# ECE 790
# Master's Research

Prof. Yu-Hen Hu

## Acceleration of MP3 encoder by using the GPU

by

Hsin-Yu Chen

# Abstract

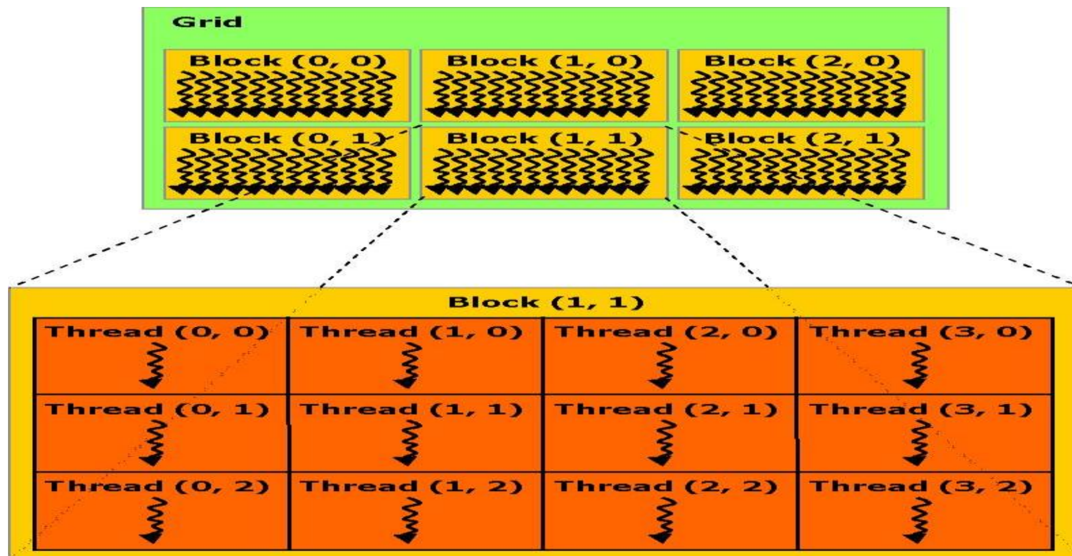
At the present time, CUDA is a useful and common computing engine used by many people. It accelerates the computing speed by running simply same tasks at the same time which is called the parallel processing. Many scientists use it as a coprocessor to decrease the computing time in various areas such as image processing, weather forecast or heart wall tracking.

Lame is a high quality and an efficient MP3 encoder. Also, it is an open source software which makes it the most popular MP3 encoder in the world. In 2008, NVIDIA had held a contest for accelerating Lame by CUDA. Only 2 out of 181 teams handed the program. The champion team got 2.4x speed up in GPU compared to CPU. It seems like there are still a lot of room to improve. In this project, I am going to use the CUDA to accelerate the lame MP3 encoder and figure out which parts are worth to do.

# Background

## CUDA

CUDA is the computing engine in NVIDIA graphics processing units or GPUs that is accessible to software developers through industry standard programming languages. The computing method in GPU is parallel, the basic computing unit in GPU is thread which is contained in the block and the block is organized into a grid. The user may choose to use 1-dimensional, 2-dimensional or 3-dimensional grid, block or thread to computing the data. The figure[7] below shows how it works. Unlike the CPU, GPU can process the data which works sequentially in CPU at the same time. That is the reason why GPU can make computing faster.




## MP3

### Overview:

MP3 (MPEG-1 Layer 3) was stated by International Organization for Standardization (ISO). The main purpose of it is to reduce the amount of data without any noticeable quality loss. Most humans cannot sense frequencies below 20 HZ nor above 20000 HZ. By using this fact, we can throw away a lot of data which we cannot percept and make it smaller than the original data.

All MP3 files are divided into smaller fragments called frames. Each frame stores 1152 audio samples. In addition a frame is subdivided into two granules each containing 576 samples. A frame consists of five parts; header, CRC, side information, main data and ancillary data as the figure[1] shown below.

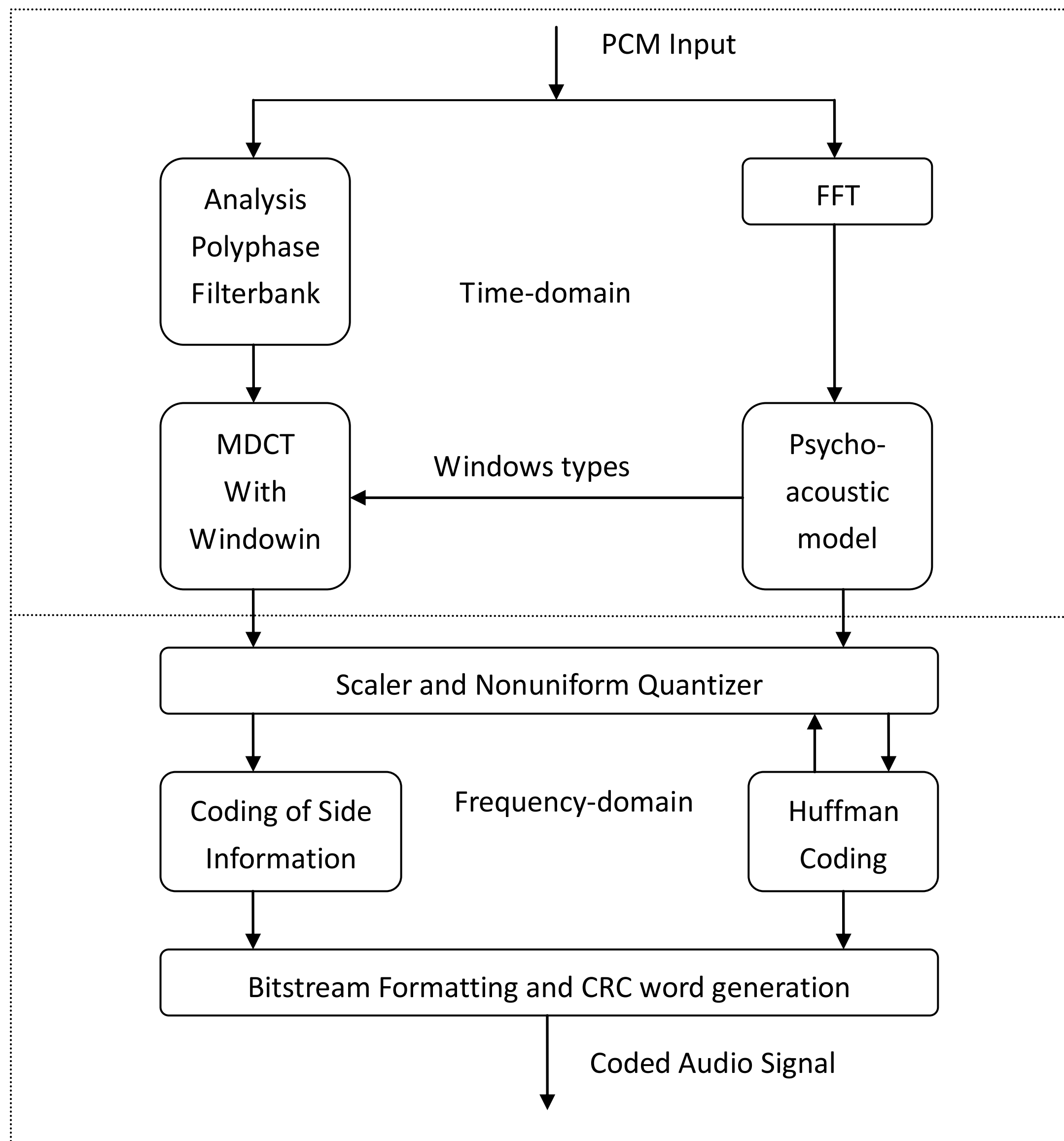
| Header | CRC | Side information | Main Data | Ancillary Data |
|---|---|---|---|---|

The header is 32 bits long and the main function of it is to tell the decoder how to decode the MP3 file. The CRC field is to check the transmission errors. If these values are incorrect, they will corrupt the whole frame. The side information part consists of information needed to decode the main data. The main data part consists of scale factors and Huffman coded bits. The purpose of scale factors is to reduce the quantization noise. The Huffman coded bit store the information after Huffman coding. The ancillary data ranges to where the next frame's

main_data_begin points to.

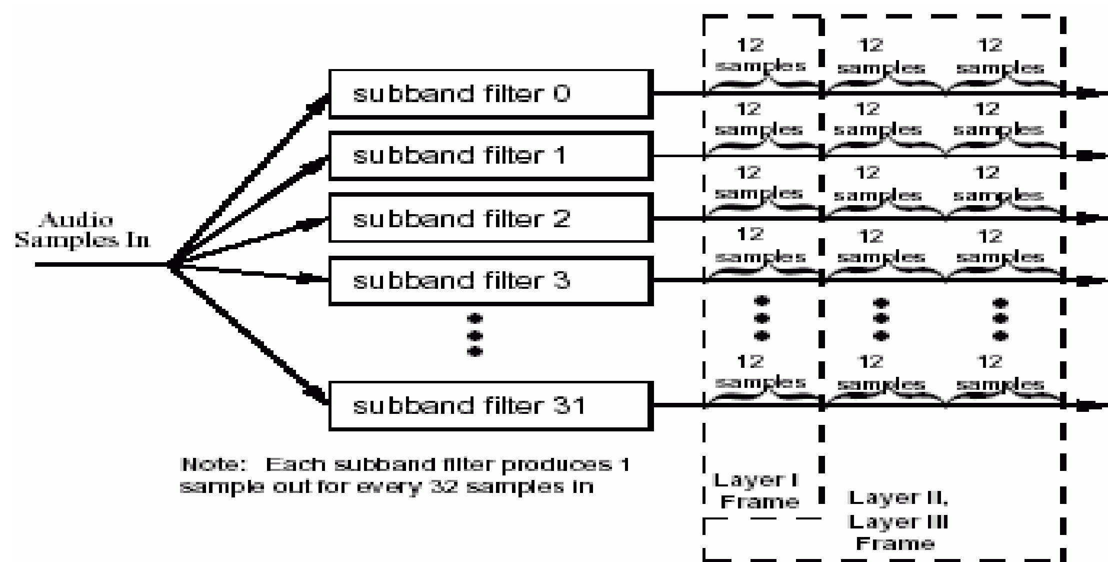
## Encoding:

The figure[1] below is the MP3 encoding scheme.




### *Analysis Polyphase Filterbank*

The purpose of the filterbank is to divide the audio signal into 32 equal-width frequency subbands depending of the Nyquist frequency of the PCM signal. A sequence of 1152 PCM samples are filtered into these filters and groups 3 groups of 12 samples per sub-band as the figure[6] below.




***Modified discrete cosine transform (MDCT)***

The MDCT can compensate the aliasing introduced by the sub-sampling in the filterbank. The 32 subband signals are subdivided further in frequency content by applying a 18-spectral point or 6-spectral point MDCT. There are two different MDCT block lengths: a long block or a short block. Block switching is triggered by psycho acoustics

Long blocks have a higher frequency resolution. Each sub-band is transformed into 18 spectral coefficients by MDCT, yielding a maximum of 576 spectral coefficients each representing a bandwidth of 41.67Hz at 48 kHz sampling rate. There is a 50% overlap between successive transform windows, so the window size is 36 for long blocks.

Short blocks have a higher time resolution. Short block length is one third of a long block and used for transients to provide better time resolution. Each sub-band is transformed into 6 spectral coefficients by MDCT, yielding a maximum of 192 spectral coefficients each representing a bandwidth of 125Hz at 48kHz sampling rate. There is a 50% overlap between successive transform windows, so the window size is 12 for short blocks.

***FFT***

The PCM data also transformed to frequency domain by a Fast Fourier Transform. Both a 1024 and a 256 point FFT are performed on 1152 PCM sample for the input of the psychoacoustic model.

***Psychoacoustic Model***

This model calculates the noise threshold which is the information for determining the quantization step size. It also decides the windows types for the MDCT.

*Scaler and Nonuniform Quantizer*

There are two nested loops, a distortion control loop and a rate control loop in this block. Rate control loop does the quantization of the frequency domain samples and thus also determines the required quantization step size. Distortion control loop control the quantization noise which is produced by the quantization of the frequency domain lines within the rate control loop.

*Huffman Coding*

In this part, the quantized values are Huffman coded. This step can reduce the data size without loss any information.

*Coding of Side Information*

The parameters produced by the encoder are stored here. It provides the decoder to reproduce the audio signal.

*Bitstream Formatting CRC word generation*

The components of frame, header, side information, CRC, Huffman coded bit are put together to form frames here.

# Simulation

My system is based on the windows Microsoft visual C++ 2008, and I use the lame MP3 encoder 398.2 version to implement by CUDA. I focus on some parts, FFT, psychoacoustic model, quantizer and the transforming part from input buffer to sample_t. These parts are corresponding to lame as FFT→fft.c, psychoacoustic model→psymodel.c, quantizer→quantize.c and transforming part→lame.c.

The reason why I choose these parts is that they use much more “for” loops compared to other parts. The “for” loops are time consuming in CPU. I think it is worth to rewritten them into CUDA. The codes are in appendix.

# Results

The PC spec I use is Intel Core i7-920, 2.66 GHZ and 32 bit windows 7 system. The GPU is N275GTX TwinFrozr OC, 666 MHZ, 240 cores and 896 MB. My wav data is 61MB, 1411kbps. I use the V2 option 128 k bitrates to encode the wav file. The encoding time is 41 sec in CPU. After I implement the CUDA codes, some indeed improve a little bit, but some don’t. Here are my results:FFT(fft.cu)→40.5 sec, psychoacoustic model(psymodel.cu)→40.8 sec, quantizer(quantize.cu)→43 sec, transforming part(lame.cu)→47 sec.

# Conclusion

In the result, the quantizer and transforming parts can't be improved by GPU. The main reason is that the data transferring time from CPU to GPU takes too long. The GPU can't compensate the transferring time. In the other hand, FFT and psychoacoustic model take the advantage of the GPU. Although the increasing time is not obvious, it is worth to implement CUDA in these parts. There is still room for improvement, either the optimization of CUDA codes or other worthy part in lame for CUDA. In fact, my GPU is not the best one, I think if I can implement the codes with the newest GPU, the performance might be much better.

# References


[1] Rassol Raissi, The Theory Behind MP3, December 2002.

[2] Davis Yen Pan, Digital Audio Compression, Digital Technical journal Vol. 5 No. 2, Spring 1993.

[3] International Organization for Standardization webpage, http://www.iso.ch

[4] MP3 Tech, http://www.mp3-tech.org

[5] ISO/IEC 11172-3 "Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s – Part 3"

[6] The hydrogenaudio knowledgebase, http://wiki.hydrogenaudio.org

[7] NVIDIA_CUDA_Programming_Guide_3.0

[8] CUDA_Reference_Manual_3.0

[9] Lame, http://lame.sourceforge.net/


# Appendix

## fft.c

```
.
.
.
init_fft(lame_internal_flags * const gfc)
{
    int      i;

    /* The type of window used here will make no real difference, but */
    /* in the interest of merging nspsytune stuff - switch to blackman window */
cuda_fft( window, window_s);
        /* blackman window
/*    for (i = 0; i < BLKSIZE; i++)
        window[i] = 0.42 - 0.5 * cos(2 * PI * (i + .5) / BLKSIZE) +
            0.08 * cos(4 * PI * (i + .5) / BLKSIZE);

    for (i = 0; i < BLKSIZE_s / 2; i++)
        window_s[i] = 0.5 * (1.0 - cos(2.0 * PI * (i + 0.5) / BLKSIZE_s));*/
```

Add this part in original code

Comment this part

```
#ifdef HAVE_NASM
    if (gfc->CPU_features.AMD_3DNow) {
        gfc->fft_fht = fht_3DN;
    }
    else if (gfc->CPU_features.SSE) {
        gfc->fft_fht = fht_SSE;
    }
    else
#endif
    {
        gfc->fft_fht = fht;
    }
}
.
.
.
```

## fft.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define PI 3.14159265358979323846
#define size 1024
#define size_s 256
#define tw 32
#define th 16
#define bh 2
#define bw size / (bh * th * tw)
#define bsh 1
#define bsy size_s / (bsh * th * tw /4)

__global__ void kernel_a( float* a )
{
int bx = blockIdx.x;
int by = blockIdx.y;

int tx = threadIdx.x;
int ty = threadIdx.y;
int idx = ( bx * bw + by ) * ( th * tw ) + ( tx * tw + ty );

a[idx] = 0.42 - 0.5 * cos( 2 * PI * (idx + .5) / size ) +
0.08 * cos( 4 * PI * (idx + .5) / size );
}

__global__ void kernel_b( float* b )
{
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
int idx = ( bx * bsy + by ) * ( th * tw / 4 ) + ( tx * tw + ty );

b[idx] = 0.5 * (1.0 - cos(2.0 * PI * (idx + 0.5) / size_s));
}

extern "C"
void cuda_fft( float* a, float* b )
{
int data_size_a = size * sizeof(float);
int data_size_b = size_s * sizeof(float) / 2;

float* dev_A;
float* dev_B;

cudaMalloc( (void**)&dev_A, data_size_a ) ;
cudaMalloc( (void**)&dev_B, data_size_b ) ;

dim3 threads_a( th, tw );
```

```
dim3 grid_a( bh, bw );
kernel_a<<< grid_a, threads_a >>>( dev_A );

dim3 threads_b( th/2, tw/2 );
dim3 grid_b( bsh, bsy );
kernel_b<<< grid_b, threads_b >>>( dev_B );

cudaMemcpy( a, dev_A, data_size_a, cudaMemcpyDeviceToHost ) ;
cudaMemcpy( b, dev_B, data_size_b, cudaMemcpyDeviceToHost ) ;
cudaFree( dev_A ) ;
cudaFree( dev_B ) ;
}
```

## Psymodel.c

```
.
.
.
/*****************************************************************
  * determine the block type (window type)
  *****************************************************************/
        /* calculate energies of each sub-shortblocks */
        for (i = 0; i < 3; i++) {
            en_subshort[i] = gfc->nsPsy.last_en_subshort[chn][i + 6];
            assert(gfc->nsPsy.last_en_subshort[chn][i + 4] > 0);
            attack_intensity[i]
                = en_subshort[i] / gfc->nsPsy.last_en_subshort[chn][i + 4];
            en_short[0] += en_subshort[i];
        }

        if (chn == 2) {
                cuda_block_type(ns_hpfsmpl[0], ns_hpfsmpl[1]);
         /*   for (i = 0; i < 576; i++) {
                FLOAT   l, r;
                l = ns_hpfsmpl[0][i];
                r = ns_hpfsmpl[1][i];
                ns_hpfsmpl[0][i] = l + r;
                ns_hpfsmpl[1][i] = l - r;
            }*/
        }
.
.
.
```

Add this part in original code

comment this part

## Psymodel.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define sqrt2 1.41421356237309504880
#define size 576
#define th 16
#define tw 32
#define bh 3
#define bw 6

__global__ void kernel_block (float * xr0, float * xr1)
{
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
int idx = ( bx * bw + by ) * ( th * tw ) + ( tx * tw + ty );
__shared__ float l;
__shared__ float r;
 l = xr0[idx];
 r = xr1[idx];
xr0[idx] = (l + r);
xr1[idx] = (l - r);
```

```
}

extern "C"
void cuda_block_type(float * xr0, float * xr1)
{
float * dev_xr0;
float * dev_xr1;

int data_size = size * sizeof(float);

cudaMalloc( (void**)&dev_xr0, data_size);
cudaMalloc( (void**)&dev_xr1, data_size);

cudaMemcpy( dev_xr0, xr0, data_size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_xr1, xr1, data_size, cudaMemcpyHostToDevice );

dim3 threads( th, tw );
dim3 grid( bh, bw);

kernel_block<<< grid, threads >>>(dev_xr0, dev_xr1);

cudaMemcpy( xr0, dev_xr0, data_size, cudaMemcpyDeviceToHost );
cudaMemcpy( xr1, dev_xr1, data_size, cudaMemcpyDeviceToHost );

cudaFree( dev_xr0 );
cudaFree( dev_xr1 );

}
```

## Lame.c

```
.
.
.
lame_encode_buffer_int(lame_global_flags * gfp,
                       const int buffer_l[],
                       const int buffer_r[],
                       const int nsamples, unsigned char *mp3buf, const int mp3buf_size)
{
    lame_internal_flags *const gfc = gfp->internal_flags;
    int     i,channel;
    sample_t *in_buffer[2];

    if (gfc->Class_ID != LAME_ID)
        return -3;

    if (nsamples == 0)
        return 0;

    if (update_inbuffer_size(gfc, nsamples) != 0) {
        return -2;
    }

    in_buffer[0] = gfc->in_buffer_0;
    in_buffer[1] = gfc->in_buffer_1;
channel = gfc->channels_in;
cuda_sample_t(nsamples, in_buffer[0], in_buffer[1], buffer_l, buffer_r, channel);
    /* make a copy of input buffer, changing type to sample_t */
    /*for (i = 0; i < nsamples; i++) {
        /* internal code expects +/- 32768.0 */
       in_buffer[0][i] = buffer_l[i] * (1.0 / (1L << (8 * sizeof(int) - 16)));
        if (gfc->channels_in > 1)
            in_buffer[1][i] = buffer_r[i] * (1.0 / (1L << (8 * sizeof(int) - 16)));
    }*/

    return lame_encode_buffer_sample_t(gfp, in_buffer[0], in_buffer[1],
                                       nsamples, mp3buf, mp3buf_size);
}
.
.
.
```

Add this part in original code

Comment this part

## Lame.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define size 1152
#define th 16
#define tw 32
#define bh 3
#define bw 6

__global__ void kernel_sample_t( float* dev_A, float* dev_B, const int buffer_l[], const int
buffer_r[], int channels_in)
{
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
int idx = ( bx * bw + by ) * ( th * tw ) + ( tx * tw + ty );

dev_A[idx] = buffer_l[idx] * (1.0 / (1L << (8 * sizeof(int) - 16)));
if(channels_in > 1)
dev_B[idx] = buffer_r[idx] * (1.0 / (1L << (8 * sizeof(int) - 16)));
}

extern "C"
void cuda_sample_t(int nsamples, float* a, float* b, const int buffer_l[], const int buffer_r[],
int channel)
{
int data_size = nsamples * sizeof(float);
float* dev_A;
float* dev_B;
int* dev_L;
int* dev_R;

cudaMalloc( (void**)&dev_A, data_size);
cudaMalloc( (void**)&dev_B, data_size);
cudaMalloc( (void**)&dev_L, nsamples * sizeof(int));
cudaMalloc( (void**)&dev_R, nsamples * sizeof(int));

cudaMemcpy( dev_L, buffer_l, nsamples * sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( dev_R, buffer_r, nsamples * sizeof(int), cudaMemcpyHostToDevice );

dim3 threads( th, tw );
dim3 grid( bh, bw );

kernel_sample_t<<< grid, threads >>>( dev_A, dev_B, dev_L, dev_R, channel);

cudaMemcpy( a, dev_A, data_size, cudaMemcpyDeviceToHost );
cudaMemcpy( b, dev_B, data_size, cudaMemcpyDeviceToHost );

 cudaFree( dev_A ) ;
 cudaFree( dev_B ) ;
 cudaFree( dev_L ) ;
 cudaFree( dev_R ) ;
}
```

## quantize.c

```
.
.
.
ms_convert(III_side_info_t * l3_side, int gr)
{
     convert_cuda(l3_side->tt[gr][0].xr, l3_side->tt[gr][1].xr);
     /*
    int      i;
    for (i = 0; i < 576; ++i) {
        FLOAT    l, r;
```

Add this part in original code

```
        l = l3_side->tt[gr][0].xr[i];
        r = l3_side->tt[gr][1].xr[i];
        l3_side->tt[gr][0].xr[i] = (l + r) * (FLOAT) (SQRT2 * 0.5);
        l3_side->tt[gr][1].xr[i] = (l - r) * (FLOAT) (SQRT2 * 0.5);
    }
     */
}
.
.
.
```

comment this part

quantize.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define sqrt2 1.41421356237309504880
#define size 576
#define th 16
#define tw 16
#define bh 4
#define bw 4

__global__ void kernel_convert(float * xr0, float * xr1)
{
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;

int idx = ( bx * bw + by ) * ( th * tw ) + ( tx * tw + ty );
__shared__ float l;
__shared__ float r;
 l = xr0[tx];
 r = xr1[tx];
xr0[tx] = (l + r) * (float)(sqrt2 * 0.5);
xr1[tx] = (l - r) * (float)(sqrt2 * 0.5);
}

extern "C"
void convert_cuda(float * xr0, float * xr1)
{
float * dev_xr0;
float * dev_xr1;

int data_size = size * sizeof(float);

cudaMalloc( (void**)&dev_xr0, data_size);
cudaMalloc( (void**)&dev_xr1, data_size);

cudaMemcpy( dev_xr0, xr0, data_size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_xr1, xr1, data_size, cudaMemcpyHostToDevice );

dim3 threads( th, tw );
dim3 grid( bh, bw);

kernel_convert<<< grid, threads >>>(dev_xr0, dev_xr1);

cudaMemcpy( xr0, dev_xr0, data_size, cudaMemcpyDeviceToHost );
cudaMemcpy( xr1, dev_xr1, data_size, cudaMemcpyDeviceToHost );

cudaFree( dev_xr0 );
cudaFree( dev_xr1 );

}
```