Note on CHIME (not CHORD) FRB beamforming

Kendrick Smith

July 15, 2024

1 First FFT ($256 \to 512$)

We want to compute the following FFT:

$$Y_q = \sum_n \underbrace{e^{2\pi i qn/512}}_{\equiv E(q,n)} X_n \qquad 0 \le q < 512 \quad 0 \le n < 256 \tag{1}$$

The output index $0 \le q < 512$ has 9 bits, and the input index 0 has 8 bits:

$$q = q_8 q_7 q_6 q_5 q_4 q_3 q_2 q_1 q_0 \qquad\qquad n = n_7 n_6 n_5 n_4 n_3 n_2 n_1 n_0 \tag{2}$$

In this section, we'll present an in-register "microkernel" with the following register assignments for the input/output arrays:

$$[X] \qquad b \leftrightarrow \text{ReIm} \qquad t_0t_1t_2t_3t_4 \leftrightarrow n_7n_6n_5n_4n_3 \qquad r \leftrightarrow n_2n_1n_0 \tag{3}$$

$$[Y] \qquad b \leftrightarrow \text{ReIm} \qquad t_0t_1t_2t_3t_4 \leftrightarrow q_4q_5q_6q_7q_8 \qquad r \leftrightarrow q_0q_1q_2q_3 \tag{4}$$

1.1 Overview

We can write the phase $E(q,n)$ defined in (1) in terms of index bits as follows:

$$\begin{aligned} E(q,n) &= \exp\left[\frac{2\pi i}{512}\Big(\sum_j q_j 2^j\Big)\Big(\sum_k n_k 2^k\Big)\right] \\ &= \prod_{jk} \underbrace{\exp\left[\frac{2\pi i}{512} q_j n_k 2^{j+k}\right]}_{\equiv E(q_j,n_k)} \end{aligned} \tag{5}$$

Note that $E(q_j, n_k) = 1$ if $j + k \ge 9$.

Now we define the following generalization of the $E(q_j,n_k)$ notation, which is easiest to define by example. Consider a symbol like $E(q_0q_1q_2, n_0n_1n_6n_7)$ whose first argument is a subset of the q-bits, and whose second argument is a subset of the n-bits. We define it as follows:

$$E(q_0q_1q_2, n_0n_1n_6n_7) \equiv \prod_{\substack{j\in\{0,1,2\} \\ k\in\{0,1,6,7\}}} E(q_j,n_k) \tag{6}$$

Our FFT microkernel is based on the following factorization of the phase $E(q,n)$ appearing in Eq. (1):

$$\begin{aligned} E(q,n) = E(q_6q_7q_8, n_0n_1n_2)\, E(q_3q_4q_5, n_0n_1n_2)\, E(q_3q_4q_5, n_3n_4n_5) \\ \times E(q_0q_1q_2, n_2n_3n_4n_5)\, E(q_0q_1q_2, n_0n_1n_6n_7) \end{aligned} \tag{7}$$

leading to the following sequence ($X \to Z_1 \to Z_2 \to Z_3 \to Z_4 \to Y$) of array operations:

$$Z_1[q_0q_1q_2n_0n_1n_2n_3n_4n_5] = \sum_{n_6n_7} E(q_0q_1q_2, n_0n_1n_6n_7)X_n \tag{8}$$
$$Z_2[q_0q_1q_2n_0n_1n_2n_3n_4n_5] = E(q_0q_1q_2, n_2n_3n_4n_5)Z_1[q_0q_1q_2n_0n_1n_2n_3n_4n_5] \qquad \text{(no sum)} \tag{9}$$
$$Z_3[q_0q_1q_2q_3q_4q_5n_0n_1n_2] = \sum_{n_3n_4n_4} E(q_3q_4q_5, n_3n_4n_5)Z_2[q_0q_1q_2n_0n_1n_2n_3n_4n_5] \tag{10}$$
$$Z_4[q_0q_1q_2q_3q_4q_5n_0n_1n_2] = E(q_3q_4q_5, n_0n_1n_2)Z_3[q_0q_1q_2q_3q_4q_5n_0n_1n_2] \qquad \text{(no sum)} \tag{11}$$
$$Y_q = \sum_{n_0n_1n_2} E(q_6q_7q_8, n_0n_1n_2)Z_4[q_0q_1q_2q_3q_4q_5n_0n_1n_2] \tag{12}$$

These computational steps can be done as follows:

- To compute $X \to Z_1$ using Eq. (8), we use `m16n8k8` MMAs (Appendix B.1).
- To compute $Z_1 \to Z_2$ using Eq. (9), we use `__half2` FMAs.
- To compute $Z_2 \to Z_3$ using Eq. (10), we use `m16n8k16` MMAs (Appendix B.2).
- To compute $Z_3 \to Z_4$ using Eq. (11), we use `__half2` FMAs.
- To compute $Z_4 \to Y$ using Eq. (12), we use `m16n8k16` MMAs (Appendix B.2).

1.2 Register assignments

The register assignments for the X, Y, Z arrays are:

$[X]$	$b \leftrightarrow \mathrm{ReIm}$	$t_0t_1t_2t_3t_4 \leftrightarrow n_7n_6n_5n_4n_3$	$r \leftrightarrow n_2n_1n_0$	(13)
$[Z_1, Z_2]$	$b \leftrightarrow n_5$	$t_0t_1t_2t_3t_4 \leftrightarrow n_4n_3q_0q_1q_2$	$r \leftrightarrow n_2n_1n_0\mathrm{ReIm}$	(14)
$[Z_3']$	$b \leftrightarrow q_0$	$t_0t_1t_2t_3t_4 \leftrightarrow q_1q_2q_3q_4q_5$	$r \leftrightarrow n_2n_1n_0\mathrm{ReIm}$	(15)
$[Z_3, Z_4]$	$b \leftrightarrow n_2$	$t_0t_1t_2t_3t_4 \leftrightarrow n_1n_0q_3q_4q_5$	$r \leftrightarrow q_0q_1q_2\mathrm{ReIm}$	(16)
$[Y']$	$b \leftrightarrow q_3$	$t_0t_1t_2t_3t_4 \leftrightarrow q_4q_5q_6q_7q_8$	$r \leftrightarrow q_0q_1q_2\mathrm{ReIm}$	(17)
$[Y]$	$b \leftrightarrow \mathrm{ReIm}$	$t_0t_1t_2t_3t_4 \leftrightarrow q_4q_5q_6q_7q_8$	$r \leftrightarrow q_0q_1q_2q_3$	(18)

Note that for Z_3 and Y, we have given two register assignments ("primed" and "unprimed"). In both cases, we go from the primed assignment to the unprimed assignment using local/warp transpose operations (Appendix A).

For the main computational steps ($X \to Z_1 \to Z_2 \to Z_3'$) and ($Z_3 \to Z_4 \to Y'$), we need some precomputed phases in registers. We'll use the following notation for these arrays of precomputed phases (see Eqs. (8)–(12)):

$$E_{X\to Z_1} = E(q_0q_1q_2, n_0n_1n_6n_7) \tag{19}$$
$$E_{Z_1\to Z_2} = E(q_0q_1q_2, n_2n_3n_4n_5) \tag{20}$$
$$E_{Z_2\to Z_3'} = E(q_3q_4q_5, n_3n_4n_5) \tag{21}$$
$$E_{Z_3\to Z_4} = E(q_3q_4q_5, n_0n_1n_2) \tag{22}$$
$$E_{Z_4\to Y'} = E(q_6q_7q_8, n_0n_1n_2) \tag{23}$$

The register assignments for the E-arrays are:

$[E_{X\to Z_1}]$	$b \leftrightarrow \mathrm{ReIm}_{\mathrm{in}}$	$t_0t_1t_2t_3t_4 \leftrightarrow n_7n_6q_0q_1q_2$	$r \leftrightarrow n_0n_1\mathrm{ReIm}_{\mathrm{out}}$	(24)
$[E_{Z_1\to Z_2}]$	$b \leftrightarrow n_5$	$t_0t_1t_2t_3t_4 \leftrightarrow n_4n_3q_0q_1q_2$	$r \leftrightarrow n_2\mathrm{ReIm}$	(25)
$[E_{Z_2\to Z_3'}]$	$b \leftrightarrow n_5$	$t_0t_1t_2t_3t_4 \leftrightarrow n_4n_3q_3q_4q_5$	$r \leftrightarrow \mathrm{ReIm}$	(26)
$[E_{Z_3\to Z_4}]$	$b \leftrightarrow n_2$	$t_0t_1t_2t_3t_4 \leftrightarrow n_1n_0q_3q_4q_5$	$r \leftrightarrow \mathrm{ReIm}$	(27)
$[E_{Z_4\to Y'}]$	$b \leftrightarrow n_2$	$t_0t_1t_2t_3t_4 \leftrightarrow n_1n_0q_6q_7q_8$	$r \leftrightarrow \mathrm{ReIm}$	(28)

I leave it to you to verify that the register assignments in Eqs. (13)–(28) are **consistent**, in the sense that no additional transposing is needed to do the FMAs and MMAs listed at the end of §1.1. (Achieving this consistency is the most difficult part of designing the FFT microkernel. I think I got it right, but let me know if something seems wrong!)

From the E-array register assignments in Eqs. (24)–(28), it may appear that $8+4+2+2+2 = 18$ registers are needed to store the E-arrays. However, the register footprint may be reduced as follows:

- For the $E_{X\to Z_1}$ array (8 registers/thread), it's possible to store the array in the following "packed" form (4 registers/thread):

$$[E_{X\to Z_1}] \qquad b \leftrightarrow \text{ReIm} \qquad t_0t_1t_2t_3t_4 \leftrightarrow n_7n_6q_0q_1q_2 \qquad r \leftrightarrow n_0n_1 \tag{29}$$

  and "unpack" on the fly with a few `__byte_perm` instructions and sign flips. (Let me know if this description is too cryptic and I can give more detail!)

- The $E_{Z_3\to Z_4}$ array (2 registers/thread) has the same register content as the $n_2 = 0$ subset of the $E_{Z_1\to Z_2}$ array (4 registers/thread). Therefore, no additional registers are needed to store $E_{Z_3\to Z_4}$.

- The $E_{Z_4\to Y'}$ array (2 registers/thread) has the same register content as the $E_{Z_2\to Z_3'}$ array (2 registers/thread). Therefore, no additional registers are needed to store $E_{Z_4\to Y'}$.

Using these observations, the E-array footprint can be reduced to $4+4+2 = 10$ registers/thread.

1.3 Flop-counting

The FFT microkernel uses:

- 8 `m16n8k8` MMAs (2 SM-cycles each).
- 16 `m16n8k16` MMAs (4 SM-cycles each).
- 64 `__half2` FMAs (0.5 SM-cycles each).
- 16 `__shfl_sync` instructions (2 SM-cycles each).
- 32 `__byte_perm` instructions (0.5 SM-cyles each).

The total cost is:

$$(8\cdot 2) + (16\cdot 4) + (64\cdot 0.5) + (16\cdot 2) + (32\cdot 0.5) = 160 \ \frac{\text{SM-cycles}}{\text{FFT}} \tag{30}$$

The real-time FFT throughput (per GPU) is:

$$\frac{(4 \text{ cylinders})\cdot(2 \text{ polarizations})\cdot(16 \text{ freqs})}{2.56\ \mu\text{s}} = 5\times 10^7 \ \frac{\text{FFTs}}{\text{sec}} \tag{31}$$

The L40S compute envelope is:

$$(142 \text{ SMs})\cdot(2100 \text{ MHz}) = 3\times 10^{11} \ \frac{\text{SM-cycles}}{\text{sec}} \tag{32}$$

Combining these estimates, the "GPU load" (defined as fraction of GPU compute needed to run at real-time) is:

$$\text{GPU load} = \frac{(160 \text{ SM-cycles/FFT})\cdot(5\times 10^7 \text{ FFTs/sec})}{(3\times 10^{11} \text{ SM-cycles/sec})} = 0.0267 \tag{33}$$

That is, we predict that **2.67% of the GPU is needed**. (This is not the cost of the entire FRB beamforming kernel – just the first FFT!)

A Local/warp transposes

This appendix is cut-and-paste from the CHORD FRB beamformer doc, but included to make these notes self-contained.

A.1 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:

$$b_1 b_0 \leftrightarrow XY \qquad r \leftrightarrow Z \tag{34}$$

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:

$$b_1 b_0 \leftrightarrow XZ \qquad r \leftrightarrow Y \tag{35}$$

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:

$$b_1 b_0 \leftrightarrow ZY \qquad r \leftrightarrow X \tag{36}$$

Either of the local transpose operations defined in Eqs. (35), (36) 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).

A.2 Warp transpose operation

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

$$r \leftrightarrow X \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow Y_4 Y_3 Y_2 Y_1 Y_0 \tag{37}$$

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:

$$r \leftrightarrow Y_i \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow Y_4 \cdots \underbrace{X}_{\text{replacing } Y_i} \cdots Y_0 \tag{38}$$

This can be done efficiently with one warp shuffle instruction as follows:[1]

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

int i = ...; // thread index bit 0 <= i < 5
int bit = 1 << i;
```

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

```
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.

B Float16 tensor core reference

This appendix is cut-and-paste from the CHORD FRB beamformer doc, but included to make these notes self-contained.

B.1 Float16 m16n8k8

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

$$[(16 \times 8) \text{ float16 } A_{ij}] \qquad b_0 \leftrightarrow j_0 \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2i_0i_1i_2 \tag{39}$$
$$[(8 \times 8) \text{ float16 } B_{jk}] \qquad b_0 \leftrightarrow j_0 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2k_0k_1k_2 \tag{40}$$
$$[(16 \times 8) \text{ float16 } C_{ik}] \qquad b_0 \leftrightarrow k_0 \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow k_1k_2i_0i_1i_2 \tag{41}$$

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

B.2 Float16 m16n8k16

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

$$[(16 \times 16) \text{ float16 } A_{ij}] \qquad b_0 \leftrightarrow j_0 \qquad r_0r_1 \leftrightarrow i_3j_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2i_0i_1i_2 \tag{42}$$
$$[(16 \times 8) \text{ float16 } B_{jk}] \qquad b_0 \leftrightarrow j_0 \qquad r_0 \leftrightarrow j_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow j_1j_2k_0k_1k_2 \tag{43}$$
$$[(16 \times 8) \text{ float16 } C_{ik}] \qquad b_0 \leftrightarrow k_0 \qquad r_0 \leftrightarrow i_3 \qquad t_0t_1t_2t_3t_4 \leftrightarrow k_1k_2i_0i_1i_2 \tag{44}$$

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