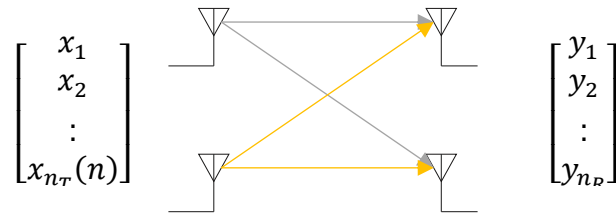


Computer Architecture Final Project

MMSE detector implemented on GPU system

1. Background :

MIMO technology is very basic in modern communication system.



As more transmission antenna increase in the transmitter, the more column vector that the transmission Matrix has.

Then we can form the transmission Matrix as below process :

$$\begin{bmatrix} h_{11} \\ h_{21} \\ \vdots \\ h_{n_R 1} \end{bmatrix} x_1(n) + \begin{bmatrix} h_{12} \\ h_{22} \\ \vdots \\ h_{n_R 2} \end{bmatrix} x_2(n) + \dots + \begin{bmatrix} h_{1 n_T} \\ h_{2 n_T} \\ \vdots \\ h_{n_R n_T} \end{bmatrix} x_{n_T}(n) = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n_R} \end{bmatrix}$$

Then we integrate all the channel effect of transmitter and receiver to form a neat matrix form :

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ \vdots & \vdots & \vdots \\ h_{n_R 1} & h_{n_R 2} & h_{n_R 3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_T}(n) \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n_R} \end{bmatrix}$$

However, how can we eliminate the channel gain matrix in the receiver? We can deploy the MMSE detector to receive the signal in the receiver.

◆ MMSE detector

In the channel model:

$$y = Hx + w$$

Detector aims to get the signal x by the channel model. Therefore, we can apply the MMSE principle to get the signal x :

$$\min_w E\{(x - Wy)^H(x - Wy)\}$$

The physical meaning of this formula is that we aim to find a transformation W that we can make the distance between x and Wy has the minimum distance.

Then we will get the below solution:

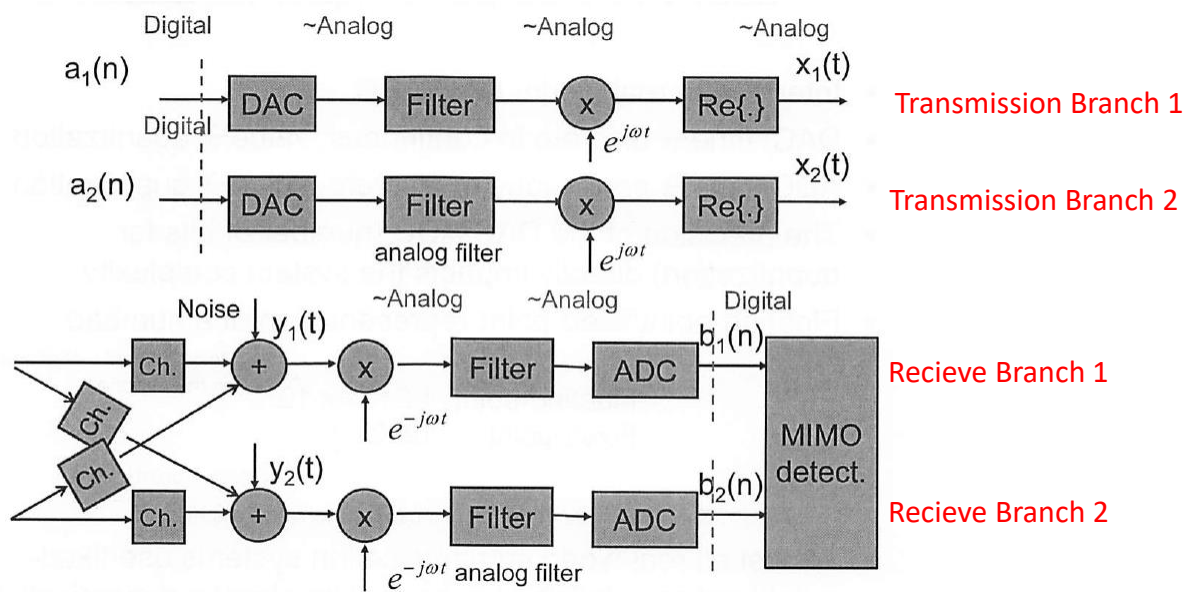
$$W = H^H(HH^H + \rho^{-1}I)^{-1}, \text{ if } n_T > n_R$$

$$W = (H^H H + \rho^{-1}I)^{-1}H^H, \text{ if } n_T \leq n_R$$





By the core design principle of this detector, we can get the closely approach of signal x by

$$x = Wy$$

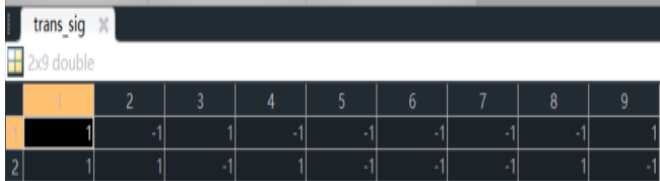
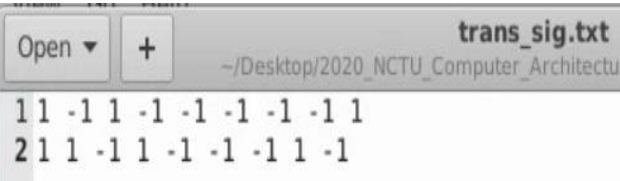
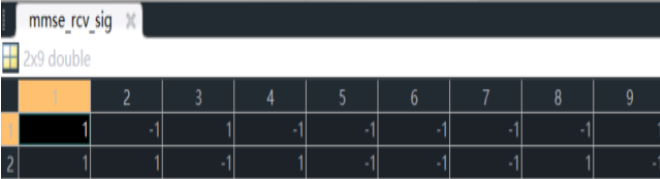
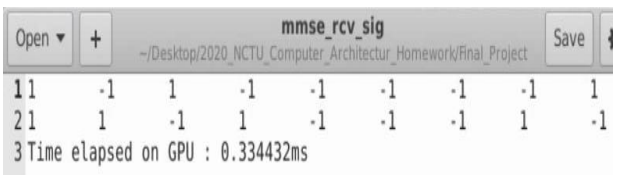
The overall system diagram is showed as below:



2. Experiment enviroment and file system :

MMSE detector	
Enviroment	UBUNTU 18.04
GPU	RTX 1080 8G RAM
File System	
 matlab_version. m	Verification code based on matlab
 mmse_detector. cu	MMSE Detector excuting on GPU
 ori_signal.txt	Transmitted signal in the transmitter side
 recieved_signal	Received signal in the recuever side MMSE demodulation result

3. Verification with Matlab :

	Matlab	CUDA
Transmitter	 <pre> trans_sig 2x9 double 1 2 3 4 5 6 7 8 9 1 -1 1 -1 -1 -1 -1 -1 1 2 1 1 -1 1 -1 -1 -1 -1 </pre>	 <pre> trans_sig.txt ~/Desktop/2020_NCTU_Computer_Architectu 1 1 -1 1 -1 -1 -1 -1 1 2 1 1 -1 1 -1 -1 -1 1 </pre>
Reciever	 <pre> mmse_rcv_sig 2x9 double 1 2 3 4 5 6 7 8 9 1 -1 1 -1 -1 -1 -1 -1 1 2 1 1 -1 1 -1 -1 -1 1 </pre>	 <pre> mmse_rcv_sig ~/Desktop/2020_NCTU_Computer_Architectur_Homework/Final_Project 1 1 -1 1 -1 -1 -1 -1 1 2 1 1 -1 1 -1 -1 -1 1 3 Time elapsed on GPU : 0.334432ms </pre>

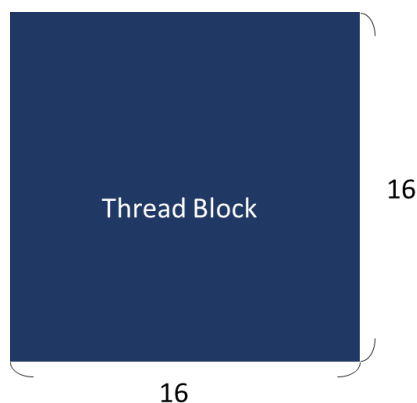
4. Some program explanation :

- Dimension Segmentation :

```
unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;  
unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;  
dim3 dimGrid(grid_cols, grid_rows);  
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
```

- Block_Size is set as 16
- Both m and n are 2
- K = 7
- grid_rows is $\lfloor (2 + 16 - 1) / 16 \rfloor = 1$
- grid_cols is $\lfloor (7 + 16 - 1) / 16 \rfloor = 1$
- Therefore we can new
- Block size is 1X1
- Thread size per block is 16X16

We assume all the task should be handled with arbitrary matrix, therefore we should resolve different operation in various way.



In my project , I used the matrix shape is showed as below :

$$(H_{2 \times 2}^H H_{2 \times 2} + \rho^{-1} I_{2 \times 2})^{-1} H_{2 \times 2}^H Y_{2 \times 7} = X_{2 \times 7}$$

Where

- $H_{2 \times 2}$: channel effect of MIMO
- $I_{2 \times 2}$: identity matrix
- $X_{2 \times 7}$: signal matrix
- $Y_{2 \times 7} = H_{2 \times 2} * X_{2 \times 7}$: signal matrix with channel effect.

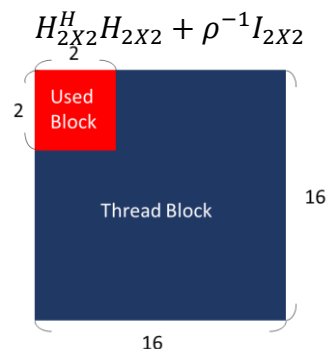
- Matrix Add

- Kernel code and explanation :

```
/* GPU matrix add*/
__global__ void gpu_add_matrix(double* matrixA, double* matrixB, double* matrixC, unsigned int row, unsigned int col)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    int j = blockDim.y*blockIdx.y + threadIdx.y;

    if (i < row && j < col)
    {
        matrixC[i*row+j] = matrixA[i*row+j] + matrixB[i*row+j];
    }
}
```

Matrix add put the element onto the 2 dimensional thread. I use a 16X16 thread to parallelize the calculation. I plot this diagram for my case as below :



I use large enough block in my implementation for general purpose.

- Matrix transpose

- Kernel code and explanation :

```
__global__ void gpu_matrix_transpose(double* mat_in, double* mat_out, unsigned int rows, unsigned int cols)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < cols && idy < rows)
    {
        unsigned int pos = idy * cols + idx;
        unsigned int trans_pos = idx * rows + idy;
        mat_out[trans_pos] = mat_in[pos];
    }
}
```

I use the same partition method as the adding matrix method. The difference between transpose and adding is the core calculation. I just replace pair of their position. I used this method for the below calculation :

$$H_{2 \times 2}^H$$

- Matrix square multiplication
 - Kernel code and explanation :

```
__global__ void gpu_square_matrix_mult(double *left, double *right, double *res, int dim) {
    int i,j;
    float temp = 0;
    __shared__ float Left_shared_t [BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Right_shared_t[BLOCK_SIZE][BLOCK_SIZE];
    // Row i of matrix left
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int tileNUM = 0; tileNUM < gridDim.x; tileNUM++) {
        // Column j of matrix left
        j = tileNUM * BLOCK_SIZE + threadIdx.x;
        i = tileNUM * BLOCK_SIZE + threadIdx.y;
        // Load left[i][j] to shared mem
        Left_shared_t[threadIdx.y][threadIdx.x] = left[row * dim + j]; // Coalesced access
        // Load right[i][j] to shared mem
        Right_shared_t[threadIdx.y][threadIdx.x] = right[i * dim + col]; // Coalesced access
        // Synchronize before computation
        __syncthreads();
        // Accumulate one tile of res from tiles of left and right in shared mem
        for (int k = 0; k < BLOCK_SIZE; k++) {
            temp += Left_shared_t[threadIdx.y][k] * Right_shared_t[k][threadIdx.x]; //no shared memory bank conflict
        }
        // Synchronize
        __syncthreads();
    }
    // Store accumulated value to res
    res[row * dim + col] = temp;
}
```

In my case I used the following calculation :

$$H_{2 \times 2}^H H_{2 \times 2}$$

This algorithm handles all matrices as square matrix. In the kernel because of the shared memory usage and its size limitations I have found solution named “tiling”. By dividing the matrices to square tiles algorithm finds the one part of the resulting element and then considering other tiles and their result it finds one element of the resulting matrix. While using tiling solution and shared memory, there are two important things: Coalesced memory access and bank conflict. In order to prevent from uncoalesced access tiles are taken from global memory to shared memory row by row by as big as block size. When reaching to shared memory matrices elements corresponds to different banks which can be seen from code, so bank conflict is prevented in this way.

Another important point using shared memory is synchronization of threads. `__syncthreads()` is used in order to fill shared memory before calculation start. If calculation phase starts before filling the shared memory threads will reach to empty places.

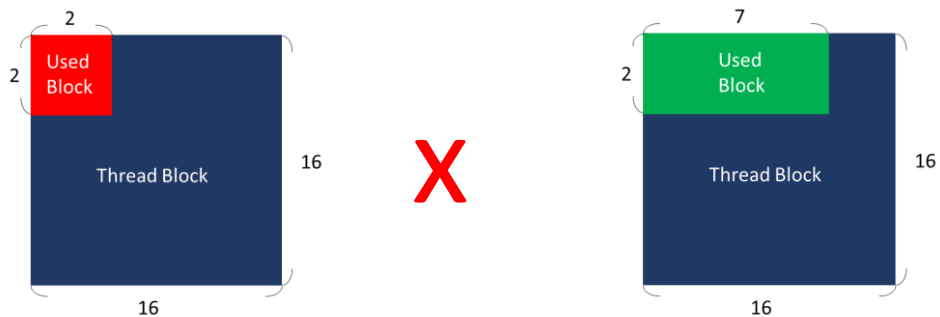
- Matrix non-square multiplication

- Kernel code and explanation :

```
void cpu_matrix_mult(double *h_a, double *h_b, double *h_result, int m, int n, int k) {
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int tmp = 0.0;
            for (int h = 0; h < n; ++h)
            {
                tmp += h_a[i * n + h] * h_b[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}
```

In my implementation I used this function to calculate the following formula :

$$(H_{2 \times 2}^H H_{2 \times 2} + \rho^{-1} I_{2 \times 2})^{-1} H_{2 \times 2}^H \times Y_{2 \times 7}$$



Both $i * n + h$ and $h * k + j$ will handle the 2×2 and 2×7 matrix row and column just like the below diagram.

