CHORD 8-bit baseband beamformer

kmsmith

December 6, 2022

1 Kernel specification

Inputs:

- The int4+4 electric field array $E_{\tau f p d}$, where τ is a time index, $0 \le f < 16$ indexes a frequency channel, $0 \le p < 2$ indexes a polarization, and $0 \le d < 512$ indexes a dish.

  Note that we use τ (not t) to index a time sample, and β (not b) to index a beam. This is because in our register assignment notation (see Appendix A), we use t to index a cuda thread, and b to index a byte within a 32-bit register.

- A precomputed int8+8 beamforming matrix $A_{p\beta d}$, where $0 \le \beta < B$ indexes a formed beam.

- An integer "bit shift" parameter $s_{pf\beta}$ which plays the role of a gain (see Eq. (1) below).

Output:

- The int4+4 beamformed electric field[1]

$$J_{\beta f p \tau} = \text{Quantize}_{4+4}\left[\left(\sum_d A_{p\beta d} E_{\tau f p d}\right) >> s_{pf\beta}\right] \tag{1}$$

  Here, rather than multiplying the sum $(\sum_d \cdots)$ by a floating-point gain $G_{pf\beta}$ before quantizing, we have chosen to right-shift by some number of bits $s_{pf\beta}$. This was motivated by Erik's work on the 4-bit beamformer, where we learned that float→int conversion on the GPU is slow.

  Minor detail: for maximum generality, I allowed the bit-shift parameter s to depend on the beam β, but I doubt this will actually be necessary. So if the β-dependence is awkward to implement, then feel free to remove it.

Differences versus previous beamformer:

- I'm now proposing that we use 8 bits instead of 4 bits. This is because Erik's 4-bit beamformer was so fast that I think we may as well use 8 bits, and remove all uncertainty over whether quantization artifacts can be an issue.

- Different input ordering of the E-array (dish index now assumed faster varying than polarization).

- After discussion on the chord-all list, I'm now proposing that we process each polarization independently. (Minor comment: this is technically easier in the new E-array ordering since the two polarizations now live on different cache lines.)

[1]In Eq. (1), we have used a right shift $(\cdots) >> s$ to divide by 2^s before quantizing. However, I think it would be slightly better to do $((\cdots) + 2^{s-1}) >> s$, i.e. round to the nearest multiple of 2^s instead of rounding down. The same comment applies to Eqs. (2), (19), (25).

Each frequency f and polarization p is processed independently in a different threadblock. For the rest of this note, we will focus on a single threadblock, and streamline notation by omitting the f, p indices. Thus the beamforming operation (1) can be written:

$$J_{\beta\tau} = \text{Quantize}_{4+4}\left[\left(\sum_d A_{\beta d} E_{\tau d}\right) >> s_\beta\right] \tag{2}$$

The number of formed beams B isn't known yet, but will probably be in the range $20 \lesssim B \lesssim 100$, depending on how much funding we can raise for backends. At this stage, I think it makes sense to be conservative and assume a large value of B. In the rest of the notes, I've assumed $B = 96$, and I suggest that we fix this value in a first pass at implementing the kernel. I'm just mentioning in advance that the value of B may change in the future, in case this influences your design decisions.

Throughout this note we use the following notation:

$$B = 96 = \text{Number of beams} \tag{3}$$
$$D = 512 = \text{Number of dishes} \tag{4}$$
$$F = 16 = \text{Frequency channels per GPU} \tag{5}$$
$$t_s = 1.7\ \mu\text{sec} = \text{Sampling time} \tag{6}$$
$$(W_b, W_d) = \text{Dimensions of } A\text{-matrix warp tiling (TBD, see §2)} \tag{7}$$
$$T = \text{"Inner" time cadence of kernel (TBD, see §2)} \tag{8}$$

2 Overview of kernel

- We keep the A-matrix $A_{\beta d}$ in registers throughout the kernel, in an appropriate tensor core ordering (see §4). The required number of 32-bit registers is:

$$\frac{\text{Registers}}{\text{Threadblock}} = \frac{BD}{2} = 256B \tag{9}$$

If B is large, then the high register count will force us to use one threadblock per SM. If B is small, then using multiple threadblocks per SM would be preferable, since we could use fewer threads per block (while keeping reasonably high occupancy), which improves shared memory bandwidth (see next bullet point).

- We divide the B-by-D matrix $A_{\beta d}$ into $(W_b \times W_d)$ tiles, where each tile is handled by a single warp. Thus the number of warps per block is $W = W_b W_d$. We mention in advance (derived in Eq. (35)) that the shared memory bandwidth is:

$$\text{Shared memory BW} = \frac{2F \text{ bytes}}{t_s}\Big[D(W_b + 1) + 8BW_d\Big] \tag{10}$$

Thus when choosing (W_b, W_d), the basic tradeoff is between occupancy (which gets better with increasing W_b, W_d) and shared memory bandwidth (which gets worse with increasing W_b, W_d).

Here is a partial criterion for choosing (W_b, W_d). If we formally minimize shared memory bandwidth in Eq. (10) at fixed occupancy $W = (W_b W_d)$, the minimum occurs when

$$\frac{W_b}{W_d} = \frac{8B}{D} \tag{11}$$

For $B = 96$, the following possibilities look interesting to me: $(W_b, W_d) = (6, 4)$ with 768 threads/block, $(W_b, W_d) = (4, 4)$ with 512 threads/block, and $(W_b, W_d) = (4, 2)$ with 256 threads/block.[2]

[2] Although I'm advocating fixing $B = 96$ on a first pass, I'm including some footnotes with thoughts on smaller values of B, in case this influences design decisions. For small B, say $B = 16$ for concreteness, Eq. (11) suggests that we'd want to use something like $(W_b, W_d) = (1, 4)$. Note that high occupancy could still be achieved by using multiple threadblocks per SM, since by Eq. (9) we only need 4K registers/block to store the A-matrix. I'll speculate that $W_d = 4$ is a good choice for all values of B, whereas the optimal choice of W_b is proportional to B, and the optimal number of threadblocks per SM is inversely proportional to B. But this is something we should test with benchmarks.

The number of rows (beams) in each tile must be divisible by 8, and the number of columns (dishes) in each tile must be divisible by 16, for tensor core reasons described in §4. Note that the values of (W_b, W_d) suggested at the end of the previous paragraph have been chosen so that each tile contains an equal number of beams/dishes (assuming $B = 96$), while satisfying these divisibility constraints.

- The main steps in the kernel are as follows. First (§3), we transfer the E-array from global memory to shared memory. Second (§4), for each warp in the $(W_b \times W_d)$ tiling, we read a tile of the E-array, and multiply it by a tile of the A-array, to obtain a tile of the "unreduced" J-array $J^u_{r\tau\beta}$. The new index $0 \le r < W_d$ corresponds to the position of the warp in the dish tiling. Third (§5), we reduce the J^u array over the r-index, to obtain the (reduced) J-array $J_{\beta\tau}$. Fourth (§6), we write the J-array to global memory.

  These four steps have different minimum time cadences. The fourth step (writing J) operates in chunks of 128 time samples. The second and third steps (matrix multiplication + reduction) can operate in chunks as short as 8 time samples, but it is probably advisable to use larger a larger chunk size, in order to reduce calls to the expensive `__syncthreads()` operation. (I'm defining the "chunk size" of a step in the kernel to be the number of time samples between calls to `__syncthreads()`.)

  On the other hand, using large chunk sizes increases the shared memory requirement. This is particularly harmful for small B, where we want to get high occupancy by scheduling multiple threadblocks per SM, so we want to keep shared memory per threadblock small. The shared memory layouts for the beamforming kernel are specified in Eqs. (13), (21) below.

  A final consideration: in the first step (transferring E from global to shared memory) we probably want to use a large enough chunk size that all threads can participate. I'll leave it to you to balance the above considerations and decide what chunk sizes are optimal.[3]

- Internally, the kernel performs matrix multiplication using a permuted dish ordering. We will used a primed index d' to refer to this alternate ordering. Representing an unpermuted dish index as a 9-digit binary integer $d = \big[d_8 d_7 \cdots d_0\big]_2$, and a permuted dish index as $d' = d'_8 d'_7 \cdots d'_0$, the reordering operation can be taken to be:

$$d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0 \leftrightarrow d'_8 d'_7 d'_3 d'_2 d'_6 d'_5 d'_4 d'_1 d'_0 \tag{12}$$

  As described in §4, the dish reordering will allow us to avoid a shared memory bank conflict without needing local/warp transpose operations. (The part of the reordering (12) which matters is $d_6 d_5 d_1 d_0 \leftrightarrow d'_3 d'_2 d'_1 d'_0$.)

  We use the dish reordering in two places. First, the warp tiling (described in the second bullet point in this section) is defined using a primed dish index d'. (More precisely, the set of dish indices held by each warp in the tiling should consist of blocks of 16 contiguous primed indices.) Second, and closely related, the A-matrix is stored internally using a primed index $A_{\beta d'}$.

3 Step 1: transferring global memory to shared memory

In this step, we load E-array elements from global memory, and write them to shared memory. We use the following shared memory layout, for a chunk of int4+4 data with T time samples and $D = 512$ dishes:

```
__shared__ int4+4 Earr[T][512+4];    // (time, dish)                (13)
```

with 4 bytes (one int32) padding on the dish axis. We have left the time chunk size T unspecified (see discussion in §2), but T should be divisible by 4.

We describe a "building-block" operation in which a single warp processes 4 time samples and 128 dishes. A larger chunk of T time samples and D dishes can be processed by appropriately distributing the building-block operation across warps in the threadblock.

[3] For $B = 96$, where we only want one threadblock per SM and can therefore use up to 99KB shared memory, I'll speculate that a good choice is to outer-loop over blocks of 128 time samples, and inner-loop over blocks of $T = 32$ time samples. Steps 1–3 of the kernel run at 32-sample cadence, and step 4 (writing the J array) runs at 128-same cadence. For smaller values of B, we may want to use a smaller value of T.

In the building-block operation, we use a 16-byte load instruction to read 16 contiguous bytes (4 registers) on each thread.[4] If we arrange the loads so that thread $t = 8i + j$ reads time $\tau = i$ and dishes $16j \le d < 16(j+1)$, then we get the following register assignment:[5]

$$[\text{int4+4 } E_{\tau d}] \qquad b_1 b_0 \leftrightarrow d_1 d_0 \qquad r_1 r_0 \leftrightarrow d_3 d_2 \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow \tau_1 \tau_0 d_6 d_5 d_4 \tag{14}$$

This can be written to shared memory by looping over the 4 registers and issuing 4-byte stores. The stores are bank conflict free, if the shared memory layout (13) is used.

One final comment. In my N^2 kernel, I found it useful to implement prefetching to hide the latency of shared memory loads. Rather than copying directly from global memory to shared memory, I issue global memory loads (into registers) one time chunk in advance, leave the registers "untouched" for one time chunk, then issue stores to shared memory. (Unfortunately, this way of implementing prefetching "by hand" seemed to be necessary, since NVIDIA's PTX prefetch instruction didn't seem to help.[6]) It would be interesting to see if prefetching also helps in the baseband beamformer.

4 Step 2: matrix multiplication

In this step, the warps are arranged in a $W_b \times W_d$ grid, where each warp holds one tile of the A-matrix in persistent registers. We read one E-matrix tile from shared memory, and multiply it by the A-matrix tile, obtaining a tile of the unreduced J-matrix J^u which is written to shared memory.

We will describe a "building-block" operation in which a single warp processes 8 beams, 8 time samples, and 16 dishes. The larger chunk of (B/W_b) beams, T time samples, and $(512/W_d)$ dishes can be implemented by adding loops within the warp.

We read the E-tile from shared memory, in the register assignment:

$$[\text{int4+4 } E_{\tau d'}] \qquad b_0 b_1 \leftrightarrow d'_0 d'_1 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d'_2 d'_3 \tau_0 \tau_1 \tau_2 \tag{15}$$

Using shared memory layout (13), this can be done with one bank conflict free int32 load. (Note that if we had used an unprimed dish index here, then the load would have bank conflicts – this is the reason for the dish permutation in Eq. (12).)

We next unpack each int4+4 into real and imaginary parts, storing the results as int8s in separate registers. (Note that sign extension needs to be done here.) This gives the following register assignment:

$$[\text{int8 } E_{\tau d'}] \qquad b_0 b_1 \leftrightarrow d'_0 d'_1 \qquad r \leftrightarrow \text{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d'_2 d'_3 \tau_0 \tau_1 \tau_2 \tag{16}$$

We assume that at the beginning of the kernel, the A-matrix tile has been loaded into persistent registers, with register assignment:

$$[\text{int8 } A_{\beta d'}] \qquad b_0 b_1 \leftrightarrow d'_0 d'_1 \qquad r \leftrightarrow \text{ReIm} \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d'_2 d'_3 \beta_0 \beta_1 \beta_2 \tag{17}$$

These register assignments have been chosen to apply the tensor core MMA instruction described in Appendix A. Applying the MMA instruction (or rather, four MMA instructions since $A_{\beta d'}$ and $E_{\tau d'}$ are complex), we get the J^u-matrix tile:

$$[\text{int32 } J^u_{\tau\beta}] \qquad r_1 r_0 \leftrightarrow \text{ReIm}, \tau_0 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow \tau_1 \tau_2 \beta_0 \beta_1 \beta_2 \tag{18}$$

Before writing to shared memory, I propose reducing the J^u bit depth from 32 to 16. To do this, we'll need to first right-shift by a small bit count σ:

$$J^u_{\tau\beta} \leftarrow (J^u_{\tau\beta} >> \sigma) \qquad \text{where } \sigma \equiv \begin{cases} 5 & \text{if } W_d = 1 \\ 4 & \text{if } W_d = 2 \\ 3 & \text{if } W_d = 4 \end{cases} \tag{19}$$

[4]In CUDA, a 16-byte load can be issued by dereferencing an `(int4 *)` pointer. I don't know how it's done in Julia! I recently learned that 16-byte load/stores can increase global memory bandwidth by up to 30% (relative to 4-byte load/stores) so I'm trying to use them in GPU kernels from now on.

[5]I'm using this "register assignment" notation in all of my GPU kernel notes now. To make these notes self-contained, I reviewed the notation in Appendix A (mostly cut-and-paste from other notes).

[6]`https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-conversion-instructions-prefetch-prefetchu`

The value of σ has been chosen so that after right-shifting by σ, J^u is a 16-bit quantity.

After right-shifting, we pack the real and imaginary parts of the J^u array into a single int16+16 register, obtaining:

$$[\text{int16+16 } J^u_{\tau\beta}] \qquad r \leftrightarrow \tau_0 \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_1\tau_2\beta_0\beta_1\beta_2 \tag{20}$$

We then write the J^u array to shared memory. Note that on a single warp, J^u is a 2-d array $J^u_{\tau\beta}$, but in shared memory it becomes a 3-d array $J^u_{r\tau\beta}$. The new index $0 \le r < W_d$ corresponds to the position of the warp in the dish tiling. We will reduce over this index in the next step (§5). We use the following shared memory layout for $J^u_{r\tau\beta}$:

```
__shared__ int16+16 Ju[W_D][T][B+4];   // (r, time, beam)        (21)
```

with the beam axis padded by 4 elements (16 bytes). With this shared memory layout, and the J^u register assignment in Eq. (20), the J^u array can be written to shared memory with two bank conflict free 32-bit stores per thread.

5 Step 3: reduce and quantize

In this step, we reduce the array $J^u_{r\tau\beta}$ over the index $0 \le r < W_d$, and quantize the result to int4+4 (applying the bit shift in Eq. (2)), obtaining the beamformed J-array $J_{\beta\tau}$. The J-array will be saved in registers, and written to global memory (on a slower 128-sample cadence) in §6.

First we will describe a "building-block" reduce operation which processes 4 beams and 8 times on a single warp. To process a larger array of B beams, we assign sets of four beams to different warps in a round-robin fashion. This assignment of beams to warps is unrelated to the $(W_b \times W_d)$ warp tiling in §4. To process a larger array of T times, we loop (within each warp) over chunks of 8 time samples.

In the building-block operation, for each $0 \le r < W_d$, we read the 4-by-8 J^u array in register assignment

$$[\text{int16+16 } J^u_{\tau\beta}] \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_0\tau_1\tau_2\beta_0\beta_1 \tag{22}$$

We unpack the real and imaginary parts into separate 32-bit registers:

$$[\text{int32 } J^u_{\tau\beta}] \qquad r \leftrightarrow \text{ReIm} \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_0\tau_1\tau_2\beta_0\beta_1 \tag{23}$$

We perform steps (22), (23) for each $0 \le r < W_d$ and accumulate the result, to obtain the *reduced* J-array:

$$[\text{int32 } J_{\tau\beta}] \qquad r \leftrightarrow \text{ReIm} \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_0\tau_1\tau_2\beta_0\beta_1 \tag{24}$$

We then apply the following right shift:

$$J_{\beta\tau} \leftarrow (J_{\beta\tau} >> (s_\beta - \sigma)) \tag{25}$$

as specified in Eq. (2), and keeping in mind that we already right-shifted by σ in Eq. (19). The value of s_β can be loaded into a register at the beginning of the kernel, and held persistently throughout the kernel.

We quantize to 4 bits, and pack the real/imaginary parts into a single register:

$$[\text{int4+4 } J_{\beta\tau}] \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_0\tau_1\tau_2\beta_0\beta_1 \tag{26}$$

At this point, we have distilled 8 time samples into a single byte per thread. We pack this byte into one of four "persistent" J-registers which store J-array elements until they can be written out at slower cadence (128 time samples) in §6. Collectively, the four persistent J-registers represent an array $J^{\text{per}}_{\beta\tau}$ with four beams and 128 time samples. It will be convenient in §6 to use the following slightly awkward register assignment:

$$[\text{int4+4 } J^{\text{per}}_{\beta\tau}] \qquad b_0b_1 \leftrightarrow \tau_5\tau_6 \qquad r_0r_1 \leftrightarrow \tau_3\tau_4 \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_0\tau_1\tau_2\beta_0\beta_1 \tag{27}$$

This register assignment specifies how the byte in Eq. (26) should be packed into one of the four J^{per} registers in Eq. (27), as a function of time τ within the 128-byte cadence for writing to global memory.

One final comment to conclude this section. Note that if the number of warps is not divisible by $4B$, then the number of J^{per} registers is not the same on every warp. This is awkward in implementation since the length of a register array must be a compile-time constant (not a warp-dependent quantity). This can be handled by assigning the maximum possible number of registers at compile time, and using a runtime if-statement to short-circuit the logic on warps which need fewer registers.

6 Step 4: shuffle and write to global memory

Every 128 time samples, we write the J-array to global memory. Restricting attention to a set of four beams on a single warp, we have the array (this equation is the same as Eq. (27)):

$$[\text{int4+4 } J^{\text{per}}_{\beta\tau}] \qquad b_0 b_1 \leftrightarrow \tau_5\tau_6 \qquad r_0 r_1 \leftrightarrow \tau_3\tau_4 \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_0\tau_1\tau_2\beta_0\beta_1 \tag{28}$$

We need some local transpose (Appendix B) and warp transpose (Appendix C) operations, to get a register assignment which is suitable for writing to global memory. First exchange r_1, t_0 using a warp transpose:

$$[\text{int4+4 } J^{\text{per}}_{\beta\tau}] \qquad b_0 b_1 \leftrightarrow \tau_5\tau_6 \qquad r_0 r_1 \leftrightarrow \tau_3\tau_0 \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_4\tau_1\tau_2\beta_0\beta_1 \tag{29}$$

Exchange b_0, r_1 using a local transpose:

$$[\text{int4+4 } J^{\text{per}}_{\beta\tau}] \qquad b_0 b_1 \leftrightarrow \tau_0\tau_6 \qquad r_0 r_1 \leftrightarrow \tau_3\tau_5 \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_4\tau_1\tau_2\beta_0\beta_1 \tag{30}$$

Exchange r_1, t_1 using a warp transpose:

$$[\text{int4+4 } J^{\text{per}}_{\beta\tau}] \qquad b_0 b_1 \leftrightarrow \tau_0\tau_6 \qquad r_0 r_1 \leftrightarrow \tau_3\tau_1 \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_4\tau_5\tau_2\beta_0\beta_1 \tag{31}$$

Exchange b_1, r_1 using a local transpose:

$$[\text{int4+4 } J^{\text{per}}_{\beta\tau}] \qquad b_0 b_1 \leftrightarrow \tau_0\tau_1 \qquad r_0 r_1 \leftrightarrow \tau_3\tau_6 \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_4\tau_5\tau_2\beta_0\beta_1 \tag{32}$$

Exchange r_1, t_2 using a warp transpose:

$$[\text{int4+4 } J^{\text{per}}_{\beta\tau}] \qquad b_0 b_1 \leftrightarrow \tau_0\tau_1 \qquad r_0 r_1 \leftrightarrow \tau_3\tau_2 \qquad t_0t_1t_2t_3t_4 \leftrightarrow \tau_4\tau_5\tau_6\beta_0\beta_1 \tag{33}$$

In this register assignment, the $J^{\text{per}}_{\beta\tau}$ array can be written to global memory by issuing one contiguous 16-byte store instruction on each thread. (See discussion of 16-byte load/store instructions in §3.)

7 Cost estimates

All estimates assume $B = 96$ and $(W_b, W_d) = (6, 4)$. I don't know whether this choice of (W_b, W_d) is optimal (see §2 for more discussion).

$$\begin{aligned}
\text{Global memory bandwidth} &= (\text{Read } E) + (\text{Write } J) \\
&= \frac{2FD \text{ bytes}}{t_s} + \frac{2FB \text{ bytes}}{t_s} \\
&= (8.94 + 1.68) \text{ GB/s}) \\
&= 10.62 \text{ GB/s} \\
&= 2.1\% \text{ of an A40} \qquad (500 \text{ GB/sec assumed}) \qquad (34) \\
\text{Shared memory bandwidth} &= (\text{Write } E) + (\text{Read } E) + (\text{Write } J^u) + (\text{Read } J^u) \\
&= \frac{2F \text{ bytes}}{t_s}\Big(D + W_b D + 4W_d B + 4W_d B\Big) \\
&= (8.9 + 53.6 + 26.8 + 26.8) \text{ GB/s} \\
&= 1.3\% \text{ of an A40} \qquad (8700 \text{ GB/sec assumed}) \qquad (35) \\
\text{Tensor core math} &= \frac{16FBD \text{ flops}}{t_s} \\
&= 6.7 \text{ Tflops} \\
&= 2.2\% \text{ of an A40} \qquad (300 \text{ Tflops assumed}) \qquad (36) \\
\text{Warp shuffles (just checking)} &= (2F)\left(\frac{B}{4}\right)\left(\frac{1}{128t_s}\right)(192 \text{ shuffles}) \\
&= 0.62 \text{ gigashuffles/sec} \\
&= 0.03\% \text{ of an A40} \qquad (2300 \text{ Gshuffles/sec assumed}) \qquad (37) \\
\text{int32 math (rough)} &\sim \frac{2W_d FB}{t_s}(15 \text{ int32 ops}) \\
&= 100 \text{ int32 Gops/sec} \\
&= 0.6\% \text{ of an A40} \qquad (17 \text{ Tops/sec assumed}) \qquad (38)
\end{aligned}$$

When counting warp shuffles, we treat each `__shfl_sync()` as 32 "shuffles" (following NVIDIA). For int32 math, I made a very rough estimate assuming 15 instructions per element of the J^u array.

Adding everything up (which may be pessimistic since there is scope for overlapping I/O and compute), the 8-bit beamformer should take **6.3% of an A40**. Too good to be true?

A Register assignment notation and tensor core MMA

In 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, using the tensor core MMA operation.

Throughout these notes, "tensor core MMA" always refers to the signed int8 matrix-multiply-accumulate with tile size $(m, n, k) = (8, 8, 16)$ in NVIDIA's notation.[7] This operation multiplies an 8-by-16 matrix A_{ij} by a 16-by-8 matrix B_{jk} to produce an 8-by-8 matrix C_{ik}. The A, B matrices are int8, and the C-matrix is int32.

[7]As far as I can tell, all `int8` MMA operations run at the same speed, *provided* that the `mma.sync.*` PTX instruction (not `wmma.mma.sync.*`) is wrapped in inline assembly. Therefore, it makes sense to use the MMA operation with the smallest tile size. If a reason to use a larger tile size emerges in the future, it would be a minor change to the beamforming kernel (I checked that the register assignments for the larger tile sizes are "copies" of the smallest size, so we wouldn't need to change details like shared memory layouts).

Consider the 8-by-16 matrix int8 A_{ij}, which is distributed among threads in one warp. Each matrix entry has a "logical" location (i,j) in the matrix, and a "physical" location as a byte in a register somewhere. We describe both logical and physical locations using index bits as follows.

A logical location is described by integers $0 \le i < 8$ and $0 \le j < 16$, which we represent by their binary digits $i = \big[i_2 i_1 i_0\big]_2$ and $j = \big[j_3 j_2 j_1 j_0\big]_2$. Thus, we label "logical" locations by 7 index bits $i_2 i_1 i_0 j_3 j_2 j_1 j_0$.

A physical location is indexed by a 5-bit thread id $t = \big[t_4 t_3 t_2 t_1 t_0\big]_2$, and a 2-bit byte id $b = \big[b_1 b_0\big]_2$ which indexes the location of the `int8` within the 32-bit register. (Note that the A-matrix uses one register per thread. For an array with multiple registers per thread, we would introduce index bits $r_0 r_1 r_2 \cdots$ to indicate which register contains an array element.) Thus, we label "physical" locations by 7 index bits $t_4 t_3 t_2 t_1 t_0 b_1 b_0$.

Our register assignment notation works by writing down the correspondence between logical and physical index bits. In this notation, the register assignments for the A, B, C matrices are:[8]

$$[\text{int8 } A_{ij}] \qquad b_0 b_1 \leftrightarrow j_0 j_1 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow j_2 j_3 i_0 i_1 i_2 \tag{39}$$

$$[\text{int8 } B_{jk}] \qquad b_0 b_1 \leftrightarrow j_0 j_1 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow j_2 j_3 k_0 k_1 k_2 \tag{40}$$

$$[\text{int32 } C_{ik}] \qquad r_0 \leftrightarrow k_0 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow k_1 k_2 i_0 i_1 i_2 \tag{41}$$

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, no byte index bits for int32).
- The number of registers per thread is 2^R, where R is the number of "register" bits r_i. For example, the A and B-matrices in Eqs. (39), (40) use one register per thread ($R = 0$), and the C-matrix in Eq. (41) uses two registers per thread ($R = 3$).
- 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.

In the beamformer, the indices i, j, k represent respectively a beam β, a dish d' with index permutation (12) applied, and a time τ. The B-matrix represents the input electric field array $E_{\tau d'}$, and the C-matrix represents the "unreduced" beamformed array $J^u_{\tau\beta}$. Therefore, we will write the register assignment this way:

$$[\text{int8 } A_{\beta d'}] \qquad b_0 b_1 \leftrightarrow d'_0 d'_1 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d'_2 d'_3 \beta_0 \beta_1 \beta_2 \tag{42}$$

$$[\text{int8 } E_{\tau d'}] \qquad b_0 b_1 \leftrightarrow d'_0 d'_1 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow d'_2 d'_3 \tau_0 \tau_1 \tau_2 \tag{43}$$

$$[\text{int32 } J^u_{\tau\beta}] \qquad r_0 \leftrightarrow \tau_0 \qquad t_0 t_1 t_2 t_3 t_4 \leftrightarrow \tau_1 \tau_2 \beta_0 \beta_1 \beta_2 \tag{44}$$

For reference, my CUDA wrapper for the $(m, n, k) = (8, 8, 16)$ MMA instruction is:

```
// D = A*B + C
__device__ __forceinline__
void mma_s8_m8_n8_k16(int d[2], int a[1], int b[1], int c[2])
{
    asm("mma.sync.aligned.m8n8k16.row.col.satfinite.s32.s8.s8.s32 "
        "{%0, %1}, {%2}, {%3}, {%4, %5};" :
        "=r" (d[0]), "=r" (d[1]) :
        "r" (a[0]), "r" (b[0]), "r" (c[0]), "r" (c[1])
    );
}
```

B 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:

$$b_1 b_0 \leftrightarrow XY \qquad r \leftrightarrow Z \tag{45}$$

[8]From `https://github.com/kmsmith137/gputils/blob/master/reverse_engineering/reverse-engineer-mma.cu`

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:

$$b_1 b_0 \leftrightarrow XZ \qquad r \leftrightarrow Y \tag{46}$$

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:

$$b_1 b_0 \leftrightarrow ZY \qquad r \leftrightarrow X \tag{47}$$

Either of the local transpose operations defined in Eqs. (46), (47) can be implemented with two calls to the `__byte_perm()` cuda intrinsic, or with a longer sequence of bitwise operations. (I'm guessing `__byte_perm()` will be faster.)

C Warp transpose operation

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

$$r \leftrightarrow X \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow Y_4 Y_3 Y_2 Y_1 Y_0 \tag{48}$$

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:

$$r \leftrightarrow Y_i \qquad t_4 t_3 t_2 t_1 t_0 \leftrightarrow Y_4 \cdots \underbrace{X}_{\text{replacing } Y_i} \cdots Y_0 \tag{49}$$

This can be done efficiently with one warp shuffle instruction as follows:[9]

```
int i = ...; // same meaning as previous equation
int in0 = ...;  // register 0
int in1 = ...;  // register 1

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

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

(flag ? out0 : out1) = dst;
```

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

[9]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 tiny (Eq. (37)), but I thought the larger issue was worth mentioning.