Haskell FFT 12: Optimisation Part 2

January 20, 2014

In the last article, we did some basic optimisation of our FFT code. In this article, we're going to look at ways of reordering the recursive Cooley-Tukey FFT decomposition to make things more efficient. In order to make this work well, we're going to need more straight line transform "codelets". We'll start by looking at our $N = 256$ example in detail, then we'll develop a general approach.

Experiments with $N = 256$

So far, we've been using a fixed scheme for decomposing the overall Fourier matrix for our transforms, splitting out single prime factors one at a time in increasing order of size, ending up with a base transform of a prime number length, which we then process using either a specialised straight line codelet, or Rader's algorithm. However, there's nothing in the generalised Danielson-Lanczos recursion step that we're using that requires us to use *prime* factors (we can build the necessary $I + D$ matrices for any factor size), and there's nothing that constrains the order in which we process factors of our input length.

For the $N = 256$ example, we've been decomposing the input length as $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$, but there's no reason why we couldn't decompose it as 16×16, $4 \times 8 \times 8$ or any other ordering of factors that multiply to give 256. There are several things that could make one of these other choices of factorisations give a faster FFT:

1. We could have a specialised straight line transform for $N = 16$ say, which would make the 16×16 factorisation more attractive;

2. We could save work by doing less recursive calls: there's going to be less overhead in doing a single Danielson-Lanczos step for the 16×16 decomposition than in doing seven steps for the $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$ decomposition;

3. We could take advantage of special structure in the Danielson-Lanczos $I + D$ matrices for certain factor sizes (we won't do this here, but it's a possibility if we ever develop specialised "twiddlets" as well as "bottomlets");

4. One factorisation might result in better cache performance than another (this obviously is more relevant for larger transform sizes).

In order to take advantage of different factorisations in the $N = 256$ case, it will be advantageous to have specialised straight line base transforms for various powers-of-two input lengths. I've implemented these specialised transforms for $N = 2, 4, 8, 16, 32, 64$ (again just converting the FFTW codelets to Haskell).

For $N = 256 = 2^8$, we can think about transform plans that make use of any of these base transforms: if we use the $N = 64$ base transform, we account for 6 of those 8 factors of two, leaving us with 2 factors of two to deal with via Danielson-Lanczos steps. We can treat these as a 2×2 decomposition, or as a single step of size 4. Similarly, if we use the $N = 32$ base transform, we have 3 factors of 2 left over to deal with using Danielson-Lanczos steps, which we can treat as one of $\{2 \times 2 \times 2, 2 \times 4, 4 \times 2, 8\}$ – note that the order of factors may be relevant here: it's quite possible that a 2×4 decomposition could have different performance behaviour than a 4×2 decomposition.

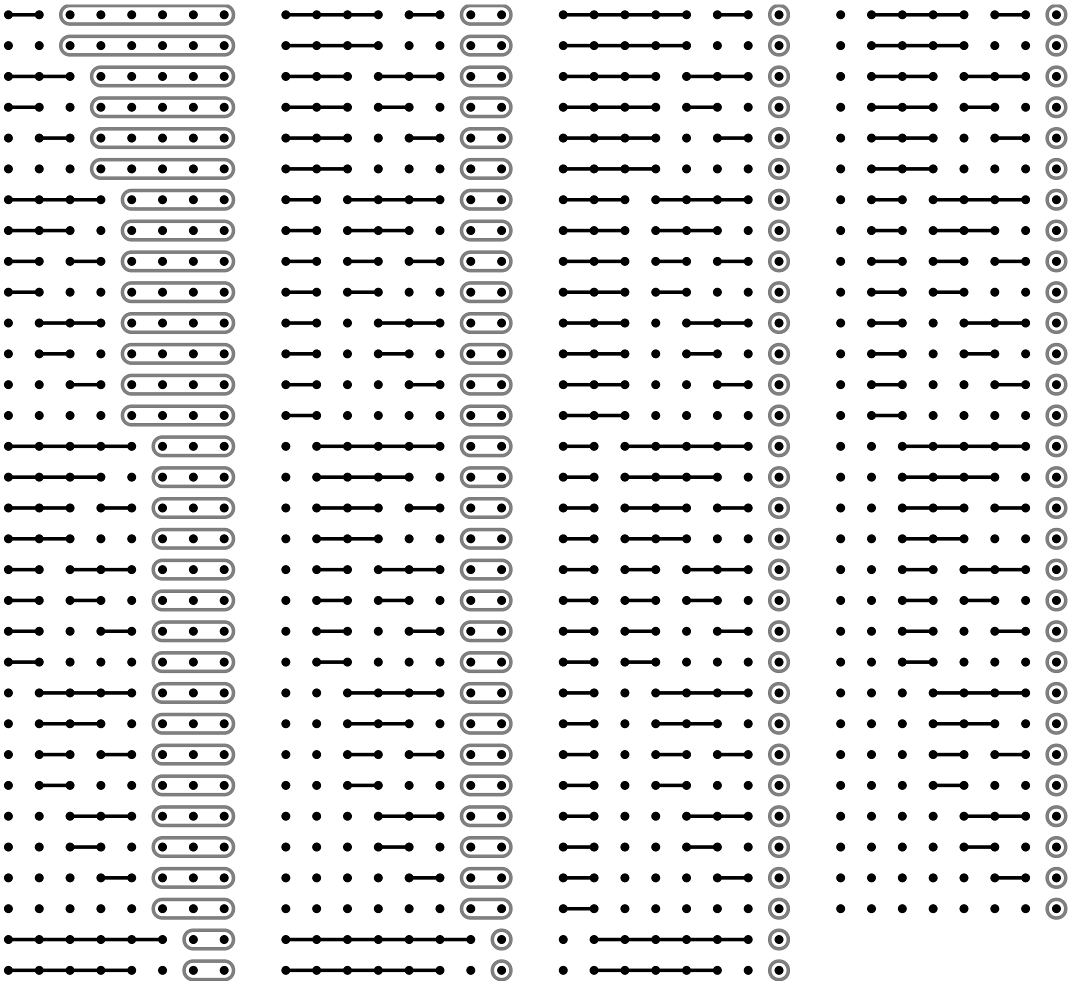
For each base transform size, we thus have a number of decomposition possibilities for dealing with the "left over" factors of two. If there are m left over factors, there are 2^{m-1} possible decompositions. To see why this is so, note that this decomposition problem is isomorphic to the combinatorical *compositions* problem, the determination of the number of ways of writing an integer m as the sum of a sequence of strictly positive integers (where the order of the sequence matters, making this distinct from the *partition* problem). For instance, for $m = 3$, we can write $3 = 1 + 1 + 1$, $3 = 2 + 1$, $3 = 1 + 2$, $3 = 3$, isomorphic to writing $8 = 2^1 \times 2^1 \times 2^1$, $8 = 2^2 \times 2^1$, $8 = 2^1 \times 2^2$, $8 = 2^3$.

The composition problem can be solved by thinking of writing down a row of m ones, with $m - 1$ gaps (shown as boxes):

$$(\quad 1 \;\square\; 1 \;\square\; \cdots \;\square\; 1 \;\square\; 1 \quad)$$

If we now assign each box either a comma (",") or a plus sign ("+"), we end up with a unique composition. Since there are $m - 1$ gaps and two possibilities for each gap, there are 2^{m-1} total compositions.

For $N = 256$, we can determine the possible compositions of the left over factors for each possible choice of base transform. In the following figure, possible FFT plans for $N = 256$ are shown, given the availability of base transforms of sizes 2, 4, 8, 16, 32, 64. Each dot represents a factor of two (eight for each instance: $256 = 2^8$). The factors for the base transform are surrounded by a frame, and factors that are treated together in a Danielson-Lanczos step are joined by a line (i.e. three dots joined by a line represent a Danielson-Lanczos step of size 8):



In total, there are 126 possiblities:

Base	***m***	$\mathbf{2^{m-1}}$
64	2	2
32	3	4
16	4	8
8	5	16
4	6	32
2	7	64
		126

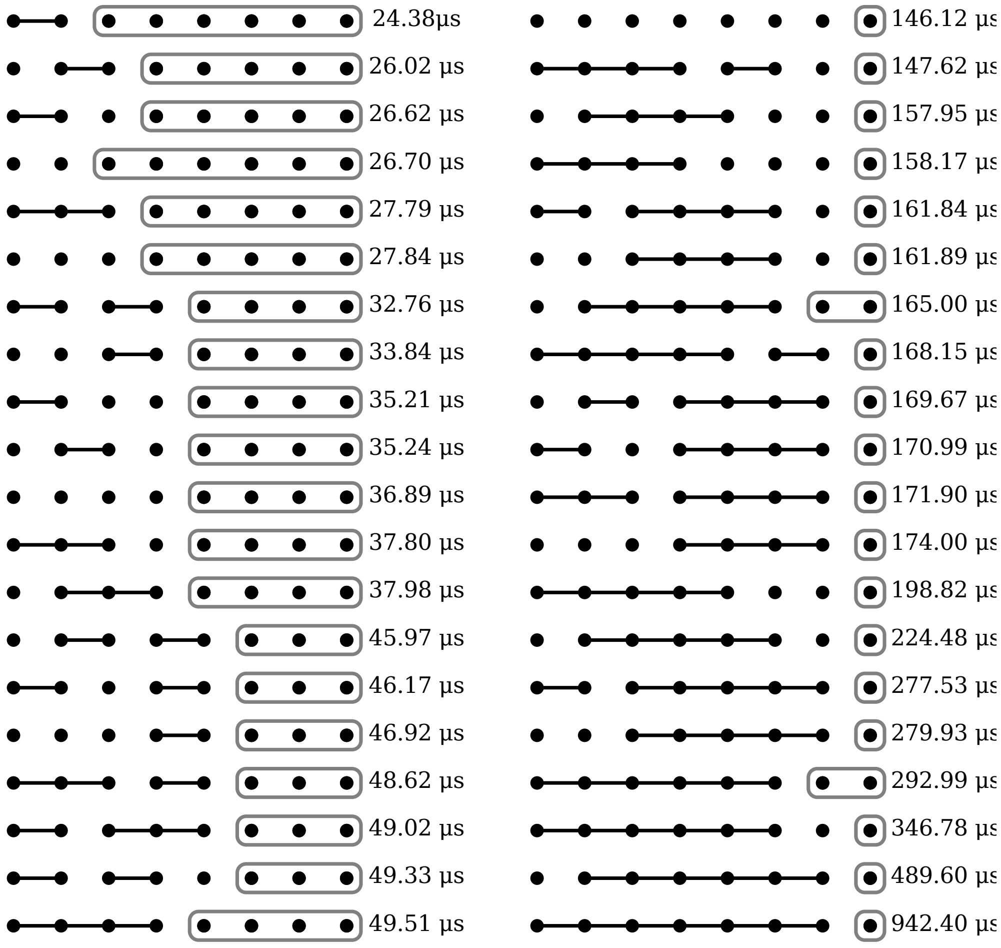
It’s not too hard to derive the number of plans for an arbitrary power-of-two input vector length. Suppose $N = 2^M$ and we have base transforms for all 2^i, $i = 1, 2, \ldots, B$. Then, if $M > B$, the total number of plans P is

$$P = \sum_{b=1}^{B} 2^{M-b-1} = 2^{M-1} \sum_{b=1}^{B} 2^{-b} = 2^{M-1} \frac{2^B - 1}{2^B} = 2^{M-B-1}(2^B - 1).$$

For $N = 256$ and bases up to $N = 64 = 2^6$, we have $M = 8$, $B = 6$ and $P = 2(2^6 - 1) = 126$. It turns out not to be much harder to find a general expression for the number of plans in the more complex mixed-radix case, as we'll see below.

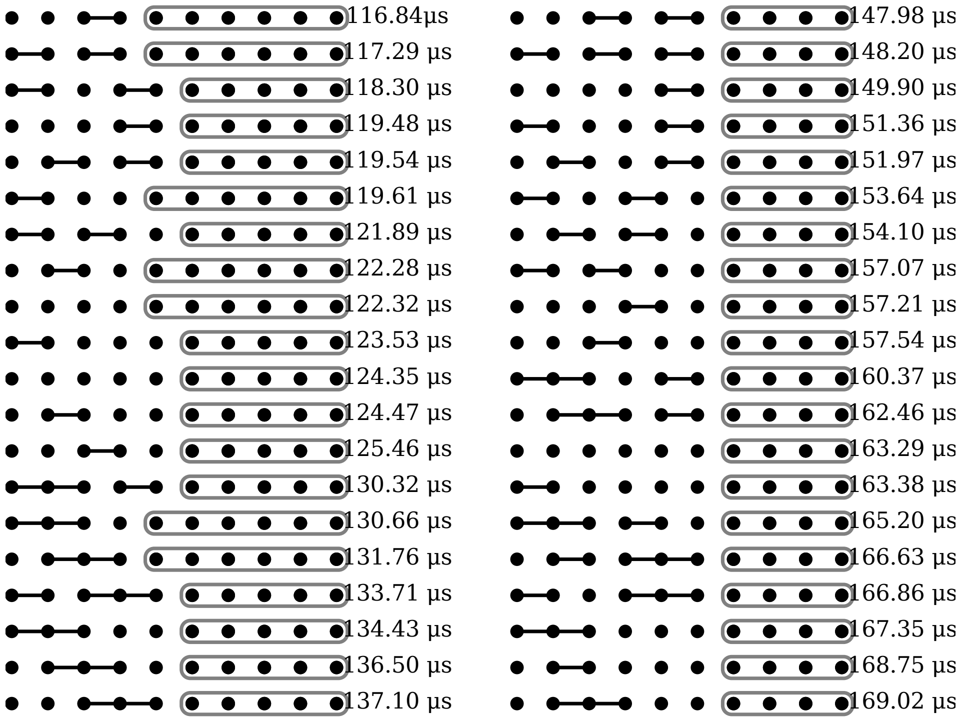
A priori, we can't say much about which of these factorisations is going to give us the best FFT performance, but we can measure the performance, using the same benchmarking approach that we've been taking all along. We need some code to generate the factorisations and to produce a value of our `Plan` type from this information, but given these things we can measure the execution time of the $N = 256$ FFT using each of the 126 plans show above. We'd expect plans using larger base transforms to do well, since these straight line base transforms are highly optimised, but it's not obvious what choice of factorisation for the "left over" factors is going to be faster.

Here are the execution times from the fastest and slowest 20 plans out of the 126 (for comparison, the execution time using the $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$ plan we've been using up to now is about 145 μs – the top entry in the right hand column here):



Two things stand out from these results. First, the fastest transforms are those using the largest base transforms ($N = 64$ or $N = 32$). This isn't too much of a surprise, since these plans offload the largest proportion of the FFT processing to optimised straight line code. Conversely, the slowest plans are those that use the smallest ($N = 2$ or $N = 4$) base transforms. Further, the slowest of the slow transforms appear to be those that use the largest Danielson-Lanczos steps. For example, the overall slowest plan (by a factor of about two in time) uses an $N = 2$ base transform and a single Danielson-Lanczos step of size 128. These larger Danielson-Lanczos steps could be more efficient if we had specialised "twiddlets" for them, but as it is currently, they do a lot of wasted work and so are less efficient than a series of smaller decompositions.

Exactly where the trade-off between larger and smaller Danielson-Lanczos steps lies isn't immediately obvious. For example, the fastest plans using an $N = 16$ base transform use 4×4, $2 \times 2 \times 4$ and $4 \times 2 \times 2$ Danielson-Lanczos steps for the other factors, but the time differences are small, perhaps not even larger than the margin of error in measurement. We can get a further idea of how this trade-off works by looking at the best plans for $N = 1024$. Here shows the best 40 plans for this input length size (for comparison, the execution time for the “standard” $N = 1024$ transform from the earlier articles is about 550 μs):



Again we see that the fastest FFTs plans use the largest specialised base transforms along with a combination of smallish Danielson-Lanczos steps to form the overall transform. In particular, we see that none of the fastest plans involve Danielson-Lanczos steps of sizes greater than eight.

The situation here is simpler than for the general mixed-radix transform case that we’ll deal with next, but these results give some idea of two approaches to use for a heuristic to choose reasonable plans to test for a given problem size:

1. Try the largest few base transforms that are available;

2. Use relatively small Danielson-Lanczos steps.

Within these constraints, the choice of which plan is best should be settled empirically by benchmarking examples for each plan.

Mixed-radix case

Now let’s think about the general mixed-radix case. In terms of base transforms, for larger prime factors we can use Rader’s algorithm (which should now be faster since we’ve improved the performance of the powers-of-two transforms that are used for the necessary convolution), but it would be useful to have some other specialised straight line base transforms available too. I’ve now implemented specialised transforms for all N up to 16, plus 20, 25, 32 and 64. This is the same set of specialised base transforms used in the standard distribution of FFTW (and indeed, the versions I’m using are just Haskell translations of the FFTW C codelets).

As possible base transforms on which to build a full FFT, we should consider any of the specialised base transform sizes that are factors of N, and we should also consider transforms using Rader's algorithm for any prime factors greater than 13 (the largest prime for which we have a specialised base transform). We should never consider the naïve DFT algorithm for a base transform, since it's almost certain that there will be a better way, now that we have specialised transforms for a range of small prime and other sizes (remember the $O(N^2)$ scaling of the naïve DFT!). Once we've selected a base transform size (which we'll call N_b in what follows) we need to decide what to do with the "left over" factors. Suppose that the prime factorisation of the input length N can be written as

$$N = f_1^{m_1} f_2^{m_2} \ldots f_D^{m_D} N_b$$

where the f_i, $i = 1, 2, \ldots, D$ are unique prime factors and the m_i are "multiplicities", i.e. the number of powers of each unique factor included in the left over part of N. Let's think about the number of FFT plans that we can build in this case. First of all, we need to decide on an order for the factors. Ignoring the issue of duplicate factors, there are $N_D = m_1 + m_2 + \ldots + m_D$ factors and the total number of possible orderings of these is just the factorial of this number. To take account of duplicate factors, we need to divide this overall factorial by the factorials of the individual multiplicities. The total number of distinct orderings of factors is thus

$$\frac{N_D!}{\prod_{i=1}^{D} m_i!}.$$

Having chosen an ordering for the factors, we then need to compute the number of distinct ways to compose adjacent prime factors to give composite factors. This is entirely analogous to the situation in the 2^N case, and we can use the same result. One might thus think that the total number of plans for this choice of base transform is then

$$\frac{N_D!}{\prod_{i=1}^{D} m_i!} 2^{N_D - 1}.$$

However, because it's possible for different compositions of different factor permutations to result in the same plan, the number is somewhat less than this. (As an example, think of $12 = 2 \times 2 \times 3$. If we use a base transform of size 2, we have two distinct permutations of the remaining factors, i.e. $(2, 3)$ and $(3, 2)$. Each of these has two compositions, $(2, 3)$ and (6) and $(3, 2)$ and (6). Because of the commutativity of ordinary multiplication we get a plan with a single size-6 Danielson-Lanczos step from each permutation.)

In order to get an accurate count of the number of plans for different values of N, we need to generate the plans to check for this kind of overlap. This requires a little bit of sneakiness. For a given N, we need to determine the possible base transforms, then we need to generate all distinct permutations of the "left over" factors to use for calculating compositions. We need to retain the unique result vectors of factors (some prime, some composite) to use as the sizes for Danielson-Lanczos steps. The most difficult part is the generation of the permutations – because we may have duplicates in the list of factors, we don't want to use a standard permutation algorithm. If we did this, for the factors 2^{10}, we would generate $10!$ permutations (all the same, because all of our factors are identical) instead of the single distinct permutation!

Wikipedia is our friend here. There's a simple algorithm to generate all distinct permutations of a multiset in lexicographic order, starting from the "sorted" permutation, i.e. the one with all the entries in ascending numerical order. It goes like this – for a sequence a_j, $1 \le j \le n$:

1. Find the largest index k such that $a_k < a_{k+1}$. If no such index exists, the permutation is the

last permutation.

2. Find the largest index l such that $a_k < a_l$.

3. Swap the value of a_k with that of a_l.

4. Reverse the sequence from a_{k+1} up to and including the final element a_n.

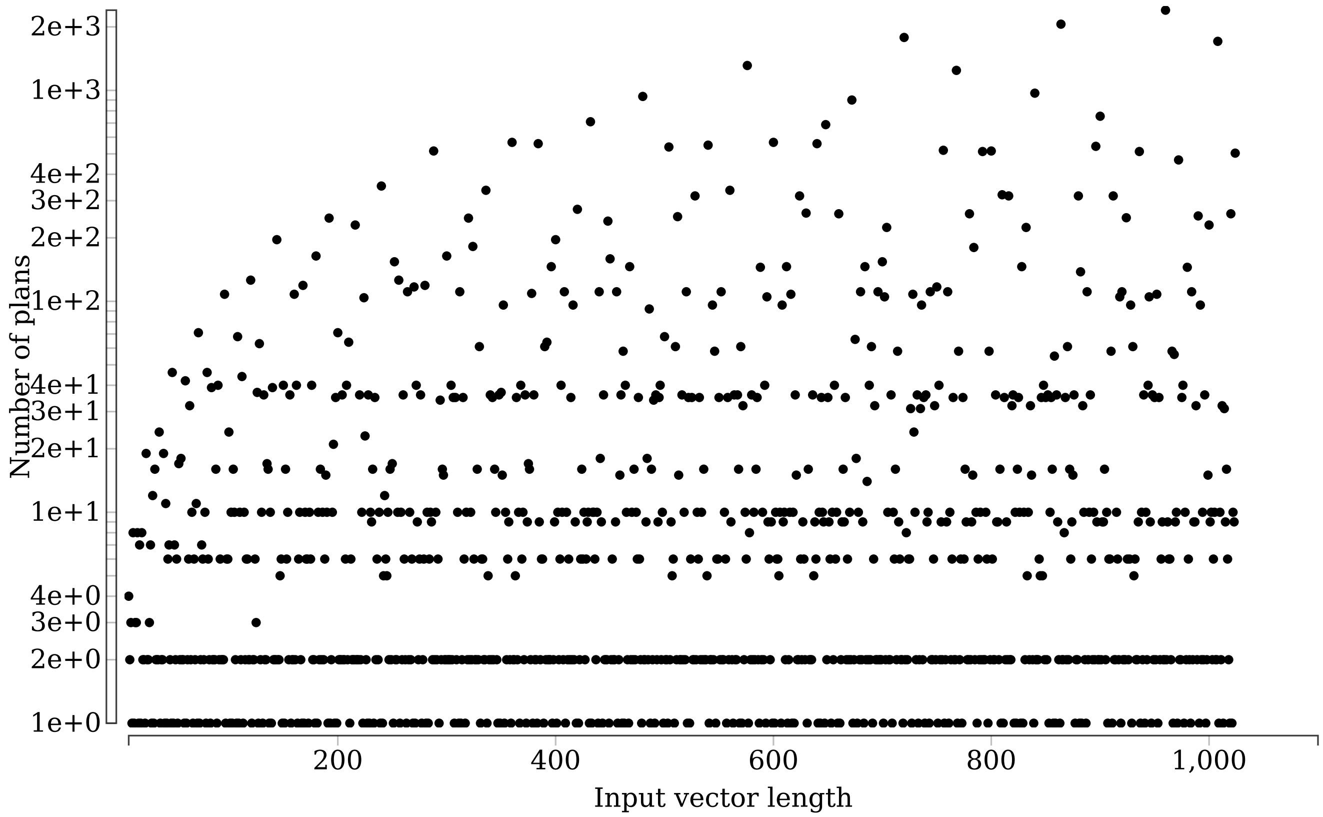
Short and sweet! Here's some (not very clever or efficient, but good enough) code (here, `Data.Vector` is imported, and `Data.List is imported qualified as`L`):

```haskell
-- One step of multiset permutation algorithm.
permStep :: Vector Int -> Maybe (Vector Int)
permStep v =
  if null ks
  then Nothing
  else let k = maximum ks
           l = maximum $ filter (\i -> v!k < v!i) $ enumFromN 0 n
       in Just $ revEnd k (swap k l)
  where n = length v
        ks = filter (\i -> v!i < v!(i+1)) $ enumFromN 0 (n-1)
        swap a b = generate n $ \i ->
          if      i == a then v!b
          else if i == b then v!a
                         else v!i
        revEnd f vv = generate n $ \i ->
          if i <= f then vv!i else vv!(n - i + f)

-- Generate all distinct multiset permutations in lexicographic order:
-- input must be the "sorted" permutation.
allPerms :: Vector Int -> [Vector Int]
allPerms idp = idp : L.unfoldr step idp
  where step v = case permStep v of
          Nothing -> Nothing
          Just p -> Just (p, p)
```

It's more or less a direct Haskell transcription of the algorithm from the Wikipedia page. I'm sure it could be sped up and cleaned up a lot, but it works well enough for this application. (There's also the Johnson-Trotter loopless permutation algorithm, described in Chapter 28 of Richard Bird's *Pearls of Functional Algorithm Design*, which looks monstrously clever and deserved some study. Probably overkill for now, since the code in here is plenty quick enough for what we need here.)

Given a way to generate all the permutations for each set of "left over" factors, we can calculate the compositions of each and retain the distinct ones for counting (we just use a `Set` from `Data.Set` to hold the results to maintain distinctness). We can do this for each possible base transform for a given N and sum them to get the total number of possible FFT plans. This plot shows the number of plans available for each input vector length in the range 8–1024 (using all the range of specialised base transforms that I've implemented):



All prime input lengths have only a single possible plan, there are some larger highly factorisable input lengths that have thousands of possible plans, and most input lengths have a number of plans lying somewhere between these two extremes.

The primary challenge in selecting a good plan from this range of possibilities is to choose a heuristic that requires us to *measure* the performance of only a few plans: out of the hundreds or thousands of possibilities for a given input vector length, most plans will be duds. We need to pick out a handful of likely candidate plans for benchmarking. In order to develop some heuristic plan selection rules, we'll do some benchmarking experiments, as we did for the $N = 256$ case in the last section. We'll look at some input vector lengths that have lots of possible plans, some that have only a few possible plans and some that lie in the middle range:

Size	Factors	No. of plans
960	$2^6 \times 3 \times 5$	2400
800	$2^5 \times 5^2$	516
512	2^9	252
216	$2^3 \times 3^3$	230
378	$2 \times 3^3 \times 7$	109
208	$2^4 \times 13$	40
232	$2^3 \times 29$	16
238	$2 \times 7 \times 17$	10
92	$2^2 \times 23$	6
338	2×13^2	5
1003	17×59	2
115	5×23	2

For each of these problem sizes, the table below shows the execution times for each of the fastest ten plans or for all possible plans, if there are less than ten (the base transform sizes are highlighted in bold):

$N = 960$ (235.25 µs)

3 × 2 × 5:**32**	115.13 µs
3 × 5:**64**	116.19 µs
3 × 5 × 2:**32**	117.68 µs
2 × 5 × 3:**32**	117.89 µs
5 × 3:**64**	118.49 µs
2 × 3 × 5:**32**	118.99 µs
5 × 2 × 3:**32**	119.04 µs
5 × 3 × 2:**32**	119.29 µs
6 × 5:**32**	121.05 µs
5 × 6:**32**	124.21 µs

$N = 800$ (192.05 µs)

5 × 5:**32**	94.64 µs
2 × 4 × 4:**25**	101.56 µs
4 × 2 × 4:**25**	101.67 µs
2 × 2 × 2 × 4:**25**	104.10 µs
4 × 4 × 2:**25**	104.20 µs
2 × 2 × 4 × 2:**25**	104.56 µs
2 × 4 × 2 × 2:**25**	106.56 µs
4 × 2 × 2 × 2:**25**	106.61 µs
2 × 2 × 2 × 2 × 2:**25**	107.50 µs
2 × 5 × 4:**20**	108.47 µs

$N = 512$ (278.94 µs)

4 × 4:**32**	55.10 µs
2 × 4:**64**	55.63 µs
4 × 2:**64**	55.66 µs
2 × 2 × 4:**32**	56.01 µs
2 × 2 × 2:**64**	56.65 µs
4 × 2 × 2:**32**	57.00 µs
2 × 4 × 2:**32**	57.34 µs
2 × 2 × 2 × 2:**32**	58.71 µs
8:**64**	60.27 µs
2 × 8:**32**	62.38 µs

$N = 216$ (69.19 µs)

3 × 6:**12**	31.10 µs
2 × 3 × 3:**12**	31.18 µs
3 × 2 × 3:**12**	31.51 µs
6 × 3:**12**	31.63 µs
3 × 3 × 2:**12**	32.60 µs
2 × 9:**12**	35.79 µs

$N = 378$ (71.52 µs)

3 × 3 × 3:**14**	52.16 µs
9 × 3:**14**	59.54 µs
3 × 9:**14**	60.95 µs
3 × 2 × 7:**9**	68.10 µs
2 × 3 × 7:**9**	68.44 µs
2 × 7 × 3:**9**	68.74 µs
7 × 6:**9**	69.49 µs
6 × 7:**9**	69.73 µs
7 × 2 × 3:**9**	69.83 µs
7 × 3 × 2:**9**	70.76 µs

$N = 208$ (32.85 µs)

4 × 4:**13**	28.97 µs
2 × 2 × 4:**13**	30.31 µs
2 × 4 × 2:**13**	31.70 µs
4 × 2 × 2:**13**	32.04 µs
2 × 8:**13**	33.21 µs
2 × 2 × 2 × 2:**13**	33.58 µs
8 × 2:**13**	34.26 µs
13:**16**	35.31 µs
16:**13**	45.04 µs
2 × 13:**8**	47.74 µs

$N = 232$ (584.14 µs)

29:**8**	87.52 µs
29 × 2:**4**	113.47 µs
2 × 29:**4**	136.21 µs
29 × 4:**2**	142.41 µs
29 × 2 × 2:**2**	159.00 µs
2 × 29 × 2:**2**	185.15 µs
2 × 2 × 29:**2**	229.41 µs
4 × 29:**2**	231.43 µs
58:**4**	254.22 µs
58 × 2:**2**	294.84 µs

$N = 238$ (316.93 µs)

17:**14**	51.85 µs
17 × 2:**7**	69.44 µs
2 × 17:**7**	69.68 µs
34:**7**	109.80 µs
17 × 7:**2**	127.19 µs
7 × 17:**2**	160.65 µs

$N = 92$ (285.60 µs)

23:**4**	43.65 µs
23 × 2:**2**	65.04 µs
2 × 23:**2**	76.00 µs
46:**2**	125.16 µs
4:**23**	295.37 µs
2 × 2:**23**	299.11 µs

$N = 338$ (67.50 µs)

13 × 2:**13**	66.91 µs
2 × 13:**13**	67.30 µs
26:**13**	106.41 µs
13 × 13:**2**	212.18 µs
169:**2**	1701.35 µs

$N = 1003$ (2413.41 µs)

59:**17**	1843.99 µs
17:**59**	2538.96 µs

$N = 115$ (362.84 µs)

23:**5**	46.50 µs
5:**23**	373.16 µs

3 × 2 × 4:**9**	36.03 µs	2 × 7:**17**	327.17 µs
6 × 4:**9**	36.67 µs	7 × 2:**17**	329.79 µs
2 × 3 × 4:**9**	37.00 µs	14:**17**	339.61 µs
9 × 2:**12**	37.04 µs	119:**2**	823.30 µs

It’s important to note that we don’t need to use these results to identify the *best* plan, just a reasonable set of plans to test empirically. When a call is made to the `plan` function, we’re going to use Criterion to benchmark a number of likely looking plans and we’ll take the fastest one. It doesn’t matter if this benchmarking takes a while (a few seconds, say) since we’ll only need to do it once for each input length (we’ll eventually save the best plan away in a “wisdom” file so that we can get at it immediately if a plan for the same input length is requested later on). If some transform takes about 200 µs on average, then we can run 5,000 tests in one second. Since Criterion runs 100 benchmarks to get reasonable timing statistics, we could test about 50 plans in a second. Let’s aim to come up with a heuristic that yields 20-50 plans for a given input length, test them all, then pick the best one.

So, based on these results and the earlier results for the powers-of-two case, what appear to be good heuristics for plan selection?

- First, make use of the larger specialised base transforms. In both the $N = 256$ and $N = 1024$ cases, the fastest plans all made use of the size-64 or size-32 base transforms, and likewise in the general experiments shown above, the faster plans tend to make use of the larger base transforms.

- Specialised base transforms are generally faster than using Rader’s algorithm for a larger prime factor as the base transform. For example, for $N = 92 = 2^2 \times 23$, plans using the size-4 or size-2 base transforms are faster than those using Rader’s algorithm for the factor of 23.

- Using multiple Danielson-Lanczos steps of smaller sizes is generally faster than trying to fold everything into one big composite step. For example, the results for the $N = 256$ case show that the *slowest* plans tend to be those using the largest Danielson-Lanczos steps.

- In cases where there’s no but to use Rader’s algorithm for the base transform, it can make a difference to use a base transform size that doesn’t need any padding for the Rader algorithm convolution. For example, for $N = 1003 = 17 \times 59$, using a base transform of size 17 is significantly faster than using a base transform of size 59, presumably because, for the size-17 transform, Rader’s algorithm requires us to perform a convolution of size 16, and no zero padding is needed, while for the size-59 transform, the convolution requires padding of an intermediate vector to length 128 (the smallest power of two that works).

Based on this, here’s a set of heuristics for choosing plans to test:

1. Determine the usable base transforms for the given N; sort the specialised base transforms in descending order of size, followed by any other prime base transforms, in descending order of size with transform sizes of the form $2^m + 1$ before any others (i.e. those that don’t require any padding for the Rader’s algorithm convolution).

2. For each base transform, generate all the plans that make use of Danielson-Lanczos steps for the remaining “left over” factors that are “small”, in the sense that they involve only one, two or three factors at a time (i.e. don’t bother generating plans that use Danielson-Lanczos steps of larger sizes).

3. Limit the number of generated plans to around 50 so that the benchmarking step doesn’t take too long.

4. Benchmark all the generated plans and select the fastest!

This approach may not find the optimal plan every time, but it should have a relatively good chance and won’t require the benchmarking of too many plans.

There is one aspect of all this that would require revisiting if we had specialised “twiddlets” for the Danielson-Lanczos steps. In that case, we would want to pick plans where we could use a large specialised base transform and make use of the largest possible specialised Danielson-Lanczos “twiddlets”. For the moment, since we don’t have such specialised machinery for the Danielson-Lanczos steps, the approach above should be a good choice.

Implementation

The code described here is the `pre-release-3` version of the GitHub repository.

Generation of the candidate plans for a given input size is basically a question of determining which base transforms are usable, then generating all of the permutations and combinations of the “left over” factors for each base transform. The plans are sorted according to the heuristic ordering described above and we take the first 50 or so for benchmarking.

For the base transforms, we define a helper type called `BaseType` to represent base transforms and to allow us to define an `Ord` instance for the heuristic ordering of the base transforms. We also define a `newtype` wrapper, `SPlan`, to wrap the whole of a plan definition, again so that we can write a custom `Ord` instance based on the heuristic ordering of plans:

```haskell
-- | Base transform type with heuristic ordering.
data BaseType = Special Int | Rader Int deriving (Eq, Show)

-- | Newtype wrapper for custom sorting.
newtype SPlan = SPlan (BaseType, Vector Int) deriving (Eq, Show)

-- | Base transform size.
bSize :: BaseType -> Int
bSize (Special b) = b
bSize (Rader b) = b

-- | Heuristic ordering for base transform types: special bases come
-- first, then prime bases using Rader's algorithm, ordered according
-- to size compensating for padding needed in the Rader's algorithm
-- convolution.
instance Ord BaseType where
  compare (Special _)  (Rader _)    = LT
  compare (Rader _)    (Special _)  = GT
  compare (Special s1) (Special s2) = compare s1 s2
  compare (Rader r1)   (Rader r2)   = case (isPow2 $ r1 - 1, isPow2 $ r2 - 1) of
    (True, True) -> compare r1 r2
    (True, False) -> compare r1 (2 * r2)
    (False, True) -> compare (2 * r1) r2
    (False, False) -> compare r1 r2

-- | Heuristic ordering for full plans, based first on base type, then
-- on the maximum size of Danielson-Lanczos step.
instance Ord SPlan where
  compare (SPlan (b1, fs1)) (SPlan (b2, fs2)) = case compare b1 b2 of
    LT -> LT
    EQ -> compare (maximum fs2) (maximum fs1)
    GT -> GT
```

Here’s the code for the generation of candidate plans:

```haskell
-- | Generate test plans for a given input size, sorted in heuristic
-- order.
testPlans :: Int -> Int -> [(Int, Vector Int)]
testPlans n nplans = L.take nplans $
                     L.map clean $
                     L.sortBy (comparing Down) $ P.concatMap doone bs
  where vfs = allFactors n
```

```
        bs = usableBases n vfs
        doone b = basePlans n vfs b
        clean (SPlan (b, fs)) = (bSize b, fs)

-- | List plans from a single base.
basePlans :: Int -> Vector Int -> BaseType -> [SPlan]
basePlans n vfs bt = if null lfs
                     then [SPlan (bt, empty)]
                     else P.map (\v -> SPlan (bt, v)) $ leftOvers lfs
  where lfs = fromList $ (toList vfs) \\ (toList $ allFactors b)
        b = bSize bt

-- | Produce all distinct permutations and compositions constructable
-- from a given list of factors.
leftOvers :: Vector Int -> [Vector Int]
leftOvers fs =
  if null fs
  then []
  else S.toList $ L.foldl' go S.empty (multisetPerms fs)
  where n = length fs
        go fset perm = foldl' doone fset (enumFromN 0 (2^(n - 1)))
          where doone s i = S.insert (makeComp perm i) s

-- | Usable base transform sizes.
usableBases :: Int -> Vector Int -> [BaseType]
usableBases n fs = P.map Special bs P.++ P.map Rader ps
  where bs = toList $ filter ((== 0) . (n `mod`)) specialBaseSizes
        ps = toList $ filter isPrime $ filter (> maxPrimeSpecialBaseSize) fs
```

The main `testPlans` function calls `usableBases` to determine what base transforms can be used for a given input size, distinguishing between specialised straight-line transforms (the `Special` constructor of the `BaseType` type) and larger prime transforms that require use of Rader's algorithm (the `Rader` constructor for `BaseType`). For each possible base transform, the `basePlans` function determines the "left over" factors of the input size and uses the `leftOvers` function to generate a list of possible Danielson-Lanczos steps, which are then bundled up with the base transform information into values of type `SPlan`. The `leftOvers` function calculates all possible permutations of the multiset of left-over factors (using the `multiPerms` function, which is essentially identical to the multiset permutation code shown earlier), and for each permutation calculates all possible compositions of the factors. Plans are collected into an intermediate `Set` to remove duplicates.

The full set of candidate plans is then sorted in `testPlans` in descending order of desirability according to the planning heuristics, and the first `nplans` plans are returned for further processing.

There are several places where this code could be made more efficient (there's quite a bit of intermediate list construction, for instance), but there's really not too much point in expending effort on it, since the planning step should only be done once before executing an FFT calculation multiple times using the same plan. In any case, the plan generation is relatively quick even for quite large input sizes.

The top-level code that drives the empirical planning process is shown here:

```
-- | Globally shared timing environment.  (Not thread-safe...)
timingEnv :: IORef (Maybe Environment)
timingEnv = unsafePerformIO (newIORef Nothing)
{-# NOINLINE timingEnv #-}

-- | Plan calculation for a given problem size.
empiricalPlan :: Int -> IO Plan
empiricalPlan n = do
  wis <- readWisdom n
  case wis of
    Just p -> return $ planFromFactors n p
    Nothing -> do
      let ps = testPlans n nTestPlans
      withConfig (defaultConfig { cfgVerbosity = ljust Quiet
                                , cfgSamples   = ljust 1 }) $ do
        menv <- liftIO $ readIORef timingEnv
        env <- case menv of
          Just e -> return e
          Nothing -> do
            meas <- measureEnvironment
            liftIO $ writeIORef timingEnv $ Just meas
            return meas
        let v = generate n (\i -> sin (2 * pi * fromIntegral i / 511) :+ 0)
        tps <- CM.forM ps $ \p -> do
          let pp = planFromFactors n p
          pptest <- case plBase pp of
            bpl@(RaderBase _ _ _ _ csz _) -> do
              cplan <- liftIO $ empiricalPlan csz
              return $ pp { plBase = bpl { raderConvPlan = cplan } }
            _ -> return pp
          ts <- runBenchmark env $ nf (execute pptest Forward) v
          return (sum ts / fromIntegral (length ts), p)
        let (rest, resp) = L.minimumBy (compare `on` fst) tps
        liftIO $ writeWisdom n resp
        let pret = planFromFactors n resp
        case plBase pret of
          bpl@(RaderBase _ _ _ _ csz _) -> do
            cplan <- liftIO $ empiricalPlan csz
            return $ pret { plBase = bpl { raderConvPlan = cplan } }
          _ -> return pret
```

We'll describe the "wisdom" stuff in a minute, but first let's look at the `Nothing` branch of the `case` expression at the top of the `empiricalPlan` function. The `testPlans` function we looked at above is used to generate a list of candidate plans and we then use Criterion to run benchmarks for each of these plans, choosing the plan with the best execution time. There are a few wrinkles to make things a little more efficient. First, we have a global `IORef` that we use to store the Criterion timing environment information – this avoids repeated calls to `measureEnvironment` to determine the system clock resolution. Using an `IORef` in this way is not thread-safe, which is something we would have to fix to make this a production-ready library. We'll not worry about it for now. Second, we make Criterion only collect a single sample for each plan, just to make the benchmarking go quicker. If we have a number of plans that are within a few percent of each other in terms of their execution times, it probably doesn't matter too much exactly which one we choose, so there's not much to be gained from running lots of benchmarking tests to get accurate timing information. In most cases, the differences between plans are large enough that we can easily identify the best plan from a single benchmark run. Finally, we have to do some slight messing around to fix up the plans for the convolution step in prime base transforms using Rader's algorithm.

From a user's point of view, all of this complexity is hidden behind the `empiricalPlan` function: you call this with your input size and you get a `Plan` back, one that's hopefully close to the optimal plan that we can generate.

Once the optimal plan for a given input size has been determined on a given machine, it won't change, since it depends only on fixed details of the machine architecture (processor, cache sizes and so on). Instead of doing the work of running the empirical benchmarking tests every time that `empiricalPlan` is called, we can thus save the planning results away in "wisdom" files for reuse later on. Here's the code to do this:

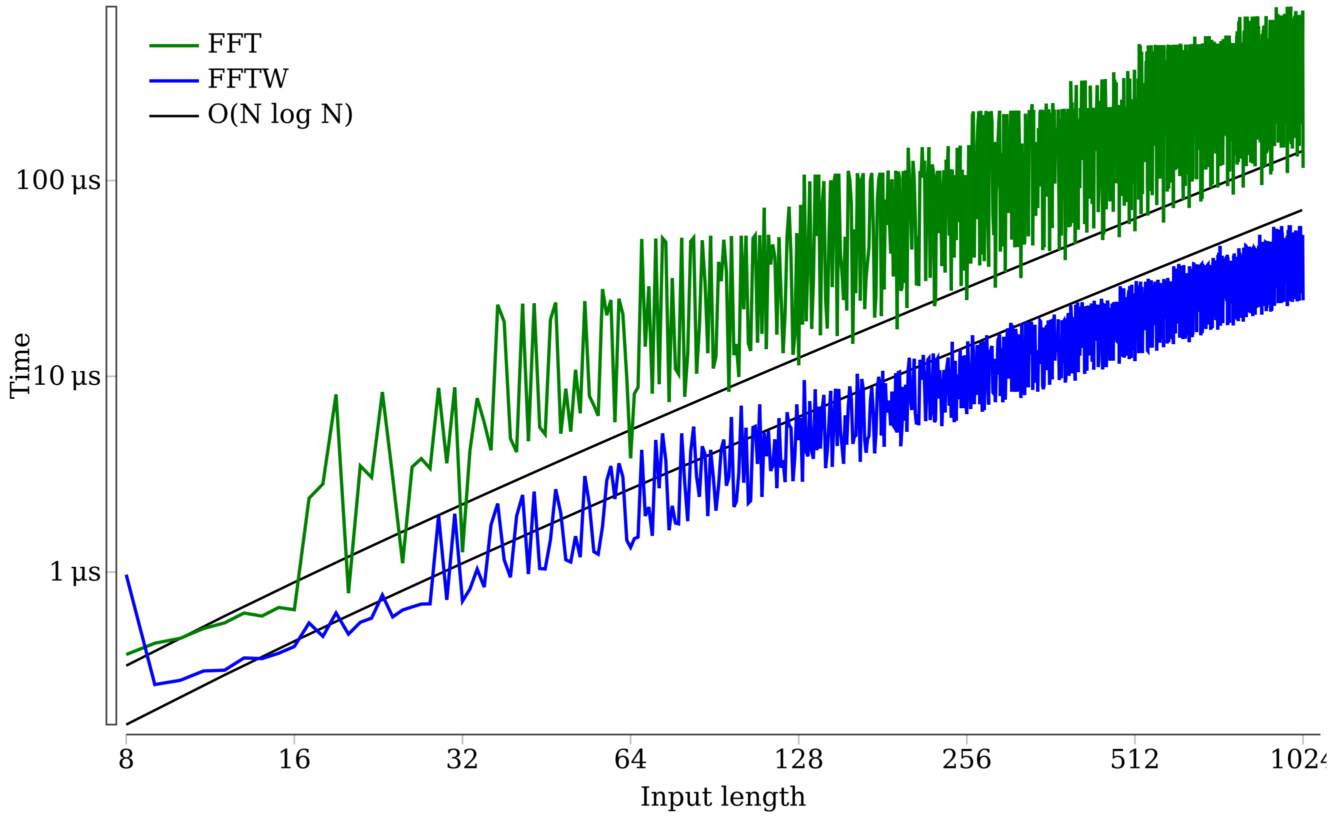
```
-- | Read from wisdom for a given problem size.
readWisdom :: Int -> IO (Maybe (Int, Vector Int))
readWisdom n = do
  home <- getEnv "HOME"
  let wisf = home </> ".fft-plan" </> show n
  ex <- doesFileExist wisf
  case ex of
    False -> return Nothing
    True -> do
      wist <- readFile wisf
      let (wisb, wisfs) = read wist :: (Int, [Int])
      return $ Just (wisb, fromList wisfs)
```

```
-- | Write wisdom for a given problem size.
writeWisdom :: Int -> (Int, Vector Int) -> IO ()
writeWisdom n (b, fs) = do
  home <- getEnv "HOME"
  let wisd = home </> ".fft-plan"
      wisf = wisd </> show n
  createDirectoryIfMissing True wisd
  writeFile wisf $ show (b, toList fs) P.++ "\n"
```

We save one file per input size in a directory called `~/.fft-plan`, writing and reading the `(Int, Vector Int)` planning information using Haskell’s basic `show` and `read` capabilities. Whenever `empiricalPlan` is called, we check to see if we have a wisdom file for the requested input size and only generate plans and run benchmarks if there is no wisdom. Conversely, when we benchmark and find an optimal plan, we save that information to a wisdom file for later.

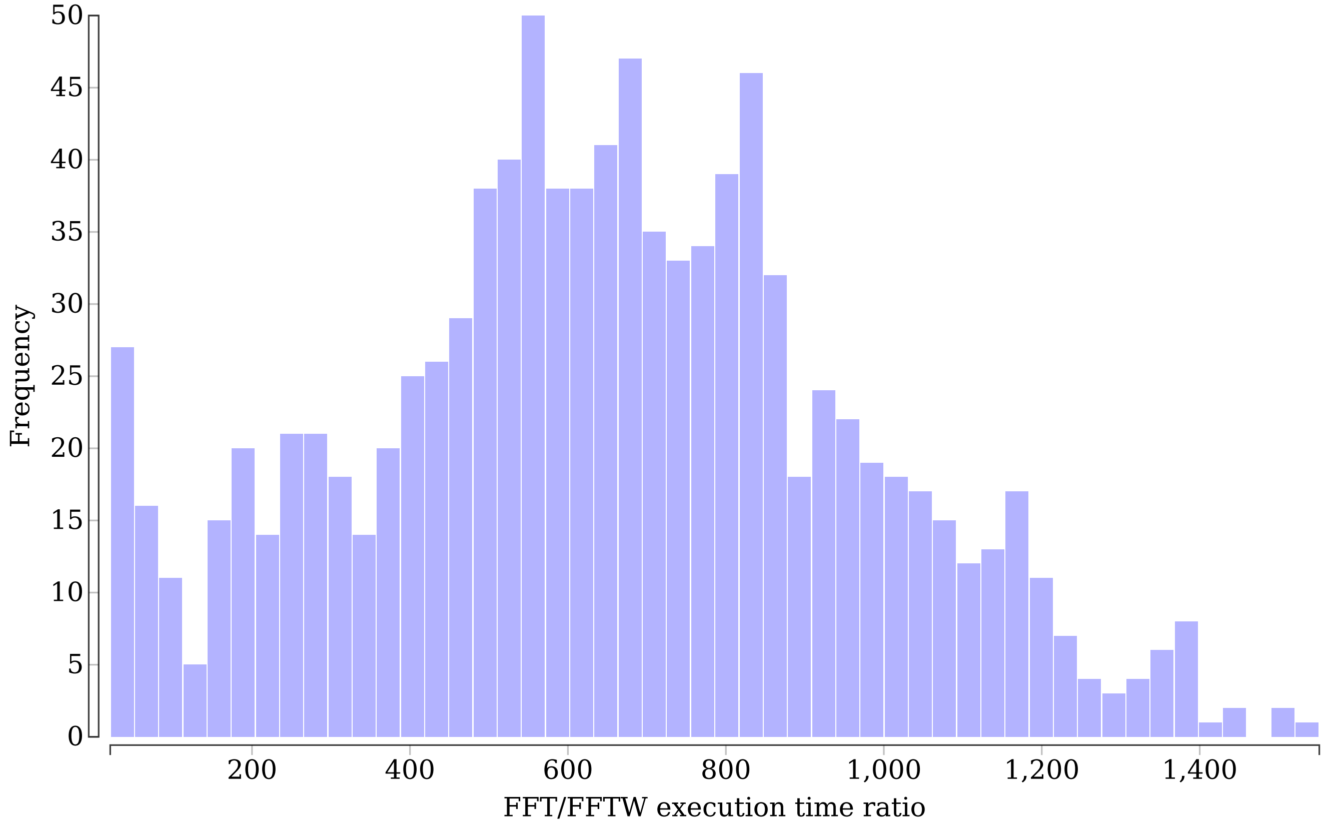
Benchmarking

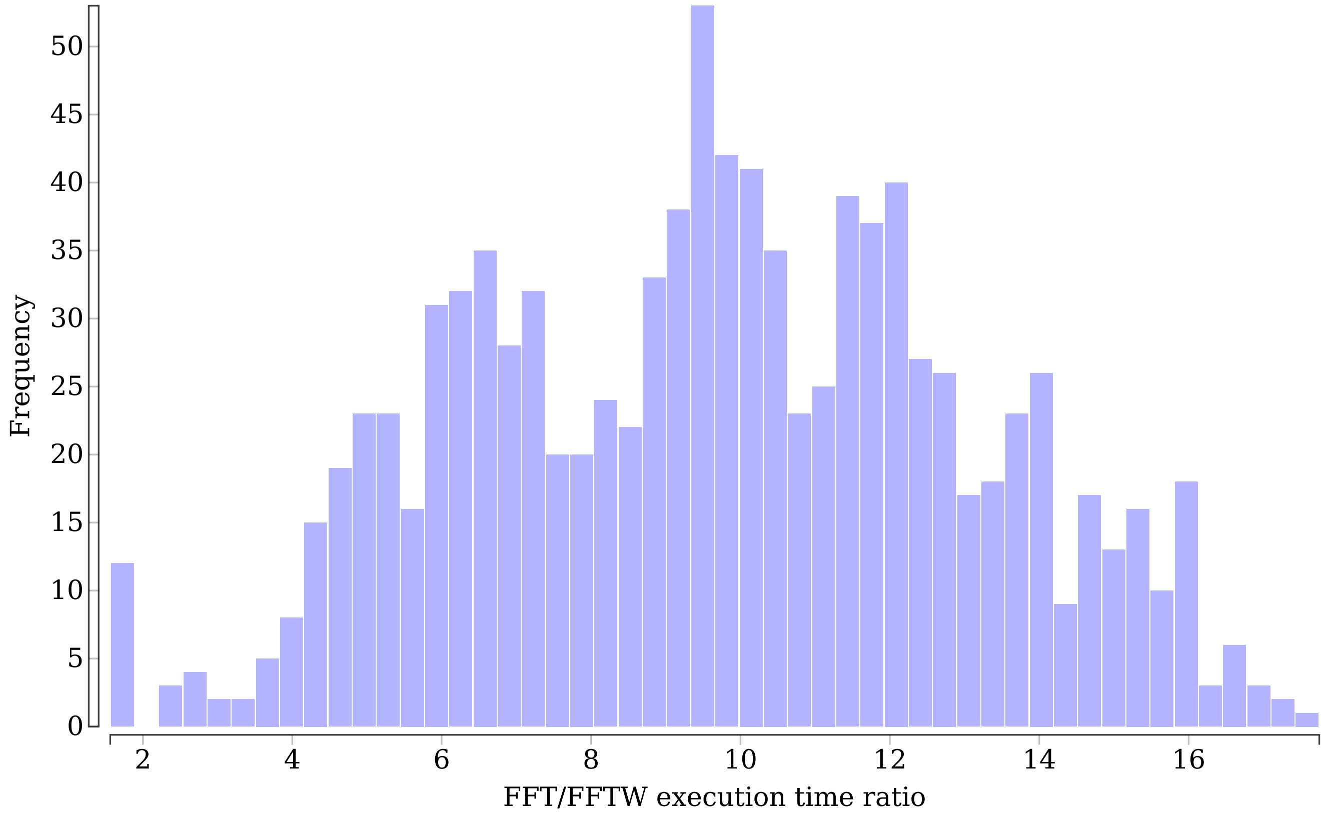
We can measure the performance of the `pre-release-3` code in the same way as we’ve done for earlier versions. Here’s a view of the performance of this version of the code that should be pretty familiar by now:

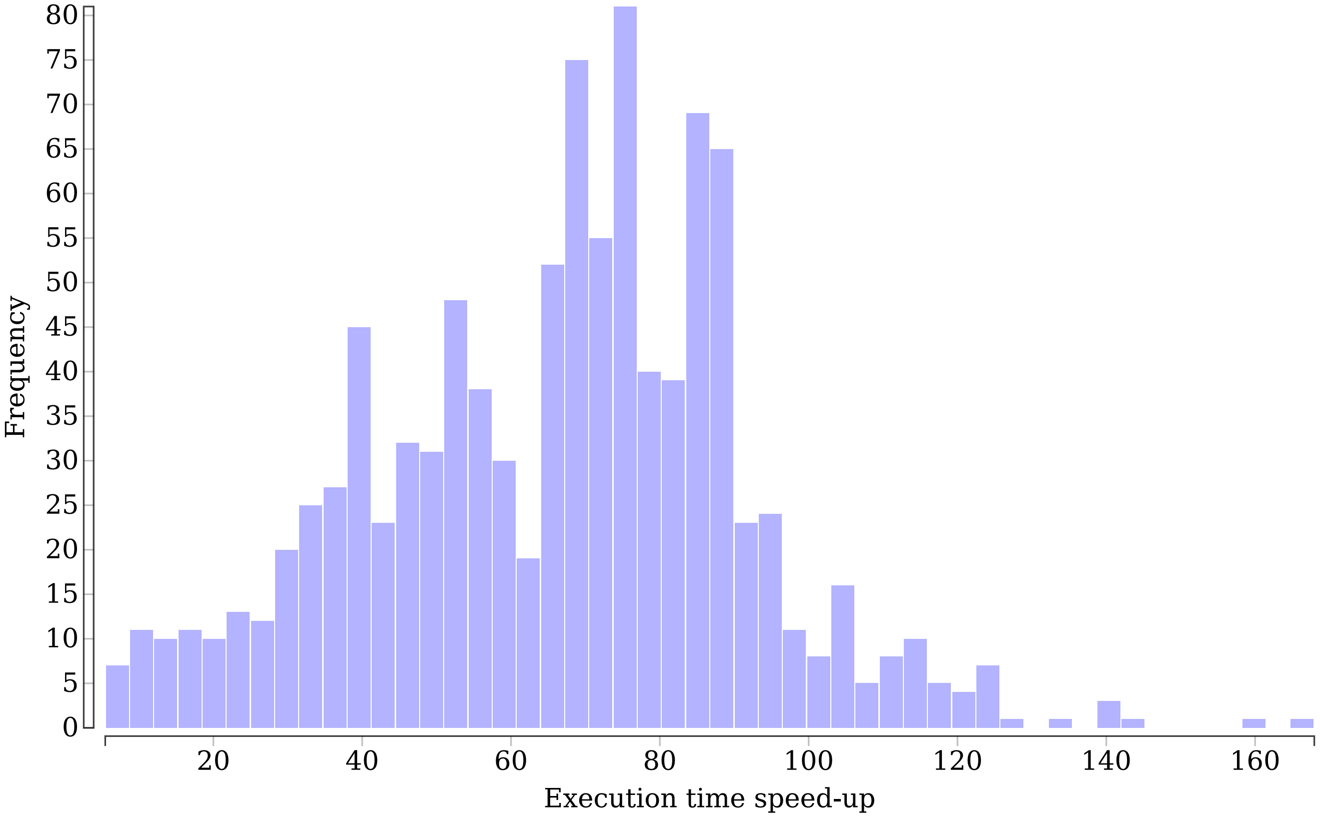


and here are the ratio plots showing the relative performance of the original unoptimised version of the code (`pre-release-1`), the current version (`pre-release-3`) and FFTW:

- pre-release-1
- pre-release-3
- Speed-up


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


Frequency
50
45
40
35
30
25
20
15
10
5
0
2
4
6
8
10
12
14
16
FFT/FFTW execution time ratio



It appears that the empirical optimisation approach we've taken here has been quite successful. The "pre-release-3" tab above shows that for most input sizes in the range that we're benchmarking, our code is around 10 times slower than FFTW, and never more than 20 times slower. In the previous article, we saw that, for the `pre-release-2` code version, most input lengths were between 40 and 100 times slower than FFTW. The "Speed-up" tab also shows that we've significantly increased the range of input lengths getting 50-fold or better speedups compared to the original unoptimised code.

Most of the remaining slower cases can be put down to our implementation of Rader's algorithm. When the input length N is not of the form $2^m + 1$, allocation is required of a zero-padded vector is required for the convolution in Rader's algorithm. It ought to be possible to avoid this allocation and speed up the code a little, which ought to help with some of the slower cases (most of which are either prime lengths, or involve a comparatively large prime factor).

In the next (and penultimate) article in this series, we'll clear up this issue a little, play with some compiler flags and catalogue the remaining opportunities for optimisation.

data-analysis haskell

0 comments

Sign in **1 person listening**

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

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

Powered by Hakyll