficient FFT plan. It also allows us to avoid recursive calls to Rader’s algorithm if we end up with a convolution transform size that has large prime factors. However, obviously there’s no reason to expect that the padded convolution will be faster in *all* cases. Sometimes the unpadded convolution may be quicker because of specialised base transforms (e.g. an $N = 22$ FFT takes about 3 μs while the power of two needed for the corresponding padded convolution, $N = 64$, takes about 33 μs). As in other cases like this, the best way to deal with the question is to test empirically which is quicker.

Unfortunately, in order to do this, we’re going to have to change the types of some of the planning functions to run in the `IO` monad. This is because we’re going to have to recursively benchmark different plans for the convolution transforms while we’re deciding just how to do the Rader’s algorithm base transform. This isn’t all that messy when you get down to it, but it made me decide that the default planning API should also be monadic, with signature `plan :: Int -> IO Plan`, since there may be reads and writes of wisdom files going on during the planning process. To make it easy to decide which convolution approach to take, we also record the execution times of the selected transform plans in the wisdom files, so once a plan has been generated for a given input length, we can just read the timing information from the wisdom file.

So, in principle, what we do is quite simple. When we’re constructing a Rader base transform, we just determine whether the unpadded or padded transform for the convolution is quicker. We save the relevant FFT-transformed “b” sequence for the convolution and record the transform size that we’re using for the convolution. We can then set the code up so that everything works the same for the padded and unpadded case without too much trouble.

The second thing we can do is to try to avoid conversions between mutable and immutable vectors in the `RaderBase` branch of `applyBase`. Doing this effectively ends up being slightly messy, because what we really need is a specialised version of `execute` that works on mutable vectors. I call this `executeM` and it has signature

```
executeM :: Plan -> Direction -> MVCD s -> MVCD s -> ST s ()
```

taking a plan, a transform direction, input and output mutable vectors, and running in the `ST` monad. This allows us to avoid a couple of vector allocations in the convolution step. The original convolution code in `applyBase` looked like this:

```
let conv = execute cplan Inverse $
             zipWith (*) (execute cplan Forward apadfr)
                         (if sign == 1 then bfwd else binv)
```

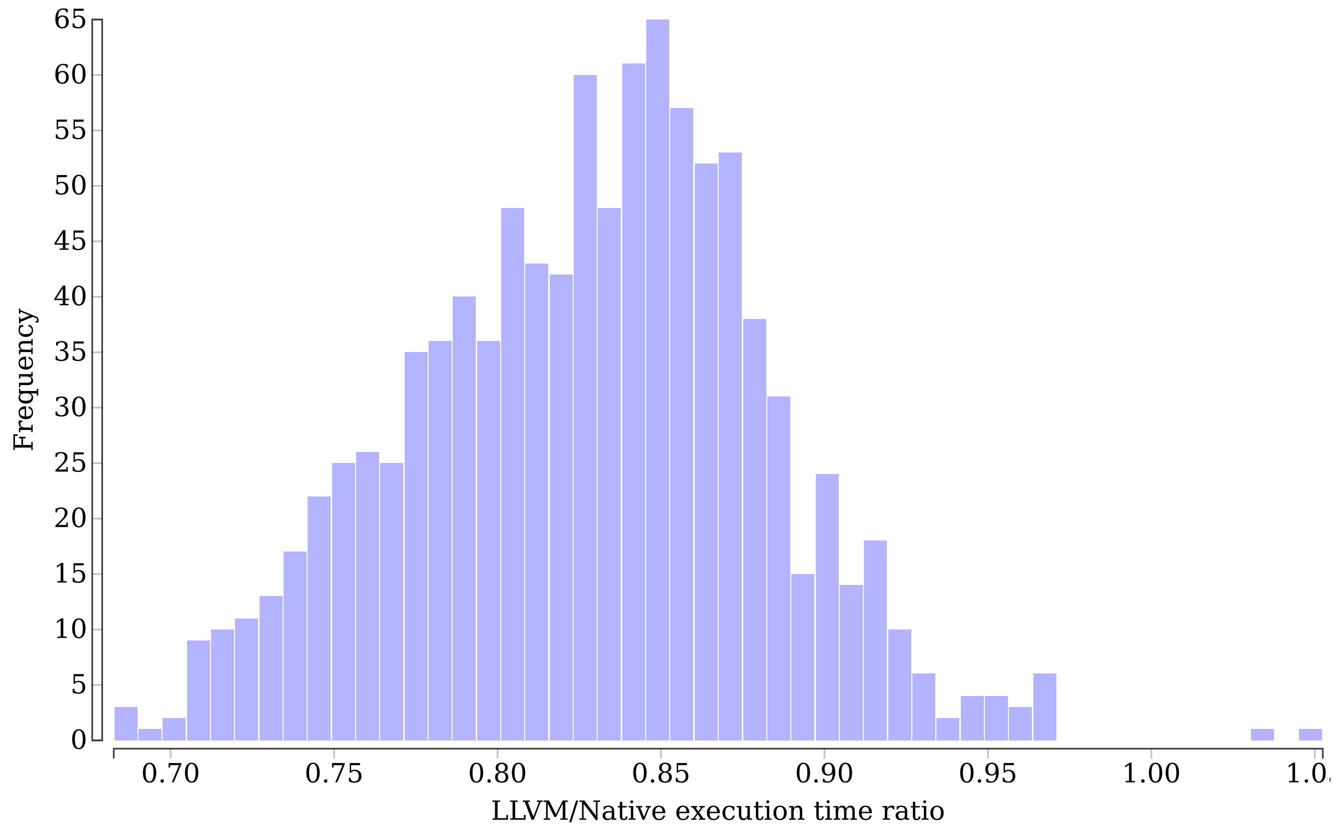
which did at least two vector allocations, one for each call to `execute`, and possibly a third associated with the call to `zipWith` (though that may possibly get fused away). Instead of this, we can use mutable vectors and the new `executeM` to avoid some of these allocations (in particular, the multiplication involved in the convolution can be done in place), giving a slight speed-up in some cases.

There are a couple of other possibilities that we might explore, but we're getting to a point of diminishing returns with this so we'll leave it for now. (One particular possibility is to experiment with padding to other sizes instead of a power of two: any highly composite size that takes advantage of specialised base transforms would be good, and the best option could be determined empirically.)

Compiler flags

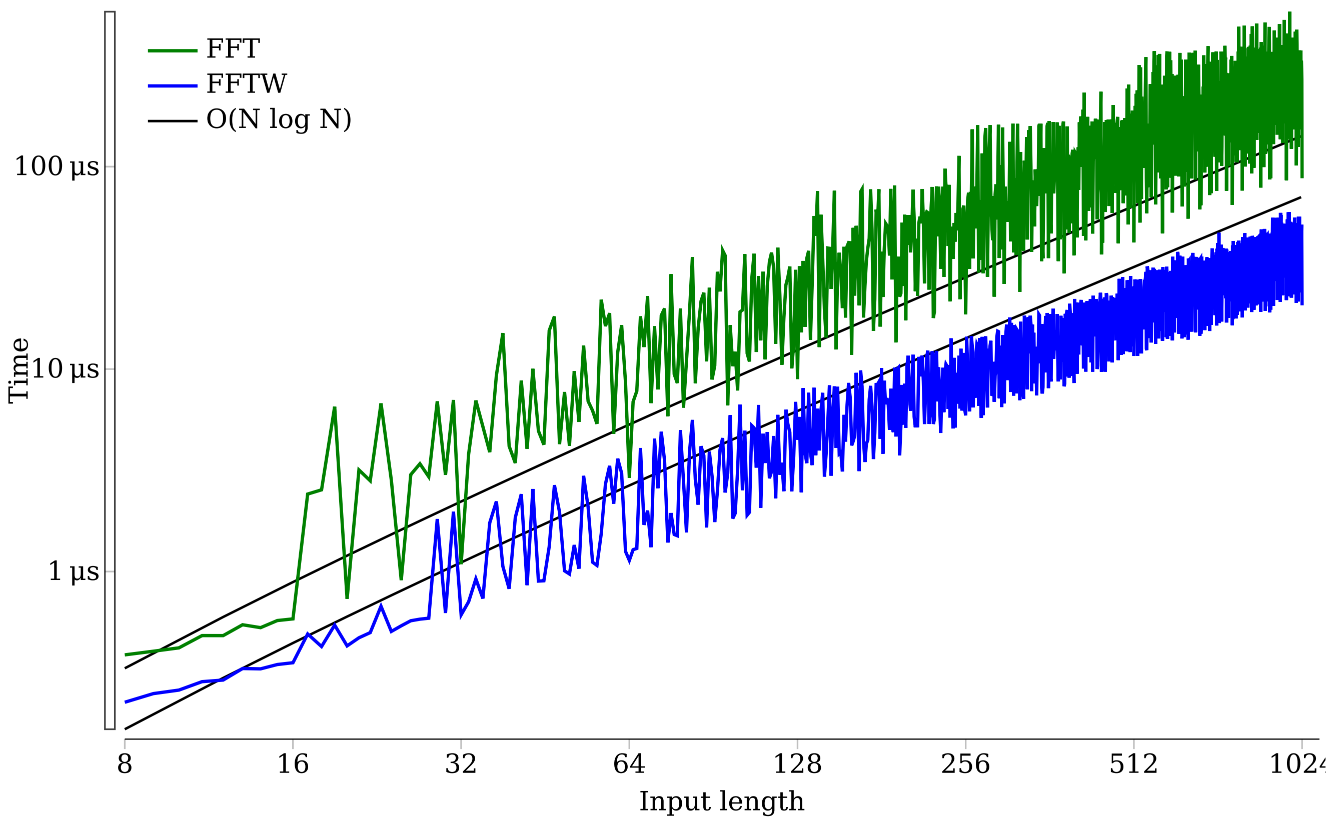
All the way through, we've been using GHC's `-O2` flag to enable the full suite of "non-dangerous" optimisations. Now, without getting into lots of messing around with specific GHC optimisation flags (of which there are lots), the best option for getting some more "free" performance improvements is to try a different code generator. Specifically, so far we've been using GHC's native code generator, but there's another LLVM-based generator that tends to give good performance for numerically-intensive code that makes a lot of use of `Vector`s. Sounds perfect for us.

We can enable the LLVM code generator simply by adding the `-fllvm` option to the `ghc-options` field in our Cabal file. Benchmarking with this version of the code reveals that we get, on average a 17% speed-up compared to the native code generator (for code version `pre-release-4`). Here's a histogram of the speed-ups of the LLVM-generated code compared to that from the native code generator for input vector lengths from 8 to 1024:



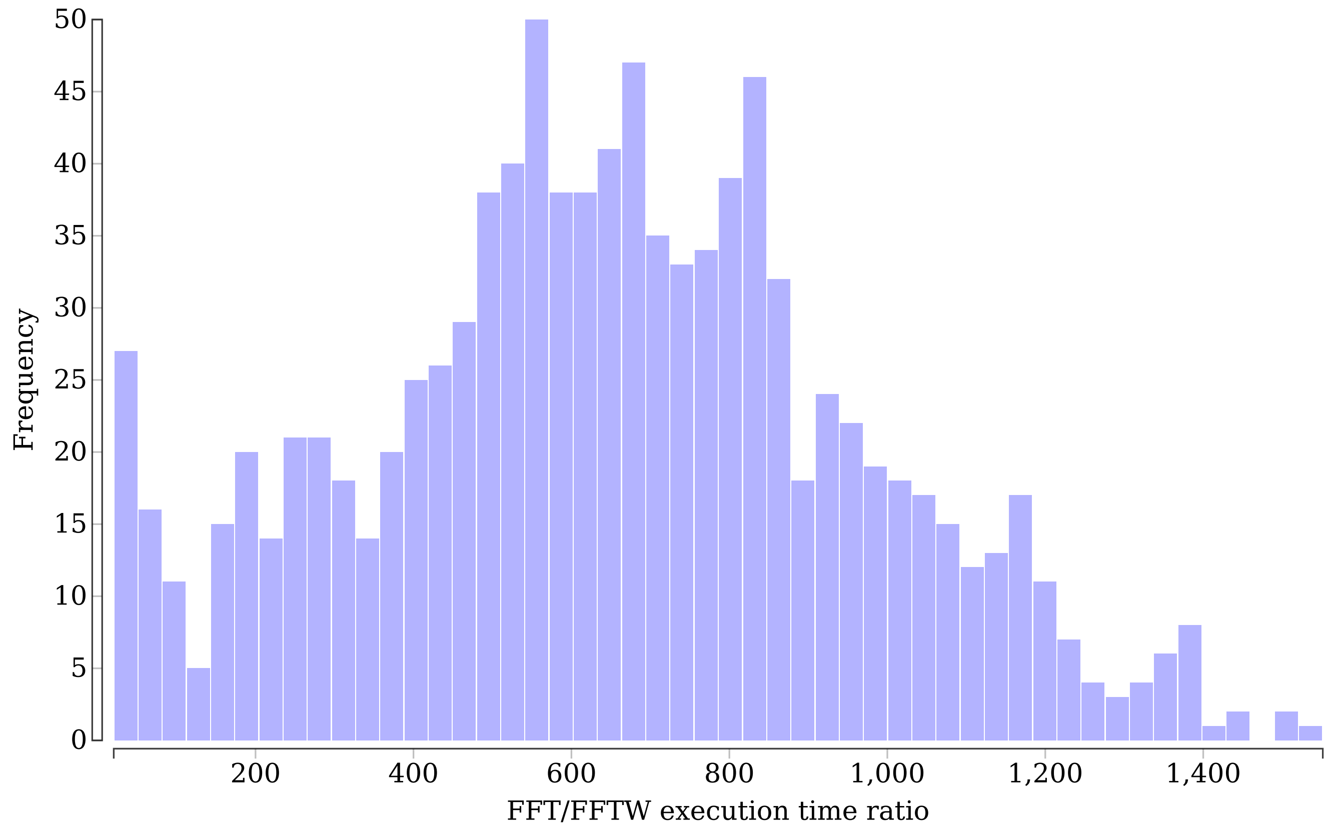
Final benchmarking

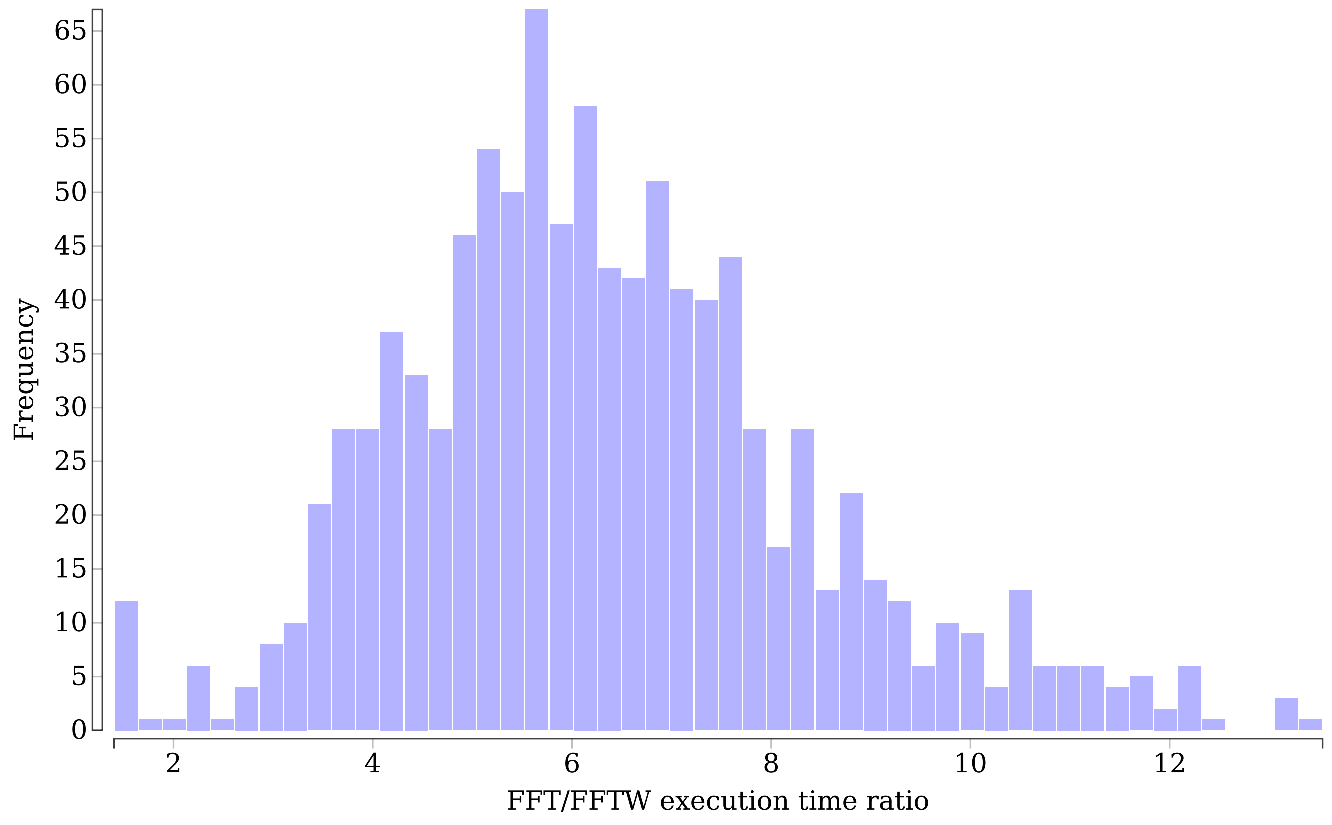
We can benchmark our final pre-release code version (`pre-release-4`) in the same way as we've done before. Here are the results shown in the usual way:

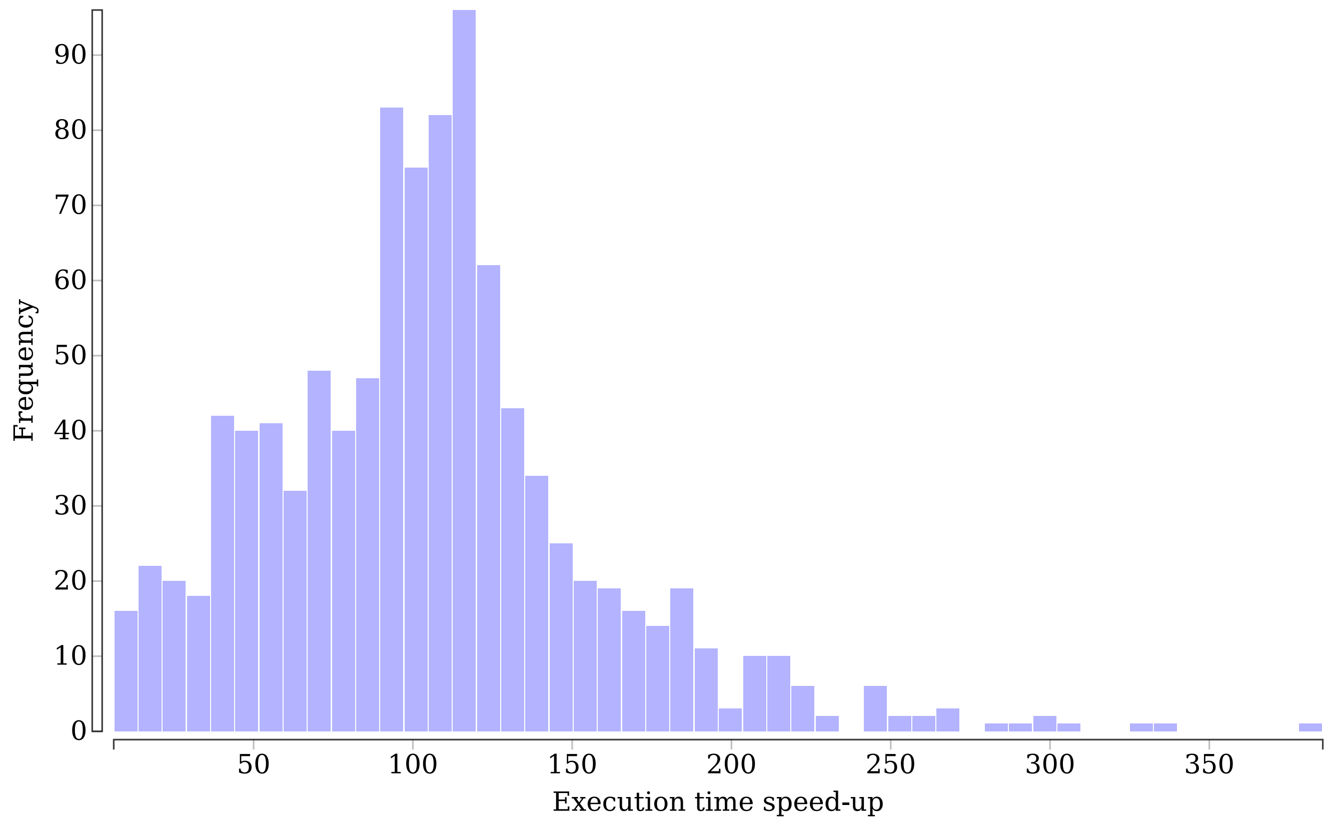


and here are the ratio plots showing the relative performance of the original unoptimised version of the code (`pre-release-1`), the current version (`pre-release-4`) and FFTW:

- pre-release-1
- pre-release-4
- Speed-up

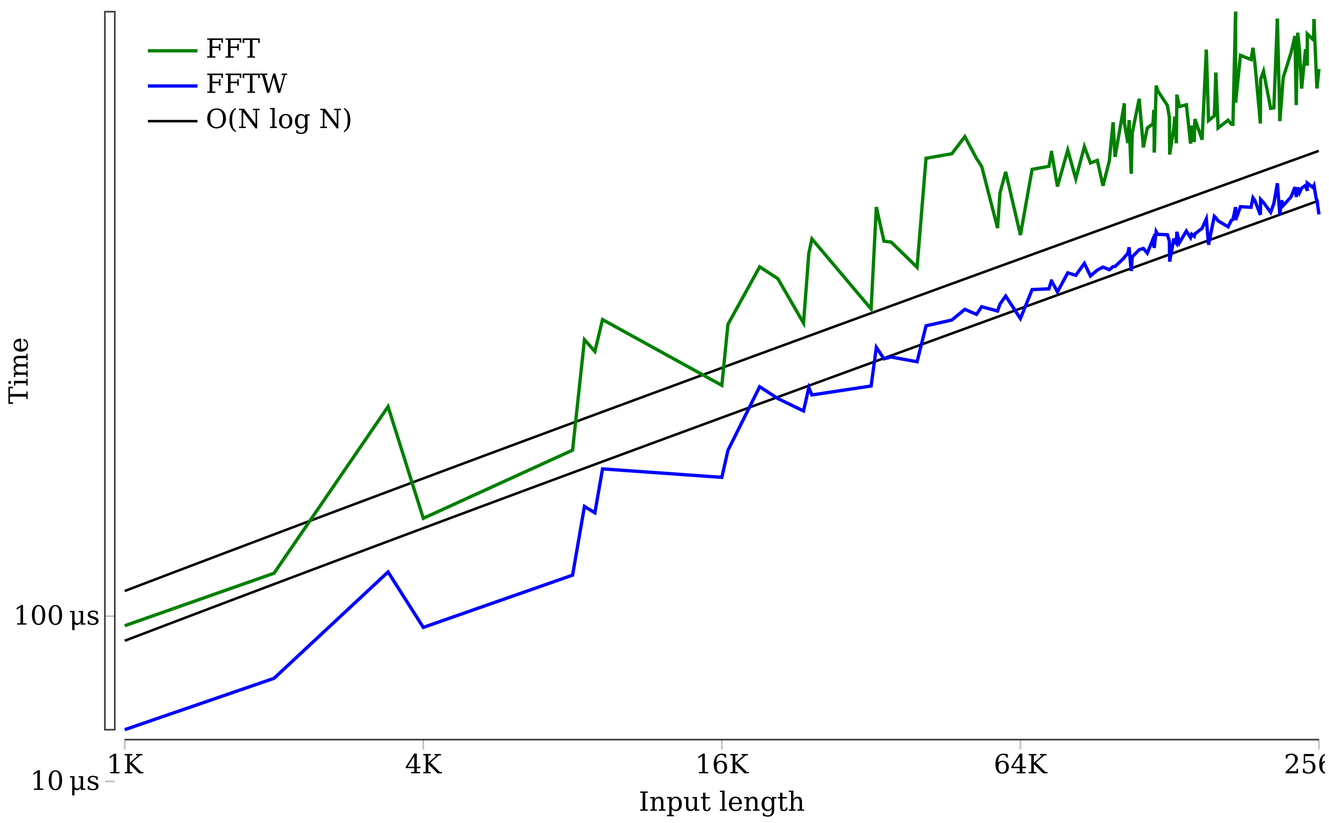

Frequency
0
5
10
15
20
25
30
35
40
45
50
200
400
600
800
1,000
1,200
1,400
FFT/FFTW execution time ratio

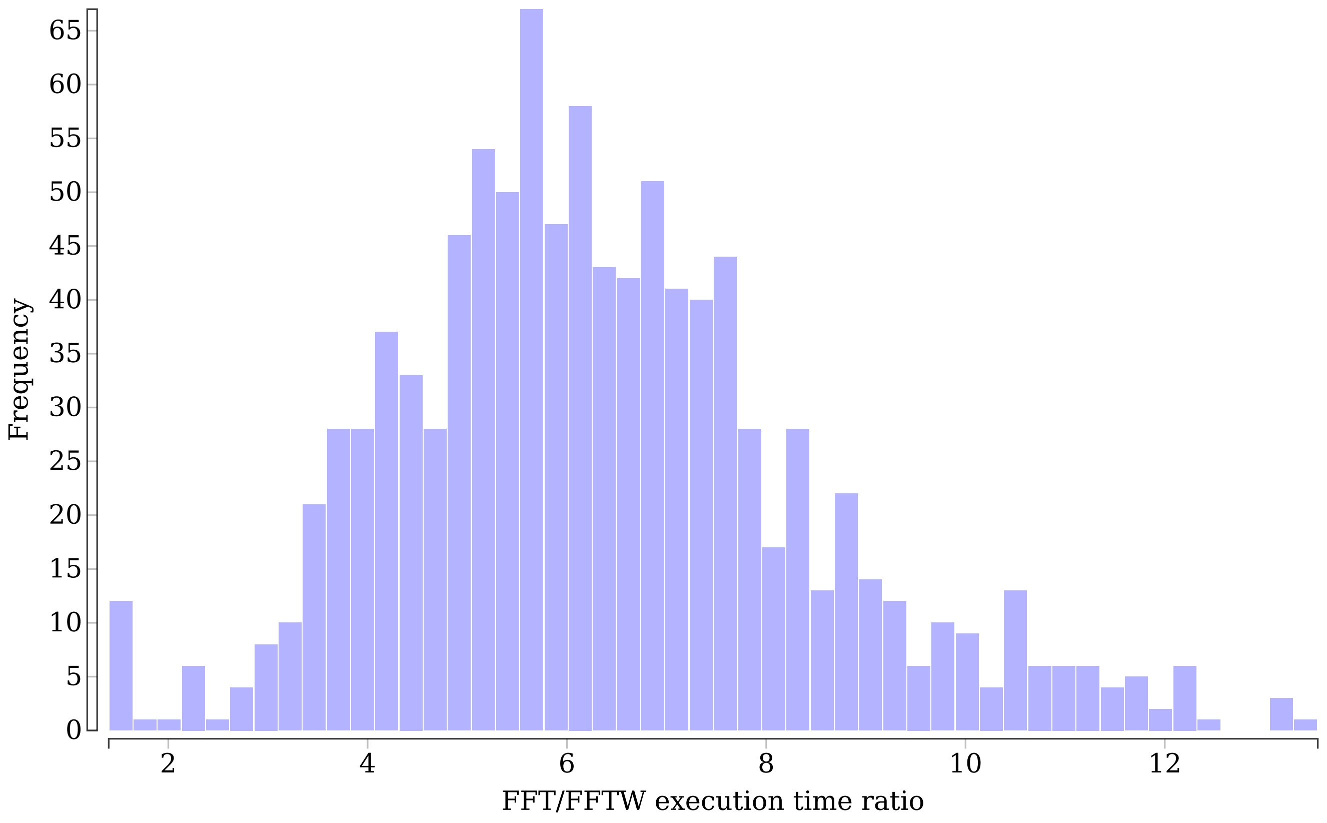

Frequency
0
5
10
15
20
25
30
35
40
45
50
55
60
65
2
4
6
8
10
12
FFT/FFTW execution time ratio



The `pre-release-4` tab is the one to look at: for most input sizes we're within a factor of 4-8 of the performance of the `vector-fftw` package. To be exact, the execution times for our code are between 1.40 and 13.54 slower than `vector-fftw`, with a median of 6.12 and quartiles of 4.98 (25%) and 7.50 (75%). To my mind, that's pretty amazing: FFTW is an *immensely* clever piece of software and I really didn't expect to get that close to it. To be sure, we've done quite a bit of work to get to this point, but it's still significantly less than the investment of time that's gone into the development of FFTW. I have a feeling that we've eaten most of the low-hanging fruit, and getting that last factor of five or so speed-up to become competitive with FFTW would be pretty hard.

I've also done benchmarking measurements with this optimised code for larger input sizes: a selection of about 100 input lengths between 2^{10} and 2^{18}. The results are shown below: they are similar to those for smaller input sizes, except that there is a longer tail to larger ratios: the maximum here is 15.22 — the median is 5.99 and the quartiles are very similar to those for the smaller input sizes.





This shows that we’re pretty successful at maintaining good $O(N\log N)$ scaling for larger input sizes, which is nice.

In the next and last article, we'll wrap-up with some comments on what we've done, the lessons learned, and things left to do.

data-analysis haskell

0 comments

Sign in **1 person listening**

+ Follow Share **Post comment as...**

Newest | Oldest | Top Comments

AI book-reviews colophonia computer-science constraints data-analysis day-job dogs embedded fiction gis haskell kayaking mathematics montpellier moocs nonsense organisation past-lives phd photos programming r regrets remote-sensing running science sleep web-programming yesod yoga