The ryg blog
When I grow up I'll be an inventor.

Notes on FFTs: for implementers

March 19, 2023
In the previous post I've talked about things you might want to know as someone who uses FFTs, this part covers all kinds of FFT implementation details, including the underlying reasons for a lot of the API complexities that showed up last time. I'll also give some recommendations on what I think are good ways write a FFT right now, and presumably also going forward.

Algorithm zoo

There's a ton of FFT algorithms to choose from, but practically speaking, the broad Cooley-Tukey family of algorithms (I include variants like split-radix as part of the general family) is by far the most important.

If you don't already know how CT FFTs work, I would recommend to avoid any sources that derive the algorithm by manipulating sums of trigonometric functions (doubly so if they do it with real sin/cos instead of complex exponentials). Way too much noise and index manipulation, nearly impossible to see what's going on. I would also not recommend the other extreme, sources that explain FFTs purely in terms of signal flow charts/butterfly diagrams, because while that form can be useful to look at *particular* transforms (until they get too large to comfortably fit in a sensibly-sized diagram anyway), the first few (or last few, depending) levels of the decomposition are in some sense special cases (because their twiddles are trivial), and with a flow-chart form that is most of what you see. Since you're forced to look at a concrete realization that is generally not too big and thus mostly special cases, it can be quite hard to see what the actual general construction is supposed to look like.

The least "noisy" introductory explanation I've ever seen for a CT radix-2 FFT (provided you know some basic linear algebra) wrote down a 8×8 DFT matrix as powers ω^0, …, ω^7 of the 8th root of unity and proceeded to find and use the symmetries on it, then explained the general rule once you'd seen the particular example (it's not too hard to figure out yourself once you start that way). I like this form: having the powers of the root of unity exposes the critical structure that is being used, but obscures the fact that of course ω^2 is i and ω^4 is -1, which is precisely the property that makes small signal-flow-graph based descriptions confusing to me because it leads to simplifications that break up the regularity somewhat.

If you have the required background (meaning you know what polynomial rings are and how they work), I've found ring-theoretic approaches the most insightful in terms of showing the actual structure of the problem without getting bogged down in the details (and in index manipulation) as tends to happen to varying degrees when working with sums of trigonometric terms, signal flow graphs or matrices. Honestly I think the most I've ever gotten out of a single source on the subject would be D. J. Bernstein's **"Multidigit multiplication for mathematicians" (https://cr.yp.to/papers/m3-20010811-retypeset-20220327.pdf)** (section 7), which covers the underlying algebra for many of the major FFT variants in about 2 pages.

DIT vs. DIF

All Cooley-Tukey family algorithms come in two dual variants, Decimation-in-Time vs. Decimation-in-Frequency. With the recursive decomposition as introduced by Gentleman and Sande, a DIT FFT for even N is decomposed into first computing two size-N/2 DFTs on the even and odd elements respectively. These two DFTs on the even and odd halves of the data can then be combined into a DFT for all N samples by first multiplying the odd-half coefficients by so-called "twiddle factors" and then adding and subtracting the even and twiddled odd halves from each other ("butterflies").

Splitting into even and odd halves is done at every level of this recursion, and always before any of the data is modified. We would like the actual computation parts of the kernel (at the various sizes) to work on contiguous elements; what this ultimately results in is a big data permutation at the start that does all of the even/odd splits at once (which turns out to be a bit-reversal permutation, where if indices i are in { 0, …, N-1 }, the item at position i is sent to the position `bit_reverse(i)` that literally reverses the order of bits in the binary expansion of the number.

Therefore this (now standard) version of Cooley-Tukey consists of first a permutation on the input, and then all the follow-up work can be conveniently done in-place. (We'll go into memory access patterns later.)

The dual version of this approach is Decimation-in-Frequency (DIF). In DIF, you start with butterflies, this time between the first and second half of the input. Then you twiddle the second half and finally recurse into size-N/2 DFTs on the first and second halves (which are already contiguous), and once those partial DFTs are done, you interleave them. Again this interleaving happens as the last step at every level of the recursion, so you can skip it in the middle of the recursion and end up with FFT coefficients out of order – bit-reversed in fact, so you have a bit-reversal permutation at the end. In short, it's the exact opposite of the DIT decomposition (if you invert a DIT FFT by reversing the flow graph, you end up with the corresponding DIF FFT, and vice versa).

The bit-reversal permutation is possible to do in-place, but it's annoying and relatively slow. DIT FFTs, which need to do the reordering first, often prefer to be out-of-place: in their first pass, they apply the input permutation while copying to the destination, and then can work in-place for the rest of the computation. This can be viewed as a feature by saying that they leave the original data intact. DIF FFTs also want to do their computation in-place and have the reordering last; so they either overwrite the input completely leaving a completed but permuted FFT in the original input, and then do a final reordering step that copies to the destination, or they can be – very conveniently – *fully* in-place and leave the coefficients in a permuted order. This latter variant is the type of FFT that can give you permuted-order coefficients more quickly, which you might not mind as long as you're only doing convolutions with it. The inverse would then use the corresponding DIT factorization, skipping the initial permutation step, and in fact ending up with zero shuffling whatsoever.

This is nice in principle, *however* the bit-reversed order is annoying for library users that do want FFT coefficients in the conventional order, and it gets worse when other types of decomposition (e.g. radix-4 or split-radix variants) are used which have a different "natural" order for their coefficients to end up in. In short, using DIF with whatever radix you want and then returning the unpermuted coefficients does save some work, but it does so at the expense of exposing a lot of implementation details, so it's a bit dicey to expose as part of a library API.

Finally, the elephant in the room: multiply-adds. A radix-2 DIT step ultimately boils down to a twiddle of the odd half then a butterfly, which works out to (the additions, subtractions and multiplications here are using complex numbers, and w is a twiddle factor)

```
out0 = in0 + w*in1
out1 = in0 - w*in1
```

This looks as though it might nicely map to multiply-add operations where available, and in fact it does, although not exactly in the obvious way (I'll get to that later). For a DIF version, remember that the order of operations was reversed, and we first add/subtract and *then* twiddle, so we get

```
out0 = in0 + in1
out1 = (in0 - in1) * w
```

which is not in multiply-add form. We might think this is a temporary setback and we just have to factor this w into the next operation using out1 to get our multiply-adds, but that doesn't work either: out1 ends up going into another DFT layer that looks just like ours, and both of its inputs need to be twiddled, so if we delay multiplying by w, we get something like

```
out0 = deferred_w0*in0 + deferred_w1*in1
out1 = deferred_w0*in0 - deferred_w1*in1
```

Note we have two twiddles now because in general the twiddle factors feeding into the two inputs at the next level will not be the same. We can do one multiply and then two multiply-adds, but this is not great at all, and we still are deferring one twiddle multiply for the next iteration. (Maybe there is some good trick here that I'm missing?)

In short, with DIT we get a form that is amenable to using FMAs (or other types of multiply-adds) throughout; with DIF, not so much. In my opinion that severely limits the usefulness DIF + out-of-order coefficients as an attractive strategy to avoid permutations for convolutions, because now your DIF FFT potentially performs quite differently from its DIT inverse that shows up in the IFFT.

Therefore, these days, I'd stay away from permuted coefficient orders at API boundaries and just eat the permutation (and/or copy) to produce conventional-order results.

Different radix, arithmetic complexity

The "canonical" way to write a Cooley-Tukey FFT for power-of-2 sizes uses a radix-2 decomposition, meaning at every level we subdivide into 2 sub-FFTs of size N/2. Another popular variant uses radix-4 steps, which do a slightly different calculation to immediately subdivide into 4 sub-FFTs of size N/4. This has slightly more complicated logistics, a different output permutation (corresponding to reversing the digits in a base-4 expansion of the index this time), and can reduce the number of multiplies relative to radix-2, which is what originally sparked the interest in it – the number of multiplications used to be the primary cost in the FFT algorithm, especially in the absence of hardware multipliers, much less pipelined ones.

There's further refinements such as split-radix, which is a hybrid between radix 2 and 4, using the fact that part of the computation in a recursive decomposition is more efficiently done as radix-2 and the remainder is more efficient as radix-4 (as you might expect, this introduces further complications to the coefficient permutation, which for split-radix is quite complex). There's a whole sub-genre of hunting down individual stray multiplications: complex multiplications with twiddle factors that are 1, -1, i, or -i are obviously unnecessary and can just be turned into real-valued additions and subtractions (optional with swapped or sign-flipped real/imaginary parts) with no actual multiplications. Slightly less obvious, multiplications with twiddles that correspond to +-45 and +-135 degrees (+-1 +- i) / sqrt(2) multiply both the real and imaginary part by the same value and can use a slightly cheaper rotation. And then there's some more tricks with leaving values scaled to reduce the number of multiplies further.

All these tricks can be used systematically in the first few (or last few, depending) steps where these are all the twiddle factors that appear. Scalar implementations can also make use of them at other levels of the decomposition to special-case individual values, but for SIMD/SIMT architectures trying to special-case these rotations in the middle of larger blocks hurts way more than it helps.

More to the point, *all* of these techniques are about minimizing arithmetic complexity, and more specifically, reducing the number of multiplications, since the number of additions is the same for all of them – and also larger than the number of multiplies. And that's a crucial caveat since, generally speaking, we're on a very different cost model from counting number of individual floating-point operations these days. Let's give a few examples to see how this matters:

Intel Ivy Bridge (2012). Older uArchs like Sandy Bridge, Nehalem etc. are similar, all the way back down to the original Pentium Pro (1995). FP/SIMD multiplier on port 0, FP/SIMD adder on part 1, both fully pipelined, can independently receive an instruction every cycle. Thus, in the limit, FP ALU throughput is limited by whichever type of operation we have more of, and that's additions in this case.
Intel Haswell (2013), Broadwell (2014). FP/SIMD fused multiply-add unit on ports 0 and 1; FP adder on port 1. That's right, HSW and BDW can do two FMAs per cycle (256b vector ones at that, too!) but only one FP add. Once again, multiplies are less of a problem than adds, funnily enough.
Intel Skylake (2015) through Rocket Lake (2021). FP FMA on ports 0 and 1, adds also accepted on ports 0 and 1. No more add imbalance, but if we're chasing op counts for throughput, it's far more helpful to use FMAs efficiently than it is to minimize multiplies. (As of 2022 Alder Lake, we do get a dedicated FP adder on port 5.)
ARM Cortex-A55 (2017). Extremely common mobile device "little" core. Two SIMD/FP pipes, the FP portion is effectively 64b wide, can do two multiplies/adds/FMAs per cycle, so again the biggest bang for the buck is by using FMAs. The newer "medium" Cortex-A710 (2021) is still two FMA-based FP/SIMD pipes although they are fully 128-bit now.
Pretty much any GPU or DSP built within the last 20 years (and that's conservative): built around pipelined multiply-accumulate operations. Not necessarily fused floating-point multiply-adds (which is a specific thing meaning there is only one rounding stage), but some form of multiply-accumulator, maybe fused, maybe with intermediate rounding, maybe with weird truncations in the middle, maybe fixed point, but there is some kind of multiply-add in there. If we do a multiply, we get a corresponding add for ~free, as long as we have a suitable candidate.

You get the idea. For a multiply-add based architecture, the operations we're counting – if we're counting anything – should be multiply-adds and not individual multiplies or adds. For a machine without multiply-adds but with separate superscalar mul and add pipes, it matters what the ratio of mul to add resources is; for integer (fixed-point), you usually get more adders than multipliers so the number of multiplications is the thing to count, but for floating-point, it was a common design point for a long time to have one pipelined FP multiplier and one pipelined FP adder, and in that case the bottleneck is whichever pipeline gets more work, which for FFTs is generally the adder. Now if your target is single-issue, has a different pipeline, or doesn't have pipelined multipliers (or adders for that matter), the classic cost model might be relevant to you, but for newer targets with beefy vector pipelines, nickel-and-diming individual multiplies is worse than useless (since it actively introduces irregularities that are more expensive than just doing the math) and utilizing FMAs (where available) well is more important.

For what it's worth, the classical operation costs of a power-of-2 sized FFT for the basic algorithm variants – just the leading terms:

Radix-2: ~5N log2(N) operations
Radix-4: ~4.25N log2(N) operations
Split-radix: ~4N log2(N) operations

In my experience anyway, radix-2 and radix-4 can make use of all-FMAs quite easily, split-radix gets a bit iffy, so these days, if you're gonna pick one, I think radix-4 is a solid choice.

How do I use FMAs in this anyway?

Right. So, let's write this out. For a DIT radix-2 step (radix-4 is analogous, just more of everything, you can figure it out), the *complex-valued* version as given above is

```
out0 = in0 + w*in1
out1 = in0 - w*in1
```

but of course we actually need to implement this in terms of individual *real* operations and expand out that complex arithmetic, which turns this into

```
out0.re = in0.re + w.re*in1.re - w.im*in1.im
out0.im = in0.im + w.re*in1.im + w.im*in1.re
out1.re = in0.re - w.re*in1.re + w.im*in1.im
out1.im = in0.im - w.re*in1.im - w.im*in1.re
```

which makes it look like it's 8 multiply-adds (I'll use multiply-adds and FMAs interchangeably in the following, even though it's not technically correct), but that's clearly not a good way to do it. We could compute the shared "w*in1" part once, which takes 2 multiplies and 2 FMAs, and then we still have 2 adds and 2 subtractions left, so that's not better in the "multiply-adds cost the same as isolated multiplies or adds" model. Now, what goes wrong here? Clearly, through the first half (the computation of `out0`), we can use 4 FMAs, and there's no redundant computations so far, so that all seems fine. But in that second half we're mostly recomputing stuff we already computed in the first half. Luckily, there is a trick, which is actually easier to see in the complex-valued version above: we can compute `out1` via

```
out1 = 2*in0 - out0
```

instead. This is a multiplication by a pure real number so we can do this using 2 more FMAs, giving us this computation once we expand out into reals:

```
out0.re = (in0.re + w.re*in1.re) - w.im*in1.im
out0.im = (in0.im + w.re*in1.im) + w.im*in1.re
out1.re = 2*in0.re - out0.re
out1.im = 2*in1.re - out1.im
```

6 FMAs total. For comparison, the "classic" version using just regular multiplies and adds ends up with 4 multiplies and 6 adds/subtracts total, so our op count goes down from 10 to 6, quite the a big change (NOTE: none of this is new and certainly not my invention, I'm just trying to spare you the time of digging it all up).

Oh, and regarding that complex multiplication in the middle: there are **old tricks (https://en.wikipedia.org/wiki/Multiplication_algorithm#Complex_number_multiplication)** to replace the 4-mul-2-add complex multiplication with 3 muls and 5 adds, 2 of which can be precomputed if one of the factors is known, as is the case for twiddle factors, so you end up with 3 muls and 3 adds. However, as you might expect, this algorithm is more irregular, needs 3 floats for the twiddle factors instead of 2 (and also prevents you from using tricks that might shrink this further), and even though it's 3 multiplies and 3 adds it's not in the form of 3 multiply-adds, although you can reduce it to an add, a multiply, and then 2 FMAs, which sounds good in theory, except…

Is minimizing arithmetic operations actually the key?

In a word? No, not usually.

Here's the problem. Take the 6-FMA snippet above, and assume we're working scalar for now. That's all the math to process a pair of complex values in a radix-2 DIT FFT step. You have to load the real and imaginary parts of both inputs (4 loads), load the real and imaginary parts of the twiddle factor (2 loads), and finally store the real and imaginary parts of both results, normally to the same place you just loaded them from (4 stores). In total, 6 loads, 4 stores and 6 arithmetic operations per 4 floats processed in each iteration.

This is not really a compute kernel so much as a memory streaming exercise with a part-time gig in arithmetic.

I was looking at a scalar version for simplicity, but the picture is exactly the same when looking at a vectorized version that processes multiple samples at once (because FFTs are so regular, vectorizing them is, for the most part, straightforward). It also doesn't actually matter here whether a vectorized version uses a split or interleaved layout internally; interleaved has the real and imaginary values next to each other, but having complex numbers packed that way in SIMD registers means that instead of N "real-valued" SIMD lanes, we effectively end up with N/2 "complex-valued" SIMD lanes. The amount useful work per individual float remains unchanged (at best; in practice, using an interleaved format inside the kernel often picks up overheads from elsewhere. I'll get to it later).

What we're looking at here is the real problem with radix-2 factorizations, which admittedly is especially amplified when using the FMA versions of the core kernel: the ratio of computation to memory operations is unfavorable. Now exactly how bad this is or isn't depends on what machine you're targeting, and what the proportion of compute to memory resources is, but more loads/stores than arithmetic ops (or even a 1:1 ratio as you would get without FMAs) is never good news. This is also the real reason why the 3-mul 3-add complex multiplication which needs 3 floats per twiddle is not that attractive: it decreases arithmetic and increases memory load even further.

Radix-4 happens to have fewer multiplies, but much more importantly, it does the work in roughly half the number of passes, and thus halves the number of coefficient loads/stores. You can reduce this even further by using yet larger radices (if you have enough registers available to keep all those values around), but the biggest single jump is from radix 2 to radix 4.

Another note on radices: there's a distinction between radix-4 and what I've seen called radix-2^2, a name that feels clunky but that I'll keep using for lack of a better suggestion. The former is an actual radix-4 step, which is to say, for DIT, we have 3 twiddles "in front" into a 4-point DFT kernel (all multiplications by 1, i, -1 and -i, which are all not actual complex multiplications, just swapping values around and some negating). The latter is essentially two unrolled radix-2 passes chained back to back. In terms of arithmetic complexity, the difference is that a "proper" radix-4 kernel has 3 twiddle multiplies, while the radix-2^2 variant contains 4 instances of a radix-2 sub-kernel that each does one twiddle multiply. However, classic radix-4 needs to load those 3 twiddles from somewhere; radix-2^2 only needs to load 2 different twiddle factors (which both get used twice). Considering our woes with the amount of computation vs. amount of memory operations required, and that FMAs change the trade-offs for the effective cost of multiplications in a radix-2 kernel, radix-2^2 is a very interesting candidate![1]

Point being: we can save a fair amount of unnecessary memory operations by using larger radices. This can use actual larger-radix algorithms, or it can use a radix-2^k-style approach where we may just load a bunch of values into registers and complete 2, 3 or more levels of the factorization at once while they're there. The difference in twiddle factor loads really starts to ramp up as we do more of these: a "real" radix-8 FFT needs to load 7 complex twiddles, a radix-2^3 step only needs 3 (and classic radix-4 into radix-2 would need 4). Combining steps like this, provided we have sufficient registers, lets us increase the amount of computation in between loads and stores to a level where we're actually keeping the FP/SIMD units respectably busy and not just spamming memory operations.

Memory operations, the second

So far, I've been looking at memory operations purely in terms of the number of instructions that need to execute. But what about actual cache effects?

Hoo boy. A lot has been written about this subject, too much of it overgeneralizing, using really broad strokes or just working from unstated, questionable assumptions. Lest I go full rant mode, I'll try to avoid editorializing for the most part. A lot of what you will read on this topic, be it in textbooks, on the web, in forums (remember forums?) or papers, is either questionable or just plain bad.

The most important news first: if you're using either the standard DIT or DIF factorizations which are in-place except for that initial (or final, depending) permutation step, then as long as you're working on a contiguous array and the transform size is small enough so the coefficients fit in about half your L1 data cache, you're pretty much going to be fine no matter what. Your data comfortably fits in the L1D cache, will be repeatedly accessed and kept hot, and nothing bad is going to happen.

L1D caches on CPUs these days are usually 32KB or larger, and that's been true for a good while (it was already true on desktops 15 years ago, for example; these days mobile devices have also caught up in that regard). That's plenty for a size-2048 complex FFT with 32-bit floats. That covers a lot of the sizes you're going to see in practice! Beyond that, here's some of the statements I've seen, and what I think are the important caveats to all of them:

"DIF has better cache behavior than DIT": this one's a bit tricky. With power-of-2 sizes, all FFT algorithms are going to do a lot of memory accesses with exact power-of-2 strides between them, and that is a recipe for running into "fun" memory subsystem problems. As discussed before, both DIF and DIT run the same steps, just in reverse order (relative to each other). What is true is that a DIT FFT has a mandatory permutation step in front, whereas DIF can avoid it entirely if reordered coefficients are acceptable. Also, in-place bit-reverse permutations are a tricky mess so DIT FFTs are more likely to work out-of-place and do the permutation as part of a copy – and it is easy to do that permutation in a way that has exceedingly poor memory access patterns. So, yes, a reordered-coefficient DIF FFT is easy to implement fully in-place and efficient, and it can avoid the reordering step with its memory access troubles entirely. A DIT FFT is more likely to be out-of-place and do extra copies, and if the reordering isn't done carefully, it can have bad cache performance – this is a solvable

problem, though.[2] And DIF has its own problems which I already went into earlier.

"Iterative FFTs are bad for the cache, use a recursive decomposition": iterative FFTs will generally have an outer loop over the pass index (which determines current transform size), and a nested loop inside it that does all sub-FFTs of that size (either as function calls or just have that one loop handle all of them). Recursive approaches use, well, recursion; a size-N transform will internally call into smaller transforms for part of the data, and then either pre- or post-process them. That means that for transforms too large to fit into the L1D (or L2, or L3, …) cache, a straight iterative approach will stream the entire dataset into and out of the cache once per pass, while a recursion subdivision will eventually narrow down to a transform size that fits in the cache, and then all the sub-transforms on that subset will have the data warm in the data cache. Don't get me wrong, this is good, but the straight linear loop structure of iterative FFTs is also good, and one recursive call per length-8 or length-16 or whatever your "leaf" transform size FFT is isn't awesome. Mostly, I disagree with the unspoken assumption that they're mutually exclusive. Iterative FFTs are great at handling the tons of small sub-FFTs near the leaf levels of the decomposition. Recursive FFTs are cache-oblivious and automatically "right-size" themselves. So just use both! Recurse down to a medium transform size that you're sure is going to fit in the cache (I'd say somewhere from 256 to 1024 complex elements, depending on just how low-spec your smallest target is), and you can let the iterative approach make short work of all the tiny sub-FFTs while still getting all the nice automatic adaptation that recursive decomposition provides for larger sizes.

Over-generalization from old HW: this is in some textbooks, papers and forum posts. Especially late-80s and early-90s RISC workstations tended to have small, direct-mapped data caches that had completely pathological behavior on memory accesses with the right (or rather wrong) power-of-2 strides, and FFTs tend to hit all of these potholes one by one. People learned some very painful lessons in those days when they got burned quite badly by this, but set-associative caches with multiple ways are not a luxury feature these days, and they have substantially defused this particular problem. It's still possible to hit pathological behavior, but you have to try a bit harder, and there's decent ways to work around it.

"You should use exclusively cache-oblivious approaches/the four-step algorithm/insert other plug here" – use whatever algorithms you want, but I would recommend doing your own research and not just blindly following recommendations that some rando typed into a forum in 2002. There's a lot of algorithms out there but many of them are quite old and written in a very different situation without floating-point hardware, with slow multipliers, little or no access to instruction-level parallelism, and very different relative costs of compute to memory. The good news is that large FFTs used to be a big-iron HPC problem and these days, even legitimately large FFTs over many million data points can be computed in a fraction of a second on a battery-powered cellphone. It's pretty hard these days to get into a problem size for FFTs that actually runs into some serious hardware limits. The bad news is that a lot of what's been written about FFTs dates back to times when this was a much bigger concern and hasn't been updated since. This post is, in part, me writing up what I still remember from about 4 weeks worth of reading the literature back in 2015, trying to give you the bits that I think are still useful and applicable and intentionally avoiding all the material that is, I think, past its sell-by date.

Finally, a disclaimer: I have personally written and used FFT kernels for small to medium problem sizes (everything still fits comfortably in a 256k L2 cache). I have never actually had reason to use FFTs with transform sizes above 32768 in an application, nor have I optimized or debugged an implementation that provides them, so anything that concerns large transforms specifically (which shifts us even more into a memory-bound scenario) is not something I can comment on.

Stockham's auto-sorter

There's a Cooley-Tukey variant usually credited to Stockham that avoids the initial (or final, for DIF) bit-reversal, digit-reversal or whatever permutation by instead performing the reordering incrementally, interleaved with the processing passes. This was popular on vector machines and is sometimes suggested for SIMD.

It works just fine, but I'm not a big fan, for two reasons. The first is that Stockham gives up on any in-place processing entirely and instead "ping-pongs" between two buffers on every pass, which means it has about twice the active working set in the cache as an in-place algorithm (and more traffic in upstream caches or memory when it doesn't fit anymore). The second is that this picks up extra data-shuffling work in every single pass; a batched reorder can be combined with other input processing (for example to turn data into an internal format), maybe an initial radix-2 or radix-4 pass, and also be done in a way that avoids thrashing the cache unnecessarily, because it's a data-movement pass at heart and can be designed with that in mind. Then the actual workhorse FFT passes can focus on the math and memory access and don't have data shuffling in the mix as well.

SIMD

So far, I've discussed everything pretending we're working with scalars. But FFTs (power-of-2 sized ones, anyway) are generally pretty easy to vectorize.

If you have the data given as separate arrays for real and imaginary parts, it can be almost as simple as replacing every scalar load/store with a vector load/store and every scalar arithmetic operation with the corresponding vector arithmetic operation. You might need to handle the first few (or last few, depending) passes specially, where you're working first with pairs of adjacent elements and then groups of 2, 4 etc., but once the distance between elements you're processing becomes larger than your vector width (which is what most of your passes will be working with), it's straightforward.

It does mean that your initial (or final) few passes end up special. However, these are the passes right next to the input/output permutation, and combining the early special-case passes with the data permutation tends to solve problems in both: the data permutation usually involves some kind of SIMD transpose, and moving some of the math across that transpose boundary separates what would otherwise be intra-vector "horizontal" dependencies out into multiple vectors. As for the data movement, that is ordinarily just loads, stores and shuffling; having some arithmetic in there helps the instruction mix and makes it less likely to be bottlenecked purely on memory operations or shuffling.

The pure deinterleaved format is probably the easiest to understand conceptually, but it's not very commonly used. Fully interleaved (all real, imaginary value pairs) is the other extreme. That's a very common format for complex-valued input data, and it usually works reasonably well to do the FFT in that form directly – for complex multiplications specifically, you do tend to need some amount of shuffling and special instructions for complex multiplication ("addsub" and "fused multiply with alternating subtract/add", most notoriously), but these instructions are often available. That said certain things which can just be dealt with implicitly in a fully deinterleaved form (such as swapping real and imaginary parts, or computing a complex conjugate) turn into actual instructions when working interleaved, so there's a lot of small random fix-ups involved; again, I'm not a big fan.

Yet another option is to keep the data interleaved, but at the granularity of full vectors (or some larger granularity that is a multiple of the vector size). For example, for 8-wide SIMD, you might store real parts of elements 0-7, then imaginary parts of elements 0-7, then real parts of elements 8-15, and so forth. This is still addressed by a single base pointer and has nice data access locality characteristics, but is just as (if not more) convenient as fully deinterleaved for SIMD code. The trade-off here is that your vector width (or other interleaving granule size) now becomes part of your interface; I like this format *inside* FFT kernels, less so as part of their public interface. (I've been experimenting a bit with this though. Maybe I'll write a follow-up on this in particular, some day.)

Whenever you don't work internally with the format you have in your API, there's going to be some translation at the ends. One end already has the input permutation and can generally rewrite the complex number format into whatever you'd prefer with no trouble. Then for the output you need to translate it back; ideally this is fused into the last pass, but one problem that does crop up with these SIMD variations is that you already have 2 or 3 versions of everything (data permutation combined with initial passes, then a standalone radix-4 or whatever pass, and adding another "final pass" with a different permutation on output can get to be a bit of a chore. It's up to you how many variations you want to generate (or hand-write if you're into that sort of thing); personally I tend to avoid having too many pre-built combinations and would rather live with a bit of inefficiency at the interface boundaries than dealing with the headache that is a combinatorial explosion of algorithm variants.

Non-pow4 sizes, inverses, real-input or real-output FFTs, trig transforms, etc.

Non-pow4 first: anything based on radix-4 (or radix-2^2) factorizations needs to deal with the fact that half of all powers of 2 aren't powers of 4. For those, you're going to need one radix-2 (or radix-8, or radix-2^3) pass in there. There's not much more to it than that, but it seemed worth writing up just to be explicit.

IFFTs are usually the same transform, just with the twiddles conjugated (scaling with 1/N or similar is often left to the caller). There's other ways to reduce IFFTs to regular FFTs by swapping around coeffs etc. but I'd advise against it; depending on how your twiddles are stored, you might be able to produce the conjugate twiddles easily, but it's also not too bad to just have a second version of the FFT kernel around for inverse transforms.

Real-input/real-output, well, there's multiple ways to handle that, but the easiest is to "bitcast" the real input array into a size-N/2 interleaved complex number array, compute a complex size-N/2 FFT (which due to linearity gives FFT[even_reals] + i*FFT[odd_reals]), and then do a final radix-2 pass to combine the even/odd halves and compute the actual outputs. The inverse of that does, well, the exact same steps in reverse. I'll not go into the details here but this is a classic algorithm. It's not the most efficient but it's reasonable and easy.

Other trigonometric transforms can also be reduced to inner FFTs with pre- and post-passes like that. If you have a decent FFT around, I've found this generally preferable to using specialized O(N log2(N)) algorithms specific to these transforms; trigonometric transforms generally have such direct algorithms, but their structure is almost always less regular than a FFT, so reduction to whatever-size FFT doesn't have a lot of wasted work and then pre-/post-passes is usually my go-to.

Fixed point

I haven't mentioned it at all. This omission is by design, not out of neglect. Floating-point and SIMD floating-point is widely available these days, and fixed-point signal processing is becoming increasingly niche. Some of the trade-offs are different with integers (adds are cheaper; DSPs usually have integer multiply-accumulate but many integer or SIMD integer instruction sets do not) and all I'll say on the topic is that the last time I've had to deal with a fixed-point FFT is about 15 years ago, and I haven't missed it. Good luck to everyone who still has to deal with it on a regular basis, but I'm just happy that's not me anymore.

Twiddle storage

Twiddle factors are, essentially, a sine/cosine table. There is definitely a temptation to try and get cute here – maybe try to save some memory by only storing part of the values and inferring the rest by symmetry, or by realizing that the twiddles for FFTs of size N/2 are just all the even-numbered elements of the twiddles for size N here, and so forth.

In short, resist the temptation to do anything fancy with them. You want them in a form that is easy to load and doesn't require a lot of bookkeeping, and trying to exploit symmetries too much to save on your sine table storage will probably not save you all that much memory but very well might make your hair go prematurely gray.

There's an exemption for *really* straightforward stuff. For example, if you're working in deinterleaved form, you need imaginary parts of twiddles (=sines) and real parts of twiddles (=cosines) both. Since cos(x) = sin(pi/2 + x), you can just store a sine table and use it for both, starting the cosine lookups a quarter of the way in. That is simple enough to not cause trouble (in my experience anyway), and it will also helps your data cache footprint, so it's actually worthwhile.

Twiddles are one aspect where my lack of experience with large transform sizes shows: I expect that for large enough sizes, once your working data set and your twiddles are both streaming through not staying in the relevant cache levels, the extra traffic is an issue and you'd much rather spend some more arithmetic in the FFT kernel to compute twiddles (incrementally or otherwise) than eat avoidable extra loads from memory. However, I've not been in a situation where I've had to care, so I don't really know one way or the other.

Takeaways

This is a long post and I've been at it all day, but I'll try to give you the summary of my personal recommendations:

Cooley-Tukey DIT is, on balance, probably your best bet.
Nonstandard-coefficient-order FFTs are cute but probably more trouble than they're worth these days.
Do use FMAs where available (which is quite common these days), and if you design any FFT code now, plan with them in mind. Also, FMAs are the great equalizer as far as the cost of different factorizations is concerned.
Don't sweat old-school operation counts, they're optimizing for the wrong things these days.
Prefer radix-4 (or radix-2^2), less so for lower number of multiplies, and more because having fewer passes and fewer loads/stores is good and memory operation count is not negligible for this.
You don't have to pick strictly between iterative and recursive approaches, and you can get the best of both worlds by recursing for larger to medium sizes, and handling the small sub-FFTs iteratively.
Do worry about number of memory accesses and your access patterns, especially for things like digit-reversal permutations.
SIMD: yes please; Stockham: probably not.
Data layout-wise, it's really up to you.
Twiddles: Keep it simple, stupid!

And I think that's it for today!

Footnotes

[1] You can also get a "proper" radix-4 kernel with 2 complex twiddle loads instead of 4. The high concept is that instead of twiddling with ω_N^k, ω_N^{2k} and ω_N^{3k}, you can substitute the 3k twiddle with -k – which is just the complex conjugate of the first twiddle, so that doesn't need a separate load. This substitution works fine but does affect the coefficient permutation (are you starting to see a pattern emerge?); see e.g. Bouguezel, Ahmad, Swamy, "Improved radix-4 and radix-8 FFT algorithms", 2004 IEEE International Symposium on Circuits and Systems. The same trick can be used on split-radix factorizations to yield what is now usually called the "conjugate-pair split-radix factorization". That said, as I keep repeating, with a FMA-counting cost model, the arithmetic operation difference between a radix-4 and radix-2 decomposition doesn't really materialize, and radix-2^2 comes out looking pretty good.
[2] Not to leave you hanging with a mysterious statement, one possible approach is in Blake, Witten, Cree, "The Fastest Fourier Transform in the South", IEEE Transactions on Signal Processing, Vol. 61 No. 19, Oct 2013 – in short, be careful about how you implement the permutation, be sure to grab some adjacent data while you're there, and also, once you have that data in registers, might as well do an initial radix-2 or radix-4 pass along with the reordering.

From → Coding, Maths

2 Comments

1. **FordPerfect permalink**
   `out1.im = 2*in1.re – out1.im`. Typo?

   Does `out1 = 2*in0 – out0` trick significantly affect FFT accuracy? Incidentally, you don't seem to mention accuracy at all in the post. A lot of implementations do seem to get it "wrong" (assuming one cares, in the first place), according to FFTW accuracy charts.

   Interestingly enough, for large (>L1) sizes, in-place permutation can be faster than out-of-place, or at least it was in my tests, with implementation based on "Towards an Optimal Bit-Reversal Permutation Program" by Larry Carter and Kang Su Gatlin (is there a better way to do it, these days?).

   Reply

2. **slembcke permalink**
   "There's other ways to reduce IFFTs to regular FFTs by swapping around coeffs etc. but I'd advise against it"

   One of the other "neat" ways to calculate the IFFT is to reverse the order of the signal. My implementation uses DIT, so I just reverse the values during reordering/scaling. It doesn't even need an extra line of code! :)

   Also, I had no idea that non-uninform block convolution was a thing from your previous post. I always wondered how insanely long reverb effects worked in real time.

   Reply