CHORD GPU upchannelization

kmsmith

December 30, 2022

Contents

1 Executive summary

The GPU correlator receives baseband data in "coarse" frequency channels with width 586 kHZ. In several places, we'd like to use higher frequency resolution than this.

- For the FRB search, we plan to use a level of upchannelization which is very frequency-dependent. At the bottom of the CHORD band (300 MHz) we might upchannelize by a factor ∼128, and at the top of the band (1500 MHz) we might not upchannelize at all!
- For the 21-cm galaxy search (and possibly RRL search?) we will want to compute visibility matrices after upchannelizing in selected frequency ranges.

In these notes, we describe a flexible GPU *upchanelization kernel* which ingests E-array (electric field) data, and outputs E-array data whose frequency resolution is higher by a specified factor U (the time resolution is coarser by a factor U, so that the number of samples per second is unchanged). The upchannelized E-array can be passed to the FRB beamformer, or to the "N^2 kernel" which computes visibility matrices.

In §2–§3, we'll describe upchannelization mathematically (setting aside implementation issues for now), and explore different algorithms:

- We conclude that a simple "two-stage PFB" algorithm which operates independently on coarse frequency channels is a good choice.
- We explore upchannelization artifacts (aliasing and "ripple" effects). While these artifacts will certainly complicate our downstream analysis pipelines, they don't seem to be showstoppers.
- We give a useful rule of thumb (42) for choosing the upchannelization factor U for a line emitter search.
- A loose end here: I'd like to do an "end-to-end" simulation of a dispersed FRB, from baseband data through upchannelization, FRB beamforming, and dedispersion. This is a mini-project that I haven't had time for yet!

In §4–§5, we plan the GPU implementation. If you are mainly interested in implementation, and not on the motivation behind our proposed upchannelization algorithm, then you could start reading in §4.

- In §4, we describe a fast algorithm for doing short power-of-two FFTs on tensor cores. The algorithm is implemented as an "inline" function which operates on a few registers, and can be coalesced into a larger kernel. This is an important building block for the upchannelization kernel, and can be implemented and tested independently.[1]
- In §5, we describe the larger upchannelization kernel. To give a rough idea of performance, we do cycle-counting cost estimates in a simple case (upchannelizing all frequencies with $M = 4$ PFB taps, upchannelization factor $U = 16$, and output bit depth $K = 4$), obtaining:

$$\begin{aligned}\text{(Estimated cost)} &= \underbrace{(1.6\%)}_{\text{input bandwidth}} + \underbrace{(1.6\%)}_{\text{output bandwidth}} + \underbrace{(3.2\%)}_{\text{compute}} \\ &= 6.4\% \text{ of GPU resources} \end{aligned} \tag{1}$$

  assuming full CHORD and an 128×A40 correlator.

[1] In the FRB beamformer notes, we presented a similar FFT microkernel, but the details are different. There, the FFT length $8 \leq N \leq 32$ was a multiple of 4, and the FFT was zero-padded by a factor 2. Here, the FFT length $U = 2^k$ is a power of two, and the FFT is not zero-padded.

2 High-level specification

In §2 and §3, we'll describe upchannelization mathematically, and explore different algorithms. If you're mainly interested in GPU implementation, you'll probably want to skip to §4!

2.1 Introduction

Recall that before the electric field data reaches the GPUs, it has been *channelized* by the FPGAs. We briefly review channelization as follows. The input to channelization is a real-valued "raw" timestream, or 1-d array:

$$E_0(t_0) \tag{2}$$

where t_0 indexes a "fast" (0.417 ns) time sample.

The output from channelization is a *complex*-valued channelized timestream, or 2-d array:

$$E_1(c, t_1) \tag{3}$$

where $c \in \{0, 1, \cdots, 2047\}$ indexes a frequency channel, and t_1 indexes a "slow" $(4096 \cdot (0.417 \text{ ns}) = 1.71\ \mu\text{s})$ time sample. The channelization operation is an invertible operation which preserves the number of real degrees of freedom per second (ignoring quantization subtleties).

The simplest channelization algorithm would be to simply split the raw timestream into 4096-sample chunks, and perform a length-4096 r2c FFT on each chunk (obtaining 2048 complex numbers). Unfortunately, this simple algorithm has poor spectral leakage, so in CHORD we use a different algorithm: the *polyphase filterbank* (PFB), which we review in §3.2.

The FPGA channelization (with 2048 frequency channels) is the first step in the CHORD real-time processing. Therefore, all science backends receive data with frequency resolution $(\Delta f) = (1200 \text{ MHz})/2048 = 586$ kHz by default. For most of the science backends, this default resolution suffices.

However, there are two exceptions (so far!): the 21-cm galaxy search backend and the FRB search backend need higher frequency resolution.[2] Therefore, we plan to implement an *upchannelization kernel*, which operates on the coarsely channelized timestream $E_1(c, t_1)$, to produce a complex-valued upchannelized timestream, or 3-d array:

$$E_2(c, u, t_2) \tag{4}$$

where $c \in \{0, 1, \cdots, 2047\}$ indexes a coarse (586 kHZ) frequency channel, $u \in \{0, 1, \cdots, U-1\}$ indexes a fine $(586/U$ kHz) frequency subchanel within a coarse channel, and t_2 indexes a time sample with duration $(\Delta t) = U \cdot (1.71\ \mu\text{s})$.

In this note, we will study the question: what upchannelization algorithm should we use, to go from the coarsely channelized timestream $E_1(c, t_1)$ to the upchannelized timestream $E_2(c, u, t_2)$?

In the X-engine, we plan to run two upchannelization kernels, with different frequency-dependent upchannelization factors U: one kernel for the 21-cm galaxy search and one kernel for the FRB search. The output of each upchannelization kernel will be written to GPU memory, for further processing on the GPU by a "downstream" kernel: either the FRB beamformer, or a visibility matrix kernel (for the 21-cm galaxy search). The two downstream kernels have different requirements (upchannelization factor U, bit depth, and subset of coarse frequencies which is upchannelized). In the next two subsections, we specify these requirements.

2.2 21-cm galaxy search

Upchannelization factor. For the 21-cm galaxy search, we want to support upchannelization factors $U = 8$, 16, 32, 64. Different frequency channels will have different upchannelization factors.

[2] After these notes were written, we started talking about a third possibility: an RRL (radio recombination line) backend. An RRL backend would be qualitatively similar to the 21-cm galaxy backend, in the sense that in both cases, the required data product from the GPU correlator is an upchannelized visibility matrix. However, the frequency ranges of interest and upchannelization factors are very different.

Rationale: Kristine's spec calls for a redshift-dependent channel width in the range 10–50 kHz. This corresponds to upchannelization factors in the range 12–60, since our current baseline FPGA creates 600 kHZ channels.

Range of frequency channels. Right now, we should assume that the upchannelization kernel will run on ∼half of the frequency channels. It is possible that in the future, sometime between the CHORD pathfinder and full CHORD, the number of frequency channels will be reduced, perhaps to as little as ∼5–10% of the full range.

Rationale: the frequency range depends on the maximum redshift $z_{\max}$ of the 21-cm galaxy search. Going to high redshift is challenging mainly because of RFI contamination (geosatellites) and large offline storage/compute costs. In the CHORD pathfinder, we will try searching to high redshift ($z_{\max} \sim 0.7$). If this works well, then we'll also search to high redshift in full CHORD. If RFI is a showstopper, then we may reduce $z_{\max}$ for full CHORD.

Bit depth. In our proposed GPU implementation (§4, §5), the upchannelization itself is done in `float16`. The only decision is whether to write the E_2 array to GPU global memory in `int8+8` or `int4+4`. Using `int8+8` is a little more computationally expensive, but may help reduce quantization systematics. It's tempting to reduce risk by using `int8+8`, but I don't know of any concrete reason why `int4+4` would create problems. Does anyone have a strong feeling either way?

2.3 FRB search

Upchannelization factor. We want to support upchannelization factors $U = 2$, 4, 8, 16, 32, 64, 128. Different frequency channels will have different upchannelization factors.

Rationale: Here is a back-of-the-envelope derivation of the optimal upchannelization factor U for the FRB search. The dispersion delay at frequency f is:

$$\text{Delay} = \frac{D}{f^2} \tag{5}$$

where $D = 4.15$ sec-GHz^2 for an FRB with DM 1000. Differentiating, the differential delay across channel width (Δf) is:

$$\text{Channel delay} = \frac{2D}{f^3}(\Delta f) \tag{6}$$

At back-of-the-envelope level, the upchannelization factor is:

$$\text{Upchannelization factor } U = \frac{\text{Channel delay}}{\text{Sample width } t_s} = \frac{2D}{f^3 t_s}(\Delta f) = \frac{5 \text{ GHz}^3}{f^3} \tag{7}$$

where we have assumed $t_s = 1$ millisecond and $(\Delta f) = 600$ kHz in the last step. Varying f over the range 300–1500 MHz in (7), we find that U varies over the range $1.5 < U < 185$. In practice, U will be rounded to a power of 2 in each frequency channel. I think we'll end up rounding 185→128, i.e. the largest upchannelization factor will be $U = 128$.

Range of frequency channels. We should assume that the FRB upchannelization kernel will run on ∼90% of the CHORD frequency channels. Rationale: by Eq. (7), the upchannelization factor satisfies $U \geq 2$ for $f \leq 1350$ MHz, corresponding to ∼90% of the CHORD frequency range.

Bit depth. The issues here are nearly identical to the 21-cm galaxy case (§2.2). The only decision is whether to write the E_2 array to GPU global memory in `int8+8` or `int4+4`. In this case, I think that `int4+4` is the right choice, since the FRB beamformer is less sensitive to systematic effects than the 21-cm galaxy search.

3 Mathematical specification

3.1 Raw timestream

The raw timestream is a real 1-d array $E_0(t_0)$, where t_0 is an integer. We use the following Fourier transform convention:

$$E_0(t_0) = \int_{-2048}^{2048} df\, E_0(f)\, e^{2\pi i f t_0/4096} \tag{8}$$

where $E_0(-f) = E_0(f)^*$. In Eq. (8) and throughout this note, Fourier modes are labelled by a real (non-integer) frequency $-2048 \leq f \leq 2048$. We have chosen the nonstandard Fourier convention (8) so that the analog frequency f will roughly correspond to a channel index in the coarse PFB (Polyphase FilterBank).

If the statistics of the electric field are time translation invariant, then the two-point function is:

$$\left\langle E_0(f)\, E_0(f')^* \right\rangle = I_0(f)\, \delta(f - f') \tag{9}$$

where this equation defines the intensity $I_0(f)$. We assume that time translation invariance is a good approximation on timescales short enough to be relevant for designing the upchannelization algorithm.

3.2 Coarse PFB

In this section we briefly review the *polyphase filterbank* (PFB) algorithm, which channelizes the raw timestream $E_0(t_0)$ to produce the coarsely channelized timestream $E_1(c, t_1)$. Here, $c = 0, 1, \cdots, 2048$ indexes a coarse frequency channel, and t_1 is a "coarse" time sample index whose cadence is 4096 times slower than the "raw" index t_0. Richard Shaw's notes[3] are also a great reference.

The PFB is very easy to specify: it is defined by the short equation

$$E_1(c, t_1) = \sum_{s=0}^{MN-1} W_1(s)\, E_0(Nt_1 + s)\, e^{-2\pi i c s/N} \qquad \text{where } (M, N) = (4, 4096) \tag{10}$$

where $W_1(s)$ is the "sinc-Hanning" weight function, defined by:[4]

$$W_1(s) = \cos\left(\frac{\pi(s - MN/2)}{MN}\right)^2 \mathrm{sinc}\left(\frac{s - MN/2}{N}\right) \qquad s = 0, \cdots, (MN - 1) \tag{11}$$

The parameter $M = 4$ is the number of *taps* in the PFB.

What is not so easy is explaining why the PFB (10) is a good channelization algorithm! We start by writing the following Fourier-space expression for the PFB, which is derived by plugging (8) into (10):

$$E_1(c, t_1) = \int_{-N/2}^{N/2} df\, \widetilde{W}_1(c - f) E_0(f) e^{2\pi i f t_1} \tag{12}$$

where $\widetilde{W}_1$ is the Fourier transformed weight function:

$$\widetilde{W}_1(f) \equiv \sum_{s=0}^{MN-1} W_0(s)\, e^{-2\pi i f s/N} \tag{13}$$

where f is not assumed to be an integer.

In Figure 1, we show the window function $W_1(s)$ and its Fourier transform $\widetilde{W}_1(f)$. The weight function $W_1(s)$ is chosen so that $\widetilde{W}_1(f)$ is as close as possible to the step function:

$$\widetilde{W}_{\text{step}}(f) \approx \begin{cases} 1 & \text{if } (-1/2) < f < (1/2) \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

[3] `https://github.com/jrs65/pfb-inverse/blob/master/notes.ipynb`

[4] I may have some off-by-one errors in Eq. (10), e.g. I'm not sure whether the MN denominator should be $(MN - 1)$. I didn't bother tracking these down since they don't affect any of the results in this note significantly.

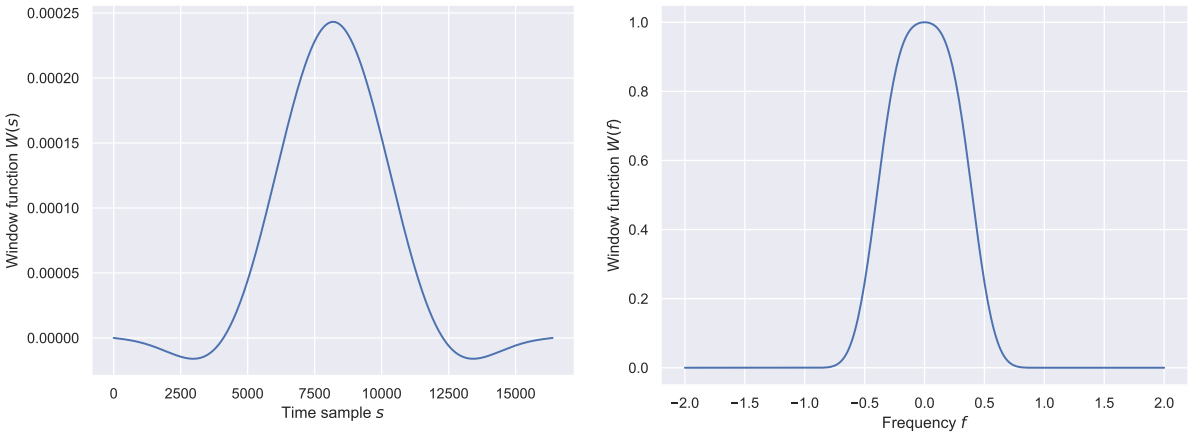

Figure 1: Sinc-Hanning window function $W_1(s)$ used in the coarse PFB (left), and its Fourier transform $\widetilde{W}_1(f)$ (right). By Eq. (10), the right plot gives the response $\widetilde{W}(c-f)$ of coarse PFB channel c (an integer) to analog frequency f (not an integer). The window function is designed so that $\widetilde{W}_1(f)$ approximates the step function (14).

Why is this desirable? If $\widetilde{W}_1(f)$ were precisely equal to $\widetilde{W}_{\rm step}$, then Eq. (12) would imply that coarse channel c has unit response to frequencies in the nominal range

$$f \in \left[c - \frac{1}{2},\, c + \frac{1}{2}\right] \tag{15}$$

and zero response to frequencies outside this range.

The sinc-Hanning weight function (11) has been chosen so that $W_1(s)$ has finite support $s = 0, \cdots, (MN-1)$, and $\widetilde{W}_1(f)$ is as close as possible to $\widetilde{W}_{\rm step}(f)$. As the number of taps M is increased, the agreement between $\widetilde{W}_1$ and $\widetilde{W}_{\rm step}$ improves. In the limit $M \to \infty$, the upchannelization algorithm becomes perfect, in the sense that $\widetilde{W}_1 \to \widetilde{W}_{\rm step}$, and each coarse channel perfectly selects frequencies in its nominal range (15.)

3.3 Upchannelization algorithms

Restriction to width 1. We will say that an upchannelization algorithm is *width* W if it operates independently on blocks of W contiguous coarse channels. For now, in this note we *restrict attention to width-1 algorithms*, i.e. algorithms which operate independently on individual coarse channels. Our eventual conclusion will be that width-1 algorithms work pretty well. Higher-width algorithms could improve sensitivity a little, but would create some minor technical challenges (for discussion see §3.8).

Kernel parameterization. A general width-1 upchannelization algorithm can be written in the form:

$$E_2(c,u,t_2) = \sum_{s=0}^{S-1} C(u,s)\, E_1(c, Ut_2 + s) \tag{16}$$

where $u = 0, \cdots, U-1$ indexes a fine channel. Here, U is the upchannelization factor, S is the kernel length (typically a small integer multiple of U), and $C(u,s)$ is the upchannelization kernel (a complex U-by-S matrix).

Within this general framework, different upchannelization algorithms correspond to different choices of kernel $C(u,s)$. In this note, we will consider two upchannelization algorithms:

- **Least-squares upchannelization.** This algorithm was proposed by Jon Sievers, and I'll just "describe" it by leaving a pointer to Jon's notes.[5] Note that Jon's algorithm generalizes to width W, but

[5] I don't think Jon's notes are on the wiki, but you may have them in your email with filename `pfb_upchannelization_v2.pdf`. Jon's code is here: `https://github.com/sievers/PFB_upchannelization`.

I'm only considering the $W = 1$ case here.

- **Two-PFB upchannelization.** Intuitively, we might try to upchannelize by applying a short "second-stage" PFB independently to each coarse PFB channel. Formally, the second-stage upchannelization algorithm is defined by:

$$E_2(c,u,t_2) = \sum_{s=0}^{MU-1} W_2(s)\, E_1(c, Ut_2+s)\, \underbrace{e^{\pi i s(U-1)/U}}_{\text{"Extra" phase}}\, e^{-2\pi i u s/U} \tag{17}$$

where the motivation for the extra phase $e^{\pi i s(U-1)/U}$ will be explained in §3.4. Note that the second-stage PFB uses a c2c FFT, whose time coarsening factor U is equal to the number of output frequencies. (In contrast to the first-stage PFB (10), which uses an r2c FFT whose time coarsening factor N is twice the number of output frequencies.) Following terminology from §3.2, we will call M the number of *taps* in the second PFB. We will use $M = 4$ taps by default.

The two-PFB upchannelization algorithm (17) is the special case of the general algorithm (16), with upchannelization kernel $C(u,s)$ given by:

$$C(u,s) = W_2(s)\, \underbrace{e^{\pi i s(U-1)/U}}_{\text{"Extra" phase}}\, e^{-2\pi i u s/U} \tag{18}$$

In subsequent sections, we will try to do calculations in a general way which applies to an arbitrary width-1 upchannelization algorithm (parameterized by kernel $C(u,s)$), so that we can treat least-squares and two-PFB upchannelization on the same footing.

3.4 Analog frequency response and aliasing

In this section, we will study the response of the upchannelized timestream to a signal with analog frequency f (not assumed to be an integer).

In Fourier space, upchannelization takes the following form:

$$E_2(c,u,t_2) = \int df\, \underbrace{\widetilde{W}_1(c-f)\, \widetilde{C}(u,-f)}_{R(c,u;f)}\, E_0(f) e^{2\pi i f U t_2} \tag{19}$$

where $\widetilde{C}(u,f)$ is defined by:

$$\widetilde{C}(u,f) = \sum_s C(u,s) e^{-2\pi i s f} \tag{20}$$

and satisfies $\widetilde{C}(u,f) = \widetilde{C}(u,f+1)$. Eq. (19) is derived by plugging (12) into (16).

The analog response $R(c,u;f)$ defined by Eq. (19) gives the (complex) response of the upchannelized timestream in coarse channel $c = 0, 1, \cdots, 2048$ and fine channel $u = 0, 1, \cdots, U-1$, to an electric field with analog frequency f. We will usually be interested in the intensity response $|R(c,u;f)|^2$.

First plot. One way to explore the intensity response $|R(c,u;f)|^2$ is to choose a few digital channels (c,u), and plot the respsonse $|R(c,u;f)|^2$ as a function of analog frequency f. We do this in Figure 2.

We see significant aliasing effects, which are largest for fine channels near the edge of a coarse channel. For example, in the top panel of Figure 2 with $U = 4$, consider the channel $(c,u) = (10,0)$. The nominal frequency range for this channel is $9.5 \le f \le 9.75$. The actual intensity response $|R(c,u;f)|^2$ consists of a "main" peak roughly covering the nominal range $9.5 \lesssim f \lesssim 9.75$, plus a smaller aliased peak at $10.5 \lesssim f \lesssim 10.75$.

For fine channels near the center of a coarse channel, there is less aliasing. In the bottom panel of Figure 2 with $U = 16$, the $u = 0$ fine channel (near the edge of the coarse channel) shows significant aliasing, but for the $u = 6$ fine channel (near the center of the coarse channel) the aliased response is tiny (~ 45 dB smaller than the main response).

Aliasing is unavoidable for width-1. The aliasing in Figure 2 is an unavoidable consequence of using a width-1 upchannelization algorithm, as we now explain. Consider two sinusoidal analog signals at frequencies f and $(f+1)$:

$$E_0(t_0) = Ae^{2\pi i f t_0/N} \qquad E'_0(t_0) = A'e^{2\pi i (f+1) t_0/N} \qquad \text{where } N = 4096 \tag{21}$$

These signals are indistinguishable in a single coarse channel due to aliasing. More precisely, by Eq. (12) the coarsely channelized signals are:

$$E_1(c, t_1) = A\,\widetilde{W}_1(c-f)\,e^{2\pi i f t_1} \qquad E'_1(c, t_1) = A'\,\widetilde{W}_1(c-f-1)\,e^{2\pi i f t_1} \tag{22}$$

In a single coarse channel c, the signals are indistinguishable except for the prefactors $\widetilde{W}_1(c-f)$ and $\widetilde{W}_1(c-f-1)$. That is, the degeneracy between frequencies f and $(f+1)$ is broken only by the *coarse* PFB response $\widetilde{W}_1$.

The worst aliasing occurs for a frequency $f = c - 1/2$ such that f and $(f+1)$ are at the edges of a coarse channel. In this case, the coarse PFB response in channel c is the same for f and $(f+1)$, the degeneracy is not broken at all, and the aliasing is perfect. The least aliasing occurs for a frequency $f = c$ at the center of a coarse channel. In this case, the coarse PFB response does a good job of breaking the degeneracy, by suppressing frequencies $(f \pm 1)$, and there is not much aliasing.

We speculate that higher-width algorithms would help a little, but will not eliminate aliasing completely. Consider for example the case $W = 2$, and a pair $(c, c+1)$ of coarse channels. Each of the three analog frequencies $(f - 1/2, f + 1/2, f + 3/2)$ will map to a linear combination of two available Fourier modes in E_1. For linear algebra reasons, there must be order-one aliasing.[6]

Second plot. In Figure 3, we explore the intensity response $|R(c, u; f)|^2$ in a different way. We fix a few choices of intensity power spectrum $I_0(f)$ in panels (a)–(e). For each such choice, we plot the upchannelized response

$$\left\langle |E_2(c, u, t_2)|^2 \right\rangle = \int df\, |R(c, u; f)|^2 I_0(f) \tag{23}$$

as a function of (c, u). In panels (a)–(d), we show the channelized response to a "narrow-line" signal $I_0(f) = \delta(f - f_0)$. The aliasing is worst in the case where the input frequency f_0 is at the edge between two coarse channels (case (d)). In this case, the input signal aliases equally into four digital channels (two channels with $f \approx f_0$ and two channels with $f \approx f_0 \pm 1$)! We will let the reader think through the details of how this behavior follows from the general discussion of aliasing above. In panel (e), we show the channelized response to a "broad" Gaussian line $I_0(f) \propto e^{-(f-f_0)^2/2\sigma_f^2}$ centered on the edge between coarse channels, showing interesting aliasing behavior.

Explanation of "extra" phase in the two-PFB algorithm. A loose end: we now explain the "extra" phase in the two-PFB algorithm (17). First recall (Eq. (15)) that the nominal frequency range of coarse PFB channel c is:

$$\big(\text{Nominal frequency range of coarse channel } c\big) = \left[c - \frac{1}{2},\, c + \frac{1}{2}\right] \tag{24}$$

What about a fine channel (c, u)? We will let the reader convince themselves of the following statements. If the extra phase is included in Eq. (17), then:

$$\big(\text{Nominal frequency range of fine channel } (c, u)\big) = \left[c - \frac{1}{2} + \frac{u}{U},\, c - \frac{1}{2} + \frac{u+1}{U}\right] \tag{25}$$

If the extra phase is not included in Eq. (17), then:

$$\big(\text{Nominal freq range of fine channel } (c, u)\big) = \begin{cases} \left[c + \frac{2u-1}{2U},\, c + \frac{2u+1}{2U}\right] & \text{if } u \neq U/2 \\ \left[c - \frac{1}{2}, c - \frac{1}{2} + \frac{1}{2U}\right] \cup \left[c + \frac{1}{2} - \frac{1}{2U}, c + \frac{1}{2}\right] & \text{if } u = U/2 \end{cases} \tag{26}$$

[6] If we wanted to pull out all the stops to eliminate aliasing in upchannelization, there is a way to it: using an oversampled PFB in the FPGAs. I'm pretty confident this would work and is the "right" solution, but I haven't worked out the details. But, it would increase data rates by 20% throughout most of CHORD! I'm assuming this is a nonstarter, so I haven't seriously explored the idea of using an oversampled PFB.

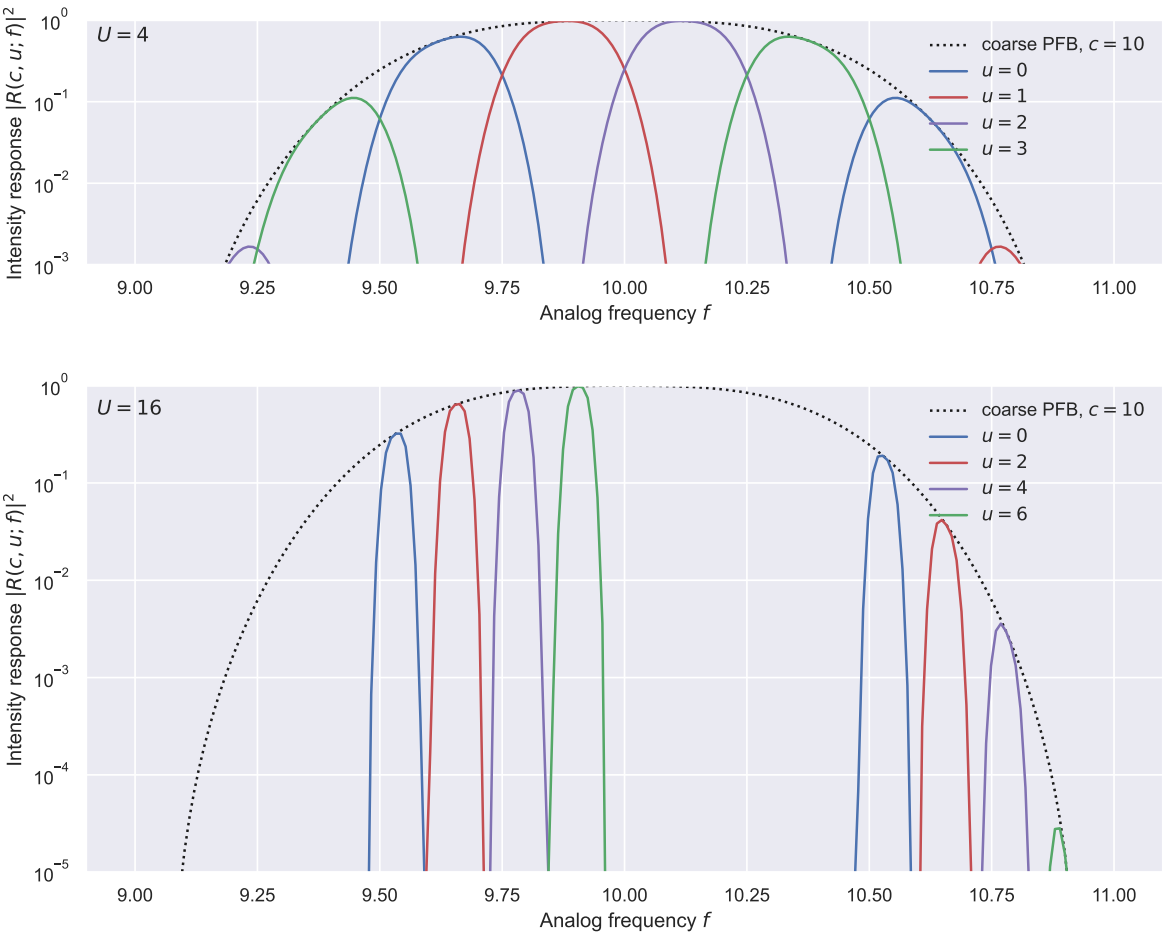

Figure 2: Intensity response $|R(c, u; f)|^2$ as a function of analog frequency f, for a few choices of fine channel u within coarse channel $c = 10$. The nominal frequency range for this coarse channel is $9.5 \leq c \leq 10.5$. Both panels use two-PFB upchannelization with $M = 4$ taps. *Top panel.* All four fine channels $u = 0, 1, 2, 3$ in a single coarse channel, for upchannelization factor $U = 4$. *Bottom panel.* Selected fine channels $u = 0, 2, 4, 6$ in a single coarse channel, for upchannelization factor $U = 16$.

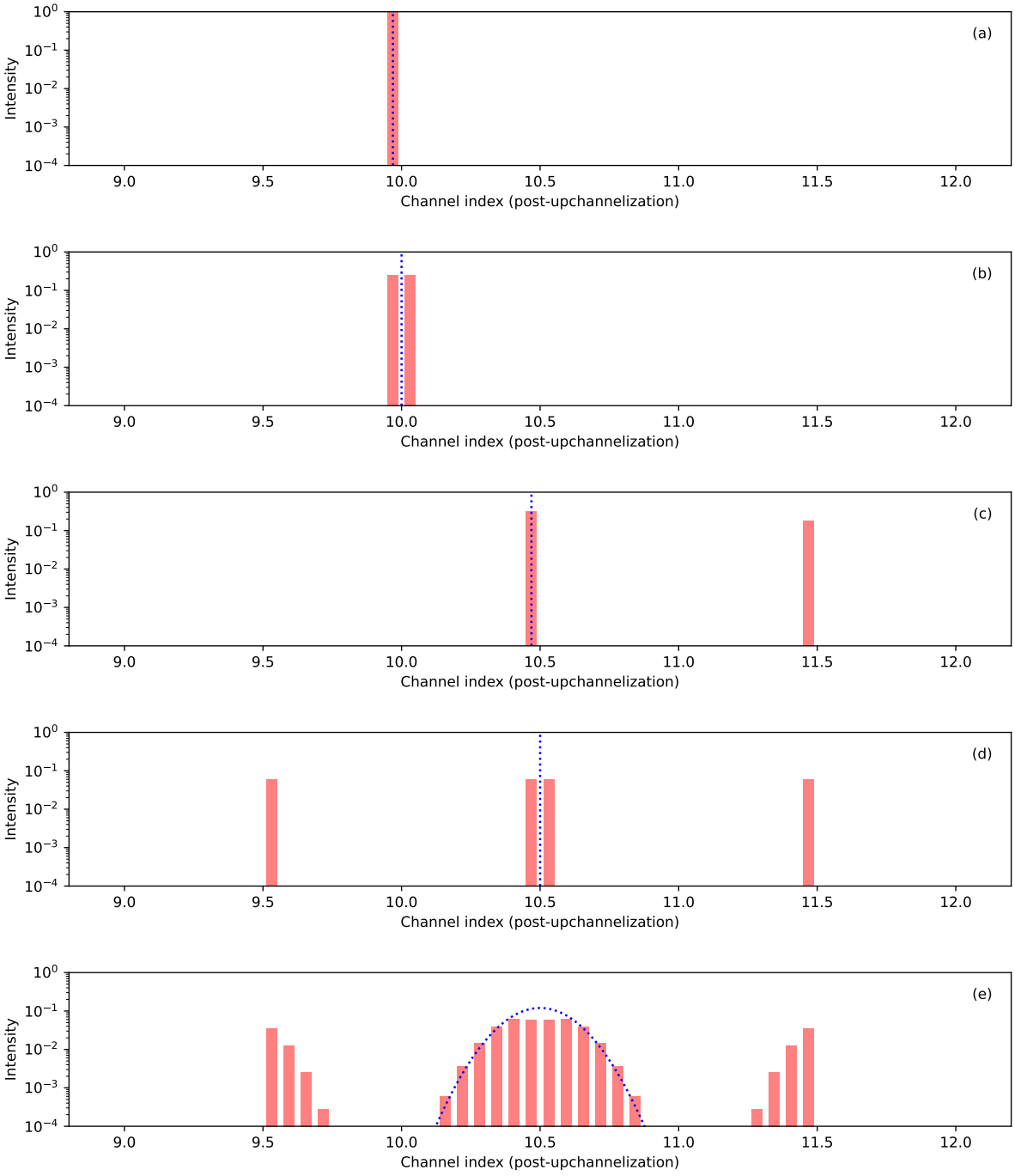

Figure 3: Upchannelized response (23) to input power spectrum $I_0(f)$. Each panel shows a different choice of input analog power spectrum $I_0(f)$ (dotted line), and the digital channelized intensity is shown as the sequence of vertical bars. In the first four panels, $I_0(f) = \delta(f - f_0)$ is a "narrow-line" signal whose frequency f_0 is shown as the dotted line. From top to bottom, the input frequency is (a) near the center of both a coarse channel and a fine channel, (b) at the edge between two fine channels near the center of a coarse channel, (c) at the center of a fine channel near the edge of a coarse channel, (d) at the edge between two coarse channels. In the bottommost panel (e), the input signal is a "broad" Gaussian line $I_0(f) \propto e^{-(f-f_0)^2/2\sigma^2}$ centered on the edge betwen two coarse channels, shown as the dotted line. We use two-PFB upchannelization with $M = 4$ taps and upchannelization factor $U = 16$ throughout.

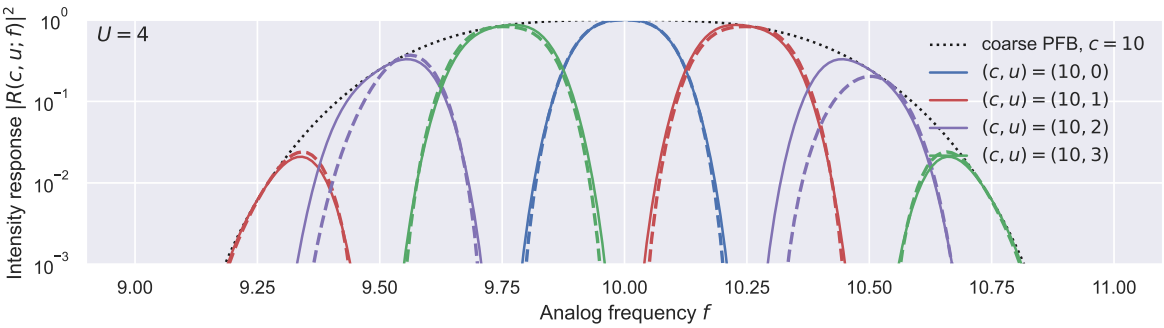

Figure 4: This plot is similar to the top panel of Figure 2, but without the "extra" phase in Eq. (17). Note the weird frequency response for $u = 2$. Besides exploring the "extra" phase, a second (and more important) purpose of the plot is to compare the two-PFB (solid curves) and least-squares (dashed curves) upchannelization algorithms (see §3.3). The two algorithms are qualitatively similar.

Thus, if the extra phase is omitted, the fine channels are permuted (relative to the intuitive ordering $u = 0, \cdots, U-1$), and fine channel $u = U/2$ has a weird frequency response.

In Figure 4, we show this visually, by plotting the intensity response $|R(c, u; f)|^2$ for upchannelization factor $U = 4$, without the extra phase in Eq. (17). Note the weird frequency response for $u = 2$, relative to the top panel of Figure 2.

Another (more important) purpose of Figure 4 is to compare the two-PFB (solid curves) and least-squares (dashed curves) upchannelization algorithms. Note that our least-squares upchannelization does not currently include the "extra" phase from Eq. (17). (This is because I'm calling Jon's code as a "black box" out of laziness – it should be easy to modify it to include the extra phase if needed.) The main conclusion from Figure 4 is that the two-PFB and least-squares algorithms are qualitatively similar. In the next section, we will also compare bottom-line SNR from the two algorithms (Figure 6), and find similar results.

Variance "ripples". The plots in this section have mainly explored frequency aliasing. In Figure 5, we explore another effect: frequency "ripples" in the upchannelized timestreams. For each fine channel (c, u), we plot the variance

$$\left\langle |E_2(c, u, t_2)|^2 \right\rangle \tag{27}$$

as a function of the nominal frequency $(c + (2u - U + 1)/(2U))$ of the fine channel. We compute the variance (27) using Eq. (38) from the next section, assuming that the raw timestream has a power spectrum $I_0(f)$ which is constant in f.

Figure 5 shows that upchannelization produces frequency "ripples" in the output. This will complicate downstream analyses but does not necessarily indicate a serious problem, since the ripples can be removed by applying a channel-dependent normalization. For example, the SNR forecasts in the next section are not significantly affected by the ripples, since the The SNR forecasts in the next section are normalization-indepenent will assume optimal matched filtering, which is independent of the channel normalizations.

3.5 Signal-to-noise forecasts for 21-cm galaxies

In the last section, we showed that upchannelization algorithms have nontrivial aliasing behavior (Figures 2, 3) and ripples (Figure 5). At minimum, this will complicate the 21-cm galaxy search. The optimal search statistic will be a matched filter (not a simple peak-finder), parameterized by a matrix which keeps track of how signal power at trial frequency f aliases into different digital channels, and a covariance matrix which keeps track of "ripples" and correlations between frequencies.

However, these extra complications do not necessarily indicate a serious problem. How can we decide whether there is a "real" problem or not? I decided to ask the following questions:

1. If we search for 21-cm galaxies with an optimal matched filter, what fraction of the SNR is lost due to upchannelization?

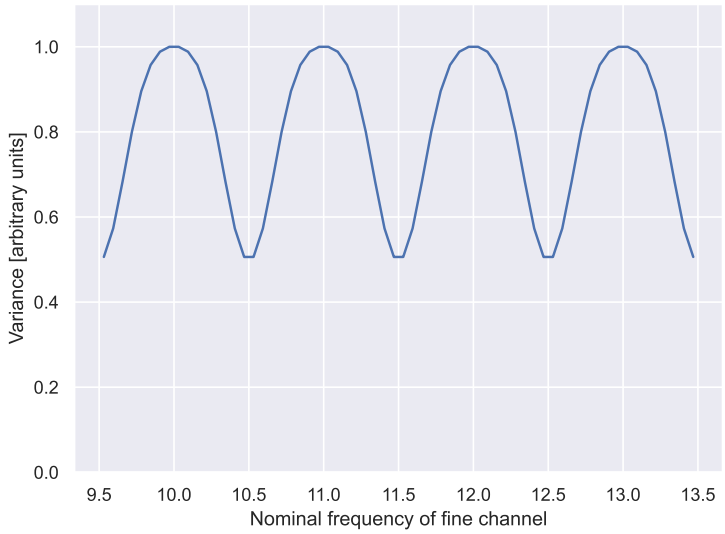

Figure 5: Variance $\langle |E_2(c,u,t_2)|^2 \rangle$ of the upchannelized timestream, as a function of fine channel $(c+u/U)$, for an input signal whose power spectrum $I_0(f)$ is independent of frequency. We see that upchannelization produces frequency "ripples" in the output. We use two-PFB upchannelization with $M = 4$ taps and upchannelization factor $U = 16$.

2. Can aliasing effects produce confusion between 21-cm galaxies at different redshifts?

Setup and forecasting machinery. Suppose we observe a faint steady source with power spectrum $I_s(f)$, in the presence of frequency-independent white noise I_n. That is, the power spectrum of the electric field is:

$$\left\langle E_0(f) E_0(f')^* \right\rangle = I_0(f)\delta(f-f') \qquad \text{where } I_0(f) = I_n + I_s(f) \tag{28}$$

where $I_s(f) \ll I_n$. We will usually be interested in the case where $I_s(f)$ is a Gaussian emission line profile with central frequency $f_{\rm mid}$:

$$I_s(f) \propto \exp\left(-\frac{(f-f_{\rm mid})^2}{2\sigma_f^2}\right) \qquad \text{where } \sigma_f = \frac{W_{50}}{2\sqrt{2\log 2}} \tag{29}$$

where we can parameterize the emission line width either by its RMS σ_f, or its full width at half maximum W_{50} (following ALFALFA [1]).

If we observe $T \gg 1$ raw timestream samples $E_0(t_0)$ without quantization noise, then the optimal SNR is:

$$\mathrm{SNR}^2_{\rm opt} = \frac{T}{2N} \int_{-N/2}^{N/2} df\, \frac{I_s(f)^2}{I_n^2} \tag{30}$$

We want to compare this with the upchannelized SNR (denoted $\mathrm{SNR}_{\rm uc}$) obtained from an optimal analysis of upchannelized visibilities, possibly in the presence of quantization noise. The upchannelized visibilities are defined by:

$$V_{cu} = \left\langle |E_2(c,u,t_2)|^2 \right\rangle_{t_2} \tag{31}$$

where $E_2(c,u,t_2)$ is the upchannelized timestream defined in §3.3. Note that the upchannelized visibilities may contain less information than the upchannelized timestream $E_2(c,u,t)$, since off-diagonal correlations $t_2 \neq t_2'$ are discarded. We will compute $\mathrm{SNR}_{\rm uc}$ as follows:

$$\mathrm{SNR}^2_{\rm uc} = S_{cu}\, C^{-1}_{cu,c'u'}\, S_{c'u'} \tag{32}$$

where S_{cu} is the mean visibility due to the signal $I_s(f)$, and $C_{cu,c'u'}$ is the visibility covariance due to the noise:

$$S_{cu} = \big\langle V_{cu} \big\rangle_{\text{signal only}} \qquad\qquad C_{cu,c'u'} = \text{Cov}\big[V_{cu}, V_{c'u'}\big]_{\text{noise only}} \tag{33}$$

The signal S_{cu} can be computed straightforwardly from (19):

$$S_{cu} = \int df \left|\widetilde{W}_1(c-f)\widetilde{C}(u,f)\right|^2 I_s(f) \tag{34}$$

The noise covariance $C_{cu,c'u'}$ is more complicated, and we will spend the next few paragraphs computing it. *Between here and Eq. (40) below, we neglect the signal term in (28), and assume* $I_0(f) = I_n$.

First, we compute the two-point function of the raw timestream $E_0(t_0)$:

$$\big\langle E_0(t_0) E_0(t_0') \big\rangle = \sigma_0^2\, \delta_{t_0 t_0'} \qquad\qquad \text{where } \sigma_0^2 = N I_n \tag{35}$$

Then we compute the two-point function of the coarsely channelized timestream $E_1(c,t_1)$:[7]

$$\big\langle E_1(c,t_1) E_1(c',t_1')^* \big\rangle = \sigma_0^2\, \zeta_1(c-c', t_1-t_1') + \sigma_1^2 \delta_{cc'} \delta_{t_1 t_1'} \tag{36}$$

where we have modelled quantization noise as an extra Gaussian noise term σ_1^2, and defined:

$$\zeta_1(f,\tau) \equiv \sum_s W_1(s) W_1(s+N\tau) e^{-2\pi i f s/N} \tag{37}$$

Next, we compute the two-point function of the upchannelized timestream $E_2(c,u,t_2)$:

$$\begin{aligned} \big\langle E_2(c,u,t_2) E_2(c',u',t_2')^* \big\rangle = \sigma_0^2 \sum_{\tau_1} \zeta_1(c-c',\tau_1)\, \zeta_2(u,u',\tau_1 + U t_2' - U t_2) \\ + \sigma_1^2\, \delta_{cc'}\, \zeta_2(u,u', U t_2' - U t_2) \\ + \sigma_2^2 \delta_{cc'} \delta_{uu'} \delta_{t_2 t_2'} \end{aligned} \tag{38}$$

where we have defined:

$$\zeta_2(u,u',\tau) \equiv \sum_s C(u,s) C(u', s-\tau)^* \tag{39}$$

Finally, the noise covariance $C_{cu,c'u'}$ is given in terms of the two-point function (38) by:

$$C_{cu,c'u'} = \frac{NU}{T} \sum_{\tau_2=-\infty}^{\infty} \left|\big\langle E_2(c,u,0) E_2(c',u',\tau_2)^* \big\rangle\right|^2 \tag{40}$$

In the rest of this section, we assume zero quantization noise ($\sigma_1 = \sigma_2 = 0$). We will include quantization noise in §3.6.

What fraction of SNR is retained after upchannelization? Using the preceding forecasting machinery, we can answer the first question from the beginning of this section. We define the *retention* to be the fraction of SNR which is retained after upchannelization:

$$(\text{Retention}) \equiv \frac{\text{SNR}_{\text{uc}}}{\text{SNR}_{\text{opt}}} \tag{41}$$

In Figure 6, we plot the retention as a function of central frequency f_{fmid} (defined in Eq. (29)) and upchannelization factor U, for fixed $W_{50} = 0.05$ (also defined in (29)).

[7] We will assume that two-point correlations of the form $\langle E_1(c,t_1) E_1(c',t_1')\rangle$, with no complex conjugate on the second factor, are zero. This is a good approximation everywhere except a few "edge channels" ($f = 0$, and to a small extent $f \in \{1, N-1\}$). The edge channels may need special treatment in the 21-cm galaxy search, but we disregard them for purposes of forecasting since they are a tiny fraction of the data.

We see that for small U, the retention is poor and contains oscillations. As U increases, the retention saturates to a non-oscillatory limiting curve. For a given value of W_{50}, we would like to choose U large enough that saturation has occurred. Based on Figure 6, we propose the following rule of thumb:

$$U \gtrsim \frac{1.5}{\text{Smallest } W_{50} \text{ of interest}} \tag{42}$$

This rule of thumb is one of the main results from this note.

It may be surprising that the retention does not approach 1 as $U \to \infty$. Instead, we see in Figure 6 that the retention saturates to a limiting value which is close to 1 if $f_{\rm mid}$ is an integer, and more like $1/\sqrt{2}$ if $f_{\rm mid}$ is a half-integer. We speculate that this is an inherent limitation of width-one algorithms. Here is a hand-waving argument. Consider a narrow ($W_{50} \ll 1$) emission line whose frequency is on the edge between two coarse frequency channels. After the coarse PFB, half (in intensity) of the signal will go into each coarse channel. If the upchannelization algorithm is width-one, then this statement will still be true after upchannelization. When we compute the visibilities V_{cu} (defined in Eq. (31)), we throw away the cross-correlation between the two channels, and the maximum retention will be $1/\sqrt{2}$.

In the left/right panels of Figure 6, we compare the retention from the two-PFB and least squares upchannelization algorithms. The two algorithms are very similar. (Neither one is uniformly better than the other – if you squint at the plots, you can find values of $f_{\rm mid}$ where one or the other is a little better. Overall there is not much difference.)

How large are correlations between different redshifts? Next, we answer the second question from the beginning of this section: in an optimal matched filter analysis, can aliasing effects produce confusion between 21-cm galaxies at different redshifts?

First a little machinery. Let $I_s^{(1)}(f)$ and $I_s^{(2)}(f)$ be emission line profiles corresponding to galaxies at different redshifts. Let $S_{cu}^{(1)}$ and $S_{cu}^{(2)}$ be the associated visibilities, computed from $I_s^{(i)}(f)$ using Eq. (34). We define a 2-by-2 Fisher matrix F_{ij} by:

$$F_{ij} = S_{cu}^{(i)} \, C^{-1}_{cu,c'u'} \, S^{(j)}_{c'u'} \tag{43}$$

where $C_{cu,c'u'}$ is the visibility covariance as before. Then the correlation between the two galaxy signals is:

$$r = \frac{F_{12}}{\sqrt{F_{11} F_{22}}} \tag{44}$$

Using this machinery, in Figure 7 we show the correlation coefficient r as a function of the central frequencies $f_{\rm mid}^{(1)}$, $f_{\rm mid}^{(2)}$. For a narrow profile ($W_{50} \ll U^{-1}$), the correlation coefficient between frequencies $(f_{\rm mid}, f_{\rm mid}+1)$ can be as large as $r = 0.4$ (left panel). We note that the correlation coefficient decreases slightly as the emission line width W_{50} increases. At the "critical" line width $W_{50} = 1.5/U$ derived from the rule of thumb in Eq. (42), the max correlation coefficient is $r = 0.3$ (right panel).

A correlation coefficient as large as $r = 0.3$ or $r = 0.4$ is unlikely to be a concern. If we set a galaxy detection threshold of 6σ, then the true redshift can be distinguished from its aliased counterparts at $6(1-r)$ sigmas. We conclude that redshift correlations are unlikely to be a significant issue for the 21-cm galaxy search, provided that an optimal matched filter analysis is used.

3.6 Quantization noise

So far our forecasts have assumed zero quantization noise ($\sigma_1 = \sigma_2 = 0$). In this section, we will study quantization noise systematically. We will assume that quantization can be fully modelled as an extra source of Gaussian noise, so that the only issue is choosing realistic values for the parameters σ_1, σ_2 defined in Eqs. (36), (38).

Note that we're assuming negligible roundoff errors in the GPU implementation of the upchannelization algorithm, so that the only place where quantization arises is when the E_1, E_2 arrays are created. I think this assumption is reasonable since the upchannelization itself will be done in either `int8` or `float16` (see last paragraph of §3.3).

Also note that we're assuming that quantization can be fully modelled as Gaussian random noise, and ignoring more subtle nonlinear effects. I haven't thought carefully about this, and I think it's an important

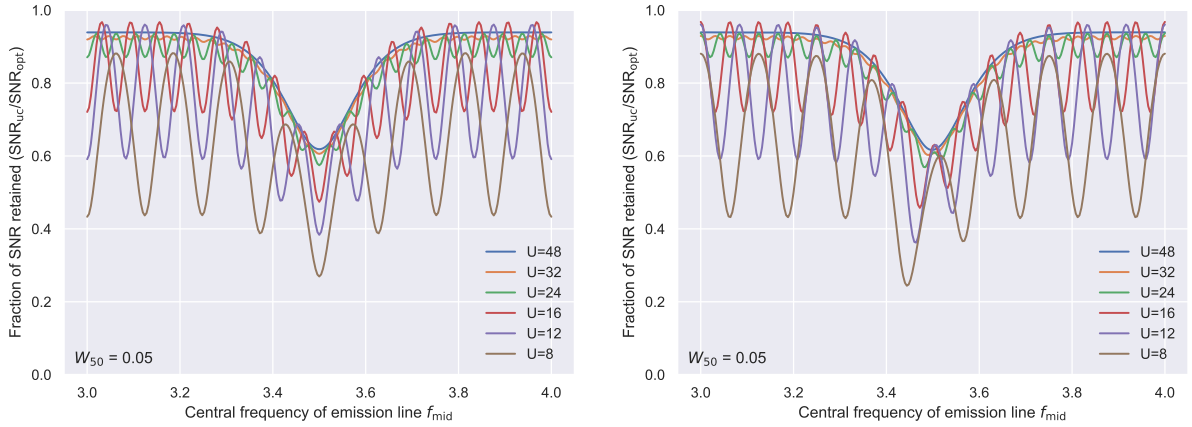

Figure 6: SNR retention (41) after upchannelization, for a Gaussian emission line with $W_{50} = 0.05$. (Throughout this note, all frequencies are multiples of the coarse PFB width 576 kHz, so $W_{50} = 0.05$ really means $W_{50} = (0.05)(576\text{ kHz}) = 28.8$ kHz.) We show the retention (41) as a function of central frequency f_{fmid} (defined in Eq. (29)) and upchannelization factor U. *Left panel.* Two-PFB upchannelization with $M = 4$ taps. *Right panel.* Least-squares upchannelization with $M = 4$ taps.

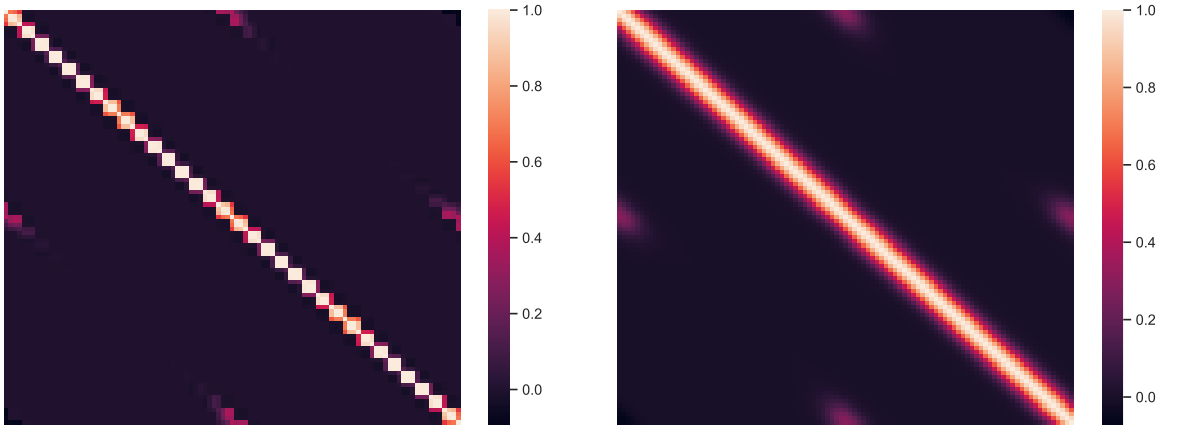

Figure 7: Correlation coefficients between emission line sources with varying central frequencies $f^{(1)}_{\text{mid}}$, $f^{(2)}_{\text{mid}}$. The x,y axes in correspond to a uniformly spaced sequence of f_{mid} values between 2.5 and 4.5. We use two-PFB upchannelization with $M = 4$ taps and upchannelization factor $U = 16$. *Left panel.* Narrow emission line $W_{50} \ll (1/U)$ which is unresolved by upchannelization. *Right panel.* Critical width emission line $W_{50} = 1.5/U$ which is barely resolved by upchannelization.

loose end. How can we 100% convince ourselves that quantization artifacts are not an issue for the 21-cm galaxy search?

Setting this difficult question aside for now, we next consider the concrete question of how to choose a realistic value for the quantization noise σ_1^2. We assume that without quantization noise, the real and imaginary parts of $E_1(c,t)$ each have RMS equal to 2 LSBs (least significant bits):

$$\sqrt{\left\langle \frac{|E_1(c,t_1)|^2}{2} \right\rangle_{\sigma_1=0}} = B_1 \times (\text{LSB}) \qquad \text{where } B_1 = 2 \tag{45}$$

We also assume that the quantization noise variance $(\sigma_1^2/2)$ for the real and imaginary parts of $E_1(c,t)$ satisfies the simple relation:

$$\frac{\sigma_1^2}{2} = \frac{(\text{LSB})^2}{12} \tag{46}$$

Combining Eqs. (36), (45), (46), we get the following expression for σ_1:

$$\sigma_1^2 = \frac{1}{6B_1^2} \left\langle \frac{|E_1(c,t_1)|^2}{2} \right\rangle_{\sigma_1=0} = \frac{\sigma_0^2 \zeta_1(0,0)}{12B_1^2} \tag{47}$$

Similarly, we assume that a realistic value for the quantization noise σ_2^2 is:

$$\sigma_2^2 = \frac{1}{6B_2^2} \left\langle \frac{|E_2(c,u,t_2)|^2}{2} \right\rangle_{\sigma_2=0} \tag{48}$$

where $B_2 = 2$ if the upchannelization output (E_2-array) is quantized as `int4+4`. If the E_2-array is quantized as `int8+8`, then we will assume σ_2 is negligibly small (since B_2 would be around 30).

In Figure 8, we show the effects of quantization noise on the bottom-line SNR retention defined in (41). We see that the effects of quantization noise are small.

3.7 Placeholder for a future section on FRBs

Questions to answer here: What does a dispersed pulse look like after upchannelization? How correlated is the pulse with the matched filter used in tree dedispersion? When we use an upchannelization factor U which depends on frequency, what U-dependent time offsets do we need to apply, in order to make everything consistent?

3.8 Summary and discussion

Summary of main results.

- Width-1 upchannelization algorithms produce significant aliasing effects (Figures 2 and 3) and intensity "ripples" (Figure 5).
- These features will make the 21-cm galaxy analysis more complicated. Rather than a simple peak finder, we will want to use a matched filter parameterized by a matrix which keeps track of the aliasing in Figure 3 and correlations between fine channels. However, if an optimal matched filter is implemented, the impact of aliasing on bottom-line SNR is modest (Figure 6), and aliasing-induced correlations are also modest (Figure 7). The main effect is a $\sim$30% drop in sensitivity at frequencies which are close to a half-integer multiple of the coarse channel width (586 KHz).
- We propose the rule of thumb
$$U \gtrsim \frac{1.5}{\text{Smallest } W_{50} \text{ of interest}} \tag{49}$$
for determining the upchannelization factor U (Eq. (42)).
- If quantization can be modelled purely as a source of Gaussian noise, then quantization effects are small (Figure 8).

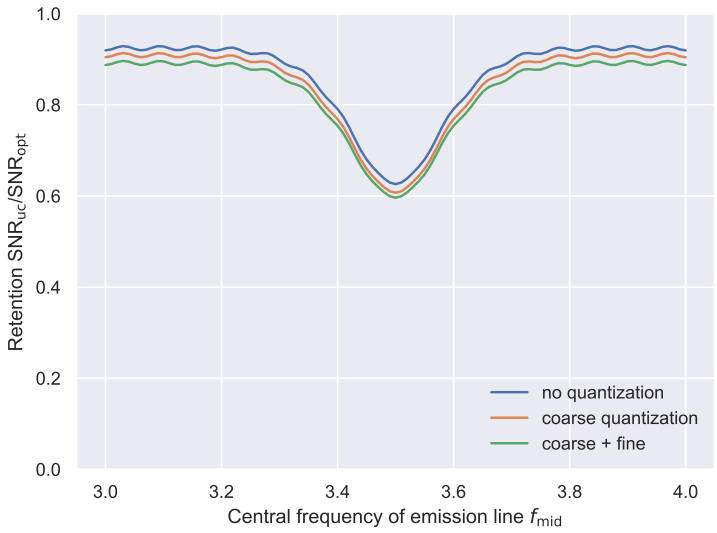

Figure 8: Effect of quantization noise on SNR after upchannelization. We emphasize that we are modelling quantization purely as an extra source of Gaussian noise. Similarly to Figure 6, this plot shows the SNR retention (41) as a function of emission line central frequency $f_{\rm mid}$. Throughout the plot, we use line width $W_{50} = 0.05$, and two-PFB upchannelization with $M = 4$ taps and upchannelization factor $U = 16$. *Top curve.* SNR retention with no quantization noise. *Middle curve.* SNR retention with `int4+4` quantization noise from the coarse PFB, computed using Eq. (47). *Bottom curve.* SNR retention with `int4+4` quantization noise from both coarse and fine PFBs, computed using Eqs. (47), (48).

- We compare two-PFB and least-squares upchannelization algorithms, and find almost no difference either in aliasing features (Figure 4) or bottom-line SNR (Figure 6). However, we have only considered the width-1 version of the least-squares upchannelization algorithm.

Loose ends.

- Simulating a dispersed FRB pulse, all the way through the channelization + upchannelization pipeline. (§3.7)
- Further quantization studies? (§3.6)

Should we explore higher-width algorithms? Throughout this note, we have only considered width-1 upchannelization algorithms. Should we explore higher width W? Here are some thoughts on the tradeoffs:

- I predict that aliasing effects (Figure 2) will be qualitatively similar, but may have smaller amplitude for $W > 1$.
- I predict that the bottom-line SNR (Figure 6) will be qualitatively similar, but the 30% drop will occur less frequently (at frequencies of the form $f = (\text{integer} \cdot W + 1/2)$ rather than half-integers).
- GPU implementation will be more difficult and may require changing the packet format for sending data between the FPGAs and GPUs.[8]

[8] A little elaboration on this non-obvious statement. Currently the packet format is constructed so that the fastest varying indices are (dish,pol). For an efficient GPU implemetation, we always want to read entire 128-byte cache lines, so the upchannelization kernel must process blocks of 64 dishes and 2 polarizations. Therefore the total memory "footprint" on each GPU compute unit is $128WUM$ bytes. If we make the rough guess that the footprint must be ≤ 64 KB before running out of registers or shared memory, this gives the constraint $WUM \leq 512$.

- Load-balancing computation across GPUs becomes more difficult as W increases, especially for the FRB search where computational cost and output bandwidth is much larger at the lowest frequencies. The seriousness of this problem will depend on how constrained we are in assigning coarse frequencies to GPUs. Note that load-balancing is a potential issue even for width-1 upchannelization, so this is something we should figure out soon in any case!

We discussed the above bullet points on a telecon, and the general sentiment was that width > 1 algorithms presented nontrivial technical challenges (load balancing, packet format) but didn't seem to have a strong benefit. Therefore, we didn't end up seriously considering width > 1 upchannelization in CHORD.

Proposed algorithm: two-PFB, width-one upchannelization. Assuming width 1, the results in this section suggest that there isn't much difference between PFB upchannelization and least squares upchannelization. Therefore, the choice should be dictated by computational cost. We're currently proposing PFB upchannelization, since we think it will be faster on the GPU. In the next two sections, we will plan a GPU implementation of PFB upchannelization.

4 FFT microkernel

In this section, we describe a "microkernel" for computing float16 c2c FFTs entirely in registers of a single warp (i.e. with no global or shared memory I/O). ***We assume that the FFT length is a power of 2, and known at compile time.***

The FFT microkernel could be implemented as a `__device__ inline` function with a few `(__half2 &)` arguments. This function would be called by the larger GPU upchannelization kernel described in the next section (§5).

4.1 Specification, spectator indices, and register assignments

We denote the FFT length by $U = 2^k$. If k is too small, then the FFT arguments will not span the registers of a single warp. Therefore, we add a spectator index $0 \leq s < S$, where

$$S = 2^{\max(6-k,0)} = \begin{cases} (64/U) & \text{if } U \leq 64 \\ 1 & \text{if } U \geq 64 \end{cases} \tag{50}$$

We denote the input index of the FFT by $0 \leq \tau < U$, and the output index by $0 \leq u < U$. Thus the FFT is defined by:[9]

$$Y_{us} = \sum_{\tau=0}^{U-1} X_{\tau s} \exp\left(-\frac{2\pi i \tau u}{U}\right) \tag{51}$$

Next we specify register assignments for the X and Y arrays in Eq. (51). We distinguish two cases:

- If $U \leq 32$ (equivalently $k \leq 5$), then the input/output register assignments are:

$$[(2^k \times 2^{6-k})\ X_{\tau s}] \quad b \leftrightarrow \tau_{k-1} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \tau_{k-2} \cdots \tau_0, s_0 \cdots s_{5-k} \quad r \leftrightarrow \text{ReIm} \tag{52}$$

$$[(2^k \times 2^{6-k})\ Y_{us}] \quad b \leftrightarrow u_0 \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow u_1 \cdots u_{k-1}, s_0 \cdots s_{5-k} \quad r \leftrightarrow \text{ReIm} \tag{53}$$

- If $U \geq 64$ (equivalently $k \geq 6$), then the input/output register assignments are:

$$[(2^k \times 1)\ X_{\tau s}] \quad b \leftrightarrow \tau_{k-1} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \tau_{k-2} \cdots \tau_{k-6} \quad r \leftrightarrow \text{ReIm}, \tau_0 \cdots \tau_{k-7} \tag{54}$$

$$[(2^k \times 1)\ Y_{us}] \quad b \leftrightarrow u_0 \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow u_1 \cdots u_5 \quad r \leftrightarrow \text{ReIm}, u_6 \cdots u_{k-1} \tag{55}$$

The number of registers per thread needed to store the X or Y array is:

$$R_{\text{data}} = \begin{cases} 2 & \text{if } U \leq 64 \\ (U/2) & \text{if } U \geq 64 \end{cases} \tag{56}$$

Important note 1: In Eqs. (52)–(55) and throughout this section, we use a permuted ordering $t_1 t_0 t_2 t_4 t_3$ for thread index bits. For some values of U, this will let us save a few cycles by using sparse tensor core MMA operations. (This is not obvious in advance – follows from details of the register assignment for the sparse `m16n8k16` tensor core MMA in §C.3.) For clarity, we use **bold typeface $\mathbf{t_1 t_0 t_2 t_4 t_3}$** whenever thread index bits are permuted.

Important note 2: In Eq. (51), we have defined the FFT with a minus sign inside the exp(). There is a 50% change that I have the wrong sign convention, and it should be a plus. Therefore, I suggest writing your code with a boolean flag which makes it easy to switch between the two sign conventions in the future.

[9] In our description of the FFT, we have used slightly weird notation (e.g. it would be less weird to replace $(U, \tau, u) \to (N, m, n)$). However, the weird notation will be convenient in the larger context of the upchannelization kernel (§5).

4.2 The FFT algorithm

In this section, we recall the idea of the FFT algorithm. Consider an FFT $X[\tau] \to Y[u]$ of length 2^{m+n}, with no spectator indices:

$$Y[u] = \sum_{\tau=0}^{2^{m+n}-1} X[\tau] \exp\left(-\frac{2\pi i \tau u}{2^{m+n}}\right) \tag{57}$$

where in Eq. (57) and throughout this subsection, we index arrays using square brackets instead of subscripts (e.g. $X[\tau]$ instead of X_τ).

The idea of the FFT algorithm is to factorize the FFT (57) into three steps: (1) an FFT of length 2^m, (2) a step where we multiply elementwise by phases, and (3) an FFT of length 2^n. First, we write the input index $0 \le \tau < 2^{m+n}$ as:

$$\tau = 2^n \tau_{\rm hi} + \tau_{\rm lo} \qquad \text{where } 0 \le \tau_{\rm lo} < 2^n \text{ and } 0 \le \tau_{\rm hi} < 2^m \tag{58}$$

Similarly, we write the output index $0 \le u < 2^{m+n}$ as:

$$u = 2^m u_{\rm hi} + u_{\rm lo} \qquad \text{where } 0 \le u_{\rm lo} < 2^m \text{ and } 0 \le u_{\rm hi} < 2^n \tag{59}$$

(Note that the roles of m, n are switched in Eqs. (58), (59).)

Using this notation, we can view the FFT as a mapping $X[\tau_{\rm lo}, \tau_{\rm hi}] \to Y[u_{\rm lo}, u_{\rm hi}]$ between 2-d arrays. Now, a short calculation shows that the FFT (57) can be factorized into three steps as follows:

$$W[\tau_{\rm lo}, u_{\rm lo}] = \sum_{\tau_{\rm hi}=0}^{2^m-1} X[\tau_{\rm lo}, \tau_{\rm hi}] \exp\left(-\frac{2\pi i \tau_{\rm hi} u_{\rm lo}}{2^m}\right) \tag{60}$$

$$Z[\tau_{\rm lo}, u_{\rm lo}] = W[\tau_{\rm lo}, u_{\rm lo}] \exp\left(-\frac{2\pi i \tau_{\rm lo} u_{\rm lo}}{2^{m+n}}\right) \tag{61}$$

$$Y[u_{\rm lo}, u_{\rm hi}] = \sum_{\tau_{\rm lo}=0}^{2^n-1} Z[\tau_{\rm lo}, u_{\rm lo}] \exp\left(-\frac{2\pi i \tau_{\rm lo} u_{\rm hi}}{2^n}\right) \tag{62}$$

The first step (60) is a length-2^m FFT $\tau_{\rm hi} \to u_{\rm lo}$, with $\tau_{\rm lo}$ acting as a spectator index. In the second step (61), we apply phases elementwise. The third step (62) is a length-2^n FFT $\tau_{\rm lo} \to u_{\rm hi}$ with spectator index $u_{\rm lo}$.

4.3 In/out notation for tensor core MMAs

ReIm in/out notation. Consider a matrix multiplication $C = AB$, where all 3 matrices are complex. To do this matrix multiplication on tensor cores, we will need to add extra ReIm indices. In the simplest example where A, B, C are 1-by-1 "matrices", the complex multiplication $C = AB$ could be implemented as:

$$\begin{pmatrix} \mathrm{Re}(C) \\ \mathrm{Im}(C) \end{pmatrix} = \begin{pmatrix} \mathrm{Re}(A) & -\mathrm{Im}(A) \\ \mathrm{Im}(A) & \mathrm{Re}(A) \end{pmatrix} \begin{pmatrix} \mathrm{Re}(B) \\ \mathrm{Im}(B) \end{pmatrix} \tag{63}$$

This example generalizes to arbitrary-shape matrices as follows. The matrices B and C get a length-2 ReIm axis in a straightforward way, whereas the matrix A gets two length-2 axes $\mathrm{ReIm}_{\rm in}$ and $\mathrm{ReIm}_{\rm out}$ (corresponding respectively to column and row indices in (63)).

For notational clarity, we temporarily denote the original complex matrix by A_c, and its real counterpart with length-2 axes $(\mathrm{ReIm}_{\rm in},\ \mathrm{ReIm}_{\rm out})$ by A_r. Then A_r and A_c are related by:

$$\begin{aligned} A_r[\mathrm{Re}_{\rm out}, \mathrm{Re}_{\rm in}, \cdots] &= \mathrm{Re}\, A_c[\cdots] \\ A_r[\mathrm{Re}_{\rm out}, \mathrm{Im}_{\rm in}, \cdots] &= -\mathrm{Im}\, A_c[\cdots] \\ A_r[\mathrm{Im}_{\rm out}, \mathrm{Re}_{\rm in}, \cdots] &= \mathrm{Im}\, A_c[\cdots] \\ A_r[\mathrm{Im}_{\rm out}, \mathrm{Im}_{\rm in}, \cdots] &= \mathrm{Re}\, A_c[\cdots] \end{aligned} \tag{64}$$

where $(\cdots)$ denotes all index bits of A_c (either row indices, column indices, or spectator indices). For an example of this notation, see Eq. (**??**) below.

Spectator bit in/out notation. Consider an 8×8 matrix multiply:

$$\underbrace{C}_{8\times 8} = \underbrace{A}_{8\times 8}\ \underbrace{B}_{8\times 8} \tag{65}$$

Suppose we want to do several such matrix multiplications, for different choices of matrix B, but with the same choice of A throughout. Also suppose we want to use the `m16n8k16` MMA (Appendix C.2), in which some matrix dimensions are 16. Then we can do two matrix multiplications $C = AB$ and $C' = AB'$ with one `m16n8k16` MMA, as follows:

$$\underbrace{\begin{pmatrix} C \\ C' \end{pmatrix}}_{16\times 8} = \underbrace{\begin{pmatrix} A & \\ & A \end{pmatrix}}_{16\times 16} \underbrace{\begin{pmatrix} B \\ B' \end{pmatrix}}_{16\times 8} \tag{66}$$

To formalize this, we introduce a length-2 spectator axis s, and combine the matrices B_{jk} and B'_{jk} into a single array B_{jks} (and likewise for C). The matrix A gets two length-2 spectator indices $s_{\rm in}, s_{\rm out}$, and depends on these indices only through an overall Kronecker delta $\delta_{s_{\rm in} s_{\rm out}}$.

This example can be generalized to other situations in which a tensor core MMA is "wider" than the matrix multiplication of interest. For example, consider this matrix multiply:

$$\underbrace{C}_{4\times 8} = \underbrace{A}_{4\times 4}\ \underbrace{B}_{4\times 8} \tag{67}$$

We can do four such matrix multiplies (for the same choice of A) with one `m16n8k16` MMA as follows:

$$\underbrace{\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}}_{16\times 8} = \underbrace{\begin{pmatrix} A & & & \\ & A & & \\ & & A & \\ & & & A \end{pmatrix}}_{16\times 16} \underbrace{\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix}}_{16\times 8} \tag{68}$$

To formalize this, we introduce a length-4 spectator axis. The B and C matrices would each get two spectator bits s_0, s_1. The A-matrix would get four spectator bits $s_0^{\rm in}, s_1^{\rm in}, s_0^{\rm out}, s_1^{\rm out}$, and would depend on these indices through two Kronecker deltas $\delta_{s_0^{\rm in} s_0^{\rm out}}\, \delta_{s_1^{\rm in} s_1^{\rm out}}$.

We mention in advance that this use of spectator bits leads to sparse matrices. The A-matrices in Eqs. (66), (68) have 50% and 25% sparsity respectively. This will allow us to use sparse tensor core MMAs (Appendix C.3) for speed.

4.4 Case 1: k=1

Starting in this section, we will use the FFT algorithm (§4.2) recursively, to build up longer FFTs from shorter FFTs. To compute the shortest "building block" FFTs, we'll sometimes use tensor core tricks, which we'll explain in context as they arise.

In the $k = 1$ case, we want to go from:

$$[(2 \times 32)\ X_{\tau s}] \qquad b \leftrightarrow \tau_0 \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow s_0 s_1 s_2 s_3 s_4 \qquad r \leftrightarrow \text{ReIm} \tag{69}$$

to:

$$[(2 \times 32)\ Y_{us}] \qquad b \leftrightarrow u_0 \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow s_0 s_1 s_2 s_3 s_4 \qquad r \leftrightarrow \text{ReIm} \tag{70}$$

This can be done with one `m16n8k8` MMA (Appendix C.1), whose arguments (A, B, C) correspond to (X, phases, Y).[10] Here, "phases" means a matrix whose elements are constants of the form $\cos(2\pi\tau u/2^k)$ or $\sin(2\pi\tau u/2^k)$. When such phase matrices arise, we assume they are precomputed and stored in persistent registers throughout the upchannelization kernel.

[10]Since the $k = 1$ FFT is thread-local, you may be wondering why we use a tensor core MMA instead of straightforward arithmetic ops. The reason is that the tensor core MMA costs 2 SM-cycles, whereas a straightforward approach would cost 4 SM-cycles (four `__half2` FMAs and four calls to `__byte_perm()`. Similarly, in the remaining cases (§4.5–**??**), we will sometimes use tensor core MMAs instead of simpler operations without explicit discussion, in cases where tensor core MMAs turn out to be faster.

4.5 Case 2: k=2

We start with the X-array, in register assignment:

$$[(4 \times 16)\ X_{\tau s}] \qquad b \leftrightarrow \tau_0 \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \tau_1 s_0 s_1 s_2 s_3 \qquad r \leftrightarrow \text{ReIm} \tag{71}$$

The entire $k = 2$ FFT can be done with one sparse `m16n8k16` MMA, whose arguments (A, B, C) correspond to (phases, X, Y). As explained in Appendix C.3, the MMA is sparse because t_0 maps to a spectator index s_0. We get the Y-array in the wrong register assignment:

$$[(4 \times 16)\ Y_{\tau s}] \qquad b \leftrightarrow s_1 \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow s_2 s_3 u_0 u_1 s_0 \qquad r \leftrightarrow \text{ReIm} \tag{72}$$

To fix the register assignment, we multiply by a 16×16 identity matrix using a sparse `m16n8k16` MMA (Appendix C.3). This doesn't change the array content, but does change the register assignment:

$$[(4 \times 16)\ Y_{\tau s}] \qquad b \leftrightarrow u_0 \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow u_1 s_0 s_1 s_2 s_3 \qquad r \leftrightarrow \text{ReIm} \tag{73}$$

4.6 Case 3: k=3,4,5,6

We will factorize the length-2^k FFT into a length-8 FFT followed by a length-2^{k-3} FFT, using the FFT algorithm (§4.2). We start with the X-array, in register assignment:

$$\mathbf{b t_1 t_0} \leftrightarrow \tau_{k-1}\tau_{k-2}\tau_{k-3} \qquad \mathbf{t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-4} \cdots \tau_0, s_0 \cdots s_{5-k}}_{3 \text{ bits}} \qquad r \leftrightarrow \text{ReIm} \tag{74}$$

We do a length-8 FFT with index bits $\tau_{k-1}\tau_{k-2}\tau_{k-3} \to u_0 u_1 u_2$, to get the the W-array (see Eq. (60)). This can be done using a dense `m16n8k16` MMA, whose arguments (A, B, C) correspond to (phases, X, W). We get the W-array in register assignment:

$$\mathbf{b t_1 t_0} \leftrightarrow \underbrace{\tau_{k-4} \cdots \tau_0, s_0 \cdots s_{5-k}}_{3 \text{ bits}} \qquad \mathbf{t_2 t_4 t_3} \leftrightarrow u_0 u_1 u_2 \qquad r \leftrightarrow \text{ReIm} \tag{75}$$

The next step is to apply phases elementwise, to get the Z-array (see Eq. (61)). This can be done with four `__half2` FMAs, with 2 registers/thread containing precomputed phases. In the case $k = 3$, the phase application step is the identity (i.e. $Z = W$) and can be skipped.

Finally, starting from the Z-array in the register assignment (75), we do a length-2^{k-3} FFT with index bits $\tau_0 \cdots \tau_{k-4} \to u_3 \cdots u_{k-1}$, obtaining the Y-array with register assignment:

$$\mathbf{b t_1 t_0} \leftrightarrow u_0 u_1 u_2 \qquad \mathbf{t_2 t_4 t_3} \leftrightarrow \underbrace{u_3 \cdots u_{k-1}, s_0 \cdots s_{5-k}}_{3 \text{ bits}} \qquad r \leftrightarrow \text{ReIm} \tag{76}$$

This final length-2^{k-3} FFT can be done with an `m16n8k16` MMA, whose arguments (A, B, C) correspond to (phases, Z, Y). This MMA is dense for $k = 6$ (Appendix C.2), or sparse for $k = 3, 4, 5$ (Appendix C.3). In the case $k = 3$, this length-2^{k-3} FFT is the identity (i.e. $Y = Z$), but as in the previous subsection (§4.5), we do want to multiply by the identity matrix in order to change the register assignment.

One final comment: if $k = 6$, then we can use the same precomputed phases in both length-8 FFTs (saving some registers).

4.7 Case 4: k ≥ 7

We will factorize the length-2^k FFT into a length-2^{k-1} FFT followed by a length-2 FFT, using the FFT algorithm (§4.2). We start with the X-array, in register assignment:

$$b \leftrightarrow \tau_{k-1} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \tau_{k-2}\tau_{k-3}\tau_{k-4}\tau_{k-5}\tau_{k-6} \qquad r \leftrightarrow \underbrace{\text{ReIm}, \tau_0, \cdots, \tau_{k-7}}_{2^{k-5} \text{ registers}} \tag{77}$$

First, we do a length-2^{k-1} FFT with index bits $\tau_1 \cdots \tau_{k-1} \to u_0 \cdots u_{k-2}$, and treating τ_0 as a spectator index. This length-2^{k-1} FFT is implemented inductively, eventually reaching the length-64 ($k = 6$) "base case" from the previous subsection. We get the W-array (see Eq. (60)), with register assignment:

$$b \leftrightarrow u_0 \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow u_1 u_2 u_3 u_4 u_5 \qquad r \leftrightarrow \underbrace{\mathrm{ReIm}, \tau_0, u_6 \cdots u_{k-2}}_{2^{k-5} \text{ registers}} \tag{78}$$

Next we apply phases elementwise, to get the Z-array (see Eq. (61)). This step needs 2^{k-5} `__half2` FMAs (not 2^{k-4}, since we only need to apply phases to registers which correspond to index bits $\tau_0 = 1$). We need $\max(2^{k-7}, 2)$ registers to store precomputed phases (not 2^{k-6}, since varying the u_{k-2} index bit multiplies phase by i, which does not lead to new precomputed register values).

Finally, we do a length-2 FFT, to compute the Y-array from the Z-array (see Eq. (62)). This step needs 2^{k-5} `__half2` FMAs, with no precomputed registers.

4.8 Computational cost and register usage

In Table 1, we show cost analysis for length-2^k FFTs with $1 \le k \le 9$. The number of registers R_{phase} used to store precomputed phases was computed as follows:

- For the `m16n8k8` MMA in the $k = 1$ case (§**??**), we need one register to store phases.
- For the `m16n6k16` MMAs which arise for $2 \le k \le 6$ (§4.5, §4.6), we need two registers if the MMA is sparse, or three registers if the MMA is dense.

  Note: in the dense case, we need 3 registers instead of 4, for the following reason. In the precomputed phase matrix, index bits ($\mathrm{ReIm}_{\text{in}}, \mathrm{ReIm}_{\text{out}}$) are mapped to registers. By Eq. (64), the (Re_{in}, Re_{out}) register stores the same value as the (Im_{in}, Im_{out}) register.

  Note 2: If $k = 6$, then the two dense MMAs use the same precomputed phases, so we need 3 registers (not 3+3).
- In several places in §4.5–§4.7, we need phase registers in `__half2` FMA operations. In each such case, we have stated the number of phase registers explicitly.

The number of SM-cycles C was computed as follows:

- An `m16n8k8` MMA costs 2 SM-cycles.
- An `m16n8k16` MMA costs 4 SM-cycles if dense, or 2 SM-cycles if sparse.
- A `__half2` FMA costs half an SM-cycle.

If you get different values in implementation for either R_{phase} or C, let me know and we can compare!

FFT length $U = 2^k$	2	4	8	16	32	64	128	256	512
Spectator indices S	32	16	8	4	2	1	1	1	1
Data registers R_{data}	2	2	2	2	2	2	4	8	16
Phase registers R_{phase}	1	4	5	7	7	5	7	9	13
SM-cycles C	2	4	6	8	8	10	24	56	128
"Cost" C/R_{data}	1	2	3	4	4	5	6	7	8

Table 1: Cost analysis of the FFT microkernel. The quantities S and R_{data} were defined in Eqs. (50), (56). The quantities R_{phase} and C are described above (in §4.8). In the last line, we have given a bottom-line "cost" (C/R_{data}) in SM-cycles per data register. To interpret this value, the global memory bandwidth of the upchannelization kernel has equivalent cost ∼15.

4.9 Extra credit: coalescing gains and/or extra phases

When the FFT arises in the larger upchannelization kernel (§5), it is sandwiched between two elementwise multiplications:

$$X_{\tau s} \to \underbrace{e^{\pi i \tau (U-1)/U}}_{\text{“Extra” phase}} X_{\tau s} \tag{79}$$

$$Y_{us} = \sum_{\tau=0}^{U-1} X_{\tau s} \exp\left(-\frac{2\pi i \tau u}{U}\right) \tag{80}$$

$$Y_{us} \to G_{us} Y_{us} \tag{81}$$

Here, G_{us} is a shape-(U, S) real-valued (float16) array of gains.

Here's a question you may enjoy thinking about: can the elementwise multiplications be coalesced into the FFT microkernel, in order to get a speedup (or use fewer registers)? Currently, we propose in §4 to do the elementwise multiplications separately, using `__half2` FMAs. For the extra phase, this has cost 1 (in the sense defined in Table 1), and uses $2\lceil U/64 \rceil$ registers per thread. For the gains, this has cost 0.5, and uses $\lceil U/64 \rceil$ registers per thread.

5 Upchannelization kernel

5.1 Specification

Compile-time inputs:

- Upchannelization factor U, and number of taps M. We assume that U is a power of two:

$$U = 2^k \tag{82}$$

  The value of M is aribtrary (e.g. M could be odd). Our implementation requires $U \leq 512$ and $U(M-1) \lesssim 1024$. (It also requires $M \lesssim 100$, but this would never be an issue in practice.)

- Output bit depth K. All intermediate computations will be `float16`, but the input data is quantized to 4 bits, and the output data will be quantized to K bits.

  We'll want to support output bit depth $K = 4$ or $K = 8$. The highest priority is $K = 4$, so feel free to postpone the $K = 8$ case if you prefer.

Runtime inputs:

- Electric field $E_{\tau f \pi d}$ (int4+4). The indexing is $(\tau, f, \pi, d) = (\text{time}, \text{coarse freq}, \text{pol}, \text{dish})$. Our implementation requires the number of dishes D to be a multiple of 64. This is the case for the CHORD pathfinder ($D = 64$), full CHORD ($D = 512$), and HIRAX ($D = 256$).

- PFB weight function (float16). This should be passed to the kernel as a length-(MU) array W_s, where $0 \leq s < MU$.

- Output gains (float16), a length-U array G_u. (I don't think the gains need to depend on the polarization and dish indices (π, d). Allowing gains to depend on (π, d) would increase the number of registers used by the kernel.)

Outputs:

- Upchannelized electric field $\bar{E}_{\bar{\tau} f u \pi d}$ (integer, $K + K$ bits), defined by:

$$\bar{E}_{\bar{\tau} f u \pi d} = \text{Quantize}_{K+K} \left[G_u \sum_{s=0}^{MU-1} W_s \underbrace{e^{\pi i s (U-1)/U}}_{\text{"Extra" phase}} e^{-2\pi i u s / U} \left(E_{\tau f \pi d} \right)_{\tau = (\bar{\tau} - M + 1)U + s} \right] \tag{83}$$

  Here, $\bar{\tau}$ is a coarse time index, whose sampling rate is U times slower than the "fast" time index τ, and $0 \leq u < U$ indexes an upchannelized channel.

Eq. (83) defines the upchannelization kernel. It agrees with our earlier definition (17) of the "width-one two-PFB" upchannelization algorithm, up to minor notational changes, and redefining the coarse time index as $\bar{\tau} \to \bar{\tau} + M - 1$. (This redefinition will simplify indexing in the kernel.)

5.2 Outline

Threadblocks. The dish index d and polarization index π are both spectator indices for upchannelization. Therefore, we can pretend there is no polarization index, by doubling the number of dishes. (Note that we can now assume that the number of dishes is a multiple of 128, rather than 64.)

Each threadblock will process one coarse frequency channel, and 128 "dishes" (i.e. dish+polarization pairs). For the rest of these notes, we will concentrate on a single threadblock. Then:

- We can treat the input electric field array as a 2-d array $E_{\tau d}$ (int4+4) with shape $(N_{\text{time}}, 128)$.

- We can treat the output upchannelized array as a 3-d array $\bar{E}_{\bar{\tau} u d}$ with shape $(N_{\text{time}}/U, U, 128)$.

After this minor notational change, the output of the upchannelization kernel is:

$$\bar{E}_{\bar{\tau}ud} = \text{Quantize}_{K+K}\left[G_u \sum_{s=0}^{MU-1} W_s \underbrace{e^{\pi i s(U-1)/U}}_{\text{"Extra" phase}} e^{-2\pi i u s/U} \left(E_{\tau d}\right)_{\tau=(\bar{\tau}-M+1)U+s}\right] \tag{84}$$

Warning: we'll make another minor notational change shortly in Eq. (88) below!

Inner/outer blocks and high-level kernel organization. The kernel processes time samples in "outer blocks" of T_{outer} time samples. Each outer block consists of one or more "inner blocks", which consist of U time samples.

For each outer block, we copy E-array data from global to shared memory, compute the $\bar{E}$-array (which ends up in shared memory), and copy $\bar{E}$-array data from shared to global memory. Thus, outer blocks define the cadence for global↔shared memory transfers, and calls to `__syncthreads()`.

Within each outer block, we loop over inner blocks. For each inner block, the E-array is copied from shared memory to registers. We also store in registers a logical ring buffer consisting of the previous $(M-1)$ inner blocks, so that we have enough thread-local data (M inner blocks) to compute one inner block of the $\bar{E}$-array (84). We copy the $\bar{E}$-array inner block to shared memory, where it overwrites the E-array inner block. Thus, inner blocks define the cadence for computing the $\bar{E}$-array.

Summarizing, at any point during processing, we store a ring buffer in registers consisting of the previous $(M-1)$ inner blocks of E-array data. We use shared memory to store one outer block of E-array data, which is incrementally overwritten (one inner block at time) by $\bar{E}$-array data.

***More compile-time parameters:* $\mathbf{W}$, $\mathbf{B}$, $\mathbf{T_{outer}}$, Packed.** The kernel has externally specified compile-time parameters (U, M, K). For each choice of (U, M, K), we define the following "derived" compile-time parameters:

- Number of warps per threadblock W, which must be a power of two:

$$W = 2^l \tag{85}$$

- Number of threadblocks per SM B.
- Outer block size T_{outer}, described above.
- Boolean parameter `Packed`, which determines whether the in-register E-array ring buffer uses datatype `int4` (`Packed` = `True`) or `float16` (`Packed` = `False`).

  For small values of MU, `Packed` = `False` is preferable since it reduces the number of `int4`→`float16` conversion instructions. For large values of MU, `Packed` = `True` will be necessary, in order to avoid running out of registers.

  Cycle-counting estimates (see §5.7) suggest that the speed improvement going from `Packed` = `True` to `Packed` = `False` is modest. Therefore, you may prefer to implement only the case `Packed` = `True`, and consider implementing `Packed` = `False` as a future optimization.

The optimal choice of $(W, B, T_{\text{outer}}, \texttt{Packed})$ for a given (U, M) will be discussed in §5.6. We mention in advance that the following compile-time asserts must be satisfied:

```
static_assert(W <= U);
static_assert((Touter % U) == 0);
static_assert((Touter % (4*W)) == 0);
```
(86)

A little more notation. Sometimes we will write the time index τ as:

$$\tau = U\bar{\tau} + \tau' \qquad\qquad \text{where } 0 \le \tau' < U \tag{87}$$

where $\bar{\tau}$ is a coarse time index, and $0 \le \tau' < U$ indexes a fine time sample within a coarse sample (or equivalently, an inner block). In register assignments, $\bar{\tau}$ will have index bits $\tau_k, \tau_{k+1}, \cdots$, and τ' will have index bits $\tau_0 \cdots \tau_{k-1}$. Using this notation:

- We'll represent the input electric field array as a 3-d array $E_{\bar{\tau}\tau' d}$ (int4+4) with shape $(N_{\rm time}/U, U, 128)$. (To be formal about this, the 3-d representation $E^{3d}_{\bar{\tau}\tau' d}$ is related to our previous 2-d representation $E^{2d}_{\tau d}$ by $E^{3d}_{\bar{\tau}\tau' d} = E^{2d}_{(\bar{\tau}U+\tau'),d}$.)
- We'll represent the PFB weight function as a 2-d array $W_{m\tau'}$ with shape (M, U). (To be formal about this, the 2-d representation $W^{2d}_{m\tau'}$ is related to our previous 1-d representation W^{1d}_s by $W^{2d}_{m\tau'} = W^{1d}_{sU+\tau'}$.)

After these minor notational changes, Eq. (84) for the output of the upchannelization kernel becomes:

$$\bar{E}_{\bar{\tau}ud} = \text{Quantize}_{K+K}\left[G_u \sum_{m=0}^{M-1}\sum_{\tau'=0}^{U-1} W_{m\tau'} \underbrace{(-1)^m e^{\pi i \tau'(U-1)/U}}_{\text{"Extra" phase}} e^{-2\pi i \tau' u/U}\, E_{(\bar{\tau}+m-M+1),\tau',d}\right] \tag{88}$$

We factor this computation into a few steps as follows, in order to introduce notation $E^{(2)}$, $E^{(3)}$, $\bar{E}^{(4)}$, $\bar{E}^{(5)}$ for intermediate quantities which will be useful later.

$$E^{(2)}_{\bar{\tau}\tau' d} = \sum_{m=0}^{M-1} (-1)^m W_{m\tau'}\, E_{(\bar{\tau}+m-M+1),\tau',d} \tag{89}$$

$$E^{(3)}_{\bar{\tau}\tau' d} = e^{\pi i \tau'(U-1)/U} E^{(2)}_{\bar{\tau}\tau' d} \tag{90}$$

$$\bar{E}^{(4)}_{\bar{\tau}ud} = \sum_{\tau'=0}^{U-1} E^{(3)}_{\bar{\tau}\tau' d}\, e^{-2\pi i \tau' u/U} \tag{91}$$

$$\bar{E}^{(5)}_{\bar{\tau}ud} = G_u\, \bar{E}^{(4)}_{\bar{\tau}ud} \tag{92}$$

$$\bar{E}_{\bar{\tau}ud} = \text{Quantize}_{K+K}\left[\bar{E}^{(5)}_{\bar{\tau}ud}\right] \tag{93}$$

(We use barred symbols $\bar{E}^{(4)}$, $\bar{E}^{(5)}$ to denote "post-upchannelization" arrays indexed by a coarse time $\bar{\tau}$ and an upchannelized index $0 \le u < U$, and unbarred symbols $E^{(1)}$, $E^{(2)}$, $E^{(3)}$ to denote "pre-upchannelization" arrays indexed by an input time sample $\tau = U\bar{\tau} + \tau'$.)

5.3 Shared memory layout and change of variable $(\mathbf{E}, \bar{\mathbf{E}}) \leftrightarrow (\mathbf{F}, \bar{\mathbf{F}})$

As explained in the previous section, we store one outer block of E-array data in shared memory, which gets incrementally replaced (one inner block at a time) by $\bar{E}$-array data. The main purpose of this section is to explain the shared memory layout.

Primed dish indices. We will sometimes use a "primed" dish index $0 \le d' < 128$, which is related to the unprimed dish index $0 \le d < 128$ by permuting index bits as follows:

$$d_0 d_1 d_2 d_3 d_4 d_5 d_6 \leftrightarrow d'_6 d'_5 d'_0 d'_1 d'_2 d'_3 d'_4 \qquad\qquad d'_0 d'_1 d'_2 d'_3 d'_4 d'_5 d'_6 \leftrightarrow d_2 d_3 d_4 d_5 d_6 d_1 d_0 \tag{94}$$

We will sometimes switch between unprimed and primed dish orderings implicitly – for example, an E-array inner block could be denoted $E_{\tau' d}$ or $E_{\tau' d'}$.

Change of variable $\mathbf{E} \leftrightarrow \mathbf{F}$. Consider an inner block $E_{\tau' d'}$ with shape $(U, 128)$ and dtype int4+4. We form 32-bit registers by varying the following index bits:

$$b_0 b_1 b_2 \leftrightarrow \text{ReIm}, d'_6, \tau_{k-1} \qquad\qquad (\text{note that } d'_6 = d_0 \text{ by Eq. (94)}) \tag{95}$$

By varying the remaining index bits $\tau_0 \cdots \tau_{k-2}, d'_0 \cdots d'_5$, we get an int32-valued array $F_{\tau' d'}$. This defines a change of variable $E \leftrightarrow F$ for one inner block. Note that the int32 array $F_{\tau' d'}$ has shape $(U/2, 64)$, whereas the int4+4 array $E_{\tau' d'}$ has shape $(U/2, 64)$, since the index bits τ'_{k-1}, d'_6 have been absorbed into the definition (95) of F.

Change of variable $\bar{\mathbf{E}} \leftrightarrow \bar{\mathbf{F}}$. Now consider an inner block of the *output* array $\bar{E}_{ud'}$ with shape $(U, 128)$. Similarly as above, we will define a change of variables $\bar{E} \leftrightarrow \bar{F}$. There are two cases: $K = 4$ and $K = 8$.

First, one small bit of notation: we will write the upchannelization index $0 \le u < U$ as:

$$u = 2u' + u_0 \qquad \text{where } 0 \le u' < (U/2) \qquad \text{and} \qquad 0 \le u_0 < 2 \tag{96}$$

We write the index $0 \le u' < (U/2)$ with index bits $u_1 \cdots u_{k-1}$ (i.e. we will not use notation like $u'_0 \cdots u'_{k-2}$).

Now consider the case $K = 4$. Given an inner block $\bar{E}_{ud'}$, we form 32-bit registers by varying the following index bits:

$$b_0 b_1 b_2 \leftrightarrow \mathrm{ReIm}, d'_6, u_0 \qquad \text{(note that } d'_6 = d_0 \text{ by Eq. (94))} \tag{97}$$

By varying the remaining index bits $u_1 \cdots u_{k-1}, d'_0 \cdots d'_5$, we get an int32-valued array $\bar{F}_{u'd'}$. This defines a change of variable $\bar{E} \leftrightarrow \bar{F}$ for one inner block. Note that the int32 array $\bar{F}_{u'd'}$ has shape $(U/2, 64)$, whereas the int4+4 array $E_{ud'}$ has shape $(U, 128)$, since the index bits u_0, d'_6 have been absorbed into the definition (97) of $\bar{F}$.

Next consider the case $K = 8$. Given an inner block $\bar{E}_{ud'}$, we form 32-bit registers by varying the following index bits:

$$b_0 b_1 \leftrightarrow d'_6, u_0 \qquad \text{(note that } d'_6 = d_0 \text{ by Eq. (94))} \tag{98}$$

By varying the remaining index bits $u_1 \cdots u_{k-1}, d'_0 \cdots d'_5$, we get an int32+32 array $\bar{F}_{u'd'}$. This defines a change of variable $\bar{E} \leftrightarrow \bar{F}$ for one inner block. In both the $K = 4$ and $K = 8$ cases, the $\bar{F}_{u'd'}$ array shape is $(U/2, 64)$, but in the $K = 8$ case there is an implicit length-2 ReIm axis.

Shared memory layout. We use shared memory to store one outer block of either E-array or $\bar{E}$-array data. The shared memory layout has logical structure:

```
union {
     int F[Touter/U][U/2][64];          // (tau_bar, tau_prime, d_prime)
     int Fbar[K/4][Touter/U][U/2][64]; // (ReIm, tau_bar, u_prime, d_prime)
};
```
(99)

where the array axes are as follows. We represent each outer block as a sequence of (T_{outer}/U) inner blocks, corresponding to array axis $0 \le \bar{\tau} < (T_{\mathrm{outer}}/U)$. We represent each inner block as a shape-$(U/2, 64)$ array using the F, $\bar{F}$ representations above. As explained above, the $\bar{F}$-array has a length-2 ReIm axis if $K = 8$, and no such axis if $K = 4$. We have covered both cases in (99) by including an array axis with length $(K/4)$.

The "union" shared memory layout (99) is designed so that as the data is processed (one inner block at a time), the output data `Fbar[]` incrementally overwrites the input data `F[]`. This property is important since it will reduce calls to `__syncthreads()`.

The shared memory layout is more complicated than the array notation (99) suggests:

- The `F[]` array is addressed as follows. Each element of `F[]` is indexed by $(\bar{\tau}, \tau', d')$. Let $0 \le \tau'_{\mathrm{rev}} < U/2$ be obtained by bit-reversing $0 \le \tau' < U/2$. Then:

  $$\text{Shared memory offset} = d' + (65 \cdot \tau'_{\mathrm{rev}}) + (\Sigma \cdot \bar{\tau}) \tag{100}$$

  where the stride Σ is defined by:

  $$\Sigma = \begin{cases} 32U + 33 & \text{if } U \le 64 \\ 65(U/2) + 1 & \text{if } U \ge 128 \end{cases} \tag{101}$$

  and satisfies $\Sigma \ge 65(U/2)$ and $\Sigma \equiv 1 \pmod{32}$.

- The `Fbar[]` array is addressed as follows. Each element of `Fbar[]` is indexed by $(\mathrm{ReIm}, \bar{\tau}, u', d')$, where $0 \le \mathrm{ReIm} < (K/4)$. Then:

  $$\text{Shared memory offset} = d' + (65 \cdot u') + (\Sigma \cdot \bar{\tau}) + \left(\frac{T_{\mathrm{outer}}\Sigma}{U} \cdot \mathrm{ReIm}\right) \tag{102}$$

The total shared memory footprint is:

$$\big(\text{Shared memory footprint}\big) = 4\Sigma \left(\frac{K}{4}\right) \left(\frac{T_{\mathrm{outer}}}{U}\right) \text{ bytes} \tag{103}$$

5.4 Register assignments

So far we've defined outer blocks (consisting of $T_{\rm outer}$ time samples and 128 dishes) and inner blocks (consisting of U time samples and 128 dishes). In this section, we'll define two more block types: the "packed miniblock" which is smaller than an inner block, and an "unpacked miniblocks" which is half the size of a packed miniblock.

We mention in advance that the upchannelization kernel will consist of four nested loops, which loop (from outermost loop to innermost) over: outer blocks, inner blocks, packed miniblocks, and finally unpacked miniblocks. (See pseudocode in §5.5.)

The register assignments for these different block types are somewhat complicated, and most of this section is devoted to writing down register assignments explicitly.

Register assignments for inner blocks. Recall that we have defined two types of inner blocks, denoted F and $\bar{F}$ in the previous section. The F-array inner block $F_{\tau' d'}$ was defined near Eq. (95) and consists of bit-shuffled E-array input data. The $\bar{F}$-array inner block $\bar{F}_{u' d'}$ was defined near Eqs. (97), (98), and consists of bit-shuffled $\bar{E}$-array output data. The definition of the $\bar{F}$-array is slightly different for $K = 4$ and $K = 8$ (in the case $K = 8$, there is an extra length-2 ReIm axis).

Suppose that we have either an inner block (either $F_{\tau' d'}$ or $\bar{F}_{u' d'}$) and want to distribute it among registers throughout a threadblock (i.e. among all threads and warps). In all cases, the array shape is $(U/2, 64)$. We will reorganize the length-$(U/2)$ time axis as a 2-d array of shape (U_t, U_r), where the two axes are mapped to (threads, registers) respectively. Similarly, we will reorganize the length-64 dish axis as a 3-d array of shape (W, D_t, D_r), where the three axes are mapped to (warps, threads, registers) respectively. The values of (U_t, U_r, D_t, D_r), and details of the F-array and $\bar{F}$-array register assignments, are given as follows (and are slightly different in the cases $U \leq 32$ and $U \geq 64$):

- If $U \leq 32$ (equivalently $k \leq 5$), then we define (U_t, U_r, D_t, D_r) by:

$$(U_t, U_r) = (2^{k-1}, 1) \qquad (D_t, D_r) = (2^{6-k}, 2^{k-l}) \tag{104}$$

  The register assignments for the F-array and $\bar{F}$-array inner blocks are:

$$\begin{aligned}
[F_{\tau' d'}] \quad & w \leftrightarrow \underbrace{d'_{k-l} \cdots d'_{k-2}, d'_5}_{l \text{ bits}} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_0, d'_{k-1} \cdots d'_4}_{5 \text{ bits}} \quad r \leftrightarrow \underbrace{d'_0 \cdots d'_{k-l-1}}_{(k-l) \text{ bits}} \\
[\bar{F}_{u' d'}]_{K=4} \quad & w \leftrightarrow \underbrace{d'_{k-l} \cdots d'_{k-2}, d'_5}_{l \text{ bits}} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_{k-1}, d'_{k-1} \cdots d'_4}_{5 \text{ bits}} \quad r \leftrightarrow \underbrace{d'_0 \cdots d'_{k-l-1}}_{(k-l) \text{ bits}} \\
[\bar{F}_{u' d'}]_{K=8} \quad & w \leftrightarrow \underbrace{d'_{k-l} \cdots d'_{k-2}, d'_5}_{l \text{ bits}} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_{k-1}, d'_{k-1} \cdots d'_4}_{5 \text{ bits}} \quad r \leftrightarrow \text{ReIm}, \underbrace{d'_0 \cdots d'_{k-l-1}}_{(k-l) \text{ bits}}
\end{aligned} \tag{105}$$

  where the notation $w \leftrightarrow (\cdots)$ denotes the assignment of warps $0 \leq w < W$ to dish index bits. In the last two equations, we have shown $\bar{F}$-array register assignments separately for the $K = 4$ and $K = 8$ cases. (In the $K = 8$ case, we just add a length-two ReIm axis, which is mapped to a register index bit.)

  Reminder: as in §4.1, we use **bold typeface** for clarity whenever thread index bits are permuted – for example $\mathbf{t_1 t_0 t_2 t_4 t_3}$ above.

- If $U \geq 64$ (equivalently $k \geq 6$), then we define (U_t, U_r, D_t, D_r) by:

$$(U_t, U_r) = (32, 2^{k-6}) \qquad (D_t, D_r) = (1, 2^{6-l}) \tag{106}$$

The register assignments for the F-array and $\bar{F}$-array inner blocks are:

$$\begin{array}{lllll}
[F_{\tau' d'}] & w \leftrightarrow \underbrace{d'_{6-l} \cdots d'_5}_{l \text{ bits}} & \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_{k-6}}_{5 \text{ bits}} & r \leftrightarrow \underbrace{\tau_0 \cdots \tau_{k-7}, d'_0 \cdots d'_{5-l}}_{(k-l) \text{ bits}} \\
[\bar{F}_{u' d'}]_{K=4} & w \leftrightarrow \underbrace{d'_{6-l} \cdots d'_5}_{5 \text{ bits}} & \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_5}_{l \text{ bits}} & r \leftrightarrow \underbrace{u_6 \cdots u_{k-1}, d'_0 \cdots d'_{5-l}}_{(k-l) \text{ bits}} \\
[\bar{F}_{u' d'}]_{K=8} & w \leftrightarrow \underbrace{d'_{6-l} \cdots d'_5}_{l \text{ bits}} & \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_5}_{5 \text{ bits}} & r \leftrightarrow \text{ReIm}, \underbrace{u_6 \cdots u_{k-1}, d'_0 \cdots d'_{5-l}}_{(k-l) \text{ bits}}
\end{array} \tag{107}$$

We will motivate these complicated register assignments at the end of this section! Note that an inner block can be declared on each thread as:

```
int F[Ur][Dr];          // F-array inner block: shape-(Ur,Dr) register array
int Fbar[K/4][Ur][Dr];  // Fbar-array inner block: gets ReIm axis if K=8
```
(108)

Ring buffer register assignment. Throughout the kernel, a logical ring buffer consisting of the previous $(M-1)$ inner blocks is held in registers. There is a compile-time boolean parameter `Packed` which controls whether the ring buffer datatype is int4+4 (`Packed` = `True`) or float16+16 (`Packed` = `False`).[11]

If `Packed` = `True`, then the ring buffer is just $(M-1)$ copies of the F-array inner block (108). It could be declared on each thread as:

```
// Ring buffer (if Packed=True)
int F_ringbuf[M-1][Ur][Dr];
```
(109)

If `Packed` = `False`, then the ring buffer is represented by "unpacking" each F-array register in (112) to four `__half2` registers. Recall from Eq. (95) that each F-array element consists of eight `int4`s, with register assignment

$$b_0 b_1 b_2 \leftrightarrow \text{ReIm}, d'_6, \tau_{k-1} \tag{110}$$

If we unpack this register in a natural way, we get four `__half2` registers with register assignment:

$$b \leftrightarrow \tau_{k-1} \qquad\qquad r \leftrightarrow \text{ReIm}, d'_6 \tag{111}$$

If we perform this unpacking for every F-array register in (112), then the resulting ring buffer representation could be declared on each thread as:

```
// Ring buffer (if Packed=False)
// The length-two axis at the end is the index bit d'_6
__half2 Ere_ringbuf[M-1][Ur][Dr][2];
__half2 Eim_ringbuf[M-1][Ur][Dr][2];
```
(112)

Packed and unpacked miniblocks. By Eq. (108), an F-array inner block is declared on each thread as `int F[Ur][Dr]`, and an $\bar{F}$-array inner block is declared as `int F[K/4][Ur][Dr]`.

We define a *packed miniblock* to be the subset of an inner block obtained by fixing the index $0 \le d' < D_r$ to a specific value. Thus, a packed miniblock is declared on each thread as:

```
int F[Ur];          // F-array packed miniblock
int Fbar[K/4][Ur];  // Fbar-array packed miniblock (with ReIm axis if K=8)
```
(113)

Note that a packed miniblock contains two values of $0 \le d'_6 < 2$. We define an *unpacked miniblock* by "unpacking" the packed miniblock to `__half2`, and fixing the index $0 \le d'_6 < 2$ to a specific value. An unpacked miniblock is declared on each thread as:

```
int Ere[Ur], Eim[Ur];     // E-array packed miniblock
int Ebar[Ur], Eim[Ur];    // Ebar-array packed miniblock
```
(114)

[11] For small values of MU, `Packed` = `False` is preferable since it reduces the number of `int4`→`float16` conversion instructions. For large values of MU, `Packed` = `True` will be necessary, in order to avoid running out of registers.

Note that an inner block consists of U_r packed miniblocks, and a packed miniblock "unpacks" to two unpacked miniblocks.

The register assignments for miniblocks are the "natural" ones, given the register assignments for inner blocks in Eqs. (104)–(107). For reference, we write out all miniblock register assignments concretely:

- If $U \leq 32$, then the miniblock register assignments are (5 cases):

$$\text{[Packed } F_{\tau' d'}\text{]} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_0, d'_{k-1} \cdots d'_4}_{5 \text{ bits}} \tag{115}$$

$$\text{[Packed } \bar{F}_{u' d'}\text{]}_{K=4} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_{k-1}, d'_{k-1} \cdots d'_4}_{5 \text{ bits}} \tag{116}$$

$$\text{[Packed } \bar{F}_{u' d'}\text{]}_{K=8} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_{k-1}, d'_{k-1} \cdots d'_4}_{5 \text{ bits}} \quad r \leftrightarrow \text{ReIm} \tag{117}$$

$$\text{[Unpacked } E_{\tau' d'}\text{]} \qquad b \leftrightarrow \tau_{k-1} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_0, d'_{k-1} \cdots d'_4}_{5 \text{ bits}} \quad r \leftrightarrow \text{ReIm} \tag{118}$$

$$\text{[Unpacked } \bar{E}_{u' d'}\text{]} \qquad b \leftrightarrow u_0 \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_{k-1}, d'_{k-1} \cdots d'_4}_{5 \text{ bits}} \quad r \leftrightarrow \text{ReIm} \tag{119}$$

  In all five cases, the assignment of warp index bits $w \leftrightarrow d'_{k-l} \cdots d'_{k-2}, d'_5$ is implicit. (This is the same assignment as in Eq. (105).)

- If $U \geq 64$, then the miniblock register assignments are (5 cases):

$$\text{[Packed } F_{\tau' d'}\text{]} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_{k-6}}_{5 \text{ bits}} \quad r \leftrightarrow \underbrace{\tau_0 \cdots \tau_{k-7}}_{(k-6) \text{ bits}} \tag{120}$$

$$\text{[Packed } \bar{F}_{u' d'}\text{]}_{K=4} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_5}_{5 \text{ bits}} \quad r \leftrightarrow \underbrace{u_6 \cdots u_{k-1}}_{(k-6) \text{ bits}} \tag{121}$$

$$\text{[Packed } \bar{F}_{u' d'}\text{]}_{K=8} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_5}_{5 \text{ bits}} \quad r \leftrightarrow \text{ReIm}, \underbrace{u_6 \cdots u_{k-1}}_{(k-6) \text{ bits}} \tag{122}$$

$$\text{[Unpacked } E_{\tau' d'}\text{]} \qquad b \leftrightarrow \tau_{k-1} \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_{k-6}}_{5 \text{ bits}} \quad r \leftrightarrow \text{ReIm}, \underbrace{\tau_0 \cdots \tau_{k-7}}_{k-6 \text{ bits}} \tag{123}$$

$$\text{[Unpacked } \bar{E}_{u' d'}\text{]} \qquad b \leftrightarrow u_0 \qquad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_5}_{5 \text{ bits}} \quad r \leftrightarrow \text{ReIm}, \underbrace{u_6 \cdots u_{k-1}}_{(k-6) \text{ bits}} \tag{124}$$

  In all five cases, the assignment of warp index bits $w \leftrightarrow d'_{6-l} \cdots d'_5$ is implicit. (This is the same assignment as in Eq. (107).)

Register asssignments for PFB weight array, gains, and extra phases. At the beginning of the kernel, three `float16` arrays are read into registers: the shape-(M, U) PFB weight array $W_{m\tau'}$, the length-U gain array G_u, and the length-U array of "extra phases":

$$X_{\tau'} = e^{\pi i \tau'(U-1)/U} \qquad (0 \leq \tau' < U) \tag{125}$$

Each warp will get its own copy of all 3 arrays, but the array elements will be distributed among threads in the warp. We use register assignments which are compatible with the register assignments for unpacked miniblocks (Eqs. (118), (119), (123), (124)). More precisely:

- If $U \leq 32$, then the register assignments are:

$$[\text{float16 } W_{m\tau'}] \quad b \leftrightarrow \tau_{k-1} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_0}_{\text{(k−1) bits}} + \underbrace{(\text{spectator bits})}_{\text{(6−}k\text{) bits}} \quad r \leftrightarrow m \tag{126}$$

$$[\text{float16 } X_{\tau'}] \quad b \leftrightarrow \tau_{k-1} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_0}_{\text{(k−1) bits}} + \underbrace{(\text{spectator bits})}_{\text{(6−}k\text{) bits}} \quad r \leftrightarrow \text{ReIm} \tag{127}$$

$$[\text{float16 } G_u] \quad b \leftrightarrow u_0 \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_{k-1}}_{\text{(k−1) bits}} + \underbrace{(\text{spectator bits})}_{\text{(6−}k\text{) bits}} \tag{128}$$

  where the presence of "spectator bits" means that the stored register value does not depend on the appropriate thread index t_i.

- If $U \geq 64$, then the register assignments are:

$$[\text{float16 } W_{m\tau'}] \quad b \leftrightarrow \tau_{k-1} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_{k-6}}_{\text{5 bits}} \quad r \leftrightarrow m,\ \underbrace{\tau_0 \cdots \tau_{k-7}}_{k-6 \text{ bits}} \tag{129}$$

$$[\text{float16 } X_{\tau'}] \quad b \leftrightarrow \tau_{k-1} \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{\tau_{k-2} \cdots \tau_{k-6}}_{\text{5 bits}} \quad r \leftrightarrow \text{ReIm},\ \underbrace{\tau_0 \cdots \tau_{k-7}}_{k-6 \text{ bits}} \tag{130}$$

$$[\text{float16 } G_u] \quad b \leftrightarrow u_0 \quad \mathbf{t_1 t_0 t_2 t_4 t_3} \leftrightarrow \underbrace{u_1 \cdots u_5}_{\text{5 bits}} \quad r \leftrightarrow \underbrace{u_6 \cdots u_{k-1}}_{(k-6) \text{ bits}} \tag{131}$$

In both cases, the arrays can be declared as:

```
__half2 Wpfb[M][Ur];
__half2 Xre[Ur], Xim[Ur];  // "Extra" phases
__half2 G[Ur];             // Gains
```
(132)

Why are these register assignments so messy? So far we haven't motivated the complicated register assignments in this section! The register assignments have been desigined to have the following nice properties:

- Consistency with FFT.

  You may have noticed that unpacked miniblocks are the same thing (including register assignments) as the FFT input/output arrays from §4. More precisely, the unpacked E-array miniblock register assignment in Eqs. (118), (123) agrees with the FFT input array register assignment given previously in Eqs. (52), (54). And the unpacked $\bar{E}$-array miniblock register assignment in Eqs. (119), (124) agrees with the FFT output array register assignment given previously in Eqs. (53), (55).

- No bank conflicts.

  An F-array or $\bar{F}$-array inner block can be read/written to shared memory without bank conflicts. This statement is not obvious, but follows from the shared memory layout given previously in Eqs. (99)–(102), and the inner block register assignments in Eqs. (105), (107) above.

5.5 Implementation details

We start with high-level pseudocode, assuming `Packed=True` for definiteness. The case `Packed=False` is described at the end of this section.

```
int F_ringbuf[M-1][Ur][Dr];  // Initialized to zero
__half2 Gains[Ur];           // Gains
__half2 Wpfb[M][Ur];         // PFB weight function
__half2 Xre[Ur], Xim[Ur];    // Extra phases

// Outermost loop over outer blocks (T_outer time samples)
for (int t_outer = 0; t_outer < NTIME; t_outer += T_outer) {
    (Copy outer block global -> shared);  // Step 1 below
    __syncthreads();

    // Loop over inner blocks (T_inner time samples)
    for (int t_inner = 0; t_inner < T_outer; t_inner += U) {

        // Loop over packed miniblocks (indexed by 0 <= dr < Dr).
        // Loops must be unrolled starting here.
        #pragma unroll
        for (int dr = 0; dr < Dr; dr++) {
            int F_in[Ur] = (Read F-array miniblock from shared);  // Step 2 below
            int Fbar_out[K/4][Ur] = 0;

            #pragma unroll
            // Loop over unpacked miniblocks (indexed by 0 <= d'_6 < 2).
            for (int dprime6 = 0; dprime6 < 2; dprime6++) {
                // Steps 3-8 below
                __half2 Ere[Ur], Eim[Ur] = (compute E by unpacking F_in);
                __half2 E2re[Ur], E2im[Ur] = (compute E2 from E);
                __half2 E3re[Ur], E3im[Ur] = (compute E3 by applying phases to E2);
                __half2 E4re[Ur], E4im[Ur] = (compute E4 by FFTing E3);
                __half2 E5re[Ur], E5im[Ur] = (compute E5 by applying gains to E4);
                (Compute Fbar_out by quantizing and packing E5);
            }

            (Write packed miniblock Fbar_out -> shared);   // Step 9 below

            // Advance ring buffer
            #pragma unroll
            for (int ur = 0; ur < Ur; ur++) {
                #pragma unroll
                for (int m = 0; m < M-2; m++)
                    F_ringbuf[m][ur][dr] = F_ringbuf[m+1][ur][dr];
                F_ringbuf[M-1][ur][dr] = F_in[ur];
            }
        }
    }

    __syncthreads();
    (Copy outer block shared memory -> global);  // Step 10 below
}
```

Next we fill in the details in the individual steps 1–10 in the pseudocode:

Step 1: copy outer block from global memory to shared memory. In this step, we copy one outer block ($T_{\rm outer}$ time samples) of data from global memory to shared memory. The destination shared memory layout was given in Eqs. (99)–(102). Along the way, we will convert the data from its E-array representation to its F-array representation, which will require some warp/local transposes.

On each warp, we read E-array data from global memory in the following register assignment:

$$b_0 b_1 b_2 \leftrightarrow \mathrm{ReIm}, d_0, d_1 \qquad\qquad r_0 r_1 \leftrightarrow d_2 d_3 \qquad\qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_4 d_5 d_6 \tau_{k-1} X \tag{133}$$

where we define index bit

$$X = \begin{cases} \bar{\tau}_0 & \text{if } U = 2 \text{ (equivalently } k = 1) \\ \tau_{k-2} & \text{if } U \geq 4 \text{ (equivalently } k \geq 2) \end{cases} \tag{134}$$

Loading global memory into register assignment (133) can be done with a 16-byte aligned load.

Define an *input tile* to be the E-array data (133) on a single warp. One input tile corresponds to 4 time samples and all 128 dishes. The 4 time samples in an input tile are not contiguous; they are obtained by varying index bits τ_{k-1} and X. We process the outer block by assigning its $(T_{\rm outer}/4)$ input tiles to warps arbitrarily. To do this cleanly, we assume (previously stated in Eq. (86)):

```
static_assert(Touter % (4*W) == 0);
```
(135)

Starting from the input tile (133), we do some warp/local transposes, to obtain register assignment:

$$b_0 b_1 b_2 \leftrightarrow \mathrm{ReIm}, d_0, \tau_{k-1} \qquad\qquad r_0 r_1 \leftrightarrow d_2 d_1 \qquad\qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_4 d_5 d_6 d_3 X \tag{136}$$

Using the definition (95) of the F-array, and the definition (94) of the primed dish index d', we reinterpret this data as four elements of the array $F_{\tau d'}$, with register assignment:

$$r_0 r_1 \leftrightarrow d'_0 d'_5 \qquad\qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d'_2 d'_3 d'_4 d'_1 X \tag{137}$$

We write these F-array elements to shared memory. Using the shared memory layout (99)–(102), one can check (nontrivial!) that the stores are bank conflict free. (Warning: the choice (134) of index bit X matters, and a different choice of X could lead to bank conflicts.)

Step 2: read F-array miniblock from shared memory. In the pseudocode above, this step appears inside the loop over packed miniblocks (i.e. $0 \leq d_r < D_r$). We read one F-array packed miniblock from shared memory. The packed miniblock is declared as `int F_in[Ur]` in the pseudocode above, and its register assignment was given in Eqs. (115), (120) above. Using these register assignments, and the shared memory layout (99)–(102), one can check (nontrivial!) that the loads are bank conflict free.

Step 3: Compute $\mathbf{E}$ by unpacking $\mathbf{F}_{\rm in}$. In the pseudocode above, this step appears inside the loop over *unpacked* miniblocks (i.e. $0 \leq d'_6 < 2$). We start with the packed miniblock `int F_in[Ur]` from the previous step. For the appropriate value of $0 \leq d'_6 < 2$, we "unpack" `int4`→`float16`, obtaining an unpacked E-miniblock (declared as `__half2 Ere[Ur], Eim[Ur]`; register assignment in Eqs. (118), (123)).

Step 4: Compute $\mathbf{E}^{(2)}$ from $\mathbf{E}$. Recall from Eq. (89) that the $E^{(2)}$-array is defined by:

$$E^{(2)}_{\bar{\tau}\tau' d} = \sum_{m=0}^{M-1} (-1)^m W_{m\tau'} \, E_{(\bar{\tau}+m-M+1),\tau',d} \tag{138}$$

The last term in the sum (i.e. $m = M-1$) is the unpacked miniblock computed in the last step. Since we are assuming `Packed=True` (see comment at beginning of §5.5), the other terms ($0 \leq m < M-1$) are held in packed miniblocks (in the ring buffer). Therefore, to compute $E^{(2)}$, we unpack $(M-1)$ miniblocks "on the fly" and accumulate their contributions to the $E^{(2)}$-array unpacked miniblock.

Step 5: Compute $\mathbf{E}^{(3)}$ by applying phases to $\mathbf{E}^{(2)}$. Recall from Eq. (90) that the $E^{(3)}$-array is defined by:

$$E^{(3)}_{\bar{\tau}\tau' d} = \underbrace{e^{\pi i \tau'(U-1)/U}}_{\text{"Extra" phase}} E^{(2)}_{\bar{\tau}\tau' d} \tag{139}$$

In this step, we compute one $E^{(3)}$-array unpacked miniblock from one $E^{(2)}$ unpacked miniblock. The register assignments for the unpacked miniblock (Eqs. (118), (123)) and the "extra" phases (Eqs. (127), (130)) have been chosen so that the complex multiplication can be implemented straightforwardly with `__half2` FMAs.

Step 6: Compute $\bar{\mathbf{E}}^{(4)}$ by FFTing $\mathbf{E}^{(3)}$. Recall from Eq. (91) that the $\bar{E}^{(4)}$-array is the FFT of the $E^{(3)}$-array:

$$\bar{E}^{(4)}_{\bar{\tau}ud} = \sum_{\tau'=0}^{U-1} E^{(3)}_{\bar{\tau}\tau' d}\, e^{-2\pi i \tau' u/U} \tag{140}$$

In this step, we compute one $\bar{E}^{(4)}$-array unpacked miniblock from one $E^{(3)}$ unpacked miniblock, by calling the FFT microkernel from §4. No change in register assignments is needed, since the FFT register assignments (Eqs. (52)–(55)) are consistent with the unpacked miniblock register assignments (Eqs. (118), (119), (123), (124)) with dishes playing the role of FFT spectator indices.

Step 7: Compute $\bar{\mathbf{E}}^{(5)}$ by applying gains to $\bar{\mathbf{E}}^{(4)}$. Recall from Eq. (92) that the $\bar{E}^{(5)}$-array is defined by applying gains:

$$\bar{E}^{(5)}_{\bar{\tau}ud} = G_{ud}\, \bar{E}^{(4)}_{\bar{\tau}ud} \tag{141}$$

In this step, we compute one $\bar{E}^{(5)}$-array unpacked miniblock from one $\bar{E}^{(4)}$-array unpacked miniblock. The register assignments for the unpacked miniblock (Eqs. (119), (124)) and the gain array (Eqs. (128), (131)) have been chosen so that the multiplication can be implemented straightforwardly with `__half2` FMAs.

Step 8: Compute $\bar{\mathbf{F}}_{\rm out}$ by applying gains to $\bar{\mathbf{E}}^{(5)}$. This step is roughly the inverse of step 3. Starting with the $\bar{E}^{(5)}$-array unpacked miniblock from the previous step, we quantize the data to either `int4` or `int8` (depending on whether $K = 4$ or 8), and put the resulting bits into the packed miniblock $\bar{F}_{\rm out}$. (Don't forget to clamp when quantizing!)

Note that in the pseudocode above, this step appears in the innermost loop over unpacked miniblocks (i.e. $0 \le d'_6 < 2$ is fixed). In each of the two iterations of the innermost loop, half of the bits in $\bar{F}_{\rm out}$ are set. After both iterations of the innermost loop, we have computed one $\bar{F}_{\rm out}$-array packed miniblock (declared as `int[K/4][Ur]`; register assignment in Eqs. (116), (117), (121), (122)).

Step 9: Write $\bar{\mathbf{F}}_{\rm out}$ to shared memory. This step is roughly the inverse of step 2. We write the $\bar{F}_{\rm out}$-array packed miniblock from the previous step to shared memory. Using the $\bar{F}_{\rm out}$-array register assignment (Eqs. (116), (117), (121), (122)) and the shared memory layout (99)–(102), one can check (nontrivial!) that the stores are bank conflict free.

Step 10a: Copy outer block from shared memory to global memory (K=4 case). This step is roughly the inverse of step 1. We assume $K = 4$ (for the $K = 8$ case, see "Step 10b" below).

On each warp, we read $\bar{F}$-array data from shared memory in the following register assignment:

$$r_0 r_1 \leftrightarrow d'_0 d'_5 \qquad\qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d'_2 d'_3 d'_4 d'_1 X \tag{142}$$

where we define index bit

$$X = \begin{cases} \bar{\tau}_0 & \text{if } U = 2 \text{ (equivalently } k = 1) \\ u_1 & \text{if } U \ge 4 \text{ (equivalently } k \ge 2) \end{cases} \tag{143}$$

Using the shared memory layout (99)–(102), one can check (nontrivial!) that these shared memory loads are bank conflict free.

Define an *output tile* to be the $\bar{F}$-array data (142) on a single warp. Previously in step 1 (near Eq. (133)) we defined input tiles similarly. Comparing the definitions, one can check that each input tile corresponds to a unique output tile, in the sense that the two tiles occupy the same shared memory addresses.

We process the outer block by assigning output tiles to warps. **Important note!!** This assignment is not arbitrary – each warp must process the output tiles which correspond to the input tiles that the warp processed in step 1. This will let us remove a call to `__syncthreads()`; see discussion at the end of this section.

Starting from the output tile (142), and using the definition (97) of the $\bar{F}$-array, we reinterpret the output tile as elements of the $\bar{E}$-array, with register assignment:

$$b_0 b_1 b_2 \leftrightarrow \mathrm{ReIm}, d_0 u_0 \qquad r_0 r_1 \leftrightarrow d_2 d_1 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_4 d_5 d_6 d_3 X \tag{144}$$

We do some warp and local transpose operations, to obtain register assignment:

$$b_0 b_1 b_2 \leftrightarrow \text{ReIm}, d_0, d_1 \qquad r_0 r_1 \leftrightarrow d_2 d_3 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_4 d_5 d_6 u_0 X \tag{145}$$

In this register assignment, the data can be written to global memory with one 16-byte cache-aligned store instruction.

Step 10b: Copy outer block from shared memory to global memory (K=8 case). In the $K = 8$ case, on each warp we read $\bar{F}$-array data in the following register assignment:

$$r \leftrightarrow \text{ReIm}, d'_0 d'_5 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d'_1 d'_2 d'_3 d'_4 X \tag{146}$$

where we define index bit

$$X = \begin{cases} \bar{\tau}_0 & \text{if } U = 2 \text{ (equivalently } k = 1) \\ u_1 & \text{if } U \geq 4 \text{ (equivalently } k \geq 2) \end{cases} \tag{147}$$

Using the shared memory layout (99)–(102), one can check (nontrivial!) that these shared memory loads are bank conflict free. Note that the definition of X is the same as the $K = 4$ case (Eq. (143)), but the thread index bits in (146) have been permuted relative to the $K = 4$ case (Eq. (142)).

Define an *output tile* to be the $\bar{F}$-array data (146) on a single warp. Comparing with the definition (133) of input tiles, one can check that each input tile corresponds to a unique output tile, in the sense that the input tile is a subset in shared memory of the output tile.

We process the outer block by assigning output tiles to warps. **Important note!!** This assignment is not arbitrary – each warp must process the output tiles which correspond to the input tiles that the warp processed in step 1. This will let us remove a call to `__syncthreads()`; see discussion at the end of this section.

Starting from the output tile (146), and using the definition (98) of the $\bar{F}$-array, we reinterpret the output tile as elements of the $\bar{E}$-array, with register assignment:

$$b_0 b_1 \leftrightarrow d_0 u_0 \qquad r \leftrightarrow \text{ReIm}, d_2 d_1 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_3 d_4 d_5 d_6 X \tag{148}$$

We do some thread-local transpose operations, to obtain register assignment:

$$b_0 b_1 \leftrightarrow \text{ReIm}, d_0 \qquad r \leftrightarrow d_1 d_2 u_0 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_3 d_4 d_5 d_6 X \tag{149}$$

In this register assignment, the data can be written to global memory with two 16-byte cache-aligned store instructions.

What if Packed=False? So far in this section, we have assumed `Packed = True`. Here, we describe the minor changes that are needed in the `Packed = False` case:

- The ring buffer is now declared as:

```
// Ring buffer (if Packed=False)
// The length-two axis at the end is d'_6.
__half2 Ere_ringbuf[M-1][Ur][Dr][2];
__half2 Eim_ringbuf[M-1][Ur][Dr][2];
```
(150)

- In step 2 ("Compute $E^{(2)}$ from E"), we no longer need unpacking operations, since the ring buffer is `float16`.

- Advancing the ring buffer is now done inside the loop over unpacked miniblocks ($0 \leq d'_6 < 2$) and the

pseudocode looks like:

```
// Advance ring buffer
#pragma unroll
for (int ur = 0; ur < Ur; ur++) {
    #pragma unroll
    for (int m = 0; m < M-2; m++) {
        Ere_ringbuf[m][ur][dr][dprime6] = Ere_ringbuf[m+1][ur][dr][dprime6];
        Eim_ringbuf[m][ur][dr][dprime6] = Eim_ringbuf[m+1][ur][dr][dprime6];
    }
    Ere_ringbuf[M-1][ur][dr][dprime6] = Ere[ur];
    Eim_ringbuf[M-1][ur][dr][dprime6] = Eim[ur];
}
```

(151)

A trick for reducing calls to syncthreads()? The pseudocode at the beginning of this section appears to have a bug: not enough calls to `__syncthreads()` to protect shared memory from race conditions between warps. I think that the pseudocode is correct as written! Here is the argument, please let me know if you agree/disagree:

- First let's argue that we do not need a call to `__syncthreads()` at the bottom of the outermost loop.

  Each warp writes to shared memory in step 1, and reads from shared memory in step 10. The details of steps 1 and 10 (and the shared memory layout in Eqs. (99)–(102)) have been constructed so that the shared memory regions which are read/written by each warp are non-overlapping. (This is not obvious, but follows from the "important notes" in steps 10a/b above, and the shared memory layouts in Eqs. (99)–(102) above.)

  Therefore, we do not need a call to `__syncthreads()` at the bottom of the outermost loop – each warp w can proceed to the next iteration without waiting for warps $w' \neq w$.

- A similar argument shows that we do not need any calls to `__syncthreads()` in the inner loop over packed miniblocks.

  We read F-array packed miniblocks from shared memory in step 2, and we write $\bar{F}$-array packed miniblocks to shared memory in step 9. The shared memory layout in Eqs. (99)–(102) ensures that when each $\bar{F}$ output miniblock is written to shared memory, it overwrites the memory locations of the corresponding F input miniblock.

  Therefore, the $\bar{F}$ output miniblocks can be written in an arbitrary order, without worrying about race conditions. In particular, we do not need any calls to `__syncthreads()` in the inner loop over packed miniblocks – each warp can proceed to the next packed miniblock without waiting for the other warps.

5.6 Choosing compile-time parameters

The compile-time parameters (U, M, K) are externally specified. For each choice of (U, M, K), we are free to choose values of the "derived" compile-time parameters $(W, B, T_{\rm outer}, \texttt{Packed})$ which lead to optimal performance.

Planner script. I wrote a short python "planner" script which predicts register usage as a function of $(U, M, K, W, \texttt{Packed})$, by adding the following contributions:

- Registers storing FFT precomputed phases ($R_{\rm phase}$ in Table 1).
- Registers needed to store the ring buffer.
- Registers needed to store the gain array G_u, the "extra" phase array X_τ, and the PFB weights $W_{\rm pfb}$ (see §5.4).
- Registers needed to store the `F_in` and `Fbar_out` arrays, and one temporary unpacked miniblock `Ere+Eim` (see pseudocode in §5.5).

- 10 miscellaneous registers (global memory pointers, shared memory offsets, loop counters, etc.)

The planner also reports shared memory usage, and a cycle-counting estimate of GPU resources used (see §5.7). I'm happy to share the planner script if it's useful!

The planner script should be a useful tool for predicting which values of $(W, B, T_{\rm outer}, \texttt{Packed})$ are interesting candidates for given values of (U, M, K). However, to choose between candidate values, we'll need an implementation for timing. Here's an example of using the planner script:

Example: $(\mathbf{U}, \mathbf{M}, \mathbf{K}) = (\mathbf{128}, \mathbf{2}, \mathbf{4})$. These are plausible upchannelization parameters for the FRB search near the bottom of the CHORD band ($f \sim 300$ MHz).

- Let's try setting `Packed = False` to minimize cost of `int4`→`float16` conversion, and let's try to fully occupy the GPU by setting $(W, B) = (32, 1)$. With these parameters, the planner predicts 57 registers/thread (out of 64 available). Since the planner is a rough estimate, these parameters are worth trying as a baseline, but may or may not lead to spill code.

  Next we choose $T_{\rm outer}$ based on shared memory footprint (larger $T_{\rm outer}$ is better since fewer calls to `__syncthreads()`, but also uses more shared memory). According to the planner, $T_{\rm outer} = 512$ uses 66576 bytes of shared memory – this is probably a good choice, given $W = 32$ warps per threadblock.

  Summarizing so far, $(W, B, T_{\rm outer}, \texttt{Packed}) = (32, 1, 512, \texttt{False})$ is an interesting candidate set of derived compile-time parameters. It may perform well, or it may lead to spill code – we need an implementation to check!

- If the parameters from the first bullet point don't work well, then we need to reduce register usage. One option is $(\texttt{Packed}, W, B, T_{\rm outer}) = (\texttt{True}, 32, 1, 512)$, i.e. storing the ring buffer in packed form. The planner now predicts 45 registers/thread (out of 64 available). Even though this value is a rough estimate, this seems like a large enough margin that spill code is unlikely.

- If the parameters from the first bullet point don't work well, then another option to reduce register usage is $(\texttt{Packed}, W, B, T_{rmouter}) = (\texttt{False}, 16, 1, 512)$, i.e. reducing the number of active warps. The planner now predicts 81 registers/thread (out of 128 available). This seems like a large enough margin that spill code is very unlikely.

5.7 Computational cost and discussion

The planner script also produces a cycle-counting estimate of computational cost. The key assumptions are as follows:

- Computational cost of the FFT was described in §4.8.

- We assume that `int4`→`float16` conversion takes 1 clock cycle per output register, and `float16` → (`int4` or `int8`) conversion takes 2 clock cycles per output register. These values are based on a `#bx-engine-dev` slack exchange between Erik and me on 2022 March 18.

- We assume that a `__half2` FMA costs 0.5 clock cycles.

- We assume that shared memory I/O costs 2 clock cycles per register.

- We assume that the transposing operations which convert $E \to F$ cost 2 cycles per register, and likewise for the transposing operations which convert $\bar{F} \to \bar{E}$ (steps 1 and 10 in

- We assume a 128×A40 correlator: 16 frequencies per GPU, 512 dual-polarization dishes, 1.7 μsec time sampling.

- We assume each A40 GPU has 84 SMs, 600 GB/sec global memory bandwidth, and clock rate 1.7 GHz.

To get a bottom-line estimate of computational cost, we need to know something about which values of (U, M, K) are specified, for each coarse frequency channel. Working this out precisely will take some time, and some coordination with the different CHORD science groups!

In the meantime, I propose that we assume that every frequency channel is upchannelized with parameters $(U, M, K) = (16, 4, 4)$, to get a rough bottom-line estimate of the cost of upchannelization. Here, I think good values of the "derived" parameters $(\texttt{Packed}, W, B, T_{\rm outer})$ are pretty unambiguous. This upchannelization is short enough that register pressure should be a nonissue, so let's choose `Packed` = `False` to save a few cycles, and choose $W = 16$ to get as much occupancy as possible (recall from Eq. (86) that W must be $\leq U$). With these parameters, the planner predicts 41 registers/thread, so there is no problem taking $B = 2$ to fully occupy the GPU. Then we take $T_{\rm outer} = 256$, to end up with around 64 KB of shared memory usage per SM (more precisely, 69760 bytes).

With the above values of $(U, M, K, \texttt{Packed}, W)$, the planner predicts:

$$\text{(Computational cost)} = \underbrace{(1.6\%)}_{\text{input bandwidth}} + \underbrace{(1.6\%)}_{\text{output bandwidth}} + \underbrace{(3.2\%)}_{\text{compute}} = 6.4\% \text{ of GPU resources} \tag{152}$$

Let's see how this cycle-counting estimate compares with implementation!

Some final comments.

- According to the planner script, using `Packed` = `True` would change the second term in Eq. (152) from 3.2% to 3.7%. This justifies the statement from §5.2 that the speed improvement going from `Packed` = `True` to `Packed` = `False` is modest.
- We might also be able to save a few cycles by doing the "extra credit" in §4.9. Otherwise I don't see much scope for optimization!
- As usual, it's hard to be 100% confident about the argument that there are no shared memory bank conflicts, using the complicated memory layouts in §5.3. Let's try to verify this, either with the optimizer, or with a runtime check such as the following (untested):

```
template<typename T>
__device__ inline void assert_bank_confict_free(const T *p)
{
    static_assert(sizeof(T)==4);

    extern __shared__ char shmem_base[];
    ptrdiff_t n = reinterpret_cast<const char *>(p) - shmem_base;

    assert((n >= 0) && (n < 128*1024));  // 'p' points to shared memory
    assert(n % 4 == 0);                  // 'p' is 32-bit aligned

    int bank = (n >> 2) & 0x1f;      // 0 <= bank < 32
    unsigned int flags = __reduce_or_sync(0xffffffff, 1U << bank);
    assert(flags == 0xffffffffU);    // no bank conflicts
}
```

Changes needed for kotekan integration. In these notes, we have neglected some nuisance issues which are unlikely to affect kernel timing, but must be addressed before putting upchannelization into `kotekan`.

- Based on these notes, we would implement the upchannelization kernel as a separately compiled kernel for each choice of (U, M, K).

  In the CHORD correlator, we want to run upchannelization kernels for many choices of (U, M, K). This would be straightforward if we could launch multiple kernels in parallel, but kotekan doesn't currently allow that (instead, all GPU kernel launches are serialized on a single compute stream). This is a technical problem that we'll need to solve (perhaps with `cudaGraph`s?)

- On a related note, we've described the upchannelization kernel as operating on a single long time series. In reality, in order to fully occupy the GPU, we'll want to break the time series into chunks which are processed in separate threadblocks, with a little bit of overlap (at least $(M-1)U$ time samples) between consecutive chunks.
- Similarly, kotekan processes data in chunks (of say 256K time samples), so there is the issue of saving overlaps (at least $(M-1)U$ time samples) from one chunk to the next. One simple approach is to pass an extra pointer argument to the kernel, so that the kernel can save/restore all ring buffer registers to global GPU memory.

References

[1] M. P. Haynes *et al.*, The Astrophysical Journal **861**, 49 (2018).

A Some ALFALFA plots for my own reference

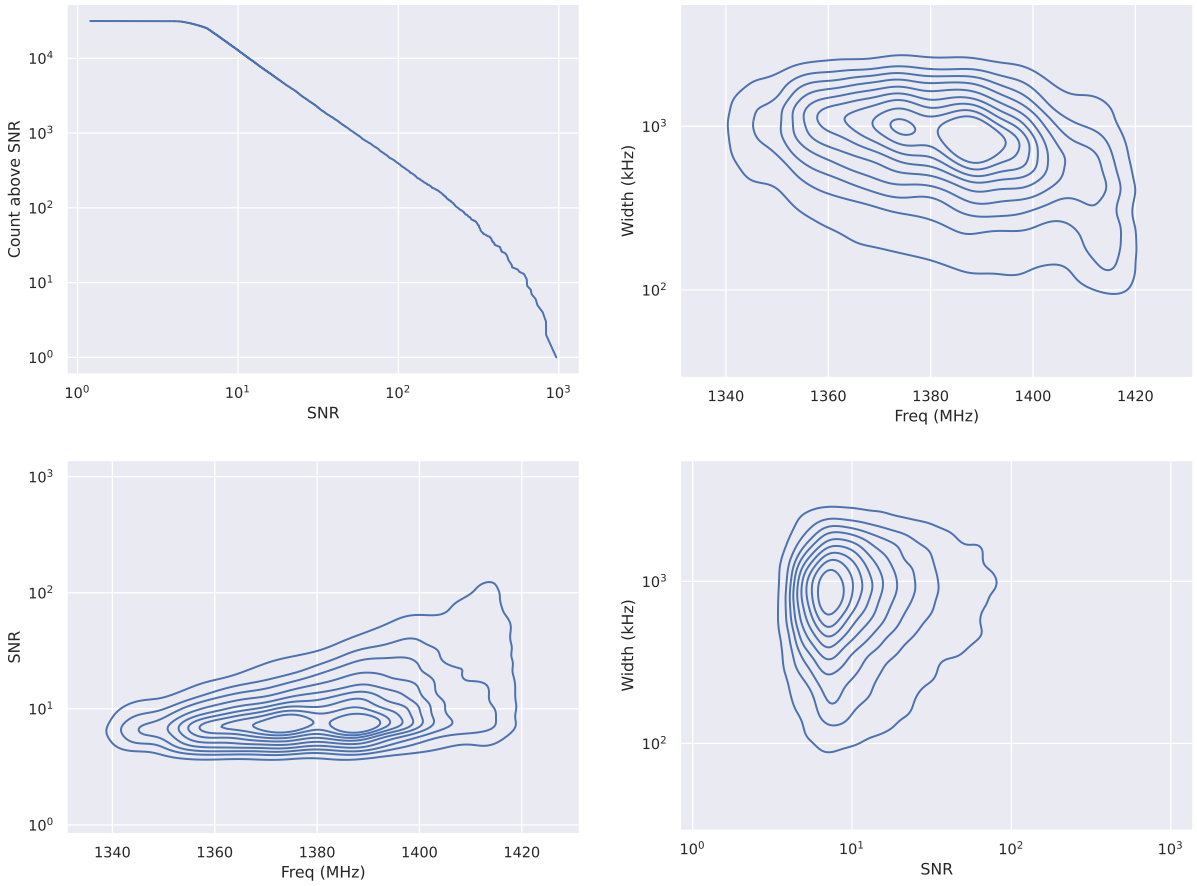

Figure 9: Just leaving these ALFALFA plots here for my own reference. The curves are 10, 20, $\cdots$, 90% level contours.

B GPU kernel preliminaries

In previous kernel design documents, we've developed some concepts which we now re-use in every kernel: register assignment notation, local transposes, and warp transposes. In order to make this design document self-contained, I included appendices which explain these concepts. This is all cut-and-paste from other design documents, so you'll probably skip this appendix entirely!

B.1 Register assignment notation

Throughout these notes, we will frequently encounter situations where an array has been distributed among threads of a warp, and/or among registers on each thread, and/or (if the datatype is smaller than 32 bits) packed into the bytes of registers. In this section, we will introduce notation to keep track of this type of register assignment.

It's easiest to explain our register assignment notation by example. One of the arguments of the `m16n8k16` float16 tensor core MMA (see §C.2) is a 16-by-16 float16 matrix A_{ij}, distributed among threads in a single warp. Each matrix entry has a "logical" location (i, j) in the matrix A_{ij}, and a "physical" location as two bytes in a register somewhere. We describe both logical and physical locations using index bits as follows.

A logical location is described by integers $0 \le i < 16$ and $0 \le j < 16$, which we represent by their binary digits $i = \big[i_3 i_2 i_1 i_0\big]_2$ and $j = \big[j_3 j_2 j_1 j_0\big]_2$. Thus, we label "logical" locations by 8 index bits $i_3 i_2 i_1 i_0 j_3 j_2 j_1 j_0$.

A physical location is indexed by a 5-bit thread id $t = \big[t_4 t_3 t_2 t_1 t_0\big]_2$, a 2-bit register id $r = \big[r_1 t_0\big]_2$ which indexes one of four registers on each thread, and a 1-bit byte id b_0 which indexes the location of the `float16` within the 32-bit register. Thus, we label "physical" locations by 9 index bits $t_4 t_3 t_2 t_1 t_0 r_1 r_0 b_0$.

Our register assignment notation works by writing down the correspondence between logical and physical index bits:

$$[(16 \times 16) \text{ float16 } A_{ij}] \qquad b_0 \leftrightarrow j_0 \qquad r_0 r_1 \leftrightarrow i_3 j_3 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow j_1 j_2 i_0 i_1 i_2 \tag{153}$$

This one-line equation compactly describes how the matrix entries A_{ij} are distributed among registers in the 32 threads which comprise one warp. Some comments on this notation:

- We show the array and its datatype in square brackets, and the number of "byte" index bits b_i will be consistent with the datatype (e.g. two bits $b_1 b_0$ for int8, one bit b_0 for float16).
- The number of registers per thread is 2^R, where R is the number of "register" bits r_i. The example (153) uses four registers per thread.
- For complex-valued arrays, we sometimes use a real datatype, and add an extra logical index bit "ReIm" to indicate how the real/imaginary parts are distributed.

B.2 Local transpose operation

Suppose we have a situation where each thread holds two registers, and each register stores four 8-bit quantities. In our register assignment notation, we write:

$$b_1 b_0 \leftrightarrow XY \qquad r \leftrightarrow Z \tag{154}$$

to indicate that the three "physical" index bits $b_1 b_0 r$ correspond to "logical" index bits XYZ, where the meaning of the logical bits depends on the larger context. (We have omitted the physical thread index bits $t_4 t_3 t_2 t_1 t_0$, since the operation we will describe is thread-local.)

Now suppose that we want to change the register assignment, by swapping the roles of physical index bits b_0 and r, to get the register assignment:

$$b_1 b_0 \leftrightarrow XZ \qquad r \leftrightarrow Y \tag{155}$$

We will call this a "local transpose" operation, since it shuffles data between different registers of the same thread.

Similarly, we might want to transpose physical index bits b_1 and r, so that we obtain the register assignment:

$$b_1 b_0 \leftrightarrow ZY \qquad r \leftrightarrow X \tag{156}$$

Either of the local transpose operations defined in Eqs. (155), (156) can be implemented with two calls to the `__byte_perm()` cuda intrinsic. According to my benchmark, `__byte_perm()` has a throughput of two instructions per cycle (i.e. one local transpose per cycle).

One last comment: to implement the $E \to F$ shuffle operation from §**??**, we may need local transposes on 4-bit boundaries. Such a local transpose can't be implemented with `__byte_perm()`, but could be implemented with two bit-shift instructions and one `LOP3` instruction.

B.3 Warp transpose operation

Now suppose we have a situation where each thread in a warp holds two 32-bit registers:

$$r \leftrightarrow X \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow Y_4 Y_3 Y_2 Y_1 Y_0 \tag{157}$$

where we are now keeping track of the 5-bit thread index $t = [t_4 t_3 t_2 t_1 t_0]_2$, but not keeping track of byte index bits (i.e. we are treating register contents as 32-bit, not 4×8-bit).

Suppose that we want to transpose index bits r and t_i, so that we obtain the register assignment:

$$r \leftrightarrow Y_i \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow Y_4 \cdots \underbrace{X}_{\text{replacing } Y_i} \cdots Y_0 \tag{158}$$

This can be done efficiently with one warp shuffle instruction as follows:[12]

```
int in0 = ...;  // contents of register 0
int in1 = ...;  // contents of register 1

int i = ...; // thread index bit 0 <= i < 5
int bit = 1 << i;
bool flag = (threadIdx.x & bit) != 0;

int src = flag ? in0 : in1;
int dst = __shfl_xor_sync(0xffffffff, src, bit);

 // Compiles to conditional move, not warp-divergent branch.
(flag ? out0 : out1) = dst;
```

We will call this a "warp transpose" operation, since it shuffles data between different threads in the same warp.

[12] Based on my microbenchmarks, the code below will be warp shuffle limited, i.e. the computation of `bit`/`flag` and the conditional assignments involving `src`/`dst` are faster than the warp shuffle and can run in parallel. I also find that warp shuffle throughput is 16 shuffles per clock cycle (where a warp shuffle involving all 32 threads in a warp is defined as 32 shuffles). A puzzle here is that this contradicts nvidia's throughput tables at `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#maximize-instruction-throughput`, which claim 32 shuffles per cycle. If you have any insight on how to get 32 shuffles per cycle, that would be really valuable, since the FRB *search* kernels are sometimes warp shuffle bound. (For the beamforming kernel which is the subject of this note, the cost of warp shuffles turns out to be small (Table **??**), but I thought the larger issue was worth mentioning.

C Float16 tensor core reference

C.1 Float16 m16n8k8

The PTX instruction `mma.sync.aligned.m16n8k8.row.col.f16.f16.f16.f16` performs the following matrix multiplication $C = AB$:

$$[(16 \times 8) \text{ float16 } A_{ij}] \quad b_0 \leftrightarrow j_0 \quad r_0 \leftrightarrow i_3 \quad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2i_0i_1i_2 \tag{159}$$

$$[(8 \times 8) \text{ float16 } B_{jk}] \quad b_0 \leftrightarrow j_0 \quad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2k_0k_1k_2 \tag{160}$$

$$[(16 \times 8) \text{ float16 } C_{ik}] \quad b_0 \leftrightarrow k_0 \quad r_0 \leftrightarrow i_3 \quad t_0t_1t_2t_3t_4 \leftrightarrow k_1k_2i_0i_1i_2 \tag{161}$$

This instruction performs 2048 flops, and costs 2 SM-cycles on an A40.

C.2 Float16 m16n8k16

The PTX instruction `mma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16` performs the following matrix multiplication $C = AB$:

$$[(16 \times 16) \text{ float16 } A_{ij}] \quad b_0 \leftrightarrow j_0 \quad r_0r_1 \leftrightarrow i_3j_3 \quad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2i_0i_1i_2 \tag{162}$$

$$[(16 \times 8) \text{ float16 } B_{jk}] \quad b_0 \leftrightarrow j_0 \quad r_0 \leftrightarrow j_3 \quad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2k_0k_1k_2 \tag{163}$$

$$[(16 \times 8) \text{ float16 } C_{ik}] \quad b_0 \leftrightarrow k_0 \quad r_0 \leftrightarrow i_3 \quad t_0t_1t_2t_3t_4 \leftrightarrow k_1k_2i_0i_1i_2 \tag{164}$$

This instruction performs 4096 flops, and costs 4 SM-cycles on an A40.

C.3 Sparse float16 m16n8k16

Conceptually, the PTX instruction `mma.sp.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16` performs an `m16n8k16` MMA with the same matrix dimensions as the dense case (§C.2), but the A-matrix has 50% sparsity. More precisely, each 1×4 submatrix of the 16×16 matrix A has 50% sparsity, as shown in Figure 10. The computational cost of the sparse MMA is 50% of the dense case.

The sparse MMA defines 6 operands $(A^{\text{sp}}, B, C, D, E, f)$. In the rest of this section, we explain the details of these operands.

Definitions of A^{sp} and E. We first describe a reparameterization of the sparse matrix A_{ij} as a pair of arrays $(A^{\text{sp}}_{ij'J}, E_{ij'J})$. The `float16` array A^{sp} contains the nonzero elements from A_{ij} (and has half the size), and the `int2` array $E_{ij'J}$ describes the sparsity pattern. Both arrays have shape $(16, 4, 2)$.

We split the length-16 column axis of A into (high, low) 2-bit integers (j', E).

$$j = 4j' + E \qquad \text{where } 0 \le j' < 4 \text{ and } 0 \le E < 4 \tag{165}$$

Each 1×4 submatrix in A (Figure 10, left panel) is indexed by a pair (i, j'), where $0 \le i < 16$ and $0 \le j' < 4$. For each such submatrix (i, j'), let $E_{ij'0}, E_{ij'1}$ be 2-bit integers describing the locations of the nonzero entries within the 1×4 submatrix. Let $A^{\text{sp}}_{ij'0}, A^{\text{sp}}_{ij'1}$ be the corresponding `float16` matrix elements. The reparameterization of the sparse A-matrix by $(A^{\text{sp}}_{ij'J}, E_{ij'J})$ is shown visually in Figure 10. Formally, the reparameterization is described by the equation:

```
Asp[i,j',J] = A[i, 4j'+E[i,j',J]]
```
(166)

Register assignments. The register assignments for $(A^{\text{sp}}_{Ij'j}, B_{jk}, C_{ik})$ are straightforward to describe:

$$[(16 \times 4 \times 2) \text{ float16 } A^{\text{sp}}_{ij'J}] \quad b_0 \leftrightarrow J \quad r_0 \leftrightarrow i_3 \quad t_0t_1t_2t_3t_4 \leftrightarrow j_2j_3i_0i_1i_2 \tag{167}$$

$$[(16 \times 8) \text{ float16 } B_{jk}] \quad b_0 \leftrightarrow j_0 \quad r_0 \leftrightarrow j_3 \quad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2k_0k_1k_2 \tag{168}$$

$$[(16 \times 8) \text{ float16 } C_{ik}] \quad b_0 \leftrightarrow k_0 \quad r_0 \leftrightarrow i_3 \quad t_0t_1t_2t_3t_4 \leftrightarrow k_1k_2i_0i_1i_2 \tag{169}$$

Note that, in the register assignment (167) for $A^{\rm sp}_{ij'J}$, we denoted the j' index bits by $j_2 j_3$, rather than $j'_0 j'_1$. The two are equivalent by Eq. (165). We also note that the B and C sparse register assignments (168), (169) are the same as their dense counterparts (163), (164).

Finally, we describe the register assignment for $E_{ij'J}$. The E-array fits into eight 32-bit registers, which are mapped to thread index bits $t_2 t_3 t_4$:

$$[(16 \times 4 \times 2)\ \text{int2}\ E_{ij'J}] \qquad b_0 b_1 b_2 b_3 \leftrightarrow J j_2 j_3 i_3 \qquad t_2 t_3 t_4 \leftrightarrow i_0 i_1 i_2 \tag{170}$$

The sparse MMA instruction includes an additional operand $0 \le f < 4$ which determines which eight threads in the warp are used, by specifying thread index bits $t_0 t_1$. Note that in Eq. (170), the j' index bits are denoted $j_2 j_3$, as in (167) above.

Cuda wrapper. For reference, here is my cuda wrapper for the sparse `m16n8k8` MMA $D = C + AB$, with operands $(A^{\rm sp}, B, C, D, E, f)$ as described above. (The D operand has the same register assignment (169) as C.)

```
// Sparse MMA D = A*B + C
template<unsigned int F>
__device__ __forceinline__
void mma_sp_f16_m16_n8_k16(__half2 d[2], const __half2 asp[2], const __half2 b[2],
                           const __half2 c[2], unsigned int e)
{
    asm("mma.sp.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16 "
        "{%0, %1}, {%2, %3}, {%4, %5}, {%6, %7}, %8, %9;" :
        "=r" (*(unsigned int *) &d[0]), "=r" (*(unsigned int *) &d[1]) :
        "r" (*(const unsigned int *) &asp[0]), "r" (*(const unsigned int *) &asp[1]),
        "r" (*(const unsigned int *) &b[0]), "r" (*(const unsigned int *) &b[1]),
        "r" (*(const unsigned int *) &c[0]), "r" (*(const unsigned int *) &c[1]),
        "r" (e),
        "n" (F)
    );
}
```

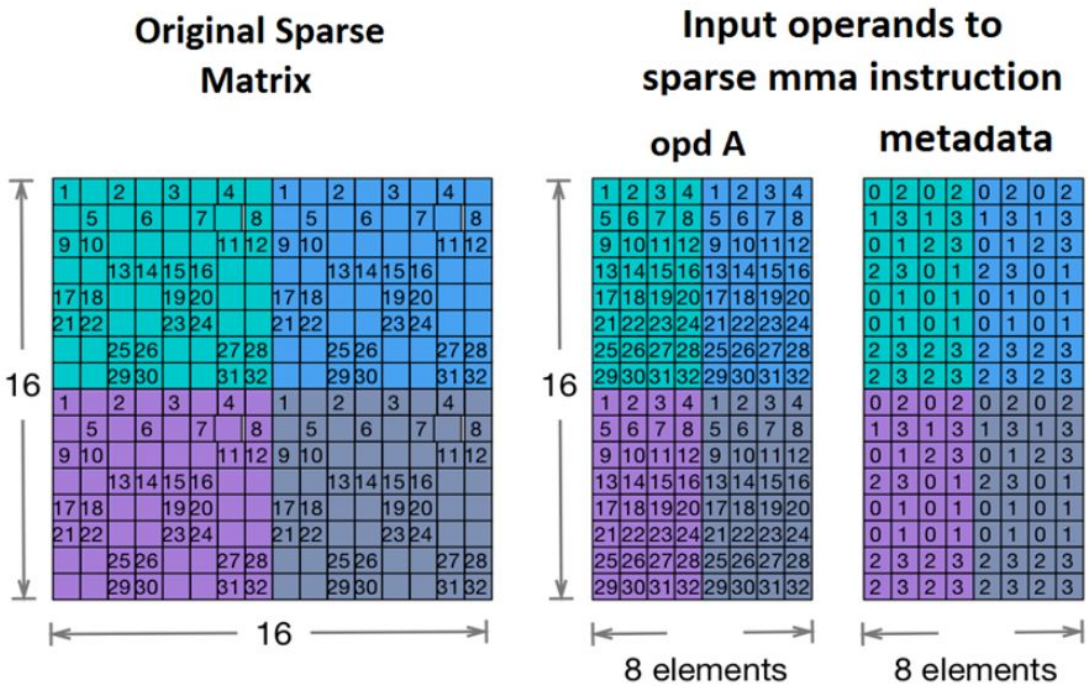

Figure 10: This figure from the nvidia documentation depicts the sparse MMA in §C.3. The 16×16 A-matrix (left) has the property that each 1×4 submatrix is 50% sparse. We split this into two smaller arrays (right): the A^{sp} array which contains nonzero elements of A, and the E array which describes the location of each nonzero element within its 1×4 submatrix. The sparse MMA instruction uses (A^{sp}, E) as operands, instead of A. (Note: the example in the figure also has the property that the same sparsity pattern is repeated 8 times. This property isn't a requirement – the sparse MMA instruction allows an independent sparsity pattern for each 1×4 submatrix.)