CHORD FRB beamformer

Kendrick Smith

November 4, 2022

Contents

1 Executive summary

The FRB beamformer is the most complicated and expensive computation on the GPU correlator. I estimate that it will take about half of the compute time, and about half of the human (programming) time. This monster document describes algorithmic planning for the FRB beamformer. So far we haven't started implementation, but Erik will start soon. This would be a great time to get feedback!

In §2, we describe our beamforming algorithm in mathematical form. It has the following properties:

- Computational cost is $\mathcal{O}(D \log D)^*$, where D is the number of dishes.

  The asterisk means that we are making an approximation, by neglecting terms which are suppressed by inverse powers of the time downsampling factor $T_{\rm ds}$. This turns out to be a good approximation in CHORD, even if the number of beams is large (say $B \sim 5000$).

- Beamformed sky locations are arbitrary, and can be either tracking or non-tracking.

- Spatial kernels are exact, with no interpolation artifacts.

- Float16 arithmetic is used for all intermediate results. (At the very end of the computation, the beamformed intensities will be quantized to `int4` or `int8` for network transmission.)

This is a very appealing set of properties! However, there is one big restriction:

- We assume that the dishes partially populate a **regular** M-by-N grid. Here, $(M, N) = (24, 24)$ for full CHORD (Appendix A). The CHORD pathfinder will either use $(M, N) = (8, 8)$ or $(M, N) = (8, 12)$. We will also consider HIRAX, with either $(M, N) = (16, 16)$ or $(M, N) = (16, 20)$.

In subsequent sections, we plan the GPU implementation.

- In §3, we describe an extremely fast algorithm for doing short float16 FFTs with 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 FFT implementation is the computational foundation for the FRB beamformer.

For technical reasons that will be explained later, we factorize the beamformer into two GPU kernels:[1]

- The first kernel (§4) computes beamformed intensities on a regular grid of sky locations which is oversampled by a factor 2, and writes them to global GPU memory. This is the most expensive and complicated step. For both full CHORD and the CHORD pathfinder (A40 GPUs assumed), we predict it will take ∼14% of GPU resources (see Table 3 in §4.13).

- The second kernel (§5) resamples the beamformed intensities on a specified set of sky locations. For full CHORD, we estimate that the cost will be ∼2% or ∼6% of total GPU resources (A40 GPUs assumed), depending on whether the spatial beam pattern is "factorizable" or "non-factorizable".[2]

The estimated total cost (∼16% or ∼20%) is significantly lower than our earlier estimates, even with a complete set of ∼5000 beams, and with arbitrary beam locations allowed. If our cost estimates prove to be accurate, for the FRB beamformer and other GPU kernels in progress (visibility matrix, baseband beamformer, upchannelization), it seems likely that we will be able to use A10 GPUs instead of A40 GPUs, decreasing the cost of the correlator substantially.

[1]We'll also need a third kernel, to put the intensities in an appropriate form for network transmission, by transposing and quantizing. The third kernel isn't specified in this document, since we haven't decided on details of the packet format yet. But I expect that the third kernel will be easy to write, and its running time will be small.

[2]We say that a spatial beam pattern is *factorizable* if the set of 2-d beam locations $\{(\theta, \theta')\}$ is the Cartesian product of 1-d sets $\{\theta\} \times \{\theta'\}$ (see Figure 1 in §2). Note that Kiyo's beamforming paper [1] uses the term "factorizable" to mean something else!

2 Beamforming algorithm

In this section, we present our beamforming algorithm in mathematical form, deferring details of the GPU implementation to later sections (§3–§5). We will slightly simplify by considering a single frequency channel, neglecting polarization, and neglecting gains.

2.1 Setting up the problem

The inputs to FRB beamforming are:[3]

- The electric field $E_{\tau mn}$, sampled in time τ on a regular dish grid (m, n). This is a complex-valued array with shape (N_{time}, M, N). Grid positions which are unoccupied by dishes are represented by zeros.
- An integer time downsampling factor T_{ds}.
- A sequence of B beam locations $(\theta_\beta, \theta'_\beta)$, where $\beta \in \{0, \cdots, B-1\}$ indexes a beam.

  We say that a spatial beam pattern is *factorizable* if the set of 2-d beam locations $\{(\theta, \theta')\}$ is the Cartesian product of 1-d sets $\{\theta\} \times \{\theta'\}$ (Figure 1). Our FRB beamforming algorithm will be a little faster if the beams are factorizable.

The output is:

- Beamformed intensity $J_{\bar{\tau}\beta}$, a real-valued array with shape $(N_{\text{time}}/T_{\text{ds}}, B)$. Here, we use $\bar{\tau}$ to index a "slow" time sample, after downsampling the "fast" E-array time index τ by a factor T_{ds}.

The beamformed intensity $J_{\bar{\tau}\beta}$ is computed as follows. First, for each beam β, we define the beamformed electric field $\widetilde{E}_{\tau\beta}$ by:[4]

$$\widetilde{E}_{\tau\beta} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} E_{\tau mn} \exp\left(\frac{2\pi i m}{M} \theta_\beta + \frac{2\pi i n}{N} \theta'_\beta \right) \tag{2}$$

Then $J_{\bar{\tau}\beta}$ is given by squaring, and downsampling by a factor T_{ds}:

$$J_{\bar{\tau}\beta} = \sum_{\tau=\bar{\tau}T_{ds}}^{\bar{\tau}(T_{ds}+1)-1} |\widetilde{E}_{\tau\beta}|^2 \tag{3}$$

We are interested in algorithms for computing the beamformed intensity $J_{\bar{\tau}\beta}$ from the electric field $E_{\tau mn}$ and the beam locations $(\theta_\beta, \theta'_\beta)$.

2.2 Proposed $\mathcal{O}(D \log D)^*$ algorithm

FFT beamforming. If the beam locations $(\theta_\beta, \theta'_\beta)$ form a suitable regular grid, then beamforming can be done very efficiently using the FFT algorithm.

As a warmup, suppose that we want to beamform at all *integer* beam locations (θ, θ'). Note that there are MN distinct (up to aliasing) integer beam locations, since Eq. (2) implies that $\widetilde{E}_{\tau\beta}$ is invariant under $\theta \to \theta + M$ or $\theta' \to \theta' + N$. In this case, beamforming could be done straightforwardly, by using a 2-d c2c FFT to compute $\widetilde{E}_{\tau\beta}$ (Eq. (2)).

[3] We denote time indices by τ rather than t, and beam indices by β rather than b. This is to avoid notational confusion in later GPU-centric sections, where GPU threads are indexed by t, and bytes within a register are indexed by b.

[4] In Eq. (2) and throughout these notes, we are using a normalized definition of the sky location (θ, θ') which is related to true sky location as follows. Let $\hat{\boldsymbol{n}}$ be a unit 3-vector pointing toward the true sky location. Let $\boldsymbol{\sigma}$, $\boldsymbol{\sigma}'$ be 3-vectors representing dish displacements along the NS and EW axes (with $|\boldsymbol{\sigma}| = 6.3$ m, and $|\boldsymbol{\sigma}'| = 8.5$ m), and let λ be the radio wavelength. Then:

$$\theta = \frac{M(\hat{\boldsymbol{n}} \cdot \boldsymbol{\sigma})}{\lambda} \qquad \theta' = \frac{N(\hat{\boldsymbol{n}} \cdot \boldsymbol{\sigma}')}{\lambda} \tag{1}$$

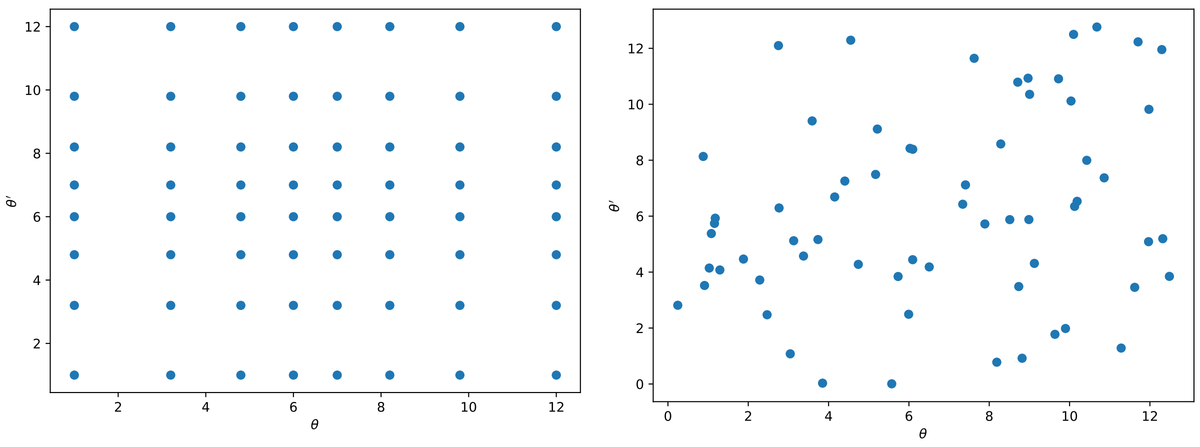

Figure 1: Visual illustration of a factorizable (left) and non-factorizable (right) set of beam locations $\{(\theta, \theta')\}$.

Next, suppose we want to beamform at all *half-integer* beam locations (θ, θ'). Then, to compute $\widetilde{E}_{\tau\beta}$, we could zero-pad the m, n axes by a factor 2, and take a 2-d FFT with shape $(2M, 2N)$. Since the half-integer case will turn out to be fundamental to our algorithm, we introduce special notation for it. We define:

$$\widetilde{E}_{\tau pq} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} E_{\tau mn} \exp\left(\frac{2\pi i m p}{2M} + \frac{2\pi i n q}{2N} \right) \qquad \text{where } 0 \le p < 2M \text{ and } 0 \le q < 2N \tag{4}$$

The shape-$(N_{\rm time}, 2M, 2N)$ complex array $\widetilde{E}_{\tau pq}$ represents the beamformed electric field (2) at all half-integer beam locations $(\theta, \theta') = (p/2, q/2)$. Note that the RHS of Eq. (4) is a 2-d c2c FFT, zero-padded to shape $(2M, 2N)$. We define the half-integer beamformed intensity $I_{\bar\tau pq}$ by:

$$I_{\bar\tau pq} = \sum_{\tau=\bar\tau T_{ds}}^{\bar\tau(T_{ds}+1)-1} |\widetilde{E}_{\tau pq}|^2 \tag{5}$$

a real-valued array of shape $(N_{\rm time}/T_{ds}, 2M, 2N)$.

Main theorem. In §2.3, we will prove the following "main theorem". The half-integer beam locations are a complete basis, in the sense that the beamformed intensity $J_{\bar\tau\beta}$ at an arbitrary location $(\theta_\beta, \theta'_\beta)$ is a linear combination of intensities $I_{\bar\tau pq}$ at half-integer locations. More precisely, we will show that:

$$J_{\bar\tau\beta} = \sum_{p=0}^{2M-1} \sum_{q=0}^{2N-1} U_p^M(\theta_\beta)\, U_q^N(\theta'_\beta)\, I_{\bar\tau pq} \tag{6}$$

where we give two expressions for the function $U_q^N(\theta)$:

$$U_q^N(\theta) = \frac{1}{N} \sum_{s=0}^{N} A_s \cos\left(\frac{\pi(2\theta - q)s}{N} \right) \qquad \text{where } A_s = \begin{cases} 1 & \text{if } 0 < s < N \\ 1/2 & \text{if } s = 0 \text{ or } s = N \end{cases} \tag{7}$$

$$= \frac{1}{2N} \sin\Big(\pi(2\theta - q) \Big) \cot\left(\frac{\pi(2\theta - q)}{2N} \right) \tag{8}$$

The second expression (8) is more compact, but degenerates to $(0/0)$ if $(2\theta - q)$ is an integer multiple of $(2N)$. The first expression (7) is slower, but evaluates straightforwardly for all values of θ. This may be preferable in a vectorized setting (either a GPU kernel, or a simd CPU computation).

Algorithm. Our FRB beamforming algorithm is as follows:

1. For each beam $(\theta_\beta, \theta'_\beta)$, we precompute the weights $U_p^M(\theta_\beta)$, $U_q^N(\theta'_\beta)$ using Eq. (7) or (8).

2. Using FFT beamforming, we compute the intensity $I_{\bar{\tau}pq}$ at half-integer sky locations (Eq. 5).
3. We compute (or "resample") the beamformed intensity $J_{\bar{\tau}\beta}$ at specified sky locations $(\theta_\beta, \theta'_\beta)$ using the main theorem (6). This resampling step does not use spatial interpolation and is exact.

We considered a few different beamforming algorithms, and decided that this algorithm should fastest, on Ampere GPUs with CHORD parameters.

The intuition for why this algorithm is fastest is simple: it does the minimum possible computation (a shape-$(2M, 2N)$ FFT) to compute beamformed intensities in a complete basis. This creates complications (the kernel $U_p^M(\theta_\beta)$, $U_q^N(\theta'_\beta)$ is non-sparse, and requires trig evaluations), but all the complication happens at coarse time resolution, where computational cost turns out to be subdominant.

Factorizability. The computation of $J_{\bar{\tau}\beta}$ using the main theorem (6) is faster if the beam locations are factorizable. To see this, note that in the factorizable case, the beam index β can be reinterpreted as an index pair $\beta = (\gamma, \gamma')$. The first beam coordinate θ_β only depends on the first index γ, and the second beam coordinate θ'_β only depends on the second index γ'. Then we can factor Eq. (6) into two steps as follows:

$$K_{\bar{\tau}\gamma q} = \sum_{p=0}^{M-1} U_p^M(\theta_\gamma) I_{\bar{\tau}pq} \tag{9}$$

$$J_{\bar{\tau}\gamma\gamma'} = \sum_{q=0}^{N-1} U_q^N(\theta'_{\gamma'}) K_{\bar{\tau}\gamma q} \tag{10}$$

This two-step method factorizes the resampling computation into independent resampling steps along the two spatial axes. This only works if the beam pattern is factorizable.

Computational cost. Assuming $D = \mathcal{O}(MN)$ dishes, B beams, and time downsampling factor $T_{\rm ds}$, the computational cost per fast time sample is:

$$\text{Cost} = \left\{ \begin{array}{cl} \mathcal{O}\big(D\log D\big) + \mathcal{O}\big((B^{1/2}D + D^{1/2}B)/T_{\rm ds}\big) & \text{if beam locations are factorizable} \\ \mathcal{O}\big(D\log D\big) + \mathcal{O}\big(BD/T_{\rm ds}\big) & \text{if beam locations are arbitrary} \end{array} \right. \tag{11}$$

where we have split computational cost into (FFT + resampling) terms. In the GPU-centric sections of this note (§3–§5), we will estimate computational cost more precisely. See Table 3 in §4.13, and Table 4 in §5.4. For full CHORD, we estimate that the (FFT + resampling) cost is:

$$\text{Cost} = \left\{ \begin{array}{ll} (14+2)\% \text{ of total GPU resources} & \text{if beam locations are factorizable} \\ (14+6)\% \text{ of total GPU resources} & \text{if beam locations are arbitrary} \end{array} \right. \tag{12}$$

assuming a 128×A40 correlator. Thus, terms which are suppressed by $1/T_{\rm ds}$ (the resampling terms) are subdominant, as claimed previously in §1.

The FRB beamformer should still be fast if the spatial beam pattern is non-factorizable. This allows tracking beams to be implemented straightforwardly if desired. For tracking beams, we would adjust the beam positions $(\theta_\beta, \theta'_\beta)$ every ∼1 second or so (between kernel launches). This may require writing some small GPU kernels to recompute phase matrices – see §5.4 for discussion.

2.3 Proof of the main theorem

To reduce notational clutter, we state and prove a "1-d" version of the main theorem with one spatial dimension, one time index, downsampling factor $T_{\rm ds} = 1$, and one beam at sky location θ. The 2-d main theorem stated previously in Eq. ((6) follows by applying the 1-d version to both spatial axes, and summing over $0 \le \tau < T_{\rm ds}$.

Theorem. Let $\{E_n\}_{0\le n<N}$ be a complex 1-d array, and define:

$$J(\theta) = \big|\widetilde{E}(\theta)\big|^2 \qquad \text{where } \widetilde{E}(\theta) = \sum_{n=0}^{N-1} E_n \exp\left(\frac{2\pi i n\theta}{N}\right) \tag{13}$$

Then $J(\theta)$ is given by:

$$J(\theta) = \sum_{q=0}^{2N-1} U_q(\theta)\, I_q \qquad \text{where } I_q = \left| \sum_{n=0}^{N-1} E_n \exp\left(\frac{2\pi i q n}{2N} \right) \right|^2 \tag{14}$$

where $U_q(\theta)$ was defined previously in Eqs. (7), (8).

Proof. Expanding the definition (13), we write $J(\theta)$ as a double sum:

$$J(\theta) = \sum_{n=0}^{N-1} \sum_{n'=0}^{N-1} E_n E_{n'}^* F_{n-n'}(\theta) \qquad \text{where } F_s(\theta) = \exp\left(\frac{2\pi i s \theta}{N} \right) \tag{15}$$

Now a little trick. We want to expand $F_s(\theta)$ in a Fourier series in its integer index s. This requires $F_s(\theta)$ to be periodic in s, which isn't formally true given the definition (15). However, we note that $F_s(\theta)$ only appears in the double sum for $-N < s < N$. Therefore, we can replace $F_s(\theta)$ by the following quantity $G_s(\theta)$, which is equal to $F_s(\theta)$ for $-N < s < N$, and is $(2N)$-periodic in the integer index s (i.e. $G_s(\theta) = G_{s+2N}(\theta)$):[5]

$$G_s(\theta) = \left\{ \begin{array}{cl} \cos(2\pi\theta) & \text{if } s \equiv N \pmod{2N} \\ \exp(2\pi i u \theta / N) & \text{if } s \equiv u \pmod{2N} \text{ with } -N < u < N \end{array} \right. \tag{16}$$

Since $G_s(\theta)$ is $(2N)$-periodic in s, we can apply a length-$(2N)$ FFT. We denote the Fourier transform (in s) of $G_s(\theta)$ by $U_q(\theta)$, for consistency with previous notation. Here, $0 \le q < 2N$ is the conjugate wavenumber to $0 \le s < 2N$, and the functional argument θ does not participate in the FFT. The quantities $G_s(\theta)$ and $U_q(\theta)$ are related by:

$$G_s(\theta) = \sum_{q=0}^{2N-1} U_q(\theta) \exp\left(\frac{2\pi i s q}{2N} \right) \qquad \text{where } U_q(\theta) = \frac{1}{2N} \sum_{s=0}^{2N-1} G_s(\theta) \exp\left(-\frac{2\pi i s q}{2N} \right) \tag{17}$$

Now we simplify Eq. (15) for $J(\theta)$ as follows:

$$\begin{aligned} J(\theta) &= \sum_{n=0}^{N-1} \sum_{n'=0}^{N-1} E_n E_{n'}^* G_{n-n'}(\theta) && \text{replacing } F_s(\theta) \to G_s(\theta) \text{ in Eq. (15)} \\ &= \sum_{n=0}^{N-1} \sum_{n'=0}^{N-1} E_n E_{n'}^* \sum_{q=0}^{2N-1} U_q(\theta) \exp\left(\frac{2\pi i (n-n') q}{2N} \right) && \text{plugging in Eq. (17) for } G_s(\theta) \\ &= \sum_{q=0}^{2N-1} U_q(\theta)\, I_q && \text{recognizing definition (14) of } I_q \end{aligned}$$

This proves the main theorem, except for the task of showing that $U_q(\theta)$ is given by the expressions in Eqs. (7), (8). This follows from a short calculation, plugging the definition (16) of $G_s(\theta)$ into the definition (17) of $U_q(\theta)$.[6]

[5] The proof still works if we replace $\cos(2\pi\theta) \to 0$ in the first line of (16). Then, instead of the function $U_q(\theta)$ defined in Eqs. (7), (8), we would get the function $V_q(\theta)$ defined by

$$V_q(\theta) = \frac{1}{2N} \frac{\sin[\pi(2\theta - q)(2N-1)/(2N)]}{\sin[\pi(2\theta - q)/(2N)]}$$

I slightly prefer $U_q(\theta)$ to $V_q(\theta)$ since it has the following property. If $\theta = q'/2$ is a half-integer, then $U_q(\theta') = \delta_{qq'}$, as you would intuitively expect.

[6] The main theorem was also proved (by a different argument!) by Kiyo in his beamforming paper [1]. We have included a short self-contained proof for completeness. Note that Kiyo's paper uses $V_q(\theta)$ [see previous footnote] instead of $U_q(\theta)$.

3 Short FFTs with tensor cores

Note: Throughout this section, and the rest of these notes, we will make heavy use of the register assignment notation from Appendix B.1.

3.1 Introduction

In the FRB beamformer, we will need fast c2c float16 FFTs, zero-padded by a factor 2. The input X_n is a length-N `float16+16` array, and the output Y_q is a length-$(2N)$ `float16+16` array:

$$Y_q = \sum_{n=0}^{N-1} X_n \exp\left(\frac{2\pi i n q}{2N}\right) \qquad \text{where } 0 \le q < 2N \tag{18}$$

In this section, we will describe a fast tensor core FFT implementation. **We assume that N is known at compile time, is divisible by 4, and satisfies $8 \le N \le 32$.**

The input array X_n and output array Y_q will be held in registers in a single warp, in a register assignment described below (§3.2). The input array uses one register/thread, and the output array uses two registers/thread. Thus FFT could be implemented as a `__device__ inline` function with one `__half2` input, and two `__half2` outputs, which does not access global or shared memory. This function could be written and tested independently of the FRB beamformer.

3.2 Zero padding, spectator indices, and register assignments

The purpose of this section is to define register assignments for the input X-array and output Y-array. We'll do this in Eqs. (25) and (26) below, but first we need to explain some details: spectator indices and zero padding. First we define:

$$r = \lceil \log_2(N) \rceil \tag{19}$$

If $r \le 4$, then we will do 2^{5-r} FFTs in parallel, by distributing 2^{5-r} instances of the input array X_n across 32 threads in a warp. Formally, we introduce a spectator index $0 \le s < 2^{5-r}$, and "promote" the input and output arrays to 2-d arrays $X^{\rm 2d}_{ns}$, Y^{2d}_{qs}. The FFT is given by:

$$Y^{\rm 2d}_{qs} = \sum_{n=0}^{N-1} X^{\rm 2d}_{ns} \exp\left(\frac{2\pi i n q}{2N}\right) \qquad \text{where } 0 \le q < 2N \tag{20}$$

If N is not a power of two, then we will zero-pad the n and q indices to powers of two. It will be convenient to do the zero-padding in a slightly strange way as follows. First, we split the input index n into high/low indices (c, d), and split the output index q into high/low indices (u, v), by defining:

$$n = \frac{N}{4}c + d \qquad \text{where } 0 \le c < 4 \text{ and } 0 \le d < \frac{N}{4} \tag{21}$$

$$q = 8u + v \qquad \text{where } 0 \le u < \frac{N}{4} \text{ and } 0 \le v < 8 \tag{22}$$

Then we zero-pad by extending the d, u indices to $0 \le d < 2^{r-2}$ and $0 \le u < 2^{r-2}$. Formally, we "promote" the input and output arrays to 3-d zero-padded arrays:

$$X^{\rm 3d}_{cds} = \begin{cases} X^{\rm 2d}_{(N/4)c+d,s} & \text{if } 0 \le d < (N/4) \\ 0 & \text{if } (N/4) \le d < 2^{r-2} \end{cases} \qquad \text{where} \quad \begin{array}{l} 0 \le c < 4 \\ 0 \le d < 2^{r-2} \\ 0 \le s < 2^{5-r} \end{array} \tag{23}$$

$$Y^{\rm 3d}_{uvs} = \begin{cases} Y^{\rm 2d}_{8u+v,s} & \text{if } 0 \le u < (N/4) \\ 0 & \text{if } (N/4) \le u < 2^{r-2} \end{cases} \qquad \text{where} \quad \begin{array}{l} 0 \le u < 2^{r-2} \\ 0 \le v < 8 \\ 0 \le s < 2^{5-r} \end{array} \tag{24}$$

Our FFT implementation will operate on the zero-padded input array X^{3d}_{cds}, and return the zero-padded output array Y^{3d}_{uvs}. These arrays have total sizes 32 and 64 respectively, and can be distributed across a single warp with 1 and 2 registers/thread respectively. We will use the following register assignments:

$$[\text{float16 } X^{\rm 3d}_{cds}] \qquad b \leftrightarrow \text{ReIm} \qquad\qquad t_0t_1t_2t_3t_4 \leftrightarrow c_0c_1d_0\ \underbrace{s_0\cdots s_{4-r}\ d_1\cdots d_{r-3}}_{\text{2 bits}} \tag{25}$$

$$[\text{float16 } Y^{\rm 3d}_{dvs}] \qquad b \leftrightarrow v_0 \qquad r \leftrightarrow \text{ReIm} \qquad t_0t_1t_2t_3t_4 \leftrightarrow v_1v_2\ \underbrace{u_0\cdots u_{r-3}\ s_0\cdots s_{4-r}}_{\text{3 bits}} \tag{26}$$

These equations can also be viewed as defining register mappings for the 2-d arrays $X^{\rm 2d}_{ns}$, $Y^{\rm 2d}_{qs}$, in which some GPU threads store zeros.

3.3 FFT algorithm

The general idea of the FFT algorithm is to factorize a length-(MN) FFT as the composition of two FFTs with lengths M and N. Usually, this idea is applied recursively to factorize a length-2^n FFT as the composition of n length-2 FFTs. Here, we will factorize a length-$(2N)$ FFT as the composition of two FFTs with lengths 8 and $(N/4)$. These FFTs are small enough that they can be done with a single tensor core MMA instruction.

The "3-d" variables X^{3d}_{cds} and Y^{3d}_{uvs} have been defined in a way which makes this factorization explicit. To derive an expression for the FFT in these variables, we start with Eq. (20) and write $n = (N/4)c + d$ and $m = 8u + v$ (as in Eqs. (21), (22)). After a little algebra we get:

$$Y^{3d}_{uvs} = \sum_{d=0}^{(N/4)-1} \exp\left(\frac{8\pi i du}{N}\right) \exp\left(\frac{\pi i dv}{N}\right) \sum_{c=0}^{3} \exp\left(\frac{\pi i cv}{4}\right) X^{3d}_{cds} \tag{27}$$

By evaluating the RHS from right to left, we get an algorithm for computing $Y^{\rm 3d}_{uvs}$ from $X^{\rm 3d}_{cds}$:

$$Z_{dvs} = \sum_{c=0}^{3} \Gamma^{(1)}_{cv} X^{\rm 3d}_{cds} \qquad \text{where } \Gamma^{(1)}_{cv} = \exp\left(\frac{\pi i cv}{4}\right) \tag{28}$$

$$W_{dvs} = \Gamma^{(2)}_{dv} Z_{dvs} \quad \text{(no sum on } d \text{ or } v) \qquad \text{where } \Gamma^{(2)}_{dv} = \begin{cases} \exp\left(\frac{\pi i dv}{N}\right) & \text{if } d < (N/4) \\ 0 & \text{if } d \ge (N/4) \end{cases} \tag{29}$$

$$Y^{\rm 3d}_{uvs} = \sum_{d=0}^{2^{r-2}-1} \Gamma^{(3)}_{du} W_{dvs} \qquad \text{where } \Gamma^{(3)}_{du} = \begin{cases} \exp\left(\frac{8\pi i du}{N}\right) & \text{if } d < (N/4) \text{ and } u < (N/4) \\ 0 & \text{if } d \ge (N/4) \text{ or } u \ge (N/4) \end{cases} \tag{30}$$

One small subtlety here: when going from Eq. (27) to Eqs. (28)–(30), we have padded the indices d, u to run over $0, \cdots, (2^{r-2} - 1)$, instead of $0, \cdots, (N/4)$. The definitions of $\Gamma^{(2)}_{dv}$ and $\Gamma^{(3)}_{du}$ in Eqs. (29), (30) have been appropriately zero-padded for consistency.

3.4 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:

$$\begin{pmatrix} \text{Re}(C) \\ \text{Im}(C) \end{pmatrix} = \begin{pmatrix} \text{Re}(A) & -\text{Im}(A) \\ \text{Im}(A) & \text{Re}(A) \end{pmatrix} \begin{pmatrix} \text{Re}(B) \\ \text{Im}(B) \end{pmatrix} \tag{31}$$

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 ReIm_{in} and ReIm_{out} (corresponding respectively to column and row indices in (31)).

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

$$\begin{aligned}
A_r[\mathrm{Re}_{\mathrm{out}}, \mathrm{Re}_{\mathrm{in}}, \cdots] &= \mathrm{Re}\, A_c[\cdots] \\
A_r[\mathrm{Re}_{\mathrm{out}}, \mathrm{Im}_{\mathrm{in}}, \cdots] &= -\mathrm{Im}\, A_c[\cdots] \\
A_r[\mathrm{Im}_{\mathrm{out}}, \mathrm{Re}_{\mathrm{in}}, \cdots] &= \mathrm{Im}\, A_c[\cdots] \\
A_r[\mathrm{Im}_{\mathrm{out}}, \mathrm{Im}_{\mathrm{in}}, \cdots] &= \mathrm{Re}\, A_c[\cdots]
\end{aligned} \tag{32}$$

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. (39) below.

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

$$\underbrace{C}_{8\times 8} = \underbrace{A}_{8\times 8} \underbrace{B}_{8\times 8} \tag{33}$$

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:

$$\underbrace{\begin{pmatrix} C \\ C' \end{pmatrix}}_{16\times 8} = \underbrace{\begin{pmatrix} A & \\ & A \end{pmatrix}}_{16\times 16} \underbrace{\begin{pmatrix} B \\ B' \end{pmatrix}}_{16\times 8} \tag{34}$$

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 $s_{\mathrm{in}}, s_{\mathrm{out}}$, and depends on these indices only through an overall Kronecker delta $\delta_{s_{\mathrm{in}} s_{\mathrm{out}}}$.

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:

$$\underbrace{C}_{4\times 8} = \underbrace{A}_{4\times 4} \underbrace{B}_{4\times 8} \tag{35}$$

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

$$\underbrace{\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}}_{16\times 8} = \underbrace{\begin{pmatrix} A & & & \\ & A & & \\ & & A & \\ & & & A \end{pmatrix}}_{16\times 16} \underbrace{\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix}}_{16\times 8} \tag{36}$$

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 $s_0^{\mathrm{in}}, s_1^{\mathrm{in}}, s_0^{\mathrm{out}}, s_1^{\mathrm{out}}$, and would depend on these indices through two Kronecker deltas $\delta_{s_0^{\mathrm{in}} s_0^{\mathrm{out}}}\, \delta_{s_1^{\mathrm{in}} s_1^{\mathrm{out}}}$.

We mention in advance that this use of spectator bits leads to sparse matrices. The A-matrices in Eqs. (34), (36) have 50% and 25% sparsity respectively. This will allow us to use sparse tensor core MMAs (Appendix C.3) for speed.

3.5 Implementation

In this section, we'll describe implementation of the FFT algorithm in Eqs. (28)–(30). Conceptually, this is straightforward: we compute (28) using a tensor core MMA, then compute (29) using `__half2` FMAs, then compute (30) using a tensor core MMA. However, the index gymnastics are surprisingly complicated!

First step: computing Z using Eq. (28). We start with X^{3d}_{cds} in register assignment from Eq. (25):

$$[\text{float16 } X^{\mathrm{3d}}_{cds}] \qquad b \leftrightarrow \mathrm{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow c_0 c_1 d_0\ \underbrace{s_0 \cdots s_{4-r}\ d_1 \cdots d_{r-3}}_{2 \text{ bits}} \tag{37}$$

To compute $Z = \Gamma^{(1)} X$, we apply a `m16n8k8` MMA (Appendix C.1). In notation from that appendix, matrices (A_{ij}, B_{jk}) correspond to $(\Gamma^{(1)}, X^{\rm 3d})$, and tensor core indices i, j, k correspond to:

$$i_0 i_1 i_2 i_3 \leftrightarrow v_0 v_1 v_2\,\mathrm{ReIm}_{\rm out} \qquad j_0 j_1 j_2 \leftrightarrow \mathrm{ReIm}_{\rm in}\, c_0 c_1 \qquad k_0 k_1 k_2 \leftrightarrow d_0\, \underbrace{s_0 \cdots s_{4-r}\ d_1 \cdots d_{r-3}}_{\text{2 bits}} \tag{38}$$

Under this index correspondence, one can check that the $X^{\rm 3d}_{cds}$ register assignment (37) matches the tensor core B-array register assignment (157). This is the reason for our choice of X-array register assignment, given previously in Eq. (25) without motivation.

Similarly, we deduce the necessary register assignment for $\Gamma^{(1)}_{cv}$ from Eq. (38) and the tensor core A-array register assignment (156):

$$[\text{float16 } \Gamma^{(1)}_{cv}] \qquad b \leftrightarrow \mathrm{ReIm}_{\rm in} \qquad r \leftrightarrow \mathrm{ReIm}_{\rm out} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow c_0 c_1 v_0 v_1 v_2 \tag{39}$$

We assume that $\Gamma^{(1)}$ has been precomputed at the beginning of the kernel, and loaded into persistent registers using this register assignment. Finally, we deduce the register assignment for the output array Z_{dvs} from Eq. (38) and the tensor core C-array register assignment (158):

$$[\text{float16 } Z_{dvs}] \qquad b_0 \leftrightarrow d_0 \qquad r \leftrightarrow \mathrm{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow \underbrace{s_0 \cdots s_{4-r}\ d_1 \cdots d_{r-3}}_{\text{2 bits}}\, v_0 v_1 v_2 \tag{40}$$

Second step: computing W using Eq. (29). We assume that $\Gamma^{(2)}_{dv}$ (defined in (29)) has been precomputed and stored in persistent registers, with the same register assignment as Z_{dvs} (Eq. (40)):

$$[\text{float16 } \Gamma^{(2)}_{dv}] \qquad b_0 \leftrightarrow d_0 \qquad r \leftrightarrow \mathrm{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow \underbrace{s_0 \cdots s_{4-r}\ d_1 \cdots d_{r-3}}_{\text{2 bits}}\ v_0 v_1 v_2 \tag{41}$$

A notational point here: since $\Gamma^{(2)}_{dv}$ does not depend on the spectator index s, the spectator bit assignment $t_0 \cdots t_{4-r} \leftrightarrow s_0 \cdots s_{4-r}$ on the RHS of (41) indicates that the data is independent of thread indices $t_0 \cdots t_{4-r}$.

To compute W_{dvs} using Eq. (29), we multiply the complex arrays $\Gamma^{(2)}_{dv}$ and Z_{dvs} elementwise. For both of these arrays, the ReIm index bit is mapped to a register ($r \leftrightarrow \mathrm{ReIm}$). Therefore, the complex multiplication can be done straightforwardly using four `__half2` FMAs. The output is the W-array, in the same register assignment as the Z-array (40):

$$[\text{float16 } W_{dvs}] \qquad b_0 \leftrightarrow d_0 \qquad r \leftrightarrow \mathrm{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow \underbrace{s_0 \cdots s_{4-r}\ d_1 \cdots d_{r-3}}_{\text{2 bits}}\, v_0 v_1 v_2 \tag{42}$$

Third step: computing Y using Eq. (30). Finally, to compute $Y = \Gamma^{(3)} W$, we apply a `m16n8k16` MMA. A complication here: the MMA is sparse (Appendix C.3) if $r \le 4$, and dense (Appendix C.2) if $r = 5$. In both cases, the MMA matrices (A, B) correspond to $(\Gamma^{(3)}, W)$, and the correspondence between MMA index bits (i, j, k) and FFT index bits is:

$$\begin{aligned} i_0 i_1 i_2 i_3 &\leftrightarrow \underbrace{u_0 \cdots u_{r-3}\ s^{\rm out}_0 \cdots s^{\rm out}_{4-r}}_{\text{3 bits}}\ \mathrm{ReIm}_{\rm out} \\ j_0 j_1 j_2 j_3 &\leftrightarrow d_0\ \underbrace{s^{\rm in}_0 \cdots s^{\rm in}_{4-r}\ d_1 \cdots d_{r-3}}_{\text{2 bits}}\ \mathrm{ReIm}_{\rm in} \\ k_0 k_1 k_2 &\leftrightarrow v_0 v_1 v_2 \end{aligned} \tag{43}$$

In both dense and sparse cases, one can check that the W-array register assignment (42) matches the tensor core B-array register assignment. Similarly, we deduce the register assignment for the output array Y^{3d}_{dvs} from the tensor core C-array register assignment. In both dense and sparse cases, the result is:

$$[\text{float16 } Y^{\rm 3d}_{dvs}] \qquad b_0 \leftrightarrow v_0 \qquad r \leftrightarrow \mathrm{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow v_1 v_2\ \underbrace{u_0 \cdots u_{r-3}\ s^{\rm out}_0 \cdots s^{\rm out}_{4-r}}_{\text{3 bits}} \tag{44}$$

This register assignment for the Y-array was given previously in Eq. (26) without derivation.

The only remaining detail is the register assignment for $\Gamma^{(3)}_{du}$. Here, we distinguish 3 cases, depending on the value of $r = \lceil \log_2(N) \rceil$.

- **Case 1: r=5.** In this case, there are no spectator bits in the FFT, and the MMA is dense (Appendix C.2). We deduce the register assignment for $\Gamma^{(3)}_{du}$ from the index correspondence (43) and the tensor core A-matrix register mapping (159):

$$[\text{float16 } \Gamma^{(3)}_{du}] \qquad b_0 \leftrightarrow d_0 \qquad r_0 r_1 \leftrightarrow \text{ReIm}_{\text{out}}, \text{ReIm}_{\text{in}} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_1 d_2 u_0 u_1 u_2 \tag{45}$$

- **Case 2: r=4.** There is one spectator bit s_0 in the FFT. As explained near Eq. (34) above, the $\Gamma^{(3)}_{du}$ array gets two extra indices s_0^{in}, s_0^{out}, with a Kronecker delta $\delta_{s_0^{\text{in}} s_0^{\text{out}}}$. Under the index correspondence (43), this becomes a Kronecker delta $\delta_{i_2 j_1}$ in the MMA matrix A_{ij}.

  In Appendix C.3, we show how to use a sparse `m16n8k16` MMA for speed, in a case where A_{ij} contains a delta function of the form $\delta_{i_n j_1}$. Here, we summarize the conclusions by writing down the (A^{sp}, E, f) operands to the sparse MMA. The E operand has the "universal" value given in Eqs. (171)–(173), and the f operand is given by $f = 2$.

  To derive the A^{sp} operand, we combine the general register mapping (170) with the index correspondence (43). We find that the A^{sp} operand is $\Gamma^{(3)}_{du}$ with register mapping (two registers/thread):

$$[\text{float16 } \Gamma^{(3)}_{du}] \qquad b_0 \leftrightarrow d_0 \qquad r_0 \leftrightarrow \text{ReIm}_{\text{out}} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_1, \text{ReIm}_{\text{in}}, u_0, u_1, s_0 \tag{46}$$

  where in this context, the spectator bit assignment $t_4 \leftrightarrow s_0$ on the RHS of (46) means that the register contents are independent of thread index t_4.

- **Case 3: r=3.** There are two spectator bits s_0, s_1 in the FFT. As explained near Eq. (36) above, the $\Gamma^{(3)}_{du}$ array gets four extra indices $(s_0^{\text{in}}, s_0^{\text{out}}, s_1^{\text{in}}, s_1^{\text{out}})$, with Kronecker deltas $\delta_{s_0^{\text{in}} s_0^{\text{out}}} \delta_{s_1^{\text{in}} s_1^{\text{out}}}$. Under the index correspondence (43), these become Kronecker deltas $\delta_{i_1 j_1} \delta_{i_2 j_2}$ in the MMA matrix A_{ij}.

  In Appendix C.3, we show how to use a sparse `m16n8k16` MMA for speed, in a case where A_{ij} contains a delta function of the form $\delta_{i_n j_1}$. Here, we summarize the conclusions by writing down the (A^{sp}, E, f) operands to the sparse MMA. The E operand has the "universal" value given in Eqs. (171)–(173), and the f operand is given by $f = 1$.

  To derive the A^{sp} operand, we combine the general register mapping (170) with the index correspondence (43). We find that the A^{sp} operand is $(\Gamma^{(3)}_{du} \delta_{s_1^{\text{in}} s_1^{\text{out}}})$ with register mapping (two registers/thread):

$$[\text{float16 } \Gamma^{(3)}_{du} \delta_{s_1^{\text{in}} s_1^{\text{out}}}] \qquad b_0 \leftrightarrow d_0 \qquad r_0 \leftrightarrow \text{ReIm}_{\text{out}} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow s_1^{\text{in}}, \text{ReIm}_{\text{in}}, u_0, s_0, s_1^{\text{out}} \tag{47}$$

  where in this context, the spectator bit assignment $t_3 \leftrightarrow s_0$ on the RHS of (46) means that the register contents are independent of thread index t_3. Note that the Kronecker delta $\delta_{s_1^{\text{in}} s_1^{\text{out}}}$ is still included.

We assume that $\Gamma^{(3)}_{du}$ (defined in (30)) has been precomputed and stored in registers, in the appropriate register assignment (either Eq. (45), (46), or (47), depending on the value of r).

3.6 Register usage

The constant matrices $\Gamma^{(i)}$ are loaded into registers at the beginning of the kernel, and held persistently throughout. I think it will be easiest to compute $\Gamma^{(i)}$ on the CPU, and pass the arrays to the kernel using either GPU global memory or constant memory. In this section we compute the number of registers needed to store the $\Gamma^{(i)}$.

One more detail: in the FRB beamformer, we will need to store $\Gamma^{(i)}$ matrices for two FFT sizes (denoted M, N), and it may be possible to share $\Gamma^{(i)}$ registers between the two FFTs. Then the register counting works as follows:

- By Eq. (39), we need 2 registers for $\Gamma^{(1)}$. Since the definition (28) of $\Gamma^{(1)}$ does not depend on N, these registers are shared between the two FFTs.
- By Eq. (41), we need 2 registers for $\Gamma^{(2)}$, and these registers cannot be shared between the two FFTs (unless $M = N$).
- The $\Gamma^{(3)}$ registers cannot be shared between the two FFTs (unless $M = N$). Each $\Gamma^{(3)}$ matrix needs either 2 or 3 registers, depending on whether the MMA is sparse or dense. (In the dense case, Eq. (45) may suggest that 4 registers are needed, but note that the $(\mathrm{Re}_{\mathrm{in}}, \mathrm{Re}_{\mathrm{out}})$ register stores the same value as the $(\mathrm{Im}_{\mathrm{in}}, \mathrm{Im}_{\mathrm{out}})$ register, by Eq. (32).) The first FFT uses a sparse MMA if $M \le 16$, and likewise for the second FFT with $M \to N$.
- If either $M \le 16$ or $N \le 16$, then we need one more register for E.

For future reference in the FRB beamformer (§4), it will be useful to count registers for the following specific choices of (M, N):

$$\text{Number of registers } R_{\mathrm{FFT}} = \begin{cases} 7 & \text{if } (M,N) = (8,8) \quad [\ 2\Gamma^{(1)} + 2\Gamma^{(2)} + 2\Gamma^{(3)} + E\ ] \\ 11 & \text{if } (M,N) = (8,12) \quad [\ 2\Gamma^{(1)} + 4\Gamma^{(2)} + 4\Gamma^{(3)} + E\ ] \\ 7 & \text{if } (M,N) = (16,16) \quad [\ 2\Gamma^{(1)} + 2\Gamma^{(2)} + 2\Gamma^{(3)} + E\ ] \\ 12 & \text{if } (M,N) = (16,20) \quad [\ 2\Gamma^{(1)} + 4\Gamma^{(2)} + 5\Gamma^{(3)} + E\ ] \\ 7 & \text{if } (M,N) = (24,24) \quad [\ 2\Gamma^{(1)} + 2\Gamma^{(2)} + 3\Gamma^{(3)}\ ] \end{cases} \tag{48}$$

where in square brackets, we have "shown our work" by counting the number of registers of each type ($\Gamma^{(1)}$, $\Gamma^{(2)}$, $\Gamma^{(3)}$, E).

3.7 Computational cost

All cycle-counting assumes GPU architecture 8.6 (RTX3090, A40, A10), not architecture 8.0 (A100, A30). The computational cost of the FFT implementation in §3.5 can be computed as follows.

- The first step does an `m16n8k8` MMA (2 SM-cycles).
- The second step does four `__half2` FMAs ($4 \times 0.5 = 2$ SM-cycles).
- The third step does either a dense `m16n8k16` MMA (4 SM-cycles) if $N > 16$, or a sparse `m16n8k16` MMA (2 SM-cycles) if $N \le 16$.

Thus, the total SM-cycle count is 8 if $N > 16$, or 6 if $N \le 16$. This sequence of operations performs 2^{5-r} FFTs in parallel, where $r = \lceil \log_2(N) \rceil$, due to spectator indices. The number of SM-cycles per FFT is shown in the first row of Table 1. In the second row of Table 1, we convert this to FFTs/second, assuming the nominal A40 value of $84 \times (1.7 \times 10^9)$ SM-cycles/second.

To help interpret this number, in the bottom row of Table 1, we convert to an inferred Tflop count, assuming the following optimistic counting of flops per FFT (derivation omitted):

$$\text{Flops/FFT} = \begin{cases} m 2^{m+4} & \text{if } N = 2^m \\ 2^{m+3} p (2m + p) & \text{if } N = 2^m p, \text{where } p \text{ is a small odd prime} \end{cases} \tag{49}$$

FFT length N	8	12	16	20	24	28	32
SM-cycles/FFT	1.5	3	3	8	8	8	8
GigaFFTs/sec	97	49	49	18	18	18	18
Inferred Tflops	37	33	50	26	32	45	47

Table 1: Cycle-counting analysis for short FFTs with tensor cores, assuming an A40 GPU. All FFTs are c2c, and zero-padded by a factor 2, so that the input and output arrays have lengths N and $2N$ respectively.

Table 1 can be summarized by the statement that FFT performance is comparable to the nominal *scalar* float16 compute capacity of an A40 (37 Tflops), but a few times slower than the nominal *tensor* float16 compute capactity (150 Tflops). This level of performance is much faster than `cufft`, and a few times faster than `SFFT`[7], which have additional overhead from loads/stores or shuffling register assignments. Our implementation avoids this overhead by using tensor core MMAs, which shuffle register assignments as a side effect. Of course, all "results" in this section are really performance predictions, since we don't have an implementation yet!

[7]`https://github.com/KAdamek/SMFFT`

4 First FRB beamforming kernel

The first FRB beamforming kernel computes beamformed intensities $I_{f\bar{\tau}pq}$ on a regular grid of half-integer beam locations, from the electric field $E_{\tau f\pi d}$. The indices are as follows:

- τ indexes a "fast" time sample, and $\bar{\tau}$ indexes a "slow" time sample, after downsampling by a factor $T_{\rm ds}$ (a kernel parameter; see §4.1).
- p, q index half-integer beam locations.
- f indexes a frequency channel.
- π indexes polarization (not p, to avoid confusion with the index p in the second bullet point).
- d indexes a dish.

The output array $I_{f\bar{\tau}pq}$ is written to global memory. Subsequently, the *second* FRB beamforming kernel (§5) will read this array (representing intensities on a regular grid), and compute intensities at a specified set of sky locations.

In this section, we will not specify a global memory layout for the output I-array. The optimal memory layout will be dictated by the second FRB beamforming kernel, and will depend on `cublas` experiments which are hard to predict in advance (see §5). The first FRB beamforming kernel in this section should be able to accommodate a variety of I-array memory layouts, without affecting performance significantly.

4.1 Specification

Inputs:

- Number of dishes D, and dish grid size parameters (M, N). The dishes are assumed to be a subset of an M-by-N regular grid. Throughout §4, we will consider the following cases:
  - **Case 1.** $(D, M, N) = (64, 8, 8)$. [CHORD pathfinder, square]
  - **Case 2.** $(D, M, N) = (64, 8, 12)$. [CHORD pathfinder, rectangular]
  - **Case 3.** $(D, M, N) = (256, 16, 16)$. [HIRAX, square]
  - **Case 4.** $(D, M, N) = (256, 16, 20)$. [HIRAX, rectangular]
  - **Case 5.** $(D, M, N) = (512, 24, 24)$. [Full CHORD]

  The CHORD pathfinder will probably be rectangular (case 2), but a square pathfinder (case 1) is still a possibility. For HIRAX, both square (case 3) and rectangular (case 4) options are being considered.

  The parameters (D, M, N) are **compile-time parameters**, i.e. we'll compile a separate kernel (using C++ templates or a Julia code generator) for each choice of (D, M, N).

- Time downsampling factor $T_{\rm ds}$, not assumed to be a power of 2 (e.g. $T_{\rm ds}$ could be odd). In Table 3, we give realistic values of $T_{\rm ds}$ for each of cases 1–5. We also give values for the number of frequency channels F, and the input sampling rate t_s.
- The 4+4 bit electric field $E_{\tau f\pi d}$ in global memory, with memory layout suggested by the index ordering.
- The dish gridding $d \to (m, n)$, defining 2-d integer grid coordinates (m, n) for each dish d. This information would be specified in one or more integer-valued arrays passed as kernel arguments. Feel free to choose whatever array arguments and memory layouts are convenient (see discussion in §4.8).

  Early in the kernel (§4.8), we will re-index (or "grid") the electric field array as $d \to (m, n)$. We will denote the gridding operation implicitly, via choice of indices: $E_{\tau f\pi d}$ denotes the ungridded electric field, and $E_{\tau f\pi mn}$ denotes the gridded electric field.

- Complex `float16+16` weights $W_{f\pi mn}$ applied to the gridded electric field before beamforming. These will be computed externally using real-time calibration and noise estimation logic, and passed to the FRB beamforming kernel (just like the baseband beamforming kernel.) Feel free to choose whatever memory layout is convenient (see discussion in §4.10).

Outputs:

- The beamformed intensity array $I_{f\bar{\tau}pq}$ on a regular complete grid

$$I_{f\bar{\tau}pq} = \sum_{\tau=T_{\rm ds}\bar{\tau}}^{T_{\rm ds}(\bar{\tau}+1)-1} \sum_{\pi=1}^{2} \big|\widetilde{E}_{\tau f\pi pq}\big|^2 \qquad 0 \le p < 2M \text{ and } 0 \le q < 2N \tag{50}$$

  where $\widetilde{E}_{\tau f\pi pq}$ is the FFT of the gridded electric field $E_{\tau f\pi mn}$ (more precisely, a two-dimensional c2c FFT along spatial axes $mn \rightarrow pq$, zero-padded by a factor 2):

$$\widetilde{E}_{\tau f\pi pq} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{f\pi mn}\, E_{\tau f\pi mn}\, \exp\left(\frac{2\pi imp}{2M} + \frac{2\pi inq}{2N}\right) \tag{51}$$

  Note the weight factor $W_{f\pi mn}$ in this equation.

  The intensity array $I_{f\bar{\tau}pq}$ is written to global memory, but the intermediate quantity $\widetilde{E}_{\tau f\pi pq}$ is not. As discussed above, this document doesn't define a global memory layout for the I-array. In particular, the index ordering $I_{f\bar{\tau}pq}$ is arbitrary and is not intended to suggest a preferred memory layout!

4.2 Outline

Warps and threadblocks. We use one threadblock per frequency channel. In the rest of §4, we will restrict attention to a single threadblock, and leave the frequency index f implicit.

The number of warps per threadblock W is a compile-time constant, which depends on (D, M, N) as shown in Table 2. The number of threadblocks per SM will need some experimentation (see §4.14).

Bank-confict free gridding. In a previous version of the FRB beamformer, the gridding operation $E_d \rightarrow E_{mn}$ had shared memory bank conflicts, whose severity depended on the details of the mapping $d \rightarrow (m, n)$ from dishes to grid locations. This was unfortunate, since it introduced a complicated dependency between several CHORD subsystems (cabling between dishes and FPGA inputs, cabling between FPGAs and GPU nodes, etc.)

In this version of the FRB beamformer, we use an alternative gridding algorithm which is always bank conflict free, but requires an extra shared memory read-write cycle. I decided this "two-pass" gridding algorithm was preferable, since the overhead of the extra read-write cycle is pretty small (a few percent of the total FRB beamformer cost; see Table 3), and it avoids the logistical challenge of tracking a complicated dependency between subsystems.

Outer and inner blocks. The outermost loop in the FRB beamformer reads the E-array in "outer blocks" of $T_{\rm outer}$ time samples. The value of $T_{\rm outer}$ is a compile-time constant, which depends on (D, M, N) as shown in Table 2.

Within the outermost loop, we do an inner loop over "inner blocks" of $T_{\rm inner}$ time samples. The value of $T_{\rm inner}$ is also a compile-time constant, shown in Table 2. The value of $T_{\rm inner}$ is always a divisor of $T_{\rm outer}$. (See pseudocode below in §4.3.)

We use this nested-loop structure because our two-pass gridding algorithm requires a large block size $T_{\rm outer}$ to be efficient (see below), whereas the 2-d FFT requires a smaller block size $T_{\rm inner}$ to avoid overflowing shared memory.

E and F arrays. One outer block of electric field data is an 8-bit `int4+4` array:

```
int4+4 E[Touter][2][D];    // (time, pol, dish)                (52)
```

Early in the kernel (§4.6), we will repackage this as an `int32` array, with 4 times fewer elements:

```
int32 F[Touter/2][D];    // (time, dish)                                  (53)
```

We now explain how the F-array is defined. We group elements of the E-array (52) into quadruples, by either varying the polarization index π, or by varying the time index τ by $(\pm T_{\rm outer}/2)$. Restricting attention to one such quadruple, we can view the quadruple as a shape-$(2,2,2)$ array

```
int4 E[2][2][2];    // (tau_hi, pol, ReIm)                                (54)
```

where we have introduced the index bit $\tau_{\rm hi}$ to represent changing the time index τ by $(T_{\rm outer}/2)$ within the E-array outer block (52). We pack this shape-$(2,2,2)$ array into an `int32` register `F`, using the following shuffled bit assignment:

$$[\text{int32 } F] \qquad b_0 b_1 b_2 \leftrightarrow \pi, \tau_{\rm hi}, \text{ReIm} \tag{55}$$

If we do this for all quadruples in the E-array outer block (52), then we get the F-array outer block (53). Note that the outer block time index runs from $0 \le \tau < T_{\rm outer}$ in the E-array representation (52), and from $0 \le \tau < (T_{\rm outer}/2)$ in the F-array representation (53).

More on the two-pass gridding algorithm. Here is an outline of our two-pass, bank conflict free, gridding algorithm. For each outer block ($T_{\rm outer}$ time samples):

- We read the E-array from global memory, repackage it as the F-array $F_{\tau d}$ from Eq. (53), and write it to a shared memory array `Fsh1[tau][d]`. (§4.6)
- We read the $F_{\tau d}$ array from shared memory into a register array `Freg1[tau,d]`, in an alternate register assignment where the thread index t corresponds to the time τ. For this to work effieicntly, $T_{\rm outer}/2$ (the number of time indices in the F-array) should either be 32, or a little less than 32. If $T_{\rm outer}/2$ is less than 32, then the `Freg1[]` array will be zero-padded, by having some thread indices store zeros in their registers. (§4.7)
- We write the register array `Freg1[tau,d]` back to shared memory, in gridded form `Fsh2[tau,m,n]`. Gridding happens in this step: the source array `Freg1[tau,d]` is indexed by a dish d, and the destination array `Fsh2[tau,m,n]` is indexed by a grid location (m,n). There are no bank conflicts because the thread index t corresponds to time τ, and the time index τ is a "spectator" in the gridding operation $d \to (m,n)$. (§4.8)
- Finally, we read the `Fsh2[tau,m,n]` shared memory array into a register array `Freg2[tau,m,n]`, in yet another register assignment which is matched to the tensor core FFT from §3. (§4.9)

At the end of this two-pass gridding algorithm, the electric field data ends up in a register array `Freg2[tau,m,n]`, not in shared memory. This will let us re-use shared memory for the 2-d FFT, which we now describe.

2-d FFT. We will factorize the 2-d FFT as two 1-d FFTs $E_{mn} \to G_{mq} \to \widetilde{E}_{pq}$:

$$G_{\tau\pi mq} = \sum_{n=0}^{N-1} W_{\pi mn} E_{\tau\pi mn} \, \exp\left(\frac{2\pi i n q}{2N}\right) \tag{56}$$

$$\widetilde{E}_{\tau\pi pq} = \sum_{m=0}^{M-1} G_{\tau\pi mq} \exp\left(\frac{2\pi i m p}{2M}\right) \tag{57}$$

The 1-d FFTs are done using our tensor core FFT algorithm from §3. After the first FFT (56), we write the G-array to a shared memory array `Gsh[]`, and read it back "transposed" for the second FFT (57).

After the second FFT, we accumulate the contribution to the intensity array:

$$I_{\bar{\tau}pq} = \sum_{\tau\pi} |\widetilde{E}_{\tau\pi pq}|^2 \tag{58}$$

The "running" I-array is kept in registers throughout the kernel. Periodically (every $T_{\rm ds}$ time samples), we write the I-array to global memory, and zero the running I-array. (See pseudocode in §4.3.)

4.3 Pseudocode

```
// Accumulator for intensity array I_{pq}.
// Number of registers/thread RI is a compile-time constant, see sec 4.4.
__half2 I_running[RI] = 0;
int t_running = 0;

for (int t_outer = 0; t_outer < NTIME; t_outer += T_outer) {

    // Two-pass gridding algorithm starts here. Read E-array from global memory,
    // repackage as F-array, write to shared memory array Fsh1[tau,d].
    copy_global_memory_to_Fsh1();  // sec 4.6
    __syncthreads();

    // Read Fsh1[] into register array Freg1[], with (thread index t) <-> (time tau).
    // Number of registers/thread RF1 is a compile-time constant, see sec 4.4.
    int Freg1[RF1] = read_Fsh1();     // sec 4.7
    __syncthreads();

    // Do gridding, by writing Freg1[tau,d] to shared memory array Fsh2[tau,m,n].
    write_Fsh2(Freg1);                // sec 4.8
    __syncthreads();

    // Read Fsh2[] array to register array Freg2[], in an FFT-friendly layout.
    int Freg2[RF2] = read_F2();     // sec 4.9 (see sec 4.4 for values of RF2)
    __syncthreads();

    // This loop is unrolled, since t_inner will index the Freg2[] register array.
    #pragma unroll
    for (int t_inner = 0; t_inner < T_outer; t_inner += T_inner) {
        // Do first 1-d FFT for one inner block. The input data is in registers
        // (Freg2[]), and the output G-array (sec 4.2) is written to shared memory.

        do_first_fft(Freg2, t_inner);  // sec 4.10
        write_G_to_shared_memory();    // sec 4.10
        __syncthreads();

        for (int t = 0; t < T_inner; t++) {
            // do_second_fft() reads the G-array from shared memory for a single
            // time sample and polarization, does the FFT to compute Etilde, and
            // accumulates |E^2| into I_running. (sec 4.11)

            for (int pol = 0; pol < 2; pol++)
                I_running += do_second_fft(t,pol);   // sec 4.11

            if (++t_running == T_ds) {
                write_I_running_to_global_memory();  // sec 4.12
                I_running = 0;
                t_running = 0;
            }
        }
        __syncthreads();
    }
}
```

4.4 Compile-time constants

	Case 1	Case 2	Case 3	Case 4	Case 5	Notes
Input parameters (template arguments in C++)						
Number of dishes D	64	64	256	256	512	
Dish grid size (M, N)	(8,8)	(8,12)	(16,16)	(16,20)	(24,24)	
Warps/threadblock W	4	6	16	20	24	
Threadblocks/SM B	4–8?	3–5?	1–2?	1	1	See discussion in §4.4
Outer time $T_{\rm outer}$	64	48	64	40	48	
Inner time $T_{\rm inner}$	8	6	8	5	4	
Derived compile-time parameters						
$(M_{\rm pad}, N_{\rm pad})$	(8,8)	(8,16)	(16,16)	(16,32)	(32,32)	Pad M, N to powers of 2
`Freg2[]` layout (§4.9)						
M_t [Eq. (100)]	4	2	2	1	1	$M_t = 32/N_{\rm pad}$
M_w [Eq. (104)]	2	2	8	4	24	$M_w = \gcd(M/M_t, W)$
T_w [Eq. (104)]	2	3	2	5	1	$T_w = W/M_w$
M_r [Eq. (105)]	1	2	1	4	1	$M_r = M/(M_t M_w)$
T_r [Eq. (105)]	16	8	16	4	24	$T_r = T_{\rm outer}/(2T_w)$
`Fsh1[]` strides						
Σ_{F1} [Eq. (65)]	260	260	257	257	257	$(D{=}{=}64)$? 260 : 257
`Fsh2[]` strides						
Σ_{F2} [Eq. (70)]	292	290	546	545	801	$32(M+1) + M_t$
`Gsh[]` strides						
Σ_{G1} [Eq. (74)]	132	98	258	161	257	$2M_{\rm pad}T_{\rm inner} + (32/N_{\rm pad})$
Σ_{G0} [Eq. (75)]	1064	1576	4144	5168	8256	$N_{\rm pad}\Sigma_{G1} + M_{\rm pad}$
Registers/thread						
`Freg1[]` R_{F1} [Eq. 93]	16	11	16	13	22	$\lceil D/W \rceil$
`Freg2[]` R_{F2} [Eq. 106]	16	16	16	16	24	$M_r T_r$
W-array R_W [Eq. 117]	2	4	2	8	2	$2M_r$
I-array R_I [Eq. 129]	1	1	1	1	2	$M_{\rm pad}N/(16W)$
FFT constants $R_{\rm FFT}$	7	11	7	12	7	See Eq. (48) in §3
Misc $R_{\rm misc}$	15	15	15	15	15	See discussion in §4.4
Total	41	47	41	52	50	$\max(R_{F1}, R_{F2}) + R_{\rm other}$
Shared memory (bytes)						
`Fsh1[]` [Eq. (66)]	8320	8320	32896	32896	65792	$D\Sigma_{F1}/4$
`Fsh2[]` [Eq. (71)]	9344	13920	34944	43600	76896	$4N\Sigma_{F2}$
`Gsh[]` [Eq. (77)]	8512	12608	33152	41344	66048	$8\Sigma_{g0}$
Bottom-line	9344	13920	34944	43600	76896	max(`Fsh1`, `Fsh2`, `Gsh`)

Table 2: Compile-time constants for first FRB kernel. Cases 1+2 are square and rectangular versions of the CHORD pathfinder (rectangular is more likely). Cases 3+4 are square and rectangular versions of HIRAX. Case 5 is full CHORD. These compile-time constants satisfy a large number of consistency relations (compile-time asserts). See discussion near Eq. (59). For a complete list of compile-time asserts, see Eqs. (59), (78), (83), (88), (98), (107), (110), (121).

Threadblocks per SM. The number of threadblocks per SM (specified in `__launch_bounds__`) is a compile-time constant B. In cases 4 and 5, $B = 1$ is the only option, since the number of warps per threadblock W is > 16. In cases 1–3, there is a tradeoff: if B is too large, then the kernel may run out of registers and emit spill code, leading to poor performance. On the other hand, if B is too small, then the kernel will have poor occupancy. In Table 2, we have listed a plausible range for B in cases 1–3, but pinning down the optimal of B will need experimentation after the kernel has been written.

Counting "miscellaneous" persistent registers. In Table 2, we have counted registers used to hold arrays (R_{F1}, R_{F2}, R_W, R_I) and FFT constants (R_{FFT}). In addition, there are some "miscellaneous" registers used to store state as follows:

- Global memory pointers (E, I): 4 registers (2 registers/pointer).
- Other kernel arguments (E-array τ-stride, total number of time samples, time downsampling factor T_{ds}): 3 registers.
- Loop counters ($t_{\rm outer}$, $t_{\rm inner}$, t): 3 registers.
- Other persistent state (`S` from §4.8, `Fsh2_offset` and `Fsh2_mask` from §4.9, `Gsh_write_offset` from §4.10, `Gsh_read_offset` from §4.11): 5 registers.

for total register count $R_{\rm misc} = 15$ (shown in Table 2). This value of $R_{\rm misc}$ is intended to be a ballpark count, not an exact count!

Compile-time asserts. The compile-time constants in Table 2 satisfy a large number of consistency relations (compile-time asserts). We list compile-time asserts which have been assumed so far:

```
// Size restrictions for 1-d FFTs on tensor cores (sec 3).
static_assert((M >= 8) && (M <= 32) && ((M % 4) == 0));
static_assert((N >= 8) && (N <= 32) && ((N % 4) == 0));

// Definition of F-array assumes T_outer is even.                          (59)
static_assert((Touter % 2) == 0);

// Each outer block is an integer number of inner blocks.
static_assert(Touter % Tinner == 0);
```

More compile-time asserts will emerge in Eqs. (78), (83), (88), (98), (107), (110), (121) below. I wrote a python script which checks all of these compile-time asserts. The python script also checks that all shared memory access is bank conflict free, using the shared memory layouts described in the next section. (There are six shared memory access patterns to check, corresponding to reading/writing the `Fsh1[]`, `Fsh2[]`, and `Gsh[]` arrays.) Let me know if sharing this python script would be useful!

4.5 Shared memory layout

The shared memory layout consists of 3 arrays which overwrite each other:

```
union {
    int Fsh1[];
    int Fsh2[];                                                   (60)
    __half2 Gsh[];
};
```

Each of these 3 arrays has its own shared memory layout which we describe individually below.

Warning: these shared memory layouts are complicated! But, they were the simplest layouts I could find which are bank conflict free in all cases. Eliminating bank conflicts in the FRB beamforming kernel turned out to be a major challenge!

Fsh1 shared memory layout. The `Fsh1[]` array in the shared memory layout (60) stores the outer chunk after gridding $F_{\tau d}$. The logical dimensions of this array are:

$$\texttt{int Fsh1[T\_outer/2][D]; // (tau, dish)} \tag{61}$$

In shared memory, we pad the time axis to length 32. We also split the dish index $0 \le d < D$ into "high" and "low" parts:

$$d = 8d_{\rm hi} + d_{\rm lo} \qquad \text{where } 0 \le d_{\rm hi} < \frac{D}{8} \text{ and } 0 \le d_{\rm lo} < 8 \tag{62}$$

Thus, the padded array dimensions in shared memory are:

$$\texttt{int Fsh1[32][D/8][8]; // (tau, dhi, dlo)} \tag{63}$$

We use the following `Fsh1[]` shared memory strides, to avoid bank conflicts in §4.6, §4.7:

$$\big(\tau\text{-stride, } d_{\rm lo}\text{-stride, } d_{\rm hi}\text{-stride}\big) = \big(1, 32, \Sigma_{F1}\big) \tag{64}$$

where we define the compile-time constant:

$$\Sigma_{F1} = \left\{ \begin{array}{lll} 260 & \text{if } D = 64 & \text{(cases 1+2)} \\ 257 & \text{if } D \in \{256, 512\} & \text{(cases 3–5)} \end{array} \right. \tag{65}$$

The shared memory footprint of the `Fsh1[]` array is:

$$\texttt{Fsh1[]} \text{ shared memory footprint} = \Sigma_{F1} \left(\frac{D}{8}\right) \big(4 \text{ bytes}\big) \tag{66}$$

F2 shared memory layout. The `Fsh2[]` array in the shared memory layout (60) stores the gridded outer chunk $F_{\tau mn}$. The logical dimensions of this array are:

$$\texttt{int Fsh2[T\_outer/2][M][N]; // (tau, m, n)} \tag{67}$$

In shared memory, we pad the time axis to length 32. The padded dimensions are:

$$\texttt{int Fsh2[32][M][N]; // (tau, m, n)} \tag{68}$$

We use the following `Fsh2[]` shared memory layout, to avoid bank conflicts in §4.8, §4.9:

$$\big(\tau\text{-stride, } m\text{-stride, } n\text{-stride}\big) = \big(1, 33, \Sigma_{F2}\big) \tag{69}$$

where we define the compile-time constant Σ_{F2} by:

$$\Sigma_{F2} = 32(M+1) + M_t \qquad \text{where } M_t = \frac{32}{N \text{ rounded up to a power of two}} \tag{70}$$

(The motivation for the notation M_t is explained near Eq. (100) below.) The shared memory footprint of the `Fsh2[]` array is:

$$\texttt{Fsh2[]} \text{ shared memory footprint} = (N\Sigma_{F2})\big(4 \text{ bytes}\big) \tag{71}$$

G shared memory layout. The `__half2 Gsh[]` array in the shared memory layout (60) stores the intermediate FFT array $G_{\tau\pi mq}$ from Eq. (56). The logical dimensions of this array are:

$$\texttt{\_\_half2 Gsh[T\_inner][2][M][2*N]; // (tau, pol, m, q)} \tag{72}$$

where we use a `__half2` to store the real and imaginary parts of $G_{\tau\pi mq}$. In shared memory, we pad the (M, N) axes to powers of 2. The padded dimensions are:

$$\texttt{float16+16 Gsh[T\_inner][2][M\_pad][2*N\_pad]} \quad \text{where } M_{\rm pad} = 2^{\lceil \log_2(M) \rceil} \text{ and } N_{\rm pad} = 2^{\lceil \log_2(N) \rceil} \tag{73}$$

We also use the following messy shared memory layout, designed to avoid bank conflicts in §4.10, §4.11. First we define compile-time constants:

$$\Sigma_{G1} = 2M_{\rm pad}T_{\rm inner} + \frac{32}{N_{\rm pad}} \qquad \text{bit-reversed } q_1\text{-stride, see pseudocode below} \tag{74}$$

$$\Sigma_{G0} = N_{\rm pad}\Sigma_{G1} + M_{\rm pad} \qquad q_0\text{-stride, see pseudocode below} \tag{75}$$

It is easiest to specify the shared memory layout using the following cuda pseudocode:

```
__device__ inline __half2 *Gsh_address(int tau, int pol, int m, int q)
{
    // Compile-time constants (see above)
    constexpr int Sigma_G1 = 2*Mpad*Tinner + (32/Npad);
    constexpr int Sigma_G0 = Npad*Sigma_G1 + Mpad;

    int q0 = (q & 1);      // lowest bit of q (either 0 or 1)
    int qr = (q >> 1);     // remaining bits of q (satisfying 0 < qr < N_pad)
    int qr_brev = brev(qr);  // bit-reversed (satisfying 0 < qr_brev < N_pad)

    extern __shared__ __half2 shmem_base[];

    return (shmem_base + m          // m has stride 1
             + pol * Mpad           // pol has stride Mpad
             + tau * (2*Mpad)       // tau has stride (2 Mpad)
             + qr_brev * Sigma_G1   // qr_brev has stride (Sigma_G1)
             + q0 * Sigma_G0);      // q0 has stride (Sigma_G0)
}
```
(76)

The shared memory footprint of the `Gsh[]` array is:

$$\texttt{Gsh[]} \text{ shared memory footprint} = (2\Sigma_{g0})\big(4 \text{ bytes}\big) \tag{77}$$

Compile-time asserts. In this section, we have made the following assumptions:

```
// Assumed in Fsh1[] and Fsh2[], where we zero-pad from length (Touter/2) to 32.
static_assert(Touter <= 64);

// Assumed in Fsh1[], when defining stride Sigma_{F1}.
static_assert((D==64) || (D==256) || (D==512));

// Ensure that Fsh2[] stride is large enough that array elements don't "collide"
static_assert(Sigma_F2 >= 33*M);
```
(78)

4.6 Copying global memory to shared memory (Fsh1)

In this step, we copy an "outer block" of electric field data, consisting of $T_{\rm outer}$ time samples, from global memory to the `Fsh1[tau,d]` array in shared memory. This step appears in the outermost loop of the kernel (see pseudocode in §4.3). Note that the outer block is stored in global memory in the E-representation (52), but we write it to shared memory in the F-representation (53), so we need to do some bit-shuffling.

E→F shuffle. First, we describe a thread-local operation which reshuffles electric field data from its E-representation (52) to its F-representation (53). The input and output will consist of 4 registers/thread.

When data is read from global memory in the E-representation, each register contains four dishes:

$$[\text{int4 } E_d] \qquad b_0b_1b_2 \leftrightarrow \text{ReIm}, d_0, d_1 \tag{79}$$

Recall from §4.2 that we group E-array elements into quadruples, by either varying the polarization index π, or by varying the time index by $(\pm T_{\rm outer}/2)$. Suppose that on a single thread, we have one such quadruple of E-array registers:

$$[\text{int4 } E_{\tau\pi d}] \qquad b_0 b_1 b_2 \leftrightarrow \text{ReIm}, d_0, d_1 \qquad r_0 r_1 \leftrightarrow \tau_{\rm hi}\pi \tag{80}$$

where the index bit $\tau_{\rm hi}$ denotes varying the time index τ by $(T_{\rm outer}/2)$. (This notation was previously used near Eq. (54).)

To convert to the F-representation, we will need to perform a thread-local bit shuffling operation, whose input is the four registers in Eq. (80), and whose output is the same data in the following bit/register assignment:

$$[\text{int4 } E_{\tau\pi d}] \qquad b_0 b_1 b_2 \leftrightarrow \pi, \tau_{\rm hi}, \text{ReIm} \qquad r_0 r_1 \leftrightarrow d_0 d_1 \tag{81}$$

I leave to you the nontrivial problem of finding the fastest way to get from (80) to (81)! (We exchanged a few emails on this, with subject "bit-shuffling operation for FRB beamformer".) Note that the ReIm bit needs to move from b_0 to b_2, which costs cycles. However, the assignment $b_2 \leftrightarrow$ ReIm will save cycles later (see "Unpacking int4 to float16" in §4.10).

Note that, by the definition of the F-array in Eqs. (53), (55), the 4 registers in Eq. (81) are the same thing as the F-array elements for 4 dishes:

$$[\text{int32 } F_d] \qquad r_0 r_1 \leftrightarrow d_0 d_1 \tag{82}$$

We will call the thread-local operation which goes from (80) to (82) the ***E* → *F* shuffle**.

Using the $E \to F$ shuffle as a building block, we now return to the higher-level task of copying global memory to shared `Fsh1[]`. We split this into two cases as follows.

Cases 1+2. In these cases, one can check using Table 2 that the following compile-time asserts are satisfied:

```
// Cases 1+2 only
static_assert(D == 64);
static_assert(T_outer % 8 == 0);                    (83)
static_assert((T_outer/8) % W == 0);
```

Since $T_{\rm outer}$ is a multiple of 8, we split the time index $0 < \tau < T_{\rm outer}$ into "high", "middle", and "low" parts:

$$\tau = \frac{T_{\rm outer}}{2}\tau_{\rm hi} + 4\tau_{\rm mid} + \tau_{\rm lo} \qquad 0 \le \tau_{\rm hi} < 2, \quad 0 \le \tau_{\rm mid} < \frac{T_{\rm outer}}{8}, \quad 0 \le \tau_{\rm lo} < 4 \tag{84}$$

We define a *tile* to be a single $\tau_{\rm mid}$ value (with 64 dishes, 2 polarizations, 2 $\tau_{\rm hi}$ values, and 4 $\tau_{\rm lo}$ values). Since W divides $N_{\rm tiles} = (T_{\rm outer}/8)$, we can parallelize by assigning tiles to warps (possibly in a several-to-one way). For the rest of "Cases 1+2", we will describe how a single tile is processed by a single warp.

First, we load the tile into registers, with register assignment (8 registers/thread):

$$[\text{int4 } E_{\tau\pi d}] \qquad b_0 b_1 b_2 \leftrightarrow \text{ReIm}, d_0, d_1 \qquad r \leftrightarrow d_2 d_3 \tau_{\rm hi} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_4 d_5 \pi \tau_0^{\rm lo} \tau_1^{\rm lo} \tag{85}$$

A little thought shows that this can be done with two 16-byte cache-aligned load instructions. (Recall that 16-byte load instructions have higher memory bandwidth than the usual 4-byte load instructions.)

Next, we do a warp transpose operation (Appendix B.3) on indices d_3, π, obtaining register assignment:

$$[\text{int4 } E_{\tau\pi d}] \qquad b_0 b_1 b_2 \leftrightarrow \text{ReIm}, d_0, d_1 \qquad r \leftrightarrow d_2 \tau_{\rm hi} \pi \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_4 d_5 d_3 \tau_0^{\rm lo} \tau_1^{\rm lo} \tag{86}$$

Next, we do the $E \to F$ shuffle above, obtaining an F-array tile with register assignment:

$$[\text{int32 } F_{\tau d}] \qquad r \leftrightarrow d_0 d_1 d_2 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d_4 d_5 d_3 \tau_0^{\rm lo} \tau_1^{\rm lo} \tag{87}$$

Note that the F-array does not contain the $\tau_{\rm hi}$ index bit, since the F-array has half as many time samples as the E-array (Eqs. (52), (53)).

Finally, we write the F-registers to the `Fsh1[tau,d]` array in shared memory, using eight 32-bit stores. One can check that these stores are bank conflict free, using the `Fsh1[]` shared memory layout in Eq. (64).

Cases 3–5. In these cases, one can check using Table 2 that the following compile-time asserts are satisfied:

```
// Cases 3-5 only
static_assert(D % 256 == 0);
static_assert(T_outer % 2 == 0);                                  (88)
static_assert(((D/256) * (T_outer/2)) % W == 0);
```

Since $T_{\rm outer}$ is a multiple of 2, we split the time index $0 < \tau < T_{\rm outer}$ into "high" and "low" parts:

$$\tau = \frac{T_{\rm outer}}{2}\tau_{\rm hi} + \tau_{\rm lo} \qquad \text{where } 0 \le \tau_{\rm hi} < 2 \text{ and } 0 \le \tau_{\rm lo} < \frac{T_{\rm outer}}{2} \tag{89}$$

We define a *tile* to be 256 dishes, and a single $\tau_{\rm lo}$ value (with 2 $\tau_{\rm hi}$ values and 2 polarizations). Since W divides $N_{\rm tiles} = (D/256)\,(T_{\rm outer}/2)$, we can parallelize by assigning tiles to warps (possibly in a several-to-one way). In the rest of this subsection, we will describe how a single tile is processed by a single warp.

First, we load the tile into registers, with register assignment (8 registers/thread):

$$[\text{int4 } E_{\tau\pi d}] \qquad b_0b_1b_2 \leftrightarrow \text{ReIm}, d_0, d_1 \qquad r \leftrightarrow d_2d_3\tau_{\rm hi} \qquad t_0t_1t_2t_3t_4 \leftrightarrow d_4d_5d_6d_7\pi \tag{90}$$

This can be done with two 16-byte cache-aligned load instructions. Next, we do a warp transpose operation (Appendix B.3) on indices d_3, π, obtaining register assignment:

$$[\text{int4 } E_{\tau\pi d}] \qquad b_0b_1b_2 \leftrightarrow \text{ReIm}, d_0, d_1 \qquad r \leftrightarrow d_2\pi\tau_{\rm hi} \qquad t_0t_1t_2t_3t_4 \leftrightarrow d_4d_5d_6d_7d_3 \tag{91}$$

Then we do the $E \to F$ shuffle above, obtaining an F-array tile with register assignment:

$$[\text{int32 } F_d] \qquad r \leftrightarrow d_0d_1d_2 \qquad t_0t_1t_2t_3t_4 \leftrightarrow d_4d_5d_6d_7d_3 \tag{92}$$

Note that the F-array does not contain the $\tau_{\rm hi}$ index bit, since the F-array has half as many time samples as the E-array (Eqs. (52), (53)).

Finally, we write the F-registers to the `Fsh1[]` array in shared memory, using eight 32-bit stores. One can check that these stores are bank conflict free, using the `Fsh1[]` shared memory layout in Eq. (64).

4.7 Reading shared memory (Fsh1) into registers (Freg1)

In this step, we read the shared memory array `Fsh1[]` into the register array `Freg1[]`, for one outer block consisting of $T_{\rm outer}$ time samples. This step appears in the outermost loop of the kernel (see pseudocode in §4.3).

The destination register array has logical dimensions `int32 Freg1[Touter/2][D]`, to be distributed across all threads in the threadblock. To distribute the array, we assign dishes to warps in round-robin fashion. Each time index $\tau = 0, \cdots, (T_{\rm outer}/2) - 1$ is assigned to the corresponding thread index $t = 0, \cdots, (T_{\rm outer}/2) - 1$. Thread indices $t = (T_{\rm outer}/2), \cdots, 31$ will read junk data, from shared memory addresses which were not initialized in §4.6. This turns out to be harmless (briefly, the junk data does get written to `Fsh2[]` in §4.8, but will not be read back into `Freg2[]` in §4.9).

The number of registers in the `Freg1[]` array is:

$$\big(\text{Number of registers/thread } R_{F1}\big) = \left\lceil \frac{D}{W} \right\rceil \tag{93}$$

The shared memory load instuctions needed to read `Fsh1[]`$\to$`Freg1[]` are bank conflict free, since the `Fsh1[]` array has τ-stride 1 (Eq. (64)).

4.8 Writing shared memory (Fsh2) from registers (Freg1)

In the previous step (§4.7), we read one outer block of electric field data into a register array `Freg1[tau,d]`. In this step, we write this data to the shared memory array `Fsh2[tau,m,n]`. Note that dish gridding $d \to (m, n)$ happens in this step, since the input data is ungridded (indexed by a dish d), whereas the output data is gridded (indexed by a grid location m, n). This step appears in the outermost loop of the kernel (see pseudocode in §4.3).

Some grid locations (m, n) are not occupied by dishes. We will set the corresponding `Fsh2[tau,m,n]` elements to zero.

We define two integer-valued length-(MN) arrays m_d, n_d as follows. For $0 \le d < D$, let (m_d, n_d) be the grid coordinates of dish d. For $D \le d < MN$, let (m_d, n_d) be the unoccupied grid locations, in an arbitrary order. That is, we treat unoccupied grid locations as dummy dishes $D \le d < MN$ whose electric fields are zero.

For $0 \le d < MN$, let s_d be the offset (relative to the beginning of shared memory) of element $(\tau, m, n) = (0, m_d, n_d)$ in the `Fsh2[tau,m,n]` shared memory array. Concretely, s_d is given by (see Eq. (69)):

$$s_d = 33\, m_d + \Sigma_{F2}\, n_d \tag{94}$$

In the GPU kernel, it may help to think of dish gridding as a copy operation `Freg1[:,d]` $\rightarrow$ `Fsh2[`s_d`:`s_d`+32]` from registers to shared memory, rather than an index mapping $d \rightarrow (m_d, n_d)$.

We use one persistent register per thread (denoted S) to store values of s_d. On warp $0 \le w < W$ and thread $0 \le t < 32$, the value of S is:

$$S = \begin{cases} s_{w+tW} & \text{if } 0 \le t < (MN/W) \\ \text{arbitrary} & \text{if } (MN/W) \le t < 32 \end{cases} \tag{95}$$

This register assignment is chosen for consistency with the `Freg1[]` register array from the previous section (§4.7). Recall that each warp $0 \le w < W$ holds the following dish indices in its `Freg1[]` array:

$$d = w + iW \qquad \text{where } 0 \le i < R_F \tag{96}$$

When dish $d = w + iW$ is written to shared memory `Fsh2[]`, a warp shuffle is needed to "broadcast" the value of s_d from thread $t = i$ to all threads. It is easiest to explain this in pseudocode:

```
int Freg1[RF1];  // Temp registers holding F-array, populated in sec 4.7.
int S;           // Persistent register for storing s_d, see above

extern __shared__ int Fsh2[];
int t = threadIdx.x & 0x1f;   // thread id 0 <= t < 32

#pragma unroll
for (int i = 0; i < RF1; i++) {
    // Freg1[i] contains F-array element for (tau,d)=(t,w+iW).
    int sd = __shfl_sync(0xffffffff, S, i);  // s_d for d=w+iW
    Fsh2[sd+t] = Freg1[i];  // Recall from sec 4.5 that Fsh2[] tau-stride is 1.
}

for (int i = RF1; i < (M*N)/W; i++) {
    // Dummy dish: write zero for (tau,d)=(t,w+iW).
    int sd = __shfl_sync(0xffffffff, S, i);  // s_d for d=w+iW
    Fsh2[sd+t] = 0;  // Recall from sec 4.5 that Fsh2[] tau-stride is 1.
}
```
(97)

Note that the shared memory writes are bank conflict free, since the `Fsh2[]` shared memory layout in Eq. (69) has τ-stride 1.

At the beginning of the kernel, we need to initialize the persistent S-register in Eq. (95). I'll leave it to you to decide how best to implement this, including details such as: whether the conversion (94) from $(m, n) \rightarrow s$ happens on the CPU or GPU, and what global memory layout is used to store the array(s) which pass the dish gridding from the CPU to the kernel.

In the pseudocode (97), we made a couple of assumptions which are captured by the following compile-time asserts:

```
// Grid locations (M,N) divide evenly between warps.
static_assert((M*N) % W == 0);

// Only one S-register per thread is needed to store the s_d array.
static_assert((M*N)/W <= 32);
```
(98)

Both assumptions were made for convenience, and could be relaxed. (One more detail: for the pseudocode (97) to work as written, some threads need to store zeros in the last register of the `Freg1[]` array, corresponding to dish indices $D \le d < WR_{F1}$.)

4.9 Reading shared memory (Fsh2) into registers (Freg2)

In this step, we read the shared memory array `Fsh2[tau,m,n]` into the register array `Freg2[tau,m,n]`, for an "outer" block of $T_{\rm outer}$ time samples (see pseudocode in §4.3). After this step, the shared memory occupied by `Fsh2[]` is no longer needed, and can be overwritten by `G[]` in subsequent steps. The `Freg2[]` registers are held persistently throughout "first FFT" and "second FFT" steps in the kernel (§4.10 and §4.11).

Freg2 register assignment. The `Freg2` register array is distributed across all $(32W)$ threads in the kernel, in a nontrivial scheme which takes some time to describe.[8]

The F-array has logical dimensions

```
int F[Touter/2][M][N];    // (tau,M,N)
```
(99)

We define a *mini-tile* to be a subset of the F-array consisting of one τ-index, M_t consecutive m-indices, and all n-indices $0 \le n < N$, where:

$$M_t = \frac{32}{N_{\rm pad}} \qquad\qquad N_{\rm pad} = \big(N \text{ rounded up to a power of two}\big) \tag{100}$$

Each mini-tile will be stored in a single warp, with 1 register/thread. The mini-tile register assignment is as follows. We first split the index $0 \le n < N$ into "high" and "low" parts:

$$n = \frac{N}{4} n_{\rm hi} + n_{\rm lo} \qquad\quad \text{where } 0 \le n_{\rm hi} < 4 \;\text{ and }\; 0 \le n_{\rm lo} < \frac{N}{4} \tag{101}$$

When the shape-(M_t, N) mini-tile is stored in registers, we zero-pad the dimensions to $(M_t, N_{\rm pad})$, by extending $n_{\rm lo}$ to an index $0 \le n_{\rm lo} < (N_{\rm pad}/4)$. The zero-padded mini-tile is stored with register assignment (one `int32` register/thread):

$$\Big[\Big(\frac{32}{N_{\rm pad}} \times N_{\rm pad}\Big)\, F_{mn}\Big] \qquad\qquad t_0t_1t_2t_3t_4 \leftrightarrow n_0^{\rm hi} n_1^{\rm hi} n_0^{\rm lo}\; \underbrace{m_0 \cdots m_{4-\nu}\; n_1^{\rm lo} \cdots n_{\nu-3}^{\rm lo}}_{\text{2 bits}} \tag{102}$$

where we have defined

$$\nu = \log_2(N_{\rm pad})\,. \tag{103}$$

Next, we define a *mega-tile* to be a $(T_w \times M_w)$ grid of mini-tiles, where the compile-time constants T_w, M_w are defined by:

$$M_w = \gcd\Big(\frac{M}{M_t}, W\Big) \qquad\qquad T_w = \frac{W}{M_w} \tag{104}$$

[8]Here is a brief explanation of why my proposed `Freg2[]` register assignment is so complicated! The choice of (M_t, N_t) and mini-tile register assignment (102) is dictated by the input register assignment for the tensor core FFT (§3). See §4.10 for details on how `Freg2` gets "unpacked" to `float16` and fed into the tensor core FFT. Then, the choice of (M_w, T_w, M_r, T_r) in Eqs. (104), (105) is designed to minimize M_r, in order to minimize the number of persistent registers R_W needed to store the W-array (see Eq. (117) below).

Each mega-tile consists of W mini-tiles, and is distributed across all W warps in the threadblock, with 1 register/thread. Note that a mega-tile contains T_w τ-indices, $(M_t M_w)$ m-indices, and all n-indices (with zero-padding as described above).

Finally, the entire shape-$(T_{\rm outer}/2, M, N)$ F-array is a $(T_r \times M_r)$ grid of mega-tiles, where the compile-time constants T_r, M_r are defined by:

$$T_r = \frac{T_{\rm outer}/2}{T_w} \qquad\qquad M_r = \frac{M}{M_w M_t} \tag{105}$$

Each mega-tile in the $(T_r \times M_r)$ grid of mega-tiles is stored in a different register. Thus:

$$\Big(\text{Number of registers/thread } R_{F2} \text{ in the } \texttt{Freg2[]} \text{ array }\Big) = T_r M_r \tag{106}$$

We have now fully described the `Freg2[]` register assignment (across all W warps in the kernel). This scheme implicitly depends on the following compile-time asserts:

```
// F-array divides evenly into mini-tiles
constexpr int Mt = 32/Npad;
static_assert(M % Mt == 0);

// F-array divides evenly into mega-tiles                                    (107)
constexpr int Mw = gcd(M/Mt, W);
constexpr int Tw = W/Mw;
static_assert((Touter/2) % Tw == 0)
static_assert(M % (Mw*Mt) == 0);
```

We checked that these asserts are satisfied for each case 1–5 in Table 2. These asserts imply $M_t M_w M_r = M$ and $T_w T_r = T_{\rm outer}/2$.

Reading from shared memory. So far in this section, we have just described the `Freg2[]` register assignment. We now describe the code for reading the shared memory array `Fsh2[tau,m,n]` into the register array `Freg2[tau,m,m]`.

Conceptually, this is straightforward. On each thread, we have a $(T_r \times M_r)$ array of registers. For each such register, we read from a shared memory address which depends on the warp index $0 \le w < W$ and thread index $0 \le t < 32$. However, there is a technical issue arising from padding the n-index. In the destination register array `Freg2[]`, some threads store "zero-padded" n-indices $N \le n < N_{\rm pad}$ (Eq. (102)). However, the n-index is unpadded in the source shared memory array `Fsh2[]` (§4.5). Therefore, straightforward indexing of the source array will lead to either some threads reading past the end of shared memory, or warp divergence to suppress load instructions on some threads.

This can be addressed with a technical trick as follows. We call a thread *zero-padded* if it corresponds to a zero-padded index $N \le n < N_{\rm pad}$, in the `Freg2[]` register assignment from Eq. (102). On zero-padded threads, we compute the shared memory address straightforwardly (which will be "out of bounds"), and then replace it with an arbitrary in-bounds address *in the same shared memory bank.* Then we can issue a shared memory load instruction which is in-bounds, bank conflict free, and warp divergence free, but it will read "junk" (i.e. arbitrary) data on zero-padded threads. To fix this, we apply a mask which zeros the data on zero-padded threads.

Here's some pseudocode to explain the trick better. At the beginning of the kernel (not shown in the pseudocode from §4.3), we initialize two persistent registers `Fsh2_offset` and `Fsh2_mask` as follows:

```
constexpr int Mt = 32/Npad;
int w = (threadIdx.x >> 5);    // warp id 0 <= w < W
int t = (threadIdx.x & 0x1f);  // thread id 0 <= t < 32

// (mt,nt) = coordinates of thread within (Mt x Npad) mini-tile.
// This complicated logic reflects the mini-tile register assignment above.
int mt = (t >> 3) % Mt;                      // 0 <= mt < Mt
int nhi = (t & 3);                           // 0 <= nhi < 4
int nlo = ((t >> 3) / Mt) + ((t >> 2) & 1);  // 0 <= nlo < Npad/4
int nt = nhi*(N/4) + nlo;                    // only valid if (is_padded == false)
bool is_padded = (nlo >= (N/4));

// (tauw,mw) = coordinates of warp within (Tw x Mw) mega-tile.
int tauw = w / Mw;  // 0 <= tauw < Tw
int mw = w % Mw;    // 0 <= mw < Mw                                                    (108)

// s0 = shared memory offset of F-register (0,0)
//      [within the (Tr x Mr) register grid on each thread]
//
// Recall from sec 4.5 that the F2[tau,m,n] array has tau-stride 1,
// m-stride 33, and n-stride Sigma_F2 (a compile-time constant).

int s0 = tauw + 33*(mw*Mt+mt) + Sigma_F2*nt;   // note m=(mw*Mt+mt) here

// On a zero-padded thread, s0 may be past the end of shared memory. Replace
// s0 by an "in-bounds" offset in the same bank (to avoid bank conflicts later).

int Fsh2_offset = is_padded ? (s0 & 0x1f) : s0;
int Fsh2_mask = is_padded ? 0 : 0xffffffff;
```

Then, in the outermost loop of the kernel (see pseudocode in §4.3), we read the shared memory array `Fsh2[]` into the register array `Freg2[]` as follows:

```
extern int __shared__ Fsh2[];
int *sp = Fsh2 + Fsh2_offset;  // Note Fsh2_offset included here.

int Freg2[Tr][Mr];  // destination array of persistent registers

#pragma unroll
for (int tr = 0; tr < Tr; tr++) {
    #pragma unroll
    for (int mr = 0; mr < Mr; mr++) {
        // Offsets from (tr,mr), to be added to Fsh2_offset above.              (109)
        // Note constexpr here, since loop is unrolled!
        constexpr int s = tr + 33*(Mt*Mw*mr);

        // The shared memory load sp[s] is in-bounds and bank conflict free.
        // But, it reads "junk" data on zero-padded threads, whereas we want zeros.
        // We fix this by applying the Fsh2_mask.
        Freg2[tr][mr] = sp[s] & Fsh2_mask;
    }
}
```

One can check that the shared memory loads in this pseudocode are bank conflict free, for each case 1–5 in Table 2, using the `Fsh2[]` shared memory layout from Eq. (69).

One important comment! The trick in (108), (109) also serves the important purpose of reducing compute time (in addition to keeping shared memory access in-bounds). After precomputing `Fsh2_offset`, all shared memory offsets in the inner loop (109) are compile-time constants. A similar trick may be useful in other places to save compute time. In particular, we will suggest doing this in §4.10, §4.11 below, by precomputing `Gsh[]` offsets.

4.10 First FFT

In this step, we perform the FFT $E_{\tau\pi mn} \to G_{\tau\pi mq}$ (see Eq. (56)), for one inner chunk consisting of $T_{\rm inner}$ time samples (see pseudocode in §4.3). The input E-array comes from the `Freg2[]` register array, which was initialized in the previous step (§4.9). The output G-array is written to shared memory `Gsh[]`.

This step uses the "mega-tile" and "mini-tile" scheme, introduced in the previous step (§4.9), for distributing (τ, m, n)-indices among threads in the kernel. Here, we add one new compile-time assert:

```
// Inner block (T_inner time indices) divides evenly into mega-tiles.
assert(Tinner % Tw) == 0;
```
(110)

The high-level logic of the first FFT is easiest to explain in pseudocode:

```
// This double-for loop runs over mega-tiles which comprise one inner chunk.
// We unroll the loops, since the loop counters (tau_tile, m_tile) will
// end up indexing the register array Freg2[].

#pragma unroll
for (int tau_tile = 0; tau_tile < T_inner; tau_tile += T_w) {
    #pragma unroll
    for (int m_tile = 0; m_tile < M; m_tile += M_t*M_w) {

        // Inner loop over polarization
        #pragma unroll
        for (int pol = 0; pol < 2; pol++) {
            // Each of these steps is described under its own heading below.
            unpack_int4_to_float16();
            multiply_W();
            do_fft();
            write_G_to_shared_memory();
        }
    }
}
```
(111)

In the rest of this section, we'll describe each of the steps in the inner loop. Throughout this discussion, we focus on a single iteration of the inner loop, on a single warp. Thus, we can pretend that the data consists of one mini-tile (one τ-index, M_t consecutive m-indices, and all n-indices), for a single polarization index π. We will usually leave τ, π indices implicit (e.g. the E-array will be denoted E_{mn}).

Unpacking int4 to float16. The $(\tau_{\rm tile}, m_{\rm tile})$ values of interest live in one `Freg2[]` register, with register assignment (same as Eq. (102) above):

$$[\text{int32 } F_{mn}] \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow n_0^{\rm hi} n_1^{\rm hi} n_0^{\rm lo} \underbrace{m_0 \cdots m_{4-\nu} \; n_1^{\rm lo} \cdots n_{\nu-3}^{\rm lo}}_{\text{2 bits}} \tag{112}$$

where $n_{\rm lo}, n_{\rm hi}, \nu$ were defined in Eqs. (101), (103). Unpacking the definition (55) of F, this register holds four `int4+4` E-array elements, with register assignment:

$$[\text{int4 } E] \qquad b_0 b_1 b_2 \leftrightarrow \pi, \tau_{\rm hi}, \text{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow n_0^{\rm hi} n_1^{\rm hi} n_0^{\rm lo} \underbrace{m_0 \cdots m_{4-\nu} \; n_1^{\rm lo} \cdots n_{\nu-3}^{\rm lo}}_{\text{2 bits}} \tag{113}$$

Within the inner loop in the pseudocode (111), we are processing a particular value of the index bits $(\tau_{\rm hi}, \pi)$. We extract the appropriate `int4` elements from the simd register (113) and convert `int4`$\rightarrow$`float16`, obtaining:

$$[\text{float16 } E_{mn}] \qquad b \leftrightarrow \text{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow n_0^{\rm hi} n_1^{\rm hi} n_0^{\rm lo} \underbrace{m_0 \cdots m_{4-\nu}\; n_1^{\rm lo} \cdots n_{\nu-3}^{\rm lo}}_{\text{2 bits}} \tag{114}$$

Erik and Kendrick had some discussion in `#bx-engine-dev` around March 18 on how to go from (113) to (114) efficiently. Note that the index bit assignment $b_2 \leftrightarrow$ ReIm in (113) is a choice that was made intentionally, in order to make this step as efficient as possible. (This choice is "baked in" to the definition of F in Eq. (55), and we also commented on it after Eq. (81), but we couldn't really justify it until now.)

Multiplying by W. Before FFT-ing the E-array E_{mn}, we must multiply it by the weight array W_{mn} (see Eq. (56)). Let's temporarily assume that W_{mn} is held in persistent registers on the same warp, with the same register assignment as the E-array (Eq. (114)):

$$[\text{float16 } W_{mn}] \qquad b \leftrightarrow \text{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow n_0^{\rm hi} n_1^{\rm hi} n_0^{\rm lo} \underbrace{m_0 \cdots m_{4-\nu}\; n_1^{\rm lo} \cdots n_{\nu-3}^{\rm lo}}_{\text{2 bits}} \tag{115}$$

Then we can multiply the complex arrays W and E with the cuda intrinsic `__hcmadd()`, obtaining the product array (WE) with the same register assignment:

$$[\text{float16 } (WE)_{mn}] \qquad b \leftrightarrow \text{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow n_0^{\rm hi} n_1^{\rm hi} n_0^{\rm lo} \underbrace{m_0 \cdots m_{4-\nu}\; n_1^{\rm lo} \cdots n_{\nu-3}^{\rm lo}}_{\text{2 bits}} \tag{116}$$

Here and below, we denote the product array by $(WE)_{mn} = W_{mn} E_{mn}$, rather than introducing separate notation.

It remains to describe the details of how the W-array gets stored in persistent registers, in the register assignment (115). First let's count the total number of registers needed to store the W-array:

$$\big(W\text{-array registers per thread } R_W\big) = 2M_r \tag{117}$$

where the factor 2 is from polarization. These registers are initialized at the beginning of the kernel (not shown in pseudocode in §4.3), by reading the W-array from global memory. Note that this is a several-to-one operation, in the sense that each W-array element gets copied onto T_w threads. Also note that the W-array register mapping (115) is zero-padded: some threads store zeros.

I'll leave to you the task of deciding how to implement the W-array initialization, including decisions such as: What global memory layout is used for the W-array? Does the several-to-one copy operation go through shared memory or L1 cache? Does the data reordering in the W-array register assignment (115) happen on the CPU or GPU?[9]

Doing the FFT. We use the tensor core FFT from §3, to do the FFT $(WE)_{mn} \rightarrow G_{mq}$. The (WE)-array register assignment (116) is consistent with the register assignment (25) for the FFT input array. Note that the n-index in $(WE)_{mn}$ corresponds to the "active" n-index in the FFT (25), and the m-index in $(WE)_{mn}$ corresponds to the spectator s-index in the FFT (25).

After doing the FFT, we get the array G_{mq} for one mini-tile, with $M_t = (32/N_{\rm pad})$ m-indices and $(2N)$ q-indices. The register assignment for G_{mq} is given by the FFT output assignment given previously in (26). We first split the index $0 \le q < (2N)$ into "high" and "low" parts:

$$q = 8q_{\rm hi} + q_{\rm lo} \qquad \text{where } 0 \le q_{\rm hi} < (N/4) \text{ and } 0 \le q_{\rm lo} < 8 \tag{118}$$

Then the G_{mq} register assignment is:

$$\big[(M_t {\times} 2N_{\rm pad})\, G_{mq}\big] \qquad b_0 \leftrightarrow q_0^{\rm lo} \qquad r \leftrightarrow \text{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow q_1^{\rm lo} q_2^{\rm lo} \underbrace{q_0^{\rm hi} \cdots q_{\nu-3}^{\rm hi}\; m_0 \cdots m_{4-\nu}}_{\text{3 bits}} \tag{119}$$

[9]Now is a good time to mention that in the highest-level interface to the FRB beamformer, we will want the weights array to be specified in ungridded form (i.e. as $W_{f\pi d}$ rather than $W_{f\pi mn}$). I think this detail can be added later. Since the W-array is small, I think we'll decide it's easiest to do the gridding $W_{f\pi d} \rightarrow W_{f\pi mn}$ on the CPU, then copy to the GPU and launch the kernel.

where the q-index has been zero padded from length-$(2N)$ to length-$(2N_{\rm pad})$, by extending the index $q_{\rm hi}$ from length-$(N/4)$ to length-$(N_{\rm pad}/4)$.

Finally, we do a local transpose (Appendix B.2) to get register assignment:

$$\big[(M_t \times 2N_{\rm pad})\, G_{mq}\big] \qquad b_0 \leftrightarrow {\rm ReIm} \qquad r \leftrightarrow q_0^{\rm lo} \qquad t_0t_1t_2t_3t_4 \leftrightarrow q_1^{\rm lo}q_2^{\rm lo}\, \underbrace{q_0^{\rm hi}\cdots q_{\nu-3}^{\rm hi}\; m_0\cdots m_{4-\nu}}_{\text{3 bits}} \tag{120}$$

Writing G to shared memory. Finally, we write the G_{mq} (two registers per thread) to the shared memory array `Gsh[]`. Using register assignment (120) for the source registers, and shared memory layout (76) for the destination array `Gsh[]`, the shared memory writes are bank conflict free. This statement is not obvious at all – I wrote python code to check it for each of our cases 1–5!

Computing `Gsh[]` shared memory destination offsets may be a source of overhead, since the shared memory layout (76) is so complicated. I suggest the following: on each thread, precompute the shared memory offset corresponding to $\tau_{\rm tile} = m_{\rm tile} = \pi = 0$ (in notation following the pseudocode (111)), and store it in a persistent register `Gsh_write_offset`. Then, in the unrolled loops in the pseudocode (111), all shared memory offsets can be obtained from `Gsh_write_offset` by adding compile-time constants. (The idea is similar to the trick in Eqs. (108), (109) from the previous section. Note that in this case, there is no issue with writing past the end of shared memory, since the q-axis of `Gsh[]` is padded from length $2N$ to length $2N_{\rm pad}$.)

4.11 Second FFT

This step appears in an inner loop over τ, π (see pseudocode in §4.3), and therefore operates on a single time sample τ and polarization π. In the rest of this section, we will not write τ, π indices explicitly.

In this step, we read the array G_{mq} from shared memory, perform the FFT $G_{mq} \to \widetilde{E}_{pq}$ (see Eq. (57)), and accumulate $|\widetilde{E}_{pq}|^2$ into a running total I_{pq}. The I-array is held in persistent registers.

We define a *tile* of the array G_{mq} to consist of all m-values, and $(32/M_{\rm pad})$ q-values. Here, $M_{\rm pad}$ is M rounded up to a power of 2. For each of our cases 1–5 in Table 2, the following compile-time asserts are satisfied:

```
constexpr int nq_total = 2*N;            // total q-indices in G-array
constexpr int nq_tile = 32/Mpad;         // q-indices per tile
constexpr int ntiles = nq_total / nq_tile;
                                                                          (121)
static_assert(nq_total % nq_tile == 0);  // G-array divides evenly into tiles
static_assert(ntiles % W == 0);          // Tiles divide evenly among warps
```

In view of these asserts, we can parallelize by dividing tiles among warps. (The number of tiles per warp turns out to be 1 in cases 1–4, and 2 in case 5.) In the rest of this section, we describe how a single tile is processed by a single warp. Thus the G_{mq} array will have shape $(M, 32/M_{\rm pad})$, and the $\widetilde{E}_{pq}$ array will have shape $(2M, 32/M_{\rm pad})$.

Reading one G-tile from shared memory. We read the G_{mq} array from shared memory into a register assignment which is tailored to the tensor core FFT (§3). We split the index $0 \le m < M$ into "high" and "low" parts:

$$m = \frac{M}{4} m_{\rm hi} + m_{\rm lo} \qquad \text{where } 0 \le m_{\rm hi} < 4 \text{ and } 0 \le m_{\rm lo} < \frac{M}{4} \tag{122}$$

We will sometimes zero-pad length-M arrays to length-$M_{\rm pad}$, by extending $m_{\rm lo}$ to an index $0 \le m_{\rm lo} < (M_{\rm pad}/4)$. We read $G_{\rm mq}$ from shared memory, with register assignment (1 register/thread):

$$\left[\left(M_{\rm pad} \times \frac{32}{M_{\rm pad}}\right) G_{mq}\right] \qquad b \leftrightarrow {\rm ReIm} \qquad t_0t_1t_2t_3t_4 \leftrightarrow m_0^{\rm hi}m_1^{\rm hi}m_0^{\rm lo}\; \underbrace{q_0\cdots q_{4-\mu}\; m_1^{\rm lo}\cdots m_{\mu-3}^{\rm lo}}_{\text{2 bits}} \tag{123}$$

where we have defined

$$\mu = \log_2(M_{\rm pad})\,. \tag{124}$$

Using the `Gsh[]` shared memory layout previously defined in (76), this 32-bit shared memory load is bank conflict free. This statement is not obvious at all – I wrote python code to check it for each of our cases 1–5!

As in the previous section (§4.10), computing `Gsh[]` shared memory offsets may be a source of overhead, so we may want to precompute a persistent register `Gsh_read_offset`, to reduce computation in the innermost loop.

Another issue: if M is not a power of 2 (this only happens in Case 5), then some thread indices in (120) will correspond to m-indices with $M \le m < M_{\rm pad}$. We want these threads to store zeros, not "junk" data read from uninitialized `Gsh[]` addresses. This could be implemented either by zeroing the appropriate elements of `Gsh[]`, or with a conditional move instruction after reading the data.

Doing the FFT. We use the tensor core FFT from §3 to do the FFT $G_{mq} \to \widetilde{E}_{pq}$. The G-array register assignment (123) is consistent with the register assignment (25) for the FFT input array. Note that the m-index in G_{mq} corresponds to the "active" n-index in the FFT (25), and the q-index in G_{mq} corresponds to the spectator s-index in the FFT (25).

After applying the FFT from §3, we get the array $\widetilde{E}_{pq}$ for one tile, with $(2M)$ p-indices and $(32/M_{\rm pad})$ q-indices. The register assignment for $\widetilde{E}_{pq}$ is given by the FFT output assignment given previously in Eq. (26). We first split the index $0 \le p < (2M)$ into "high" and "low" parts:

$$p = 8p_{\rm hi} + p_{\rm lo} \qquad \text{where } 0 \le p_{\rm hi} < (M/4) \text{ and } 0 \le p_{\rm lo} < 8 \tag{125}$$

Then the $\widetilde{E}_{pq}$ register assignment is:

$$\left[\left(2M_{\rm pad} \times \frac{32}{M_{\rm pad}}\right) \widetilde{E}_{pq}\right] \qquad b_0 \leftrightarrow p_0^{\rm lo} \qquad r \leftrightarrow \text{ReIm} \qquad t_0t_1t_2t_3t_4 \leftrightarrow p_1^{\rm lo}p_2^{\rm lo}\ \underbrace{p_0^{\rm hi}\cdots p_{\mu-3}^{\rm hi}\ q_0\cdots q_{4-\mu}}_{\text{3 bits}} \tag{126}$$

where the p-index has been zero padded from length-$(2M)$ to length-$(2M_{\rm pad})$, by extending the index $p_{\rm hi}$ from length-$(M/4)$ to length-$(M_{\rm pad}/4)$.

Accumulating the I-array. After computing $\widetilde{E}_{pq}$, we accumulate its contribution to

$$I_{pq} = \sum_{T_{\rm ds}\text{ times}} \sum_{2\text{ pols}} \big|\widetilde{E}_{pq}\big|^2 \tag{127}$$

For each tile, we store one `__half2` I-array register/thread, with register assignment:

$$\left[\left(2M_{\rm pad} \times \frac{32}{M_{\rm pad}}\right) I_{pq}\right] \qquad b_0 \leftrightarrow p_0^{\rm lo} \qquad t_0t_1t_2t_3t_4 \leftrightarrow p_1^{\rm lo}p_2^{\rm lo}\ \underbrace{p_0^{\rm hi}\cdots p_{\mu-3}^{\rm hi}\ q_0\cdots q_{4-\mu}}_{\text{3 bits}} \tag{128}$$

This register assignment mirrors the $\widetilde{E}_{pq}$ register assignment (126) above. Therefore, we can accumulate `I += |E|^2` using two `__half2` FMAs (for the real and imaginary parts).

The total number of I-array registers per thread is:

$$\begin{aligned}
\big(I\text{-array registers/thread } R_I\big) &= \big(G\text{-array tiles per warp}\big) \\
&= \frac{\text{Total } q\text{-indices } (2N)}{(\text{Warps } W)\,(q\text{-indices per tile } (32/M_{\rm pad}))} \\
&= \frac{M_{\rm pad}N}{16W}
\end{aligned} \tag{129}$$

Note that some of the I-array registers end up accumulating zeros, due to zero-padding the p-index from length-$(2M)$ to length-$(2M_{\rm pad})$. (This only happens in case 5.) This may seem wasteful, but I don't see a more efficient approach.

4.12 Writing the I array to global memory

Every T_{ds} time samples, we trigger a cyclic counter which writes the $I_{\bar\tau pq}$ array to global memory (see pseudocode in §4.3). The details of this step will depend on what global memory layout is chosen for $I_{\bar\tau pq}$, which in turn will depend on some `cublas` experiments in the context of the *second* FRB beamforming kernel (§5). I leave it to you to work out the details, once the memory layout is chosen!

Some general discussion of issues which may arise (depending on what global memory layout is chosen):

- You may find it helpful to use shared memory when writing $I_{\bar{\tau}pq}$, in order to "transpose" the I-array so that each warp writes entire cache lines. In this case, the shared memory layout (60) generalizes to:

  ```
  union {
      int Fsh1[];
      int Fsh2[];
      struct {
          __half2 Gsh[];
          __half2 Ish[];
      };
  };
  ```
  (130)

  where the new shared memory region `Ish[]` is in a "struct with `Gsh[]`", but a "union with `Fsh1[]` and `Fsh2[]`". This saves a little shared memory, since `Fsh2[]` is larger than `Gsh[]` (Table 2 in §4.4).

- You may find that it's helpful to buffer a few $\bar{\tau}$ samples, before writing $I_{\bar{\tau}pq}$ to global memory. (Especially in case 5, where the p-index is zero-padded from length $2M$ to $2M_{\text{pad}}$, and some threads store zeros which we do not want to write to global memory.) In this case, I suggest using buffering in registers instead of shared memory, since the kernel uses so much shared memory already.

4.13 Computational cost

	Case 1	Case 2	Case 3	Case 4	Case 5	Notes
Number of dishes D	64	64	256	256	512	
Dish grid size (M, N)	(8,8)	(8,12)	(16,16)	(16,20)	(24,24)	
GPU config assumed	16×A40	16×A40	16×A40	16×A40	128×A40	
Freq channels/GPU F	2048	2048	1024	1024	256	upchannelized
Input sampling t_s (μs)	27.3	27.3	41.0	41.0	27.3	upchannelized
Downsampling factor $T_{\rm ds}$	40	40	25	25	40	
Output sampling t_f (ms)	1.09	1.09	1.02	1.02	1.09	
Global memory BW [GB/s]						
Read E-array	9.6	9.6	12.8	12.8	9.6	$2FD/t_s$
Write I-array	0.96	1.44	2.05	2.56	1.08	$8FMN/t_f$
Shared memory BW [GB/s]						
Write `Fsh1[]`	9.6	9.6	12.8	12.8	9.6	$2FD/t_s$
Read `Fsh1[]`	9.6	12.8	12.8	20.5	12.8	$128FD/(T_{\rm outer}t_s)$
Write `Fsh2[]`	9.6	19.2	12.8	25.6	14.4	$128FMN/(T_{\rm outer}t_s)$
Read `Fsh2[]`	9.6	19.2	12.8	25.6	14.4	$2FMN_{\rm pad}/t_s$
Write `Gsh[]`	76.8	154	102	205	115	$16FMN_{\rm pad}/t_s$
Read `Gsh[]`	76.8	115	102	128	115	$16FM_{\rm pad}N/t_s$
SM-cycles per FFT						
M-FFT C_M	1.5	1.5	3	3	8	From Table 1
N-FFT C_N	1.5	3	3	8	8	From Table 1
Compute [giga SM-cycles/sec]						
$E \to F$ shuffle (Eq. 131)	0.96	0.96	1.28	1.28	0.96	$0.2\,FD/t_s$
S warp shuffle (Eq. 132)	0.15	0.3	0.2	0.4	0.23	$2FMN/(T_{\rm outer}t_s)$
FFT1 unpack (Eq. 133)	0.3	0.6	0.4	0.8	0.45	$0.0625\,FMN_{\rm pad}/t_s$
FFT1 `__hcmadd` (Eq. 134)	0.45	0.90	0.60	1.2	0.68	$0.094\,FMN_{\rm pad}/t_s$
FFT1 loc trans (Eq. 135)	0.3	0.6	0.4	0.8	0.45	$0.125\,FMN_{\rm pad}/t_s$
First FFT	1.8	3.6	2.4	6.4	3.6	$2FMC_N/t_s$
Second FFT	3.6	5.4	4.8	6.0	7.2	$4FNC_M/t_s$
FFT2 FMA (Eq. 136)	0.3	0.45	0.4	0.5	0.45	$0.0625\,FM_{\rm pad}N/t_s$
Total	7.9	12.8	10.5	17.4	14.0	
Bottom-line costs, as percentage of total GPU resources						
Global memory BW	1.8%	1.8%	2.5%	2.6%	1.8%	600 GB/sec assumed
Shared memory BW	2.1%	3.5%	2.7%	4.5%	3.0%	
Compute	5.4%	8.8%	7.2%	11.9%	9.6%	
Total	9.2%	14.1%	12.4%	18.9%	14.4%	

Table 3: Computational cost estimates for first FRB beamforming kernel. These estimates do not include the cost of the second FRB beamforming kernel (see §5). Cases 1+2 are square and rectangular versions of the CHORD pathfinder (rectangular is more likely). Cases 3+4 are square and rectangular versions of HIRAX. Case 5 is full CHORD. Note that we have assumed upchannelization by a factor 16, i.e. F and t_s are 16 times larger than their usual values.

Some notes on Table 3:

- The "$E \to F$ shuffle" was introduced in §4.6 (see Eqs. (80), (82)). According to Erik's email (subject "bit-shuffling operation for FRB beamformer") this operation takes 28 instructions, with 4 registers/thread (512 bytes total) in/out. Let's assume 50 SM-cycle throughput, since some instructions will take more than 1 clock cycle. This gives the cost estimate:

$$\left(50\,\frac{\text{SM-cycles}}{\text{shuffle}}\right)\left(512\,\frac{\text{bytes}}{\text{shuffle}}\right)^{-1}\left(\frac{2FD}{t_s}\frac{\text{bytes}}{\text{sec}}\right) = \frac{0.2\,FD}{t_s}\ \text{SM-cycles/sec} \qquad (131)$$

Note that in cases 1 and 2, there is an extra warp transpose between Eqs. (85) to (86), but we don't include it separately, since the cost is subdominant (∼4 SM-cycles versus ∼50).

- The "S warp shuffle" is the call to `__shfl_sync()` in the pseudocode (97), used to "broadcast" one value of s_d to all threads in a warp. We need one call to `__shfl_sync()` (2 SM-cycles) to write 128 bytes (one register/thread) to `Fsh2[]`. Therefore the cost is:

$$\left(2 \; \frac{\text{SM-cycles}}{\text{shuffle}}\right)\left(128 \; \frac{\text{bytes}}{\text{shuffle}}\right)^{-1}\left(\frac{128FMN}{T_{\text{outer}}t_s} \; \frac{\text{bytes}}{\text{sec}}\right) = \frac{2FMN}{T_{\text{outer}}t_s} \; \frac{\text{SM-cyles}}{\text{sec}} \tag{132}$$

- "FFT1 unpack" refers to the step where `Freg2[]` is unpacked to E_{mn}, just before the first FFT (Eqs. (113), (114)). According to a `#bx-engine-dev` conversation around March 18, a version of this unpacking operation with 4 output registers/thread takes 3.5 SM-cycles. Therefore, we will assume 1 SM-cycle per output register (32 E-array elts):

$$\left(1 \; \frac{\text{SM-cycle}}{\text{output reg}}\right)\left(32 \; \frac{E\text{-array elts}}{\text{output reg}}\right)^{-1}\left(\frac{2FMN_{\text{pad}}}{t_s} \; \frac{E\text{-arr elts}}{\text{sec}}\right) = \frac{0.0625\,FMN_{\text{pad}}}{t_s} \; \text{SM-cycles/sec} \tag{133}$$

- "FFT1 `__hcmadd`" is the (WE) multiplication in Eq. (116). According to my microbenchmark, `__hcmadd` throughput is 1.5 SM-cycles per instruction. Therefore the cost is:

$$\left(1.5 \; \frac{\text{SM-cycles}}{\texttt{hcmadd}}\right)\left(32 \; \frac{F\text{-arr elts}}{\texttt{hcmadd}}\right)^{-1}\left(\frac{2FMN_{\text{pad}}}{t_s} \; \frac{F\text{-arr elts}}{\text{sec}}\right) = \frac{0.094\,FMN_{\text{pad}}}{t_s} \; \text{SM-cycles/sec} \tag{134}$$

- "FFT1 loc trans" is the local transpose after the first FFT (Eq. (120)). According to my microbenchmark, a local transpose costs 1 SM-cycle (operating on 2 registers/thread, or 64 G_{mq}-array elements). Therefore the cost is:

$$\left(1 \; \frac{\text{SM-cycle}}{\text{loc trans}}\right)\left(64 \; \frac{G\text{-arr elts}}{\text{loc trans}}\right)^{-1}\left(\frac{4FMN_{\text{pad}}}{t_s} \; \frac{G\text{-arr elts}}{\text{sec}}\right) = \frac{0.0625\,FMN_{\text{pad}}}{t_s} \; \frac{\text{SM-cycles}}{\text{sec}} \tag{135}$$

- "FFT2 FMA" refers to the `__half2` FMA needed to accumulate $|\widetilde{E}_{pq}|^2$ into I_{pq} (Eq. (127)). The cost is:

$$\left(0.5 \; \frac{\text{SM-cycles}}{\text{FMA}}\right)\left(32 \; \frac{\widetilde{E}\text{-arr elts}}{\text{FMA}}\right)^{-1}\left(\frac{4FM_{\text{pad}}N}{t_s} \; \frac{\widetilde{E}\text{-arr elts}}{\text{sec}}\right) = \frac{0.0625\,FM_{\text{pad}}N}{t_s} \; \text{SM-cycles/sec} \tag{136}$$

4.14 Discussion

Frequency-dependent upchannelization. The FRB beamforming kernel will operate on an upchannelized version of the E-array. In several places (kernel specification in §4.1, and Table 3 in §4.13), we have implicitly assumed frequency-independent upchannelization factor $U = 16$. In fact, U will be frequency-dependent, and this may require some changes (e.g. frequency-dependent T_{ds}). I expect these changes to be minor, so let's postpone until after we've written an upchannelization kernel, when we'll have a better sense for what specific changes to make.

Register arrays and unrolled loops. In this design document, we have proposed storing several arrays in registers (`Freg2`, W, I) and unrolling long loops. We've been finding (in the context of the baseband beamformer) that this can lead to issues such as instruction cache misses, or mysteriously large register counts. I'm worried that these issues will be worse in the FRB beamformer, but let's see how it goes!

16-bit overflows. This may be an issue, both in testing and in production. I'm just leaving this note so we don't forget about it!

Precomputing shared memory offsets in persistent registers. I'm a big fan of doing this, since it only uses one register, and often speeds up kernels! I proposed doing this in three places (at the end of §4.9, §4.10, §4.11) but it may be useful elsewhere.

Shared memory bank conflicts. I think I checked carefully that the first FRB beamforming kernel has no bank conflicts, but I'm worried about a bug in the math. Can the profiler check for bank conflicts at runtime? If not, then I'd suggest adding a runtime check. I think something like this should work (haven't actually tried it):

```
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
}
```

5 Second FRB beamforming "kernel"

5.1 Introduction

The second FRB beamforming kernel computes the beamformed intensities

$$J_{f\tau\beta} = \sum_{pq} U_p^M(\theta_{f\beta})\, U_q^N(\theta'_{f\beta})\, I_{f\tau pq} \tag{137}$$

with notation as follows:

- Indices $0 \le p < 2M$ and $0 \le q < 2N$ index half-integer beam locations, as in §4.
- Index $0 \le f < F$ indexes a frequency channel.
- Index $0 \le \tau < N_{\rm time}$ indexes a "slow" time sample (after downsampling by a factor $T_{\rm ds}$ in the first kernel). In this section, all time samples are "slow", so we do not use the τ, $\bar\tau$ notation from §4.
- Index $0 \le \beta < B$ indexes a beam, and $(\theta_{f\beta}, \theta'_{f\beta})$ denotes the sky location of beam β in frequency channel f.
- The function $U_q^N(\theta)$ was defined in Eqs. (7), (8).
- The array $I_{f\tau pq}$ has been computed and written to global memory by the first FRB beamforming kernel.

See Table 4 for baseline values of the parameters (F, M, N), and the time sampling rate t_f.

We propose using float16 for all arrays in sight $(I, J, U$, etc). You should feel free to choose whichever memory layouts are most convenient/efficient, for all of these arrays.[10]

Erik had the nice suggestion of computing Eq. (137) using `cublasHgemm`, rather than writing a custom cuda kernel. For this reason, we have titled this section with "kernel" in quotes. This makes the task much easier, but the details are still nontrivial. There are two cases to consider, depending on whether the beam locations $(\theta_{f\beta}, \theta'_{f\beta})$ are factorizable.

5.2 Case 1: non-factorizable beam locations

In this case the beam locations $(\theta_\beta, \theta'_\beta)$ are arbitrary. We define the array $W_{fpq\beta}$ by:

$$W_{fpq\beta} = U_p^M(\theta_{f\beta})\, U_q^N(\theta'_{f\beta}) \tag{138}$$

In the rest of this section, we leave the "spectator" f-index implicit, and introduce the notation $\rho = (p, q)$ to denote a (p, q) index pair by a single index ρ. Thus we will denote the I, J, W arrays as

$$I_{f\tau pq} \to I_{\tau\rho} \qquad J_{f\tau\beta} \to J_{\tau\beta} \qquad W_{fpq\beta} \to W_{\rho\beta} \tag{139}$$

Using the above notation, we can write Eq. (137) as a matrix multiply:

$$J_{\tau\beta} = \sum_\rho I_{\tau\rho} W_{\rho\beta} \tag{140}$$

To do this matrix multiply with `cublasHgemm`, we need to store the array $W_{\rho\beta}$ in GPU global memory. For now, let's assume that $W_{\rho\beta}$ has been precomputed and stored persistently in GPU global memory, between launches of the FRB beamforming kernel. (See §5.4 for discussion of how W gets computed.)

I'll leave it to you to experiment to find the optimal way of doing the matrix multiply (140) with `cublasHgemm`, including details such as: what are the best global memory layouts for the I, J, W arrays? Should we launch all calls to `cublasHgemm` on the same stream (one call for each frequency channel), or use multiple streams for more parallelism?

[10] Note that the J-array will eventually be quantized to either 4 or 8 bits, and converted to the FRB packet format for network transmission, but for now let's just write the J-array to global memory in `float16`. We can do the quantization and packet-formatting in a separate GPU kernel later.

5.3 Case 2: factorizable beam locations

In the factorizable case, the beam index β can be reinterpreted as an index pair $\beta = (\gamma, \gamma')$. The first beam coordinate $\theta_{f\beta}$ only depends on the first index γ, and the second beam coordinate $\theta'_{f\beta}$ only depends on the second index γ'. Thus we can write Eq. (137) as (with implicit frequency index f):

$$J_{\tau\gamma\gamma'} = \sum_{pq} U_p^M(\theta_\gamma)\, U_q^N(\theta'_{\gamma'})\, I_{\tau pq} \tag{141}$$

Or, introducing the slightly more compact notation $U_{p\gamma} = U_p^M(\theta_{f\gamma})$ and $U'_{q\gamma'} = U_q^N(\theta'_{f\gamma'})$, as:

$$J_{\tau\gamma\gamma'} = \sum_{pq'} U_{p\gamma}\, I_{\tau pq}\, U'_{q\gamma'} \tag{142}$$

In this form, we see that we need to multiply three matrices ($U^T I U'$) for each time index τ (and for each frequency index f).

Here's an idea for avoiding a separate call to `cublasHgemm` for each time index τ. First, reinterpret the 3-d array $I_{\tau pq}$ as a 2-d array I_{pa}, where $a = (\tau, q)$ is an index pair. Then do the following matrix multiply with a single `cublasHgemm` call:

$$V_{\gamma a} = \sum_p U_{p\gamma} I_{pa} \tag{143}$$

Next, reinterpret the 2-d array $V_{\gamma a}$ as a 3-d array $V_{\gamma\tau q}$, and then as a 2-d array V_{bq}, where $b = (\gamma, \tau)$ is an index pair. Then do the following matrix multiply with a single `cublasHgemm` call:

$$J_{b\gamma'} = \sum_q V_{bq} U_{q\gamma'} \tag{144}$$

Then reinterpret the 2-d array $J_{b\gamma'}$ as a 3-d array $J_{\tau\gamma\gamma'}$. Note that making this idea work with `cublas` puts a lot of constraints on array memory layouts!

I'll leave it to you to experiment to find the optimal way of doing the matrix multiply (142) with `cublasHgemm`, including details such as: what are the best global memory layouts for the I, J, U, U' arrays? Should we launch all calls to `cublasHgemm` on the same stream (one call for each frequency channel), or use multiple streams for more parallelism?

5.4 Computational cost and discussion

Details of Table 4. Computational cost estimates for the second FRB kernel are shown in Table 4. Note that the quantities (F, M, N, t_f, t_k) are defined in the top few rows of the table. In the next few paragraphs, we explain how memory bandwidth and computational cost are estimated.

In the non-factorizable case, global memory bandwidth and compute requirements are calculated as follows.

$$\text{Global memory BW (non-factorizable case, bytes/s)} = \underbrace{\left(\frac{8FMN}{t_f}\right)}_{\text{Read } I} + 2\underbrace{\left(\frac{8FBMN}{t_k}\right)}_{\text{Read } W} + \underbrace{\left(\frac{2FB}{t_f}\right)}_{\text{Write } J} \tag{145}$$

$$\text{Flops (tensor float16, non-factorizable case)} = \frac{8FMN}{t_f} \tag{146}$$

In the factorizable case, we do two matrix multiplies (Eqs. (143), (144)) for each frequency f and time τ:

$$V = U^T I \qquad\qquad J = VU' \tag{147}$$

	Case 1	Case 2	Case 3	Case 4	Case 5	Notes
Dish grid size (M, N)	(8,8)	(8,12)	(16,16)	(16,20)	(24,24)	
GPU config assumed	16×A40	16×A40	16×A40	16×A40	128×A40	
Freq channels/GPU F	2048	2048	1024	1024	256	upchannelized
Time sampling t_f (ms)	1.09	1.09	1.02	1.02	1.09	downsampled
Non-factorizable beams (§5.2)						
Number of beams B	640	960	2560	3200	5760	$10MN$ assumed
Kernel cadence t_k (sec)	1	1	1	1	1	see discussion
I-array size (GB)	0.96	1.44	2.05	2.56	1.08	$8FMNt_k/t_f$
W-array size (GB)	0.67	1.5	5.4	8.4	6.8	$8FBMN$ bytes
U-array size (GB)	0.08	0.16	0.34	0.47	0.28	$4FB(M+N)$
Cost (% GPU resources)						
$I+J$ global mem BW	0.6%	0.8%	1.2%	1.5%	0.6%	See Eq. (145)
W global mem BW	0.1%	0.3%	0.9%	1.4%	1.1%	See Eq. (145)
Compute	0.4%	0.9%	3.5%	5.5%	4.1%	See Eq. (146)
Total	1.1%	2.0%	5.6%	8.4%	5.9%	
Factorizable beams (§5.3)						
Beam grid size (B_x, B_y)	(32,32)	(32,48)	(64,64)	(64,80)	(96,96)	$(4M, 4N)$
U-array size (GB)	0.004	0.007	0.008	0.011	0.005	$4F(B_xM + B_yN)$
Cost (% GPU resources)						
$I+J$ global mem BW	0.8%	1.2%	1.7%	2.1%	0.9%	See Eq. (148)
V global mem BW	0.6%	1.0%	1.4%	1.7%	0.7%	See Eq. (148)
Compute	0.1%	0.1%	0.3%	0.4%	0.2%	See Eq. (149)
Total	1.5%	2.3%	3.3%	4.2%	1.8%	

Table 4: Computational cost estimates for second FRB beamforming kernel. These estimates do not include the cost of the first FRB beamforming kernel (see §4). Cases 1+2 are square and rectangular versions of the CHORD pathfinder (rectangular is more likely). Cases 3+4 are square and rectangular versions of HIRAX. Case 5 is full CHORD. The beam counts (B in the non-factorizable case, and (B_x, B_y) in the factorizable case) are rough guesses, to be refined later with detailed survey optimization.

Global memory bandwidth and compute requirements are calculated as follows:

$$\text{Global memory BW (factorizable case, bytes/s)} = \underbrace{\left(\frac{8FMN}{t_f}\right)}_{\text{Read } I} + \underbrace{2\left(\frac{4FB_xN}{t_f}\right)}_{\text{Read/write } V} + \underbrace{\left(\frac{2FB_xB_y}{t_f}\right)}_{\text{Write } J} \tag{148}$$

$$\text{Flops (tensor float16, factorizable case)} = \underbrace{\left(\frac{8FB_xMN}{t_f}\right)}_{V=U^TI} + \underbrace{\left(\frac{4FB_xB_yN}{t_f}\right)}_{J=VU'} \tag{149}$$

In Eq. (148), we have neglected the global memory bandwidth required to read U, U', which should be small.

Kernel cadence t_k (non-factorizable case only). In the non-factorizable case, the kernel cadence t_k (time between `cublas` calls) matters, since it determines the W memory bandwidth in Eq. (145). The choice of t_k is a tradeoff: if t_k is too small, then the kernel will be inefficient since the W memory bandwidth will be large (this could be fixed by writing a custom kernel, see below). If t_k is too large, then more GPU memory will be needed to buffer the I-array (see next paragraph), and the triggering latency of the FRB search will increase. In Table 4, we have assumed $t_k = 1$ sec as a rough guess.

I expect that we'll end up using a t_k which is larger than the "natural" E-array chunk size used by most of the GPU kernels. For concreteness, suppose that $t_k = 1$ sec, whereas most of the kernels process E-array data in 125 ms chunks. Then we would still run the first FRB beamforming kernel at 125 ms cadence, and buffer eight I-array chunks before launching the second FRB beamforming kernel. The I-array buffer size is shown in Table 4 ("I-array size"). We might end up needing to multiply this buffer size by 2, depending on how buffers are managed in kotekan.

I asked Andre whether it would be a problem to run the first and second FRB beamforming kernels at different cadence. He didn't see a fundamental problem with this, but we had some discussion about how best to implement it. (See `#bx-engine-dev` thread around 13 Oct 2022. The issues raised on this thread shouldn't affect writing the GPU kernel itself, only integration with kotekan.)

We emphasize that in the non-factorizable case, the kernel cadence t_k cadence shouldn't matter much, and should be able to use kotekan's "natural" cadence.

Should we write a GPU kernel to compute W from U? (Non-factorizable case only.) In the non-factorizable case (§5.2), the W-array (138) must be stored in GPU global memory. How does the W-array get computed? There are two options: either we could compute W on the CPU and copy to the GPU, or write a GPU kernel which computes the W-array from the U-array using Eq. (138). Here are some thoughts on the tradeoffs:

- If we decide to use tracking beams, then we need to compute W on the GPU, since the W-matrix would get recomputed every $t_k \sim 1$ sec, and copying from the CPU to GPU would use too much PCIe bandwidth.
- In the non-tracking case (where W doesn't change), we might decide that the W-array is too large to store persistently on the GPU (see "W-array size" in Table 4). In this case we'd loop over frequency channels (or "tiles" consisting of a few frequency channels), and compute W for each channel on-the-fly, before running the second FRB beamforming kernel. Then we'd need to compute the W-array on the GPU, to avoid using too much PCIe bandwidth.

  Note that in either this scenario, or the one from the previous bullet point, the W-array will be written to global memory in every kernel iteration. The cost of this extra step is estimated in Table 4 ("W global mem BW" row).
- If neither of the conditions in the previous two bullet points apply, then I think both options (computing W on the CPU or GPU) are viable.

Should we write custom kernels, instead of using cublas? Here are my thoughts on what might be gained by writing custom kernels:

- In the non-factorizable case, I think that a custom kernel could avoid storing the W-array in global memory (only the U-array). Instead, the kernel would hold U-array elements in registers, and compute W-array elements "on the fly" when needed, using Eq. (138). I started designing a kernel along these lines (before Erik pointed out that we could just use cublas), and it looked like it would work, but it was pretty complicated.

  Such a kernel would have the following advantages: it would eliminate the large W-array memory footprint, avoid the memory bandwidth of reading the W-array, and eliminate the need for a t_k which is larger than kotekan's "natural" cadence (further reducing GPU memory usage, and also reducing triggering latency of the FRB search).

  My feeling is that these advantages aren't essential at first, but may be worth writing a custom kernel eventually. So I propose that we start with a cublas-based approach, and (if we settle on using non-factorizable beams, see discussion below) implement a custom kernel at some point in the future, when we have time.
- In the factorizable case, I think that a custom kernel could avoid reading/writing the intermediate V-array, by coalescing the matrix multiplication $J = U^T I U'$ into a single kernel. The cost of reading/writing V is estimated in Table 4 ("V global mem BW" row), and is pretty small (around 1% of total GPU resources). So my feeling is that a custom kernel is a low priority in the factorizable case.
- This analysis assumes that there are no cublas performance surprises, and that a cublas implementation gets close to the cost estimates in Table 4. If we do get performance surprises, we may revisit this analysis and decide to write a custom kernel.

Should we use factorizable or non-factorizable beams? Throughout this section, we've been agnostic on the question of whether we'd use factorizable or non-factorizable beams in the CHORD FRB search. For now, I think we should implement both options, to give ourselves the flexibility to decide later in the project. The decision will depend on whether we decide to use tracking beams, and whether constraining beams to be factorizable increases the cost of the FRB search.

It's also possible that we'll end up running two versions of the FRB beamformer, one to feed the FRB search (as the name suggests!), and one to feed a pulsar search. In this case, the pulsar search would use non-factorizable (tracking) beams, whereas the FRB search might use either factorizable or non-factorizable beams. We'd implement this by running one instance of the first beamforming kernel (§4), and two instances of the second kernel (§5).

References

[1] K. W. Masui *et al.*, Algorithms for fft beamforming radio interferometers, arxiv:1710.08591.

A Proposed dish layout

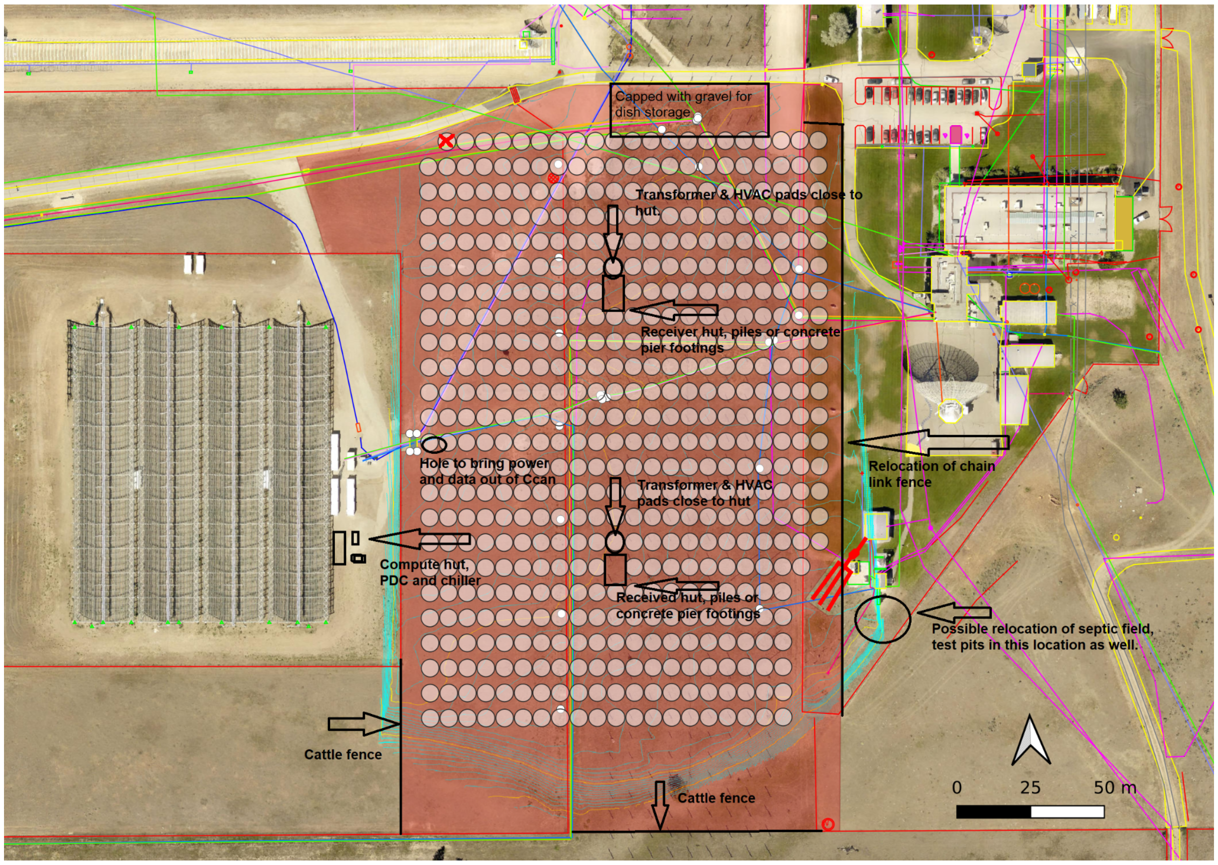
Capped with gravel for dish storage
Transformer & HVAC pads close to hut.
Receiver hut, piles or concrete pier footings
Hole to bring power and data out of Ccan
Transformer & HVAC pads close to hut
Compute hut, PDC and chiller
Received hut, piles or concrete pier footings
Relocation of chain link fence
Possible relocation of septic field, test pits in this location as well.
Cattle fence
Cattle fence
0
25
50 m

Figure 2: Proposed CHORD dish layout. The layout is a subset of a 22-by-24 regular grid. Dish spacing is 6.3m East-West and 8.5m North-South. Provisional and subject to change in the future!

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:

$$[(16 \times 16)\ \text{float16}\ A_{ij}] \qquad b_0 \leftrightarrow j_0 \qquad r_0 r_1 \leftrightarrow i_3 j_3 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow j_1 j_2 i_0 i_1 i_2 \tag{150}$$

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 (150) 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:

$$b_1 b_0 \leftrightarrow XY \qquad r \leftrightarrow Z \tag{151}$$

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:

$$b_1 b_0 \leftrightarrow XZ \qquad r \leftrightarrow Y \tag{152}$$

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:

$$b_1 b_0 \leftrightarrow ZY \qquad r \leftrightarrow X \tag{153}$$

Either of the local transpose operations defined in Eqs. (152), (153) 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 §4.6, 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:

$$r \leftrightarrow X \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow Y_4 Y_3 Y_2 Y_1 Y_0 \tag{154}$$

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:

$$r \leftrightarrow Y_i \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow Y_4 \cdots \underbrace{X}_{\text{replacing } Y_i} \cdots Y_0 \tag{155}$$

This can be done efficiently with one warp shuffle instruction as follows:[11]

```
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.

[11]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 3), 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 $C = AB$:

$$[(16\times 8)\ \text{float16}\ A_{ij}] \qquad b_0 \leftrightarrow j_0 \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2i_0i_1i_2 \tag{156}$$
$$[(8\times 8)\ \text{float16}\ B_{jk}] \qquad b_0 \leftrightarrow j_0 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2k_0k_1k_2 \tag{157}$$
$$[(16\times 8)\ \text{float16}\ C_{ik}] \qquad b_0 \leftrightarrow k_0 \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow k_1k_2i_0i_1i_2 \tag{158}$$

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 $C = AB$:

$$[(16\times 16)\ \text{float16}\ A_{ij}] \qquad b_0 \leftrightarrow j_0 \qquad r_0r_1 \leftrightarrow i_3j_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2i_0i_1i_2 \tag{159}$$
$$[(16\times 8)\ \text{float16}\ B_{jk}] \qquad b_0 \leftrightarrow j_0 \qquad r_0 \leftrightarrow j_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2k_0k_1k_2 \tag{160}$$
$$[(16\times 8)\ \text{float16}\ C_{ik}] \qquad b_0 \leftrightarrow k_0 \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow k_1k_2i_0i_1i_2 \tag{161}$$

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 3. 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).

$$j = 4j' + E \qquad \text{where } 0 \le j' < 4 \text{ and } 0 \le E < 4 \tag{162}$$

Each 1×4 submatrix in A (Figure 3, 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 3. Formally, the reparameterization is described by the equation:

```
Asp[i,j',J] = A[i, 4j'+E[i,j',J]]                                    (163)
```

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

$$[(16\times 4\times 2)\ \text{float16}\ A^{\text{sp}}_{ij'J}] \qquad b_0 \leftrightarrow J \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_2j_3i_0i_1i_2 \tag{164}$$
$$[(16\times 8)\ \text{float16}\ B_{jk}] \qquad b_0 \leftrightarrow j_0 \qquad r_0 \leftrightarrow j_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2k_0k_1k_2 \tag{165}$$
$$[(16\times 8)\ \text{float16}\ C_{ik}] \qquad b_0 \leftrightarrow k_0 \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow k_1k_2i_0i_1i_2 \tag{166}$$

Note that, in the register assignment (164) for $A^{\rm sp}_{ij'J}$, we denoted the j' index bits by j_2j_3, rather than $j'_0j'_1$. The two are equivalent by Eq. (162). We also note that the B and C sparse register assignments (165), (166) are the same as their dense counterparts (160), (161).

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 $t_2t_3t_4$:

$$[(16 \times 4 \times 2) \text{ int2 } E_{ij'J}] \qquad b_0b_1b_2b_3 \leftrightarrow Jj_2j_3i_3 \qquad t_2t_3t_4 \leftrightarrow i_0i_1i_2 \tag{167}$$

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_0t_1. Note that in Eq. (167), the j' index bits are denoted j_2j_3, as in (164) above.

A special case which will arise in §3.5. We will be interested in a special case where the 16-by-16 matrix A_{ij} is 50% sparse because it contains a Kronecker delta of the form $\delta_{i_nj_1}$:

$$A_{ij} = A'_{ij_3j_2j_0}\delta_{j_1i_n} \qquad \text{where } 0 \le n < 4 \tag{168}$$

Here, we are using the usual index bit notation $i = [i_3i_2i_1i_0]_2$ and $j = [j_3j_2j_1j_0]_2$. Let's work out the details of the reparameterization $A \to (A^{\rm sp}, E)$, and the register assignments for $(A^{\rm sp}, E)$, in this special case.

First, we leave it to the reader to show that the shape-(16,4,2) arrays $A^{\rm sp}_{ij'J}$ and $E_{ij'J}$ are given by

$$A^{\rm sp}_{ij'J} = A'_{ij'_1j'_0J} \qquad E_{ij'J} = 2i_n + J \tag{169}$$

It follows that the register assignment (164) for$A^{\rm sp}$ is equivalent to the following register assignment for A' (two registers/thread):

$$[\text{ float16 } A'] \qquad b_0 \leftrightarrow j_0 \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_2j_3i_0i_1i_2 \tag{170}$$

Next we consider the E operand. As defined in (169), the E-array depends on the value of $0 \le n < 4$. We define a "universal" E-array by including n as an additional index:

$$E^{\rm uni}_{ij'Jn} = 2i_n + J \qquad \text{where } i = [i_3i_2i_1i_0]_2 \tag{171}$$

with shape $(16, 4, 2, 4)$. We distribute $E^{\rm uni}$ across threads in a warp using register assignment (one register/thread):

$$[(16 \times 4 \times 2 \times 4) \text{ int2 } E^{\rm uni}_{ij'Jn}] \qquad b_0b_1b_2b_3 \leftrightarrow Jj_2j_3i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow n_0n_1i_0i_1i_2 \tag{172}$$

This register assignment has been chosen so that one can select a value of n by setting the f-operand of the sparse MMA to $f = n$, and setting the E-operand to the register containing $E^{\rm uni}$. Then the sparse MMA will use the appropriate E-matrix for the given value of n, in the correct register assignment (167).

Finally, we note that $E^{\rm uni}$ can be initialized with the following code:

```
int n = threadIdx.x & 3;
int ilo = ((threadIdx.x >> 2) & 7); // value of i, for t=threadIdx.x and b3=0
int ihi = ilo | 8;                  // value of i, for t=threadIdx.x and b3=1
unsigned int Elo = (ilo & (1<<n)) ? 0x0000eeee : 0x00004444;   // low 16 bits
unsigned int Ehi = (ihi & (1<<n)) ? 0xeeee0000 : 0x44440000;  // high 16 bits
unsigned int Euni = Elo | Ehi;      // one register per thread
```
(173)

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 (166) 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)
    );
}
```

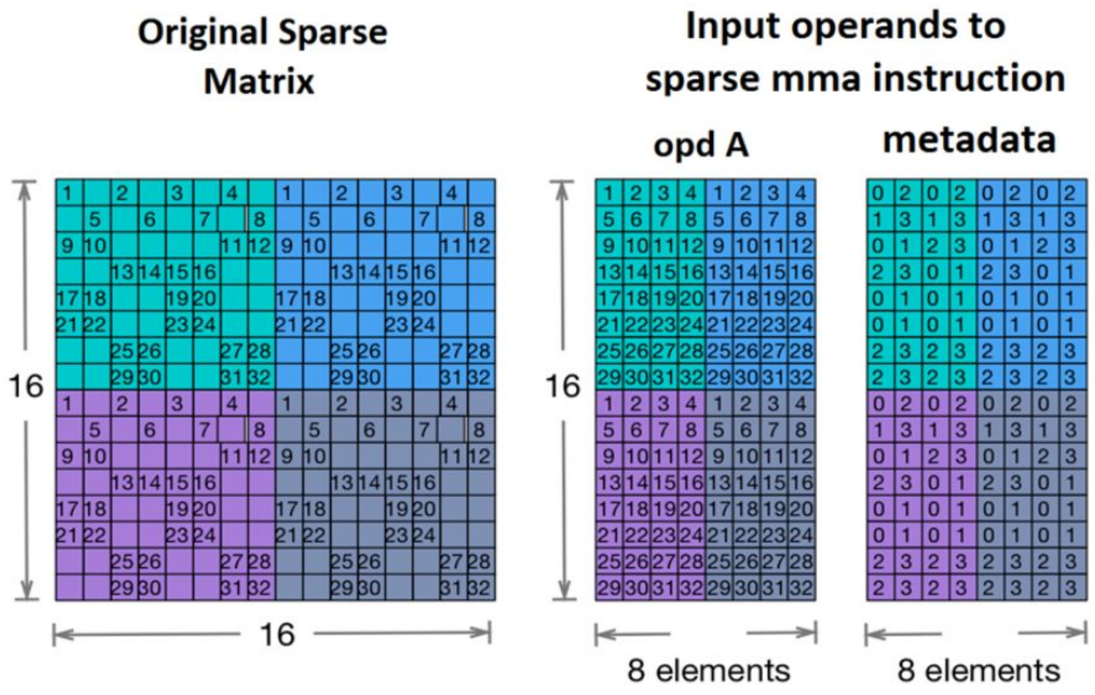

Figure 3: This figure from the nvidia documentation depicts the sparse MMA in §C.3. The 16×16 A-matrix (left) has the property that each 1×4 submatrix is 50% sparse. We split this into two smaller arrays (right): the $A^{\rm sp}$ array which contains nonzero elements of A, and the E array which describes the location of each nonzero element within its 1×4 submatrix. The sparse MMA instruction uses $(A^{\rm sp}, E)$ 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.)