Final Project

Adaptive beamforming

邱崇喆 (Chong Jhe Chiu) 108061173
陳佳詳111061627　張耀明111061549　宋乃仁 111061551

1. ABSTRACT

Beamforming can be seen as a spatial filter obtaining the signals from desired directions, while meanwhile suppressing the signals from unwanted directions. There are multiple ways to implement digital beamforming, which generally fall into two categories, conventional one and adaptive one.

Conventional beamforming is equivalent to using phase shift and amplitude scaling to compensate for the arrival delays of the narrowband signals from Rx antenna elements, to align the output signals as in-phase superposition. While conventional beamformers can point the main lobe to the direction of interest, it cannot effectively eliminate interference, or in other words increase SINR, and this is where adaptive beamformer such as MVDR, MPDR and LCMV comes in, by which we can both get the signals from desired directions and eliminate interference.

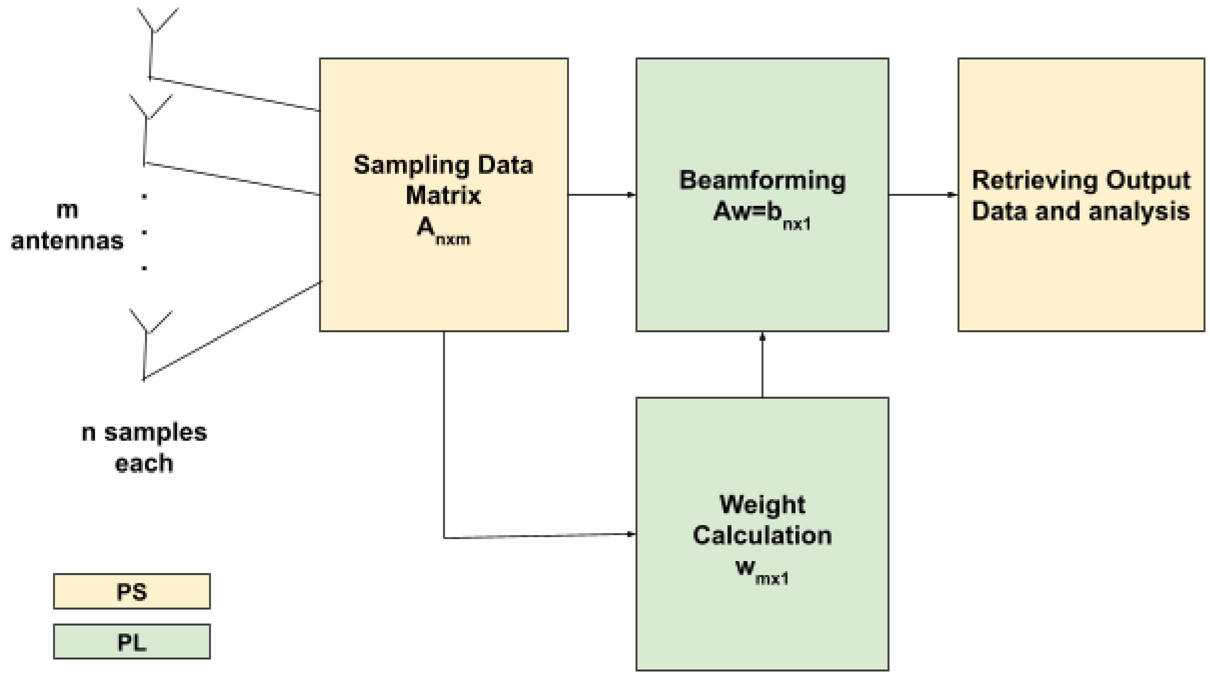


fig.1 workflow of the project

The adaptive beamforming algorithm used in this project is MPDR beamforming. QR, and Cholesky decomposition will be used, and these are the parts we want to accelerate. Besides, multiplications of matrices (sampling data matrix and the calculated weights matrix) will also be run on FPGA. Ultimately, the output sample data will be retrieved to PS for analysis.

Some other functions might be added if possible, for example, channel estimation/precoding might be added to achieve spatial multiplexing for the MIMO communication system.

target platform：U50

host program： C++/OpenCL

test data : Matlab

tasks division:
邱崇喆：whole system (kernels & host & input processing), theory
宋乃仁：input/output processing & group discussion
陳佳詳 & 張耀明：host + QRF & group discussion

2. Introduction

Generally, we can express the received data vector y from antennas as:

$$y = \sum_i v_i x_i + n = v_s x_s + \sum_{i \neq s} v_i x_i + n$$

, where v_i is the manifold vector and n is the channel noise, we can further express y with signal from the desired direction $v_s x_s$, and noise plus interference IN:

$$y = v_s x_s + IN$$

$$w^\dagger y = w^\dagger (v_s x_s + IN)$$

There are two ways to extract the desired signals or to maximize SINR, MVDR beamforming and MPDR beamforming. The former try to minimize the variance or the power of noise plus interference $E[|IN|^2]$, while the latter try to minimize the output signal power $E[|w^\dagger y|^2]$.

Note that $E[|w^\dagger y|^2] = E[w^\dagger y y^\dagger w] = w^\dagger E[y y^\dagger] w = w^\dagger R_{yy} w$, where R_{yy} is the auto-correlation of the received data.

In this implementation we are going to adopt MPDR method with Lagrange multiplier, namely minimizing $w^\dagger R_{yy} w$, given a distortionless constraint or gain $w^\dagger v_s = 1$.

$$min_{w^\dagger} (w^\dagger R_{yy} w) \quad subject\ to \quad w^\dagger v_s = 1$$

Solving the Lagrange multiplier problem above, we can obtain the solution:

$$w_{mpdr} = \frac{R_{yy}^{-1} v_s}{v_s^\dagger R_{yy}^{-1} v_s}$$

assuming that all signals x_i are uncorrelated to each other and $E[IN] = E[n] = 0$, and $p_i = E[x_i x_i^\dagger]$, then:

$$R_{yy} = E[(v_s x_s + IN)(v_s^\dagger x_s^\dagger + IN^\dagger)] = p_s v_s v_s^\dagger + E[IN \cdot IN^\dagger] = p_s v_s v_s^\dagger + R_{ININ}$$

$$\Rightarrow R_{ININ} = E[IN \cdot IN^\dagger] = \sum_{i \neq s} p_i v_i v_i^\dagger + E[nn^\dagger] = \sum_{i \neq s} p_i v_i v_i^\dagger + R_{nn}$$

$$\Rightarrow R_{yy} = p_s v_s v_s^\dagger + R_{ININ} = p_s v_s v_s^\dagger + \sum_{i \neq s} p_i v_i v_i^\dagger + R_{nn}$$

The problems right here is that R_{yy} is based on probability or randomness, so we have to figure out some ways to estimate it. In practice, we will use Sample Matrix Inversion to estimate R_{yy}:

$$R_{yy} \sim \frac{1}{N} \sum_{n=1}^{N} y_n y_n^\dagger = \frac{1}{N} Y^\dagger Y$$

, where $Y = \overline{A}$ is the sample matrix comprised of N samples each from m antennas, it's just a rearrangement of the matrix A in fig.1, and n is the index of a $m \times 1$ sample vector of time n.

To make it more computationally viable, we can use QR decomposition $Y = QR$:

$$R_{yy} \sim \frac{1}{N} Y^\dagger Y = \frac{1}{N} R^\dagger Q^\dagger Q R = \frac{1}{N} R^\dagger R$$

we can exploit Cholesky decomposition by defining $L = \frac{1}{\sqrt{N}} R^\dagger$, then:

$$R_{yy} \sim LL^\dagger$$

$$w_{mpdr} = \frac{R_{yy}^{-1} v_s}{v_s^\dagger R_{yy}^{-1} v_s} = \frac{L^{\dagger^{-1}} L^{-1} v_s}{v_s^\dagger L^{\dagger^{-1}} L^{-1} v_s}$$

Next, define $LL^\dagger u = v_s$ and $Lz = v_s$, then we can rewrite the weight as:

$$w_{mpdr} = \frac{u}{|z|^2}$$

3.Tasks details

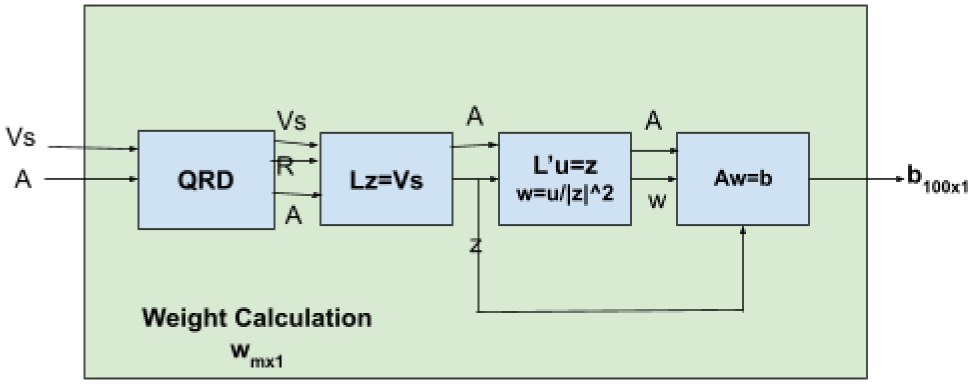


fig.2 modification of fig.1

The above task flow is in reference to the document [1]. In summary, the procedure of obtaining the weight is to calculate the following in order:

- Determine v_s
- Calculate $Y = QR$
- Calculate $L = \frac{1}{\sqrt{N}} R^{\dagger}$
- Solve $LL^{\dagger}u = v_s$ & $Lz = v_s$
- Calculate $w_{mpdr} = \frac{u}{|z|^2}$
- Calculate $A_{nxm}w_{mx1}=b_{nx1}$

Top function:

```
extern "C" void Top_Kernel(
    MATRIX_IN_T matrixA[1000],

    MATRIX_IN_T Vs[10],
    //hls::x_complex<double> matrixQ[100*100],
    MATRIX_OUT_T matrixR[1000]
    )
{
// extern "C" void kernel_geqrf_0(double dataA[MA*NA], double tau[NA]) {
//#pragma HLS INTERFACE axis port = matrixAStrm
//#pragma HLS INTERFACE axis port = matrixQStrm
//#pragma HLS INTERFACE axis port = matrixRStrm
#pragma HLS INTERFACE m_axi port = matrixA bundle = gmem0 offset = slave depth = 1000
//#pragma HLS INTERFACE m_axi port = matrixQ bundle = gmem1 offset = slave num_read_outstanding = 16 max_read_burst_length = \
//    32
#pragma HLS INTERFACE m_axi port = Vs bundle = gmem1 offset = slave depth = 10
#pragma HLS INTERFACE m_axi port = matrixR bundle = gmem2 offset = slave depth = 1000


//#pragma HLS INTERFACE s_axilite port = matrixA bundle = control
//#pragma HLS INTERFACE s_axilite port = Vs bundle = control
//#pragma HLS INTERFACE s_axilite port = matrixR bundle = control

//#pragma HLS INTERFACE s_axilite port = return bundle = control

//xf::solver::qrf<0, 100, 10, hls::x_complex<double>, hls::x_complex<double>, my_qrf_traits>(matrixAStrm, matrixQStrm,matrixRStrm);
const unsigned int rowA = 100;
const unsigned int colA = 10;
const unsigned int rowQ = 100;
const unsigned int colQ = 100;
const unsigned int rowR = 100;
const unsigned int colR = 10;
/*
int k=0;
for (int r = 0; r < 10; r++) {

      std::cout << Vs[r] << std::endl;
}
*/
pass_dataflow(
    matrixA,
    //matrixQ,
    matrixR,
```

The top function calls the pass_dataflow function

pass_dataflow:

```
void pass_dataflow(
    MATRIX_IN_T* matrixA,
    //hls::x_complex<double>* matrixQ,
    MATRIX_OUT_T* matrixR,
    MATRIX_IN_T* Vs,
    const unsigned int rowA,
    const unsigned int colA,
    const unsigned int rowQ,
    const unsigned int colQ,
    const unsigned int rowR,
    const unsigned int colR
    )
{
```

```
//std::cout<< "0" <<std::endl;
static hls::stream<MATRIX_IN_T> matrixAStrm;
//static hls::stream<MATRIX_OUT_T> matrixQStrm;
static hls::stream<MATRIX_OUT_T> matrixRStrm;
// in out stream for Vs to avoid bypass path=======
static hls::stream<MATRIX_OUT_T> qrf_A_outstream;
static hls::stream<MATRIX_OUT_T> VsStrm_out1;
static hls::stream<MATRIX_OUT_T> VsStrm_out2;
//=================================================
static hls::stream<MATRIX_IN_T> VsStrm;
static hls::stream<MATRIX_IN_T> RStrm;
//static hls::stream<MATRIX_IN_T> qrf_transpose_A_outstream;
//static hls::stream<MATRIX_OUT_T> matrixLstrm;
//==================================================
//static hls::stream<MATRIX_IN_T> inhom_Vs_instream;
static hls::stream<MATRIX_IN_T> inhom_A_outstream;
static hls::stream<MATRIX_IN_T> weight_stream;

//==================================================

#pragma HLS stream depth=1000 variable=matrixAStrm
//#pragma HLS stream depth=10000 variable=matrixQStrm
#pragma HLS stream depth=100 variable=matrixRStrm

#pragma HLS stream depth=1000 variable=qrf_A_outstream
#pragma HLS stream depth=10 variable=VsStrm_out1
#pragma HLS stream depth=10 variable=VsStrm_out2

#pragma HLS stream depth=10 variable=VsStrm
#pragma HLS stream depth=100 variable=RStrm

#pragma HLS stream depth=1000 variable=inhom_A_outstream
#pragma HLS stream depth=10 variable=weight_stream


//#pragma HLS stream depth=1000 variable=qrf_transpose_A_outstream
//Turn the 2Darray MatrixA  sent from host to kernel by axi_master to the hls:stream type
Master2Stream(matrixA, matrixAStrm,Vs,VsStrm_out1, rowA, colA);

//Vitis Library QR Factorization-------------
QRF(matrixAStrm,matrixRStrm,VsStrm_out1,VsStrm_out2,qrf_A_outstream);


/*qrf_transpose(qrf_A_outstream,qrf_transpose_A_outstream,matrixQStrm,matrixRStrm,VsStrm_out2,VsStrm,RStrm,
            rowQ, colQ, rowR, colR);*/

inhom(weight_stream,matrixRStrm,VsStrm_out2,qrf_A_outstream,inhom_A_outstream);

Weights_Mul(matrixR,inhom_A_outstream,weight_stream);
```

pass_dataflow calls subfunctions needed for the kernel.

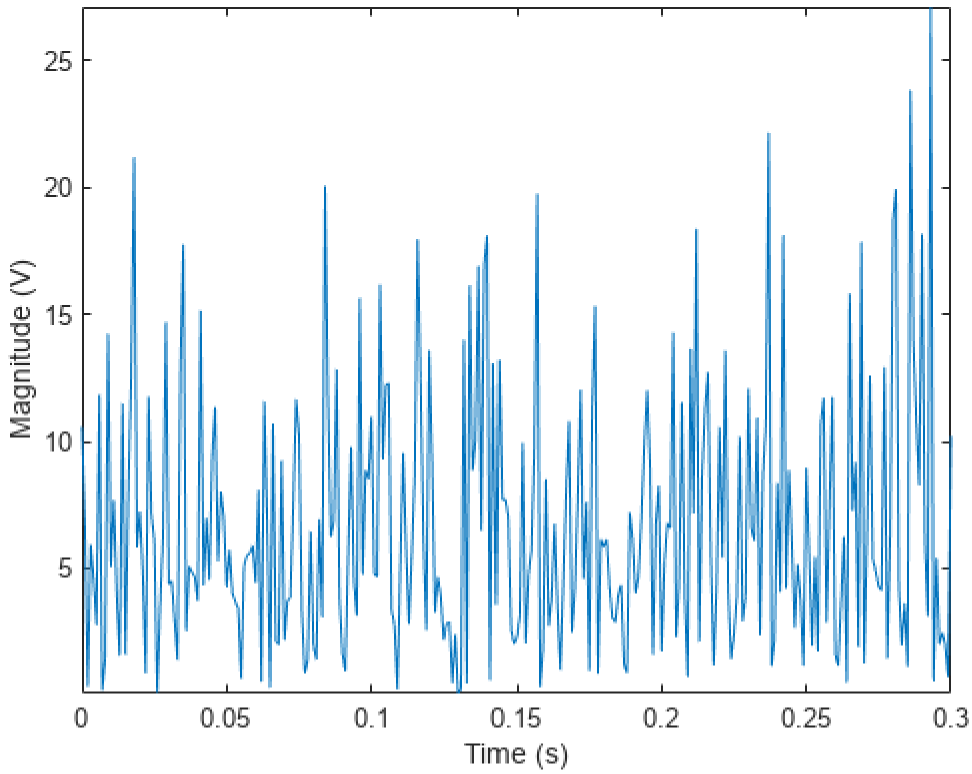
1) prepare input matrix A

generate input data from matlab:

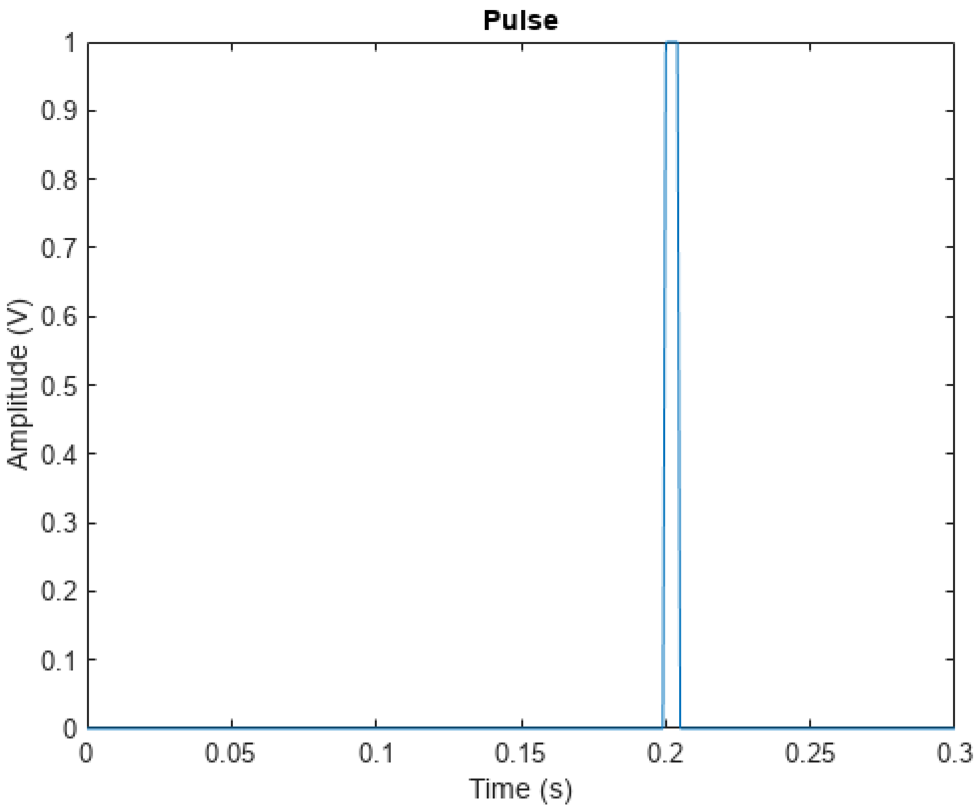
```
1 -2.7011055181846+-0.982986397387809i,-8.50996494364062+0.981686077384369i,-
  5.41815573639443+-14.1801811178475i,-
  10.8105110205246+10.9037501694024i,-9.11686858282469+6.98445727982964i,-9.1178873008493+-6.98045725017635i,-
  10.8089353521239+-10.9042383130599i,-
  5.42055447605449+14.1836060933795i,-8.51085875340653+-0.980218841245973i,-2.70428474570187+0.988027145426052
2 -11.6030616067212+8.39162777350578i,-15.3687000737728+-8.51004903730555i,-
  12.8854105982462+-19.9536451605028i,-
  18.0859655240823+17.1126639324322i,-16.4935701970758+15.4453014840139i,-16.4940968274651+-15.4418446861904i,
  18.0869902377044+-17.1145920934201i,-
  12.8851286592352+19.9588963280458i,-15.3734495031958+8.50857352231495i,-11.6082082996715+-8.39515930967039i
3 9.50765104587367+-16.2322230132609i,-11.3095012359225+17.0984738003847i,4.06172881742874+-24.4936265839877i,
  9.86705336789868+18.407697407064i,-7.74088575611278+-2.59530676611247i,-7.74173865982427+2.59377919522869i,-
  9.86862067408189+-18.4082635978814i,4.05804077722586+24.4957397470358i,-11.3045235928221+-17.0984919257579i,
  9.50685038788279+16.2355079451656i
4 -15.1196823213874+35.591509379266i,-16.1997356917615+-32.7083378933329i,-
  28.1093714006304+6.08676347204966i,-14.9365577190842+11.8151553652147i,2.42545296848139+-11.5425389885888i,-
  2.42225782974296+11.5446149602344i,-14.9380591433831+-11.8174265994094i,-
  28.1136704091613+-6.08773541410504i,-16.200520804732+32.7080019808038i,-15.1180306187837+-35.597967786234i
5 -1.65876066114104+-1.8229709682644i,10.8302815971193+0.281949980972252i,-10.9183086892627+10.451412095186i,-
  1.24892360676613+-13.3815730056833i,2.03574137177704+7.26591404278082i,2.03087599500287+-7.26466590776675i,-
  1.24550361753537+13.3721059908383i,-10.9222374287224+-10.4481602043697i,-
  10.8294287952837+-0.27947799193368i,-1.65805142393936+1.8213952027156i
6 12.828121734928+-16.8167675373653i,-12.5256392865715+18.4473083783185i,6.1799029103706+-23.9515076984669i,-
  4.50191464056496+20.2210437950649i,-4.45950999087451+-9.86461190994841i,-4.45966768717541+9.86364817617736i,
  4.50754733456859+-20.2200969834015i,6.18302509903598+23.9527662522562i,-12.5276758705874+-18.4427852299772i,
  12.8306582426575+16.821070347419i
7 -8.70193596919676+27.4591998078854i,-15.9110053412662+-24.2847639804389i,-
  25.8511731653034+2.78675135954356i,-16.0464853924234+13.5550013046595i,3.99708485543446+-16.2749078462228i,-
  3.99782461390596+16.2759363434298i,-16.0466528145469+-13.5645198387201i,-
  25.849421007297 8+-2.78174701581828i,-15.9151616856925+24.2858062470739i,-8.70031576827569+-27.4571164522005i
8 -0.121883331015871+12.2655913600827i,-23.0624470788679+-8.65069240822981i,-
  26.6215730402837+-16.0223887002426i,-7.69375608759716+26.4289038000108i,-1.81945521242273+-17.6308005647456i
  26.6190374943873+16.0291704769907i,-23.0676061871714+8.6472175343521i,-0.122608782814455+-12.2647037981731i
9 6.04580655303906+0.971445532986297i,-0.155538818541209+0.513348316896898i,-
```

and transform to a readable version for C++ (using c++ code).

```
-2.7011055181846 -0.982986397387809
-8.50996494364062 0.981686077384369
5.41815573639443 -14.1801811178475
10.8105110205246 10.9037501694024
-9.11686858282469 6.98445727982964
-9.1178873008493 -6.98045725017635
10.8089353521239 -10.9042383130599
5.42055447605449 14.1836060933795
-8.51085875340653 -0.980218841245973
-2.70428474570187 0.988027145426052
-11.6030616067212 8.39162777350578
-15.3687000737728 -8.51004903730555
12.8854105982462 -19.9536451605028
18.0859655240823 17.1126639324322
-16.4935701970758 15.4453014840139
-16.4940968274651 -15.4418446861904
18.0869902377044 -17.1145920934201
12.8851286592352 19.9588963280458
-15.3734495031958 8.50857352231495
-11.6082082996715 -8.39515930967039
9.50765104587367 -16.2322230132609
-11.3095012359225 17.0984738003847
4.06172881742874 -24.4936265839877
9.86705336789868 18.407697407064
-7.74088575611278 -2.59530676611247
-7.74173865982427 2.59377919522869
9.86862067408189 -18.4082635978814
4.05804077722586 24.4957397470358
-11.3045235928221 -17.0984919257579
9.50685038788279 16.2355079451656
-15.1196823213874 35.591509379266
-16.1997356917615 -32.7083378933329
28.1093714006304 6.08676347204966
-14.9365577190842 11.8151553652147
2.42545296848139 -11.5425389885888
2.42225782974296 11.5446149602344
-14.9380591433831 -11.8174265994094
```



a batch of input data with interference from 30,50 and 70 degree, and Gaussian noise.
extracted from MATLAB: https://www.mathworks.com/help/phased/ug/conventional-and-adaptive-beamformers.html



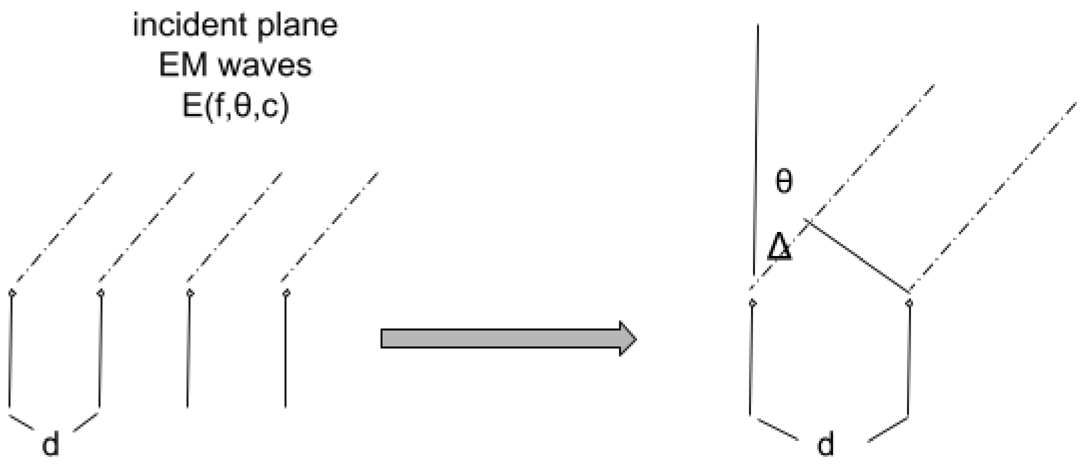
The above is the signal we want to extract. (just for example)
extracted from MATLAB: https://www.mathworks.com/help/phased/ug/conventional-and-adaptive-beamformers.html

```cpp
for(int angle_factor=51;angle_factor<53;angle_factor++){
    double incident_angle_in_rad=(angle_factor*0.9)*M_PI/180; //incident angle=angle_factor*0.9


    std::string base_path = "./organized_input/";
    std::string file_A =
        base_path + "A_" + std::to_string(angle_factor) + ".txt";

    std::cout <<"read file: "<< file_A << std::endl;


    std::complex<double>* A_ptr = reinterpret_cast<std::complex<double>*>(A);

    //std::complex<double>* Q_ptr = reinterpret_cast<std::complex<double>*>(Q_expected);
    //std::complex<double>* R_ptr = reinterpret_cast<std::complex<double>*>(R_expected);

    //prepare input A and Vs
    int A_size = numRow * numCol;
    readTxt(file_A, A_ptr, A_size);

    int k = 0;
    for (int r = 0; r < numRow; r++) {
        for (int c = 0; c < numCol; c++) {
            dataA_qrd[k] = A[r][c];
            k++;
        }
    }

    for(int i = 0; i < Vs_size; i++){
        dataVs_qrd[i] =std::polar(1.0,-i*M_PI*sin(incident_angle_in_rad));
        std::cout<<dataVs_qrd[i]<<std::endl;
    }
    //-------------------------------------------------------------
```

EMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

read different angles input data with by host code

2) determine v_s



Optical path difference between two antennas: Δ

Antennas with constant spacing: d

Incident angle of EM waves: θ

Frequency of EM waves: f

fig.3

From fig.3 we know $\Delta = dsin(\theta)$, then the phase difference between two incident waves is ϕ:

$$\phi = 2\pi \cdot \frac{\Delta}{\lambda} = 2\pi \cdot \frac{dsin(\theta)}{\lambda} = 2\pi \cdot \frac{dsin(\theta) \cdot f}{c}$$

This implies given the EM wave frequency f, the incident angle θ and the constant spacing of ULA, the corresponding manifold vector is (take 3 antennas for example):

$$v_s = \begin{bmatrix} 1 \\ e^{-j2\pi \cdot dsin\theta \cdot f/c} \\ e^{-j2\pi \cdot dsin\theta \cdot 2f/c} \end{bmatrix}$$

when $d = \lambda/2$, then $\phi = 2\pi \cdot \frac{sin(\theta)}{2} = \pi sin(\theta)$.

```
for(int i = 0; i < Vs_size; i++){
    dataVs_qrd[i] =std::polar(1.0,-i*M_PI*sin(incident_angle_in_rad));
    std::cout<<dataVs_qrd[i]<<std::endl;
}
//---------------------------------------------------------------
```

prepare Vs in host code

3) QR decomposition

For $X = QR$, we may call Vitis Solver library or resort to Householder transformation.

$$\left(\prod_{i=1}^{m} H_i\right)^{\dagger} Y = H_m H_{m-1} .... H_2 H_1 X = R$$

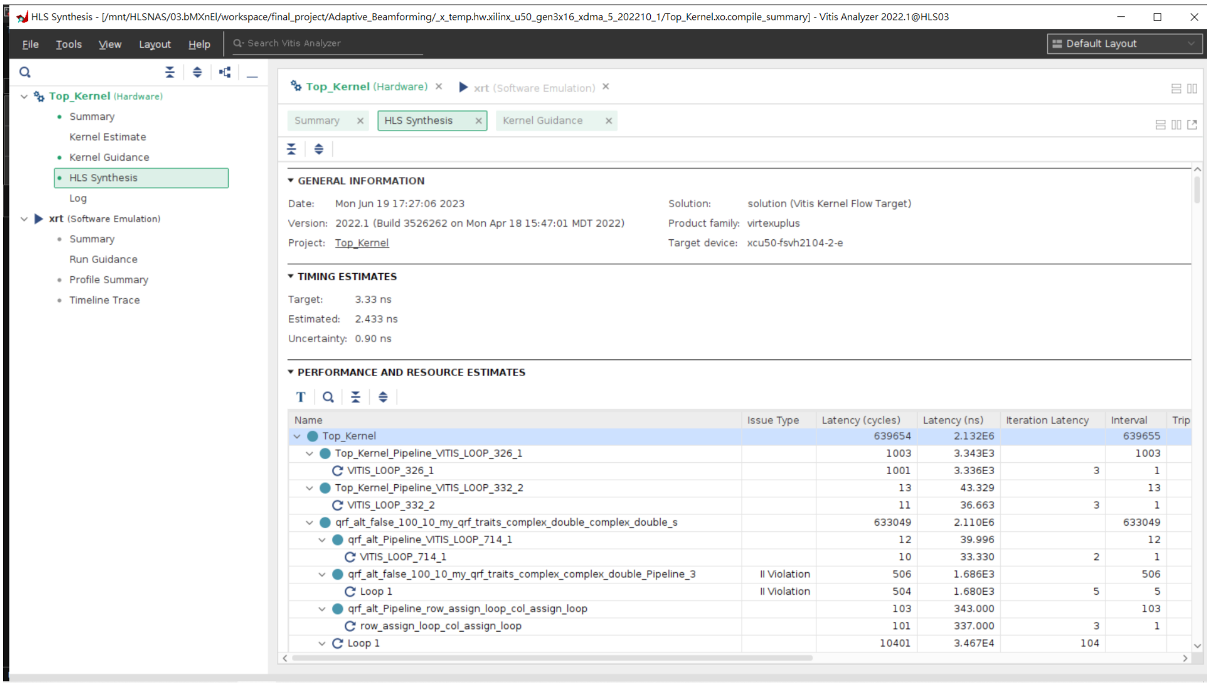
We end up making use of Vitis Solver library QR factorization kernel, and we strive to peel the library and make it as compact as possible, dismissing unnecessary files, and finally there are just few files for implementing QRF in our project.

```
void qrf_givens(int extra_pass,
                std::complex<T> a,
                std::complex<T> b,
                std::complex<T>& c,
                std::complex<T>& s,
                std::complex<T>& ss,
                std::complex<T>& cc,
                std::complex<T>& r) {
Function_qrf_givens_complex:;
    const T ONE = 1.0;
    const T ZERO = 0.0;
    const std::complex<T> CZERO = ZERO;
    T sqrt_mag_a_mag_b;
    std::complex<T> c_tmp, s_tmp;

    if (extra_pass == 0) {
        // Standard modified Givens matrix, guarding against over-/underflow
        sqrt_mag_a_mag_b = qrf_magnitude(a, b);
        if (is_zero(hls::abs(a.real())) && is_zero(hls::abs(a.imag())) && is_zero(hls::abs(b.real())) &&
            is_zero(hls::abs(b.imag()))) { // more efficient than "if (sqrt_mag_a_mag_b == ZERO)"
            c_tmp = x_copysign(ONE, a.real());
            s_tmp = ZERO;
        } else {
            c_tmp = a / sqrt_mag_a_mag_b;
            s_tmp = b / sqrt_mag_a_mag_b;
        }
        c = hls::x_conj(c_tmp);
        cc = c_tmp;
        s = hls::x_conj(s_tmp);
        ss = -s_tmp;

        r.real(sqrt_mag_a_mag_b);
    } else {
        // Transformation matrix to ensure real diagonal in R, guarding against over-/underflow
        sqrt_mag_a_mag_b = qrf_magnitude(CZERO, b);

        c_tmp = ONE;

        if (hls::abs(b.real()) == ZERO &&
            hls::abs(b.imag()) == ZERO) { // more efficient than "if (sqrt_mag_a_mag_b == ZERO)"
            s_tmp = ONE;
        } else {
            s_tmp = b / sqrt_mag_a_mag_b;
        }


        c = c_tmp;
        cc = hls::x_conj(s_tmp);
        s = ZERO;
        ss = ZERO;
        r.real(sqrt_mag_a_mag_b);
    }
}
```

The above is the code of the given rotation (this is just a part of the algorithm)

```
// Process R in batches of non-dependent array elements
px:
    for (int batch_num = 0; batch_num < CONFIG.NUM_BATCHES; batch_num++) {
    calc_rotations:
        for (int px_cnt = 0; px_cnt < CONFIG.BATCH_CNTS[batch_num]; px_cnt++) {
#pragma HLS LOOP_TRIPCOUNT min = 1 max = RowsA / 2
#pragma HLS PIPELINE II = QRF_TRAITS::CALC_ROT_II
            px_row1 = CONFIG.SEQUENCE[seq_cnt][0];
            px_row2 = CONFIG.SEQUENCE[seq_cnt][1];
            px_col = CONFIG.SEQUENCE[seq_cnt][2];
            seq_cnt++;
            extra_pass = 0;
            if (is_cmplx<InputType>::value && RowsA == ColsA && batch_num == CONFIG.NUM_BATCHES - 1) {
                extra_pass = 1;
            }
            qrf_givens(extra_pass, r_i[px_row1][px_col], r_i[px_row2][px_col], G[0][0], G[0][1], G[1][0], G[1][1], mag);
            // Pass on rotation to next block to apply rotations
            rotations[0].write(G[0][0]);
            rotations[1].write(G[0][1]);
            rotations[2].write(G[1][0]);
            rotations[3].write(G[1][1]);
            rotations[4].write(mag);
            to_rot[0].write(px_row1);
            to_rot[1].write(px_row2);
            to_rot[2].write(px_col);
        }

    rotate:
        for (int px_cnt = 0; px_cnt < CONFIG.BATCH_CNTS[batch_num]; px_cnt++) {
#pragma HLS LOOP_TRIPCOUNT min = 1 max = RowsA / 2
            G_delay[0][0] = rotations[0].read();
            G_delay[0][1] = rotations[1].read();
            G_delay[1][0] = rotations[2].read();
            G_delay[1][1] = rotations[3].read();
            mag_delay = rotations[4].read();
            rot_row1 = to_rot[0].read();
            rot_row2 = to_rot[1].read();
            rot_col = to_rot[2].read();

            extra_pass2 = 0;
            if (is_cmplx<InputType>::value && RowsA == ColsA && batch_num == CONFIG.NUM_BATCHES - 1) {
                extra_pass2 = 1;
            }
```

The above px index is the most time consuming part, which takes up to the order of 10^5 cycle per run.



The above report shows the performance (cycles) of each task, the QRF takes almost the whole execution time.

The QRF in VItis Solver library, which makes use of Given rotation, might not be as efficient as the case of exploiting Householder transformation in this project. This is because the Given rotation algorithm generally outperforms Householder transformation for the case of sparse matrix input, instead, in our project, we provide dense matrices due to noise and interference that we model. This is the reason why we think the kernel was not working efficiently.

4) Inhomogeneous equations & calculate weights

Note that $L = \frac{1}{\sqrt{N}}R^{\dagger}$ is a lower triangular matrix, the equation $Lz = v_s$ has a unique solution. For u, also, instead of solving $LL^{\dagger}u = v_s$, we solve $L^{\dagger}u = z$. To solve this equation, we can obtain each row element z_i of z by:

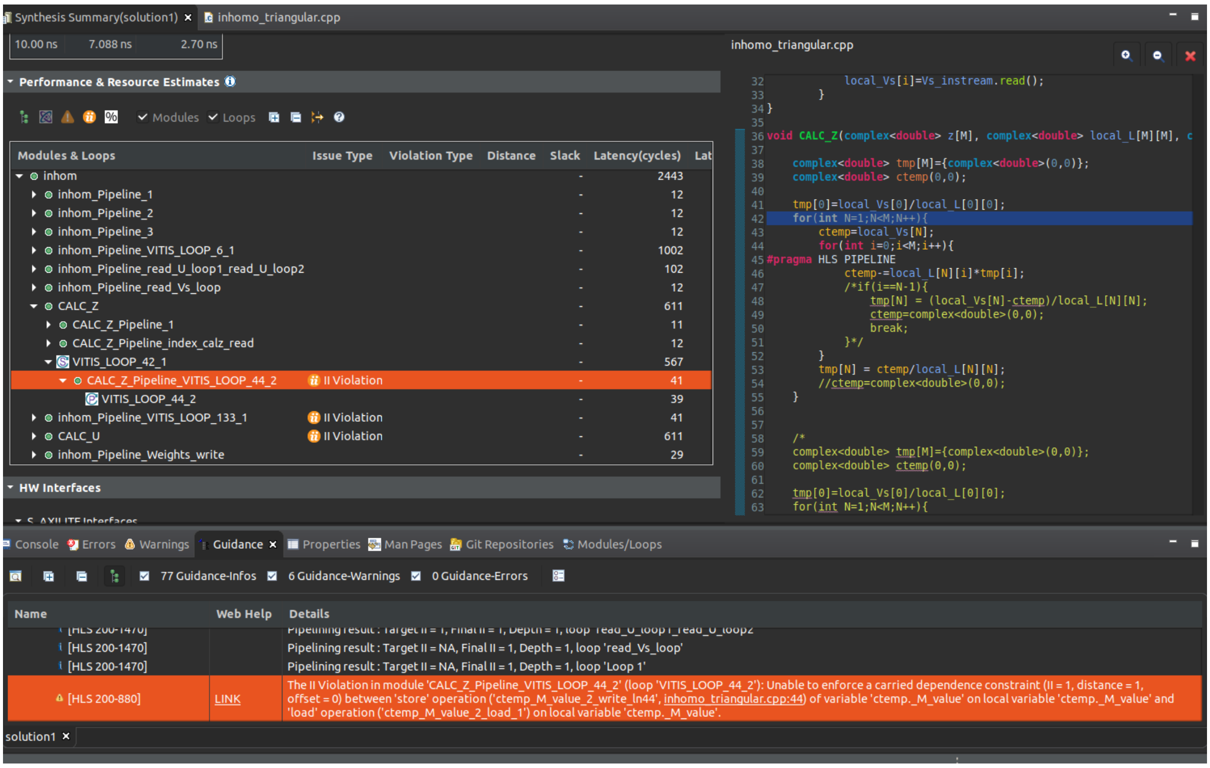
$$z_1 = \frac{v_{s_1}}{L_{11}}; \quad L_{21} \cdot z_1 + L_{22} \cdot z_2 = v_{s_2} \Rightarrow z_2 = (v_{s_2} - L_{21} \cdot z_1)/L_{22} \cdots$$

Because L and $L^{\dagger}$ are both triangular matrices. we can calculate each row element z_i fairly easily due to triangular matrices' property, well-distributed zero entries.

However, this may still take a long time because to calculate z_2 we have to wait until z_1 is calculated, and so on. This is also what prevents the task from dataflow optimization.

```
void A_stream2stream(hls::stream<complex<double>> &A_outstream, hls::stream<complex<double>> &A_instream){
        complex<double> read_value_A_instream;
        for(int i=0;i<M*S;i++){
#pragma HLS PIPELINE
                read_value_A_instream = A_instream.read();
                A_outstream.write(read_value_A_instream);
        }
}

void read_fifo_U(complex<double> local_L[M][M], complex<double> local_U[M][M], hls::stream<complex<double>> &U_instream){

        read_U_loop1:
                for(int i=0;i<M;i++){
                        read_U_loop2:
                                for(int j=0;j<M;j++){
                                        #pragma HLS PIPELINE II=1
                                        complex<double> stream_read_data;
                                        stream_read_data=U_instream.read();
                                        local_U[i][j]=stream_read_data;
                                        local_L[j][i]=conj(stream_read_data);
                                }
                }
}

void read_fifo_Vs(complex<double> local_Vs[M], hls::stream<complex<double>> &Vs_instream){

        read_Vs_loop:
                for(int i=0;i<M;i++){
                        local_Vs[i]=Vs_instream.read();
                }
}
```

```
void CALC_Z(complex<double> z[M], complex<double> local_L[M][M], complex<double> local_Vs[M]){

        complex<double> tmp[M]={complex<double>(0,0)};
        complex<double> ctemp(0,0);

        tmp[0]=local_Vs[0]/local_L[0][0];
        for(int N=1;N<M;N++){
                ctemp=local_Vs[N];
                for(int i=0;i<M;i++){
#pragma HLS PIPELINE
                        ctemp-=local_L[N][i]*tmp[i];
                        /*if(i==N-1){
                                tmp[N] = (local_Vs[N]-ctemp)/local_L[N][N];
                                ctemp=complex<double>(0,0);
                                break;
                        }*/
                }
                tmp[N] = ctemp/local_L[N][N];
                //ctemp=complex<double>(0,0);
        }

index_calz_read:
        for(int i=0;i<M;i++){
                z[i]=tmp[i];
        }
}
```



ctemp has data dependency, which prevents the function from pipelining

we've tried many ways to implement this function, but this version works the best.

```
void CALC_U(complex<double> u[M], complex<double> local_U[M][M], complex<double> z[M]){

        complex<double> tmp[M]={complex<double>(0,0)};
        complex<double> ctemp(0,0);

        tmp[M-1]=z[M-1]/local_U[M-1][M-1];
        //ctemp=z[M-2];
        for(int N=M-2;N>=0;N--){
                ctemp=z[N];
                for(int i=M-1;i>=0;i--){
#pragma HLS PIPELINE
                        ctemp-=local_U[N][i]*tmp[i];
                        /*if(i==N+1){
                                tmp[N] = (z[N]-ctemp)/local_U[N][N];
                                ctemp=complex<double>(0,0);
                                break;
                        }*/
                }
                tmp[N] = ctemp/local_U[N][N];
                //ctemp=complex<double>(0,0);
        }

index_calu_read:
        for(int i=0;i<M;i++){
                u[i]=tmp[i];
        }
}
```

```
void CALC_Z_SQUARE(complex<double> &z_sqr, complex<double> z[M]){

        complex<double> z_conj[M];
        complex<double> tmp=complex<double>(0,0);
        for(int i=0;i<M;i++){
                z_conj[i]=conj(z[i]);
                tmp+=z_conj[i]*z[i];
        }
        z_sqr=tmp;
}
```

These are the functions that solve the inhomogeneous equations.

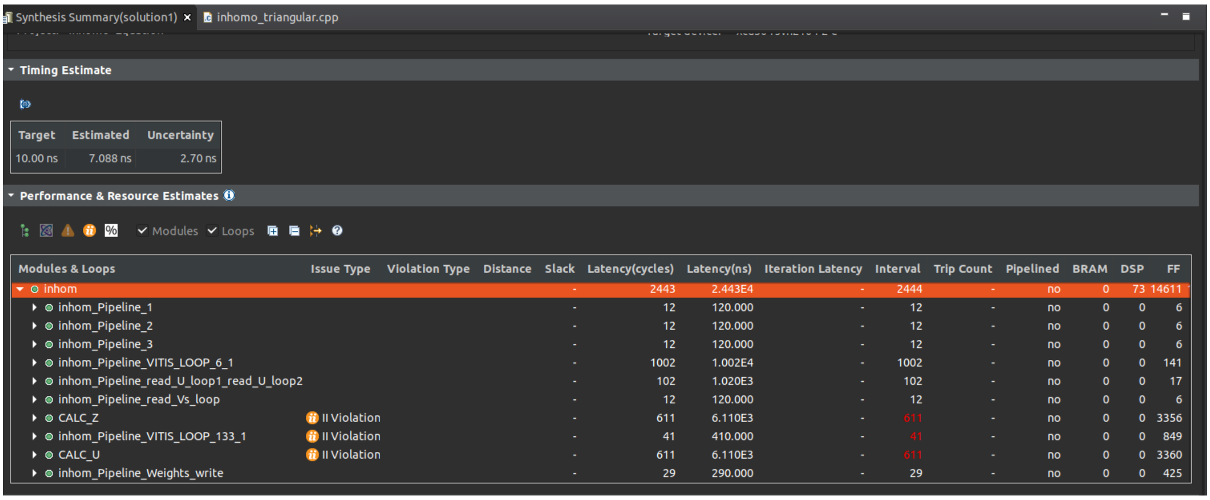
```
void inhom(hls::stream<complex<double>> &weights, hls::stream<complex<double>> &U, hls::stream<complex<double>> &Vs, hls::stream<complex<double>> &A_instream, hls::stream<complex<double>>

//#pragma HLS INTERFACE mode=ap_fifo depth=10 port=weights
//#pragma HLS INTERFACE mode=ap_fifo depth=100 port=U
//#pragma HLS INTERFACE mode=ap_fifo depth=10 port=Vs

//#pragma HLS INTERFACE mode=ap_ctrl_none port=return
//#pragma HLS STREAM depth=100 variable=U
//#pragma HLS STREAM depth=10 variable=weights
//#pragma HLS STREAM depth=10 variable=Vs
/*
#pragma HLS INTERFACE axis port=weights
#pragma HLS INTERFACE axis port=U
#pragma HLS INTERFACE axis port=Vs
#pragma HLS INTERFACE s_axilite port=return bundle=control
#pragma HLS INTERFACE s_axilite port=z_2 bundle=control
#pragma HLS INTERFACE m_axi port=z_2 offset=slave bundle=gmem
*/

complex<double> local_L[M][M];//resource limitation
#pragma HLS ARRAY_PARTITION dim=2 type=complete variable=local_L
complex<double> local_U[M][M];
#pragma HLS ARRAY_PARTITION dim=2 type=complete variable=local_U
complex<double> local_Vs[M];
complex<double> u[M]={complex<double>(0,0)};
complex<double> z[M]={complex<double>(0,0)};
complex<double> z_sqr=complex<double>(0,0);
complex<double> wt[M]={complex<double>(0,0)};;
```

```
A_stream2stream(A_outstream,A_instream);
read_fifo_U(local_L,local_U,U);
read_fifo_Vs(local_Vs,Vs);

Calc_para:

        CALC_Z(z,local_L,local_Vs);
        CALC_Z_SQUARE(z_sqr,z);
        CALC_U(u,local_U,z);




Weights_write:
        for(int i=0;i<M;i++){
                wt[i]=u[i]/z_sqr;
                weights.write(wt[i]);
        }


}
```

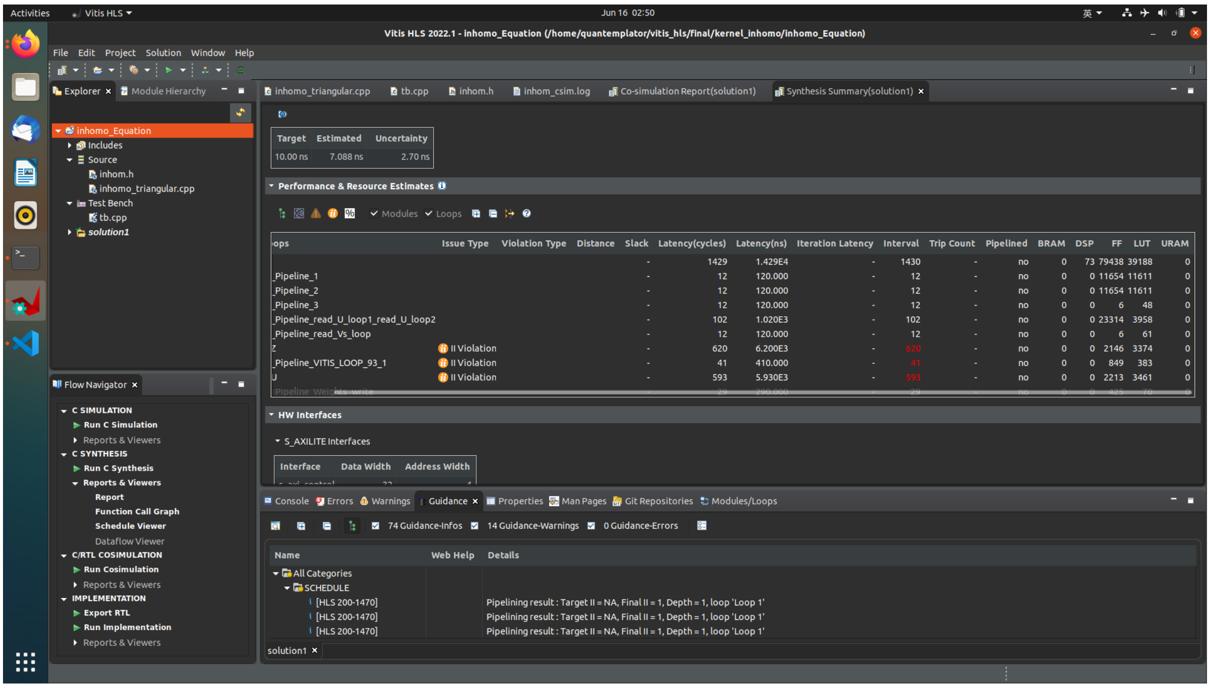
the top function of these subfunctions is called inhom

we cannot do dataflow optimization in this function since z has a single producer and multiple consumers.

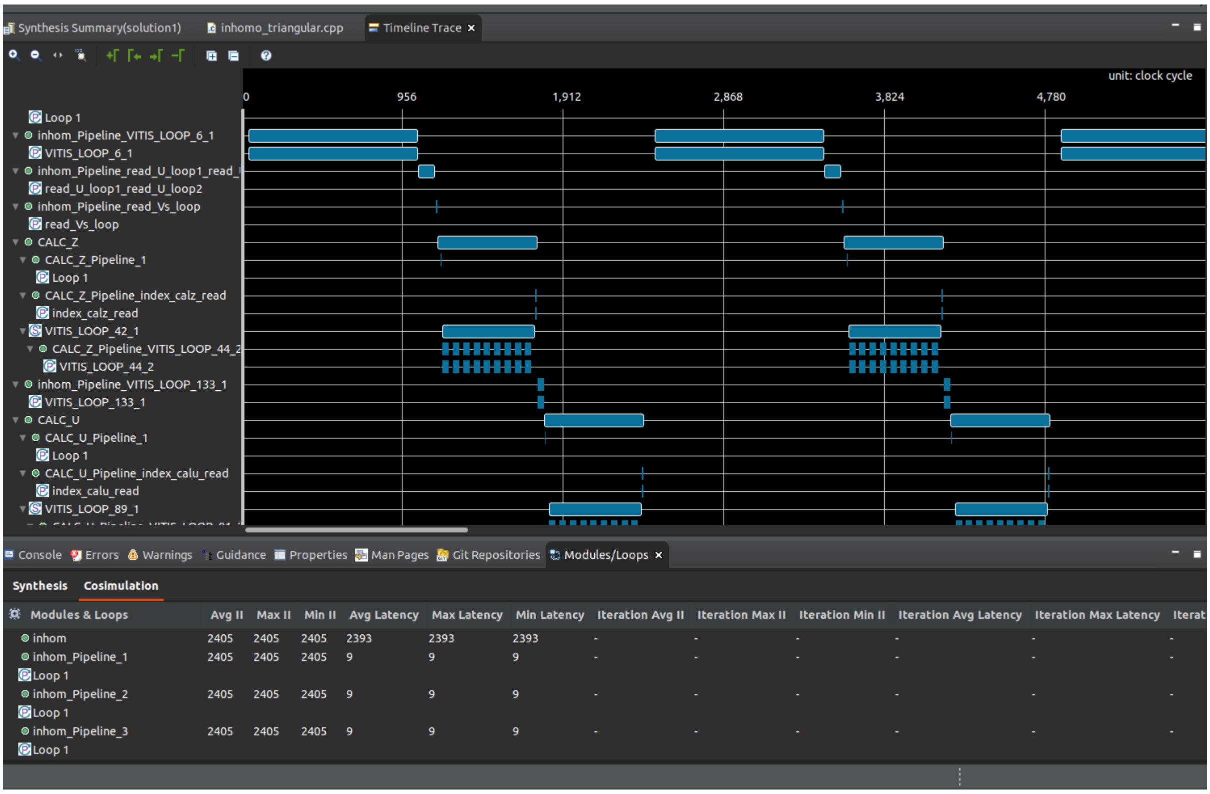
As shown in the figure 2, we propagate A until it reach the kernel that calculate the multiplication of input data A and weights, but this brings us some additional cycles from reading/writing the fifo.



synthesis with propagation of matrix A



synthesis without propagation of matrix A



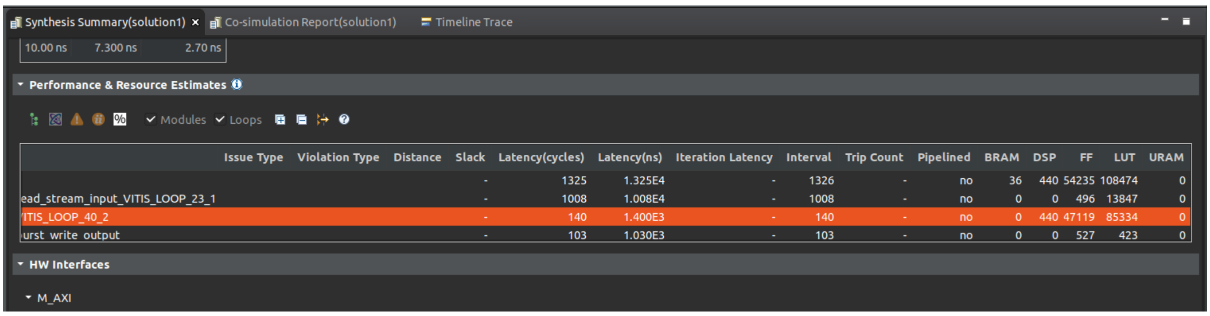
timeline trace and cosim result

5) matrix multiplication

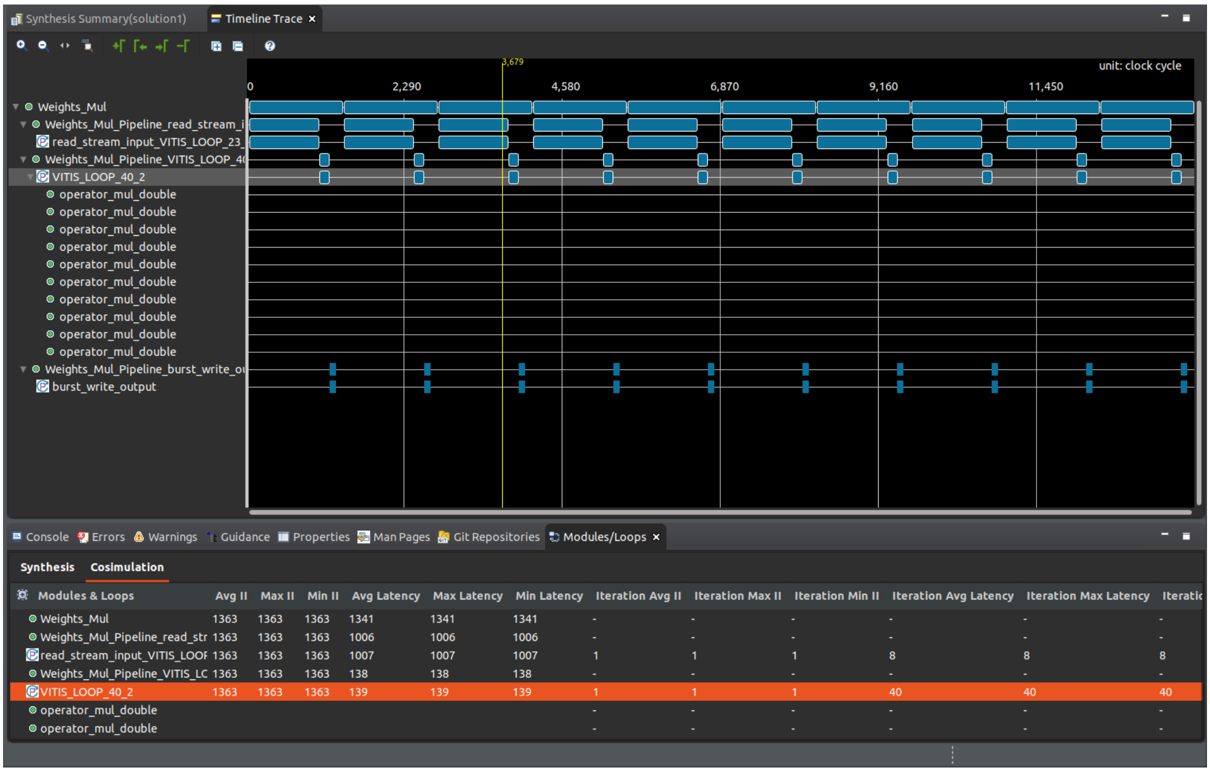
```
void Weights_Mul(complex<double> output_result[S], hls::stream<complex<double>> &A_instream, hls::stream<complex<double>> &weights_instream){

//#pragma HLS STREAM depth=1000 variable=A_instream
//#pragma HLS STREAM depth=10 variable=weights_instream
        //ADD STATIC https://docs.xilinx.com/r/2021.2-English/ug1399-vitis-hls/Array-Initialization
static  complex<double> A_matrix_w[S][M]={complex<double>(0,0)};
#pragma HLS ARRAY_RESHAPE variable=A_matrix_w type=block factor=2 dim=2
#pragma HLS BIND_STORAGE variable=A_matrix_w type=ram_1wnr

static  complex<double> INweights[M]={complex<double>(0,0)};
#pragma HLS ARRAY_RESHAPE variable=INweights type=complete
static  complex<double> results[S]={complex<double>(0,0)};
//#pragma HLS ARRAY_PARTITION variable=results type=complete


//read stream A and weights
read_stream_input:
        for(int i=0;i<S;i++){
                for(int j=0;j<M;j++){
#pragma HLS PIPELINE
                        A_matrix_w[i][j]=A_instream.read();
                        if(i==S-1){
                                INweights[j]=weights_instream.read();
                        }
                }
        }
/*
        for(int i=0;i<M;i++){
#pragma HLS PIPELINE
                for(int j=0;j<S;j++){
                        results[j]+=A_matrix_w[j][i]*weights[i];
                }
        }
*/

        for(int i=0;i<S;i++){
#pragma HLS PIPELINE
                for(int j=0;j<M;j++){
                        results[i]+=A_matrix_w[i][j]*INweights[j];
                }
        }

//burst write to host
```

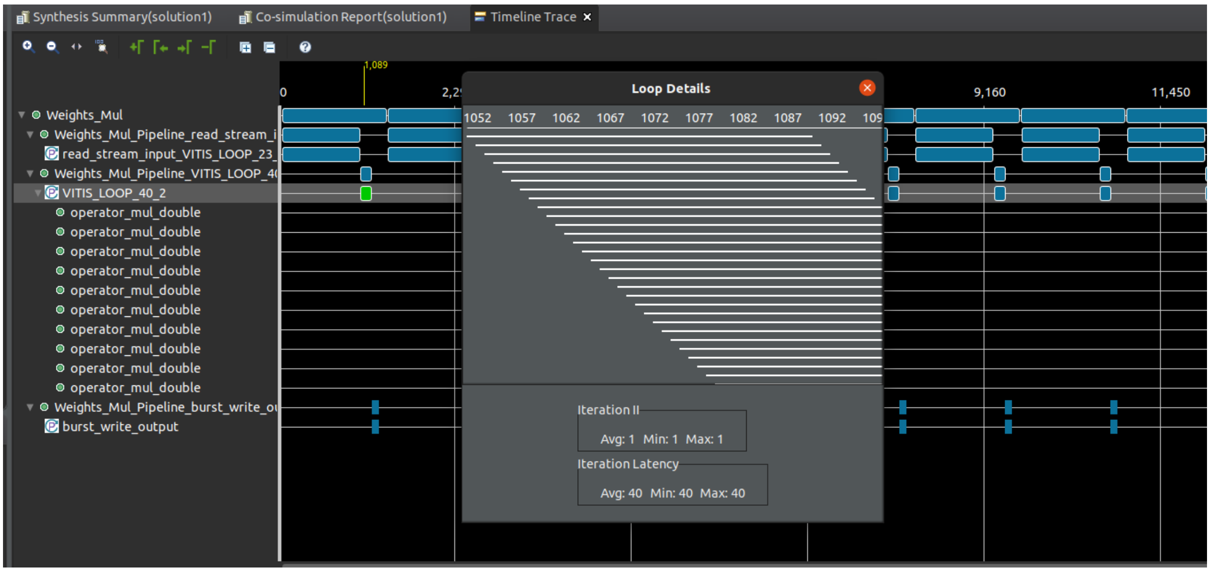
```
//burst write to host
burst_write_output:
        for(int i=0;i<S;i++){
#pragma HLS PIPELINE II=1
                output_result[i]=results[i];
        }


}
```

we used multiple port rams and array reshape to get better performance



	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
				-	1325	1.325E4	-	1326	-	no	36	440	54235	108474	0
ead_stream_input_VITIS_LOOP_23_1				-	1008	1.008E4	-	1008	-	no	0	0	496	13847	0
'ITIS_LOOP_40_2				-	140	1.400E3	-	140	-	no	0	440	47119	85334	0
urst_write_output				-	103	1.030E3	-	103	-	no	0	0	527	423	0

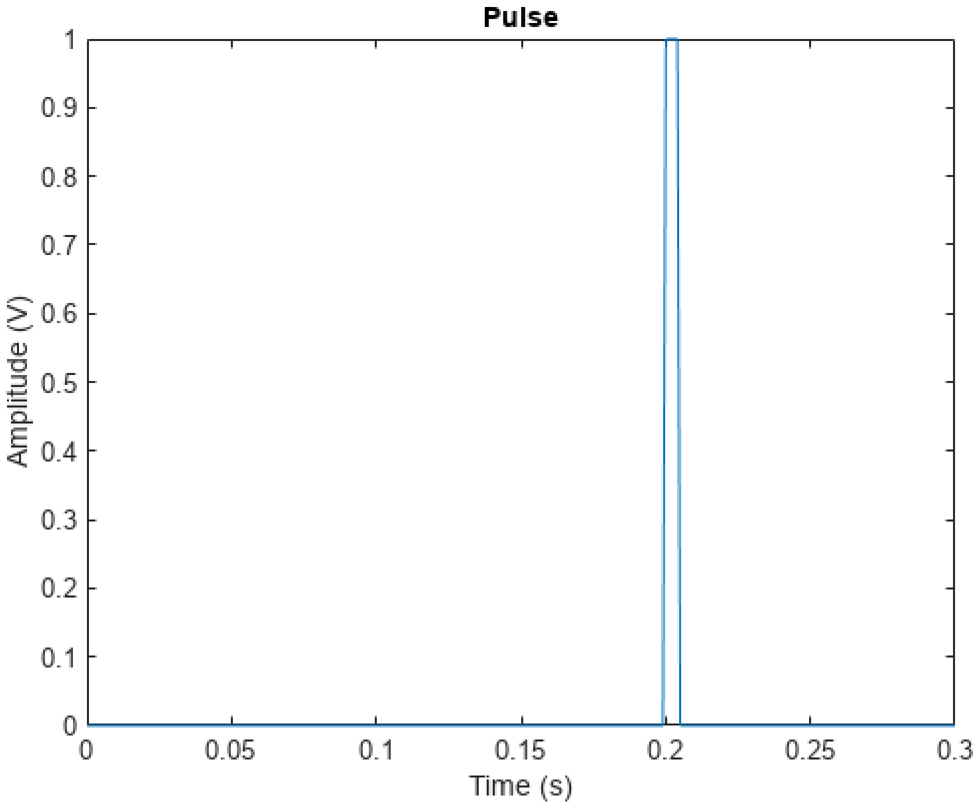




The above figures shows that we have greatly reduced operation time (to 140 cycles) for multiplication of two matrices A and weights.

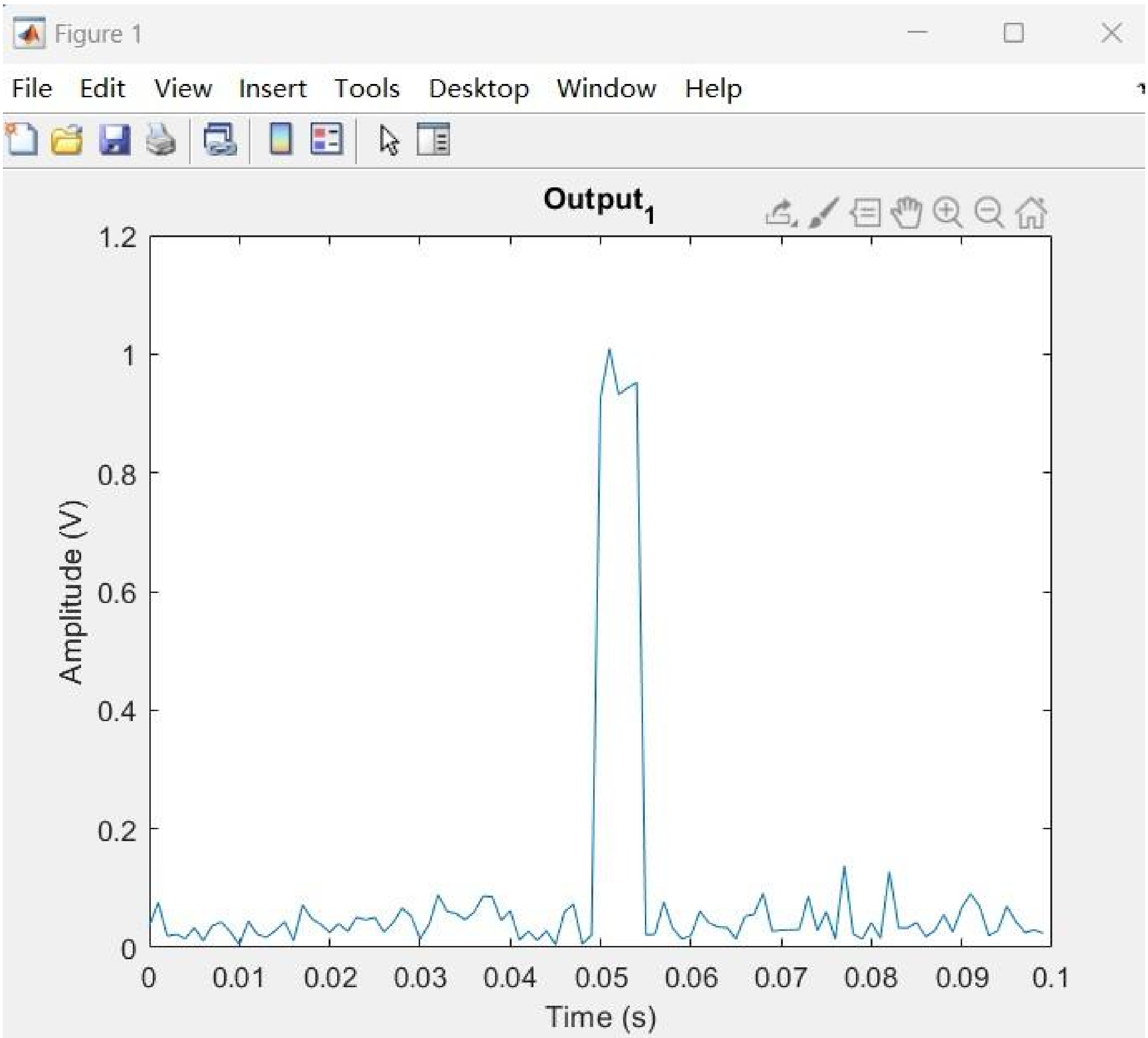
4. result analysis

expected output is: (just as example, our input settings are different)



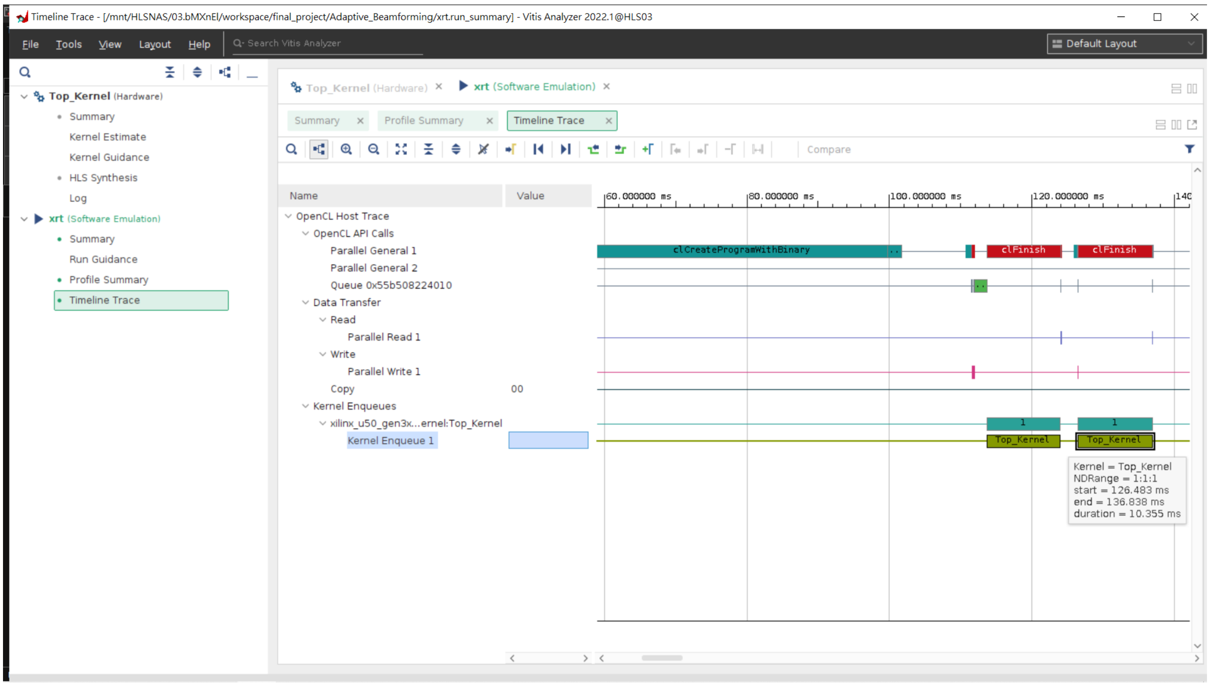
extracted from MATLAB: https://www.mathworks.com/help/phased/ug/conventional-and-adaptive-beamformers.html

The output we obtain is:

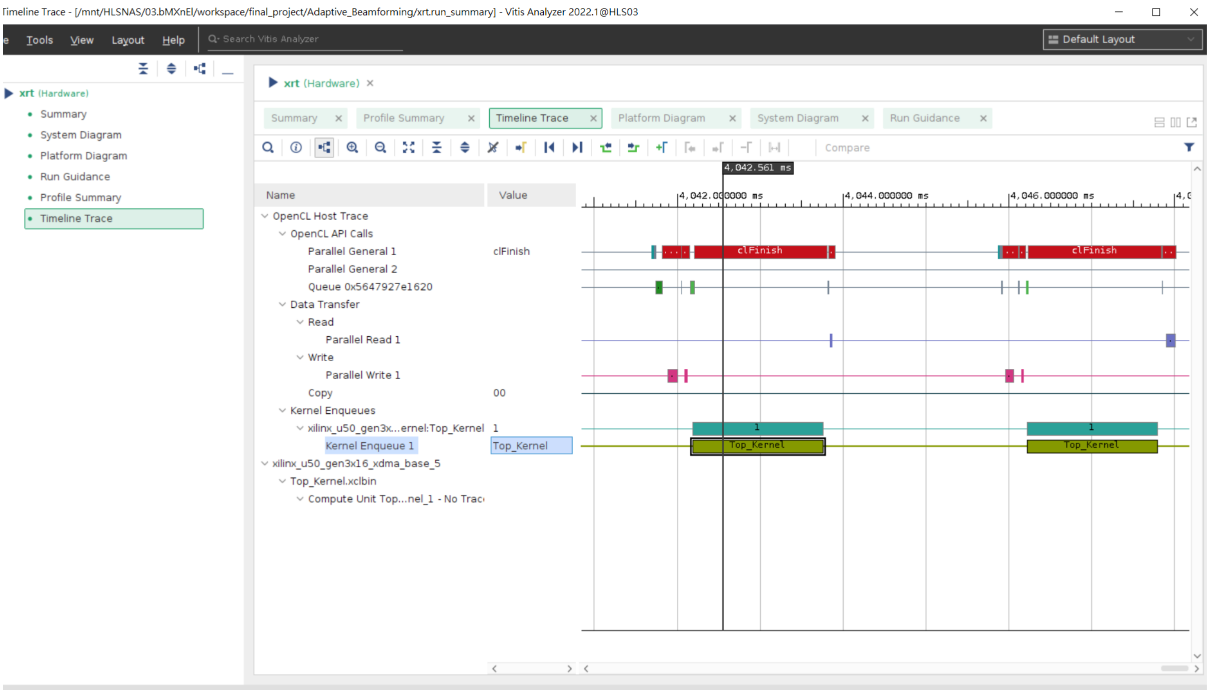


which is very close to what we want.

software emulation: (kernel execution time=10.5ms)



hardware: (kernel execution time=1~1.5ms)



The acceleration rate (sw/hw) is about seven to ten.

However, it’s worth mentioning that the hardware emulation and hardware implementation need a lot of time, more than hours,on the vitis platform to obtain the results.

synthesis report:

```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+--------+---------+-----------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+--------+---------+-----------+
    |ap_clk  |  3.33 ns|  2.433 ns|     0.90 ns|
    +--------+--------+---------+-----------+

+ Latency:
    * Summary:
    +---------+--------+----------+---------+-------+--------+---------+
    |  Latency (cycles) |  Latency (absolute)  |    Interval    | Pipeline|
    |   min   |   max   |    min    |    max   |  min  |   max  |   Type  |
    +---------+--------+----------+---------+-------+--------+---------+
    |    22700|   634108|  75.659 us|  2.113 ms|  22701|  634109|       no|
    +---------+--------+----------+---------+-------+--------+---------+

    + Detail:
        * Instance:
        +-------------------------+-------------+--------+--------+----------+---------+-------+--------+----------+
        |                         |             |  Latency (cycles) |  Latency (absolute)  |    Interval    | Pipeline |
        |         Instance        |    Module   |   min  |   max   |    min    |    max   |  min  |   max  |   Type   |
        +-------------------------+-------------+--------+--------+----------+---------+-------+--------+----------+
        |grp_pass_dataflow_fu_112  |pass_dataflow |    22698|   634106|  75.652 us|  2.113 ms|  22557|  633965|  dataflow|
        +-------------------------+-------------+--------+--------+----------+---------+-------+--------+----------+

        * Loop:
        N/A
```

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+------+---------+--------+-----+
|         Name        | BRAM_18K|  DSP |    FF   |   LUT  | URAM|
+---------------------+---------+------+---------+--------+-----+
|DSP                  |        -|     -|        -|       -|    -|
|Expression           |        -|     -|        0|       6|    -|
|FIFO                 |        -|     -|        -|       -|    -|
|Instance             |      147|   638|   167193|  151640|    0|
|Memory               |        -|     -|        -|       -|    -|
|Multiplexer          |        -|     -|        -|      92|    -|
|Register             |        -|     -|      202|       -|    -|
+---------------------+---------+------+---------+--------+-----+
|Total                |      147|   638|   167395|  151738|    0|
+---------------------+---------+------+---------+--------+-----+
|Available SLR        |     1344|  2976|   871680|  435840|  320|
+---------------------+---------+------+---------+--------+-----+
|Utilization SLR (%)  |       10|    21|       19|      34|    0|
+---------------------+---------+------+---------+--------+-----+
|Available            |     2688|  5952|  1743360|  871680|  640|
+---------------------+---------+------+---------+--------+-----+
|Utilization (%)      |        5|    10|        9|      17|    0|
+---------------------+---------+------+---------+--------+-----+
```

github:
https://github.com/ccontemplator/Adaptive-MPDR-Beamforming/tree/main

5.Reference

[1] Chengxin Xu, Ting Song, Xiangmei Li. Adaptive Beamforming Based on QR Decomposition in Smart Antenna. Proceedings of the 2018 3rd International Workshop on Materials Engineering and Computer Sciences, 2018. doi:10.2991/iwmecs-18.2018.64