ParWiBench Internal

Xuechao Wei

March 8, 2015

1 LTE Background

Frame structure[3]

The size of various fields in the time domain is expressed as a number of time units $T_s = 1/(15000 * 2048)$.

Downlink and uplink transmissions are organized into radio frames with $T_f = 3072000 * T_s = 10ms$ duration.

Frame structure type 1

$T_f = 3072000 * T_s = 10ms$ long and consists of 20 slots of length $T_{slot} = 15360 * T_s = 0.5ms$, numbered from 0 to 19. A subframe is defined as two consecutive slots where subframe i consists of slots $2i$ and $2i + 1$.

TTI

Data on a transport channel is organized into transport blocks. In each Transmission Time Interval (TTI), at most one transport block of a certain size is transmitted over the radio interface to/from a mobile terminal in absense of spatial multiplexing. In case of spatial multiplexing (MIMO), there can be up to two transport blocks per TTI.

Associated with each transport block is a Transport Format (TF), specifying how the transport block is to be transmitted over the radio interface. The transport format includes information about the transport-block size, the modulation scheme, and the antenna mapping. Together with the resource assignment, the resulting code rate can then be derived from the transport format. By varying the transport format, the MAC layer can thus realize different data rates. Rate control is therefore also known as transport-format selection.

LTE has inherited the basic principle of WCMA/HSPA that data are delivered to the physical layer in the form of Transport Blocks of a certain size. In terms of the more detailed transport-block structure, LTE has adopted a similar approach as was adopted for HSPA:

1.1 LTE Downlink Scheme

Modulation

如果想要传输一串输入数字信号（二进制 bit 流），就必须把这些信号放到正弦波上，因为无线信号在空气中是通过正弦波传输的，所以在把信号放到正弦波上以后，正弦函数的一些参量如振幅、频率、相位均可用来体现不同的信号值。

例如，如果正弦波的函数是

$$C * cos(f_c * t + \phi) \tag{1.1}$$

以 QAM 为例，将上式展开一下化为

$$A * cos(f_c * t) + B * sin(f_c * t) \tag{1.2}$$

现在希望将输入信号由 A 和 B 携带，这时就同时利用了振幅和相位来携带信息。如果 A 和 B 携带两个 bit 的信息，那么可以建立一个每两 bit 和一个实数的映射关系：

```
01-->1
10-->3
11-->-1
00-->-3
```

如果将每两个实数分别作为一个复数的实部和虚部，那么一个复数就可以表示 4bit 的信息。进一步，如果将这个复数的实部和虚部分别对应于上面第 2 个式子中的 A 和 B，那么就可以将每 4bit 的信息映射到一个正弦波上，这就完成了正弦波的信息携带。

我们需要做的只是一个映射工作，然后将每个复数的实部和虚部发送到发射电路，电路会完成“载波”的任务。相反，在接受端也会有一个滤波电路，将接受到的正弦波的 A 和 B 解出来然后发送给后面的解调及解码过程。

OFDM

OFDM 的目的是将 n 个经过调制的频域信号转化为 n 个时域信号，每一个时域信号都由一系列间隔为 Δf 的子载波叠加而成，这些子载波相互正交。时域信号实际上是由 n 个频域信号时域化后采样得到。

假设经过基带调频并经过电路调制到高频之后，在 $[0, T_u]$ 时间内，第 i 个子载波上已调的 QAM 信号可以表示为

$$s_i(t) = A_{i_c} g(t) cos(2\pi f_i t) - A_{i_s} g(t) sin(2\pi f_i t) = Re\{[A_{i_c} + jA_{i_s}] g(t) e^{j2\pi f_i t}\} = Re\{A_i g(t) e^{j2\pi f_i t}\} \tag{1.3}$$

其中，$A_i = A_{i_c} + jA_{i_s}$ 是发送的 QAM 符号的星座点，A_{i_c} 和 A_{i_s} 分别是其同相分量（I 路）和正交分量（Q 路）；$f_i = f_c + i\Delta f = f_c + \frac{i}{T_u}$ 是第 i 路的载波频率，$i = 0, 1, ..., N-1$；$g(t)$ 是脉冲成形滤波器的冲激响应，假设它为矩形脉冲。

总的 OFDM 信号可以表示为

$$s(t) = \sum_{i=0}^{N-1} s_i(t) = Re\{[\sum A_i g(t) e^{j2\pi i \Delta f t}] e^{j2\pi f_c t}\} = Re\{a(t) e^{j2\pi f_c t}\} \tag{1.4}$$

其中

$$a(t) = \sum_{i=0}^{N-1} A_i g(t) e^{j2\pi i \Delta f t} \tag{1.5}$$

是 OFDM 信号的复包络。

若令 $I(t) = Re\{a(t)\}$，$Q(t) = Im\{a(t)\}$，则 $s(t)$ 可以表示为

$$s(t) = I(t)cos(2\pi f_c t) - Q(t)sin(2\pi f_c t) \tag{1.6}$$

因此也可以先得到复包络 $a(t)$，再经过 I/Q 正交调制来得到 OFDM 信号。

需要注意的是，对于 BPSK、MASK 这样的一维调制，所考虑的载波是 $cos(2\pi f_i t)$，已调信号只有同相载波，没有正交载波 $sin(2\pi f_i t)$，因此载波正交的最小间隔是 $\frac{1}{2T_u}$，而不是 $\frac{1}{T_u}$。要保证同相载波和正交载波同时都正交，这就需要载波间隔为 $\frac{1}{T_u}$。由 $s_i t$ 可见，对于二维调制，I 路和 Q 路这两个载波可以表示为一个复数载波 $e^{j2\pi f_i t}$（这里还不太理解）。对于两个复载波 $c_n = g(t)e^{j2\pi f_n t}$ 和 $c_m = g(t)e^{j2\pi f_m t}$，假设 $g(t)$ 为矩形脉冲，则使 $c_n(t)$ 和 $c_m(t)$ 保持正交的最小间隔是 $\frac{|f_n - f_m| = 1}{T_u}$。

为了实现 OFDM 调制的基带数字处理，首先要将 OFDM 信号的复包络进行采样，成为离散时间信号。

在 $[0, T_u](T_u = \frac{1}{\Delta f})$，$T_u = N * T_{samp}$，$T_{samp}$ 为采样频率的倒数。如果 $N = 2048$ 且 $\Delta f = 15KHz$，则 $T_{samp} = T_s$，T_s 为 time unit[3]，大小为 $1/(15000 * 2048)$。若采样时刻是 $m\frac{T_u}{N}, m = 0, 1, ..., N-1$，则对 OFDM 信号的复包络 $a(t)$ 采样后的序列为

$$a_m = a(m\frac{T_u}{N}) = \sum_{i=0}^{N-1} A_i e^{j2\pi i \Delta f m \frac{T_u}{N}} = \sum_{i=0}^{N-1} A_i e^{j2\pi \frac{mi}{N}} \ m = 0, 1, ..., N-1 \tag{1.7}$$

该式恰好就是对序列 $\{A_0, A_1, ..., A_{N-1}\}$ 进行离散傅里叶反变换（IDFT）的结果。其中，IDFT 得到的 N 个时间间隔为 T_{samp} 的一组输出叫做一个 OFDM 符号（OFDM symbol）。因此，给定输入的符号 $\{A_0, A_1, ..., A_{N-1}\}$ 后，借助 IDFT 即可得到 OFDM 复包络的时间采样。

接收端通过 I/Q 正交解调后可以恢复 OFDM 信号的复包络 $a(t)$，将其采样得到的时间序列 $\{a_m\} = \{a_0, a_1, ..., a_{N-1}\}$。由于 IDFT 是可逆变换，因此对序列 $\{a_m\}$ 进行离散傅里叶变换（DFT）即可得到发送的序列 $\{A_i\}$：

$$A_i = \sum_{m=0}^{N-1} a_m e^{-j2\pi \frac{mi}{N}} \ i = 0, 1, ..., N-1 \tag{1.8}$$

由于 $\{a_m\}$ 是对时间信号的采样，故称其为时域序列。而 $\{A_i\}$ 是序列 $\{a_m\}$ 的离散傅里叶变换，故称其为频域序列。

当 N 为 2 的整幂时，DFT 和 IDFT 存在快速算法：FFT 和 IFFT。这样，可以借助 FFT 和 IFFT 来实现 OFDM 信号的调制与解调。

[**Aside:**] 对正交、频分和复用的理解

我觉得正交包含两方面的含义，一指构成每个 OFDM 符号的各子载波彼此正交；二指往高频调制时，正交调制器附加的同相载波和正交载波相互正交。前者我需要关心，后者由调制电路来做，我不用管；频分是指各子载波的频率间隔为 Δf；感觉复用最难理解，理想的正弦波的频谱其实是单值函数，即只在一个频率处有非零值，其它频率处都为 0。但实际上由计算机生成的正弦波都是锯齿状的，看起来由许许多多的台阶构成，这就使得实际的"正弦波"频谱图如图 2 中的任意一个子载波所示，图中最高波峰即为我们所需要的理想正弦波频谱的那个非零值。传统的 FDM 做法是让每个子载波间隔足够大的 Δf 值，使得每个子载波的最大频谱波形彼此分开，如图 1 所示。而 OFDM 则保证只要用一个最小的 Δf 使子载波正交，就可以用如下函数将各子载波的最大非零频谱值区分出来（虽然彼此之间存在重叠干扰）:

$$s(f) = \begin{cases} 0 & \text{if } f = n\Delta f (n \neq 0) \\ a_0 + b_0 j & \text{else} \end{cases} \tag{1.9}$$

当然，这只是非常粗略理解。

CRC

循环冗余校验是通过模 2 除法运算来建立有效信息位和校验位之间的约定关系。其中待编码的有效信息以多项式 $M(x)$ 表示，将它左移若干位后，用另一个约定好的多项式 $G(x)$ 去除，所产生的余数就是校验位。当接受方收到发来的 CRC 码后，他仍用约定的 $G(x)$ 去除，若余数为 0，表明该代码接受无误；若余数不为 0，表明某一位出错，再进一步由余数值确定出错的位置，并予以纠正。

CRC 的编码步骤如下：

- 将待编码的 N 位有效信息位表示为一个 $n-1$ 阶的多项式 $M(x)$；
- 将 $M(x)$ 左移 K 位，得到 $M(x)x^k$（k 由预选的 $k+1$ 位的生成多项式 $G(x)$ 决定）;
- 用一个预选好的 $k+1$ 位的生成多项式 $G(x)$ 对 $G(x)x^k$ 做模 2 除法；
- 把左移 k 位后的有效信息位与余数做模 2 加法，即形成长度为 $N+K$ 的 CRC 码。

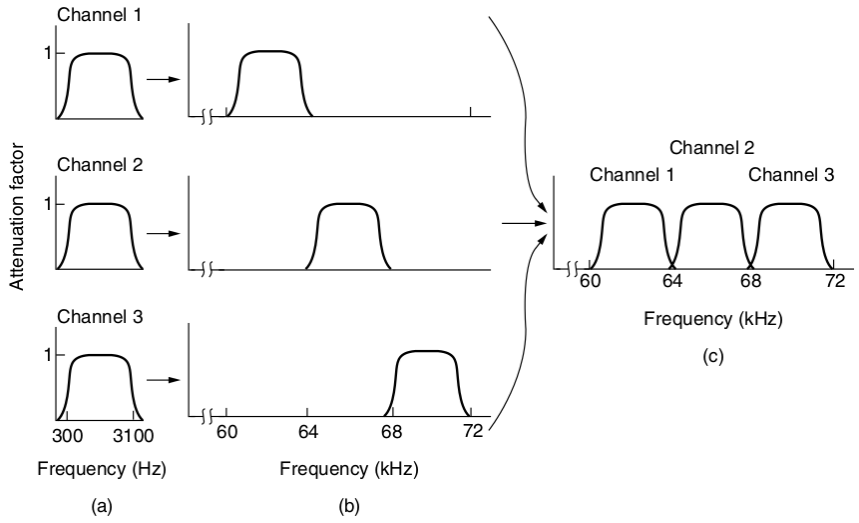

Figure 2-25. Frequency division multiplexing. (a) The original bandwidths. (b) The bandwidths raised in frequency. (c) The multiplexed channel.

Figure 1.1: Traditional FDM

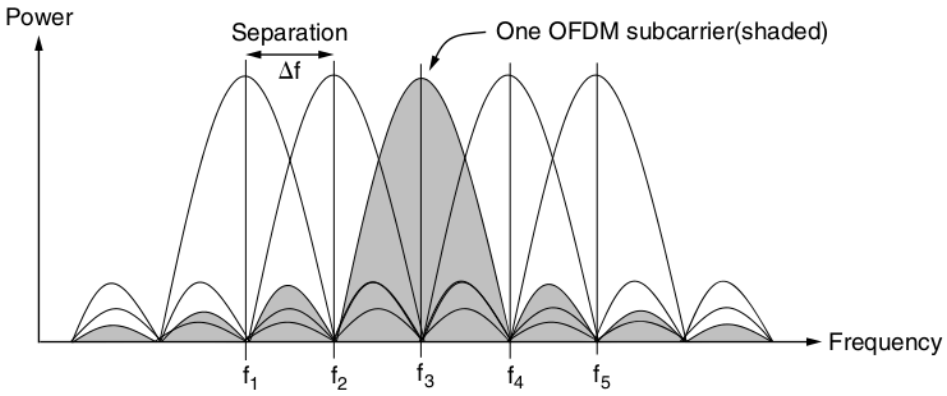

Figure 2-26. Orthogonal frequency division multiplexing (OFDM).

Figure 1.2: OFDM

Turbo coding

The transfer function of the 8-state constituent code for the PCCC (Parallel Concatenated Convolutional Code) is[1]:

$$G(D) = [1, \frac{g_1(D)}{g_0(D)}]$$

where

$$g_0(D) = 1 + D^2 + D^3 g_1(D) = 1 + D + D^3$$

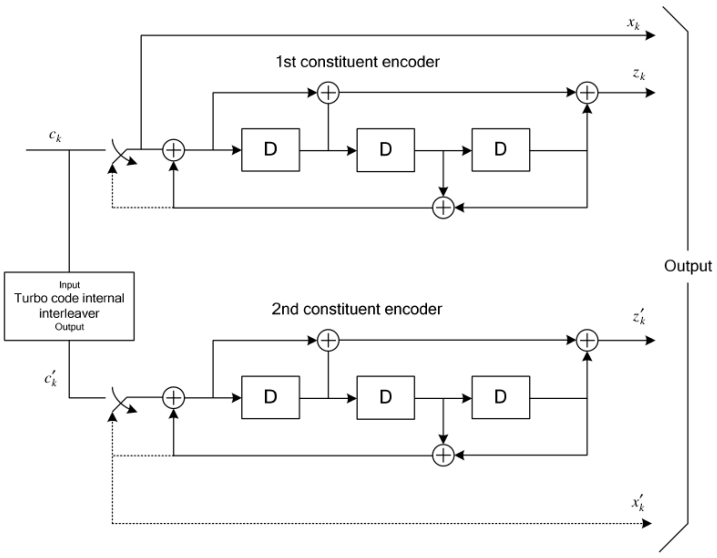

Figure 1.3: Structure of rate 1/3 turbo encoder (dotted lines apply for trellis termination only)

- In case of single-antenna transmission there is a single transport block of dynamic size for each TTI.
- In case of multi-antenna transmission, there can be up to two transport blocks of dynamic size for each TTI, where each transport block corresponds to one codeword in case of downlink spatial multiplexing. This implies that, although LTE supports downlink spatial multiplexing with up to four layers, the number of codewords and thus also the number of transport blocks is still limited to two.

Channel mapping

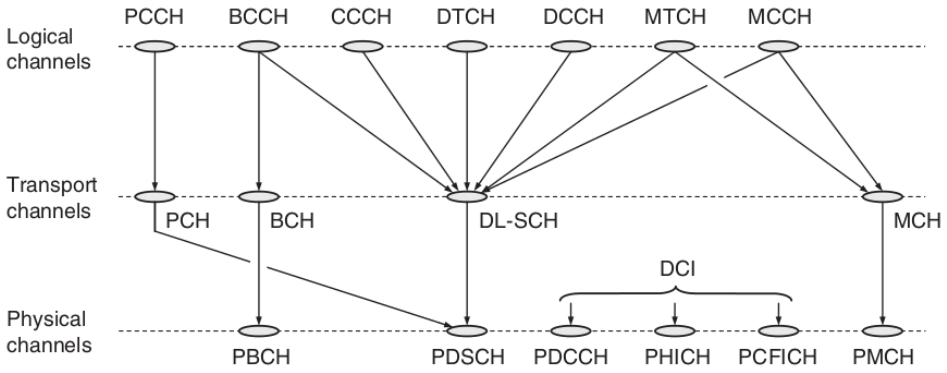

Figure 1.4: Downlink channel mapping

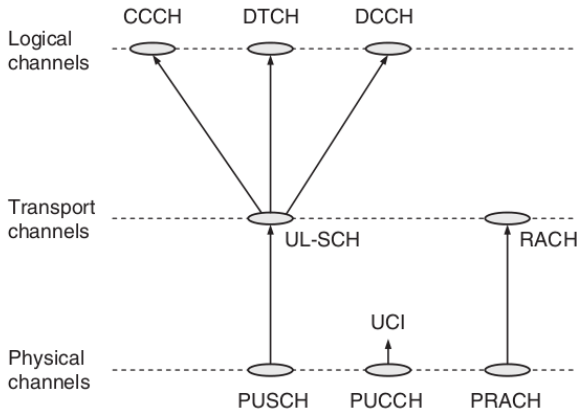

Figure 1.5: Uplink channel mapping

我对 channel mapping 的概念感到极其不理解，但从下面的文字中貌似可以看出 transport channel 和 physical channel 的联系：

To the transport block(s) to transmit on the DL-SCH, a CRC, used for error detection in the receiver, is attached, followed by Turbo coding for error correction. In case of spatial multiplexing, the processing is duplicated for each of the transport blocks. Rate matching is used not only to match the number of coded bits to the amount of resources allocated for the DL-SCH transmission, but also to generate the different redundancy versions as controlled by the hybrid-ARQ protocol.

After rate matching, the coded bits are modulated using QPSK, 16QAM, or 64QAM, followed by antenna mapping. The antenna mapping can be configured to provide different multi-antenna transmission schemes including transmit diversity, beam-forming, and spatial multiplexing. Finally, the output of the antenna processing is mapped to the physical resources used for the DL-SCH. The resources, as well as the transport-block size and the modulatin scheme, are under control of the scheduler.

A *phsical channel* corresponds to the set of time-frequency resources used for transmission of a particular transport channel and each transport channel is mapped to a corresponding physical channel.

经过师兄的指点，现在的理解是，物理信道主要负责资源块的分配，传输信道负责前面的编码部分。

The downlink physical resource

2 Benchmark Suite Implementation

2.1 Frame Parameters

Downlink

Uplink

一帧处理的 bit 长度是由 LTE 资源块的大小决定的，而资源块的大小由采样频率决定。因此一旦采样频率确定，通过从后往前推，各个模块能够处理的最大数据长度及其至少应该达到的 throughput 也就是确定的。

由于 Uplink 和 Downlink 使用的资源块结构相同，因此上行 PHY 层每帧的参数信息也可以参考1.1。与下行不同，以 Tx 为例，上行在进入 resource mapping 之前要先经过一次 DFT，因此在 resouce mapping 之前的模块，其参数与 DFT 的长度有关，而不是 IFFT 的长度。

下面以 resouce mapping 为界、采样频率为 15.36MHz 来计算各个模块的输入输出数据长度以及模块的 throughput。

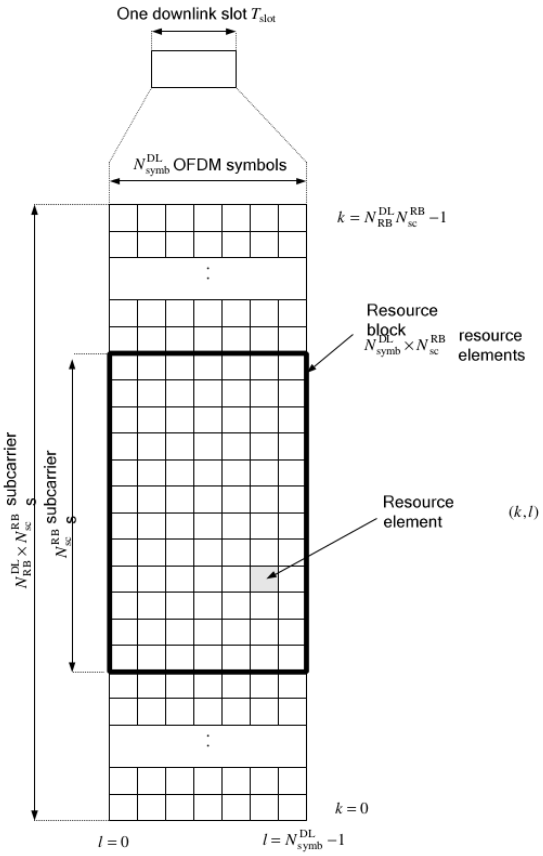

Figure 1.6: LTE downlink resource grid[2]

Channel Bandwidth (MHz)	1.25	2.5	5	10	15	20
Frame Duration (ms)	10					
Subframe Duration (ms)	1					
Sub-carrier Spacing (kHz)	15					
Sampling Frequency (MHz)	1.92	3.84	7.68	15.36	23.04	30.72
FFT Size	128	256	512	1024	1536	2048
Occupied Sub-carriers (inc. DC sub-carrier)	76	151	301	601	901	1201
Guard Sub-carriers	52	105	211	423	635	847
Number of Resource Blocks	6	12	25	50	75	100
Occupied Channel Bandwidth (MHz)	1.140	2.265	4.515	9.015	13.515	18.015
DL Bandwidth Efficiency	77.1%	90%	90%	90%	90%	90%
OFDM Symbols/Subframe	7/6 (short/long CP)					
CP Length (Short CP) (μs)	5.2 (first symbol) / 4.69 (six following symbols)					
CP Length (Long CP) (μs)	16.67					

Figure 1.7: LTE downlink physical layer parameters[5]

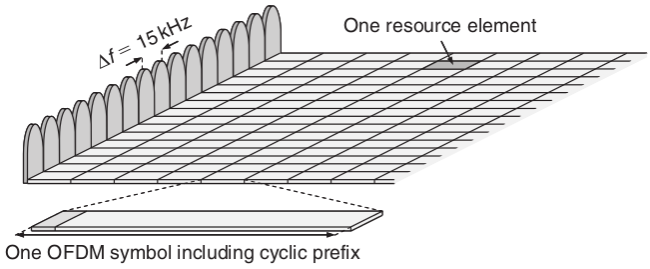

Figure 1.8: LTE downlink physical resource

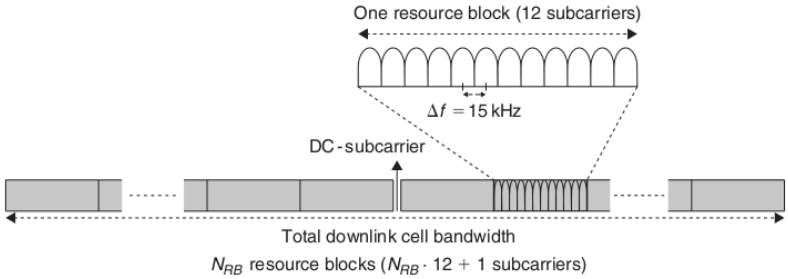

Figure 1.9: Frequency-domain structure for LTE downlink

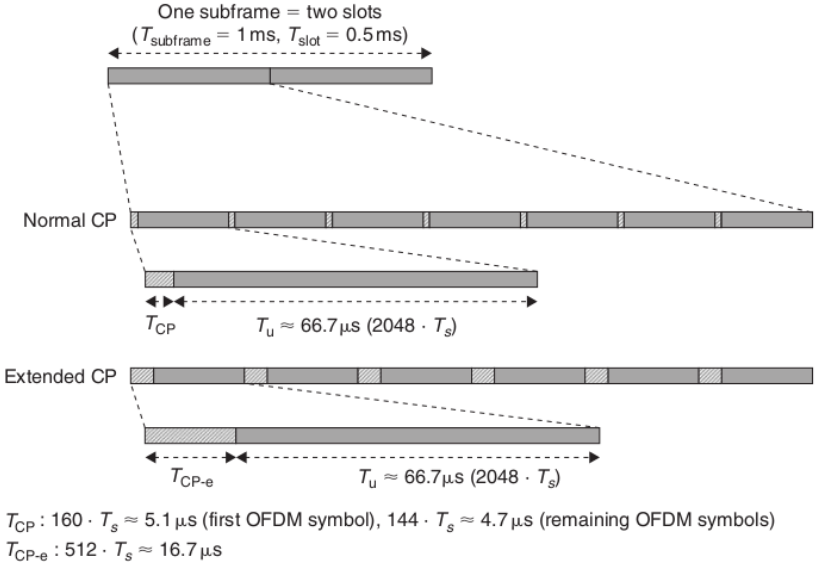

Figure 1.10: Detailed time-domain structure for LTE downlink transmission

??是1.1和1.1的结合，表示子帧（slot 或 subframe）结构，一帧中包含两个子帧。其中水平方向的长度 NumULSymbSF 表示一个子帧中的 OFDM 符号数目，目前为常数 14。垂直方向的长度为子载波的数目，是与带宽对应的量，例如带宽为 1.25MHz 时，子载波的数目为 76，这个值只有在发射端和接收端的前端才会起作用，而对于 Tx 和 Rx 的 baseband 处理时我们不会看到这个量。FFT 与子载波紧密相关，因为 FFT 各点要映射到子载波上，但 FFT 的点数与子载波的数目并没有什么关系，如果 FFT 点数大于子载波数目，超过的部分填 0，如果不足，那多出的子载波被浪费，所以对于资源块，我们真正看到的是 FFT 的点数而不是子载波的数目。

下面以 3.72MHz 采样率、20MHz 带宽为例，计算各个模块的参数。

发射端：

```
Module: Turbo encoding

InBufSz: (NumBlock - 1) * BlkSize + 1 * LastBlkSize = 18432(bit)

OutBufSz: Rate * ((NumBlock - 1) * (BlkSize + 4) + 1 * (LastBlkSize + 4)) = 55332(bit)

Throughput: 55.332M(bit)/s

Latency: 1ms

Module: Rate Matching

InBufSz: Rate * ((NumBlock - 1) * (BlkSize + 4) + 1 * (LastBlkSize + 4)) <= OutBufSz

OutBufSz: MDFT * (NumULSymbSF - 2) * (log(16)) = 57600(bit)

Throughput: 57.6M(bit)/s

Latency: 1ms
```

取`BlkSize` 为协议规定的最大值 6144 bits，同时假设输入 bit 串的长度能够被`BlkSize` 整除，则容易从上面`InBufSz` 的不等式求得`NumBlock` 的最大值为 3，得到`InBufSz` 的长度为 55332 bits。

```
Module: Scrambler
```

InBufSz: MDFT * (NumULSymbSF - 2) * (log(16)) = 57600(bit)

OutBufSz: MDFT * (NumULSymbSF - 2) * (log(16)) = 57600(bit)

Throughtput: 57.6M(bit)/s

Latency: 1ms

Module: Modulation

InBufSz: MDFT * (NumULSymbSF - 2) * (log(16)) = 57600(bit)

OutBufSz: MDFT * (NumULSymbSF - 2) = 1200 * (14 - 2) = 14400(Complex)

Throughput: 14.4M(Complex)/s

Latency: 1ms

以 16QAM 调制方式为例。

Module: Transform Precoder

InBufSz: MDFT * (NumULSymbSF - 2) = 1200 * (14 - 2) = 14400(Complex)

OutBufSz: MDFT * (NumULSymbSF - 2) = 1200 * (14 - 2) = 14400(Complex)

Throughput: 14.4M(Complex)/s

Latency: 1ms

Module: Resource Mapping

InBufSz: MDFT * (NumULSymbSF - 2) = 1200 * (14 - 2) = 14400(Complex)

OutBufSz: NFFT * NumULSymbSF = 2048 * 14 = 28672(Complex)

Throughtput: 28.672M(Complex)/s

Latency: 1ms

Module: OFDM

InBufSz: NIFFT * NumULSymbSF = 2048 * 14 = 28672(Complex)

OutBufSz: (NIFFT + SpecialCPLen) * 2 + (NIFFT + NormalCPLen) * (NumULSymbSF - 2) = (2048 + 160) * 2 + (2048 + 1

Throughput: 30.72M(Complex)/s

Latency: 1ms

Module: RF

InBufSz: (NIFFT + SpecialCPLen) * 2 + (NIFFT + NormalCPLen) * (NumULSymbSF - 2) = (2048 + 160) * 2 + (2048 + 14

OutBufSz: ?

Sampling Rate: 30.72MHz

Throughtput: ?

Latency: 30720 / 30.72MHz = 1ms

接收端:

2.2 Turbo Coding

2.2.1 Encoder

2.2.2 MAP Decoder

Pseudo-Code for the Iterative Decoder

1: **for** $k \leftarrow 1$ to N **do** ▷ N is block size
2: $\quad \gamma_k(s', s) \sim exp[\frac{1}{2}u_i(L^e(u_k) + L_c y_k^s) + \frac{1}{2}L_c y_k^p x_k^p]$
3: $\quad \tilde{\alpha}_k(s) = \frac{\sum_{s'} \tilde{\alpha}_{k-1}(s')\gamma_k(s',s)}{\sum_s \sum_{s'} \tilde{\alpha}_{k-1}(s')\gamma_k(s',s)}$
4: **end for**
5: **for** $k \leftarrow N$ downto 2 **do**
6: $\quad \tilde{\beta}_{k-1}(s') = \frac{\sum_s \tilde{\beta}_k(s)\gamma_k(s',s)}{\sum_s \sum_{s'} \tilde{\alpha}_{k-1}(s')\gamma_k(s',s)}$
7: **end for**
8: **for** $k \leftarrow 1$ to N **do**
9: $\quad L_{12}^e(u_k) = log(\frac{\sum_{S+} \alpha_{k-1}(s')\gamma(s',s)\beta_k(s)}{\sum_{S-} \alpha_{k-1}(s')\gamma(s',s)\beta_k(s)})$
10: **end for**

2.2.3 Parallel MAP

将 Turbo Decoder 并行的基本思路是将一个 block 再划分为 sub-blocks，然后以 sub-block 为单位进行译码。因为 block 是协议规定的译码算法的最小单位，因此不能按照协议规定的初始化 block 的方法来初始化 sub-block，否则可能影响误码率。目前初始化 sub-block 的方法主要有两种：Next Iteration Initialization（NII）[6] 和 Overlap[4]。

NII 对 sub-block 的初始化分为两个过程，即在第一次迭代开始之前和之后的迭代。第一次迭代开始之前的初始化按照如下过程：

对于每一个 block 的第一个 sub-block，初始化按照 Kronecker delta 函数进行，即如果状态索引 s 为 0，将 α 和 β 初始化为 1.0，否则为 0.0；对于其它的 sub-block，在所有状态下都初始化为 $1.0/N\_STATES$。

在之后的迭代中，各个 sub-block 的初始化值来自于上一次迭代计算得到的 α 和 β 的边界值。回忆我们在计算 block 的 extrinsic 值时，假设 extrinsic 的索引范围为 $[0 \cdots stride-1]$，那么为其分配的 α 和 β 的范围必须为 $[0 \cdots stride]$，其中 $\alpha[0]$ 和 $\beta[stride]$ 都已经被初始化。而最后计算 extrinsic 值时，$\alpha[stride]$ 和 $\beta[0]$ 都不会被用到，事实上，我们在计算 block 的 extrinsic 值时，都不会计算 $\beta[0]$，计算 $\alpha[stride]$ 值也仅仅是因为要计算一个 normalization 数组。但是对于 sub-block，这两个边界值被用做下一次迭代的初始化值。设 sub-block[i] 的 α 和 β 数组为 $l\_\alpha[i]$ 和 $l\_\beta[i]$，在计算完成 sub-block[i] 的 extrinsic 值

后，我们必须计算 $l\_\alpha[i][stride] l\_\beta[i][0]$ 来分别作为下一次迭代的 sub-block[i+1] 和 sub-block[i-1] 的初始值 $\_\alpha[i+1][0] l\_\beta[i-1][stride]$（经过尝试，使用 $l\_\beta[i][1]$ 作为下一次迭代的初始值会引入较大的误码率，且结果很不稳定）。当然，i-1 不能是 0，i+1 也不能是最后一个 sub-block 的索引，因为第 0 个 sub-block 的 α 和最后一个 sub-block 的 β 的初始化在第一次迭代之前的初始化完成后就不会再改变。

在实现时，如果局部空间足够，只需要为每一个 sub-block 开辟一个局部的一维 l_α 和 l_β 数组即可，不必使用全局的二维数组（第一维用来索引 sub-block）。

2.2.4 Miscellaneous

WiBench 中解调的最大似然软输出映射成 bit 流的规则是：

```
"+"->"1"
"-"->"0"
```

itpp 库中的 Turbo 解码模块认为其输入输出是“+1”或“-1”，因此如果输入是 bit 流，首先要做如下映射：

```
"1"->"-1"
"0"->"1"
```

而如果输入是软值，则认为正值代表“+1”，负值代表“-1”。因此，在目前的实现中，解调与解码的软值与 bit 的映射规则恰好相反，所以将解调模块的输出作为解码模块的输入之前，首先要做一个符号取反操作。

2.3 Rate Matching

目前的实现中，在 Rate Dematching 时，接收端将从 OFDM 传入的输入数据提取出的 DMRS 符号放到输出数据的最前面位置，然后接着再放 Data 部分。输出数据作为 Equalizer 的输入数据。

OpenMP 实现

目前的 OpenMP 多线程版本主要在 SubblockInterleaving 内部的两个循环中，而且没有计算，全都是对两个矩阵的读写，而两个矩阵的大小大概是 24KB，甚至都小于一个 core 上的 L1 cache 的大小（以 cecag1 上的 Intel Xeon E5-2620 为例，L1 cache 的大小为 35KB）。因此，如果这两个矩阵能够大部分在 cache 中，那么多线程不会带来好处，可以解释随着线程增多性能下降的现象。

2.4 Scrambling

在 Tx，scrambling 的计算公式为

$$\tilde{b}(i) = (b(i) + c(i))\%2$$

在 Rx，解扰码的计算公式为

$$\tilde{b}(i) = b(i) * c(i)$$

如果接收端一直是硬输入硬输出，如果 Tx 端的扰码运算为将扰码序列与发送序列的异或，那么 Rx 端的解扰运算也应该是异或运算。但是解扰运算的前一个模块是解调，而解调的输出是软值，且我们在之前分析过解调的软值的符号与 bit 的映射规则为：

```
"+"->"1"
"-"->"0"
```

因此，如果在发送端发送的 bit 为“1”，在接收端未解扰之前收到的是“+”，那么在发送端的扰码 bit 应为“0”(异或运算)，要想在解扰之后也能得到“+”(因为发送的是“1”)，那么解扰符号应为“+”(乘法运算)。这一串的对应关系可以表示如下：

$$1 \otimes 0 \rightarrow + * + \rightarrow +$$

同样的，剩余的扰码与解扰的关系式如下：

$$1 \otimes 1 \rightarrow - * - \rightarrow +$$

$$0 \otimes 0 \rightarrow - * + \rightarrow -$$

$$0 \otimes 1 \rightarrow - * + \rightarrow -$$

2.5 Modulation

2.5.1 HLS

外层循环迭代之间没有依赖，但有若干内层循环，对外层循环做的 pipeline 没有作用。

2.6 Resource Mapping

2.6.1 OpenCL

为输出数据 pOutData 创建 buffer 时要使用 CL_USE_HOST_PTR 类型，因为在进入 kernel 前有一个初始化过程，而在 kernel 中要保证没有被写入数据的位置这些初始化数据不丢。

2.7 OFDM

注意：ifft 的 direction 为 1，fft 的 direction 为 -1，稍候再从公式详细解释其中原因。

如果复数的数据结构为实部和虚部分布在一个数组的前半段和后半段，那么对每一个 symbol 做 fft 不能在原数组上进行，应该开一个长度为 2*IFFT 长度的 buffer，在 buffer 上进行。因为 fft 虚部的跨度为 IFFT，而原数组的跨度为输入数组的长度。

另外，fft_nrvs 版本的 fft 程序由于要使用输入数组作为迭代之间的 scratchpad，所以程序结束后输入数组已经不是原来的输入数据，所以这个程序不能被运行多次。除非在程序外或内开辟一个专门的 scratchpad 用来缓存。

2.8 DFT

Multithreading

加了 Reduction 优化后会更慢。

SIMD

在 MIC 上，不加任何向量化，运行时间为 450ms；自动向量化能够得到 60ms；只做手动向量化，280ms；手动加自动，28ms。

有一件奇怪的事，自动向量化只在 main 函数与 dft 函数放在一个文件中时才会生效，如果分放在两个文件里，加不加自动向量化运行时间差不多。

2.9 Equalizing

2.10 Synchronization

2.10.1 802.11 同步算法

802.11 OFDM 帧格式

按照 802.11a 的标准，一个 OFDM 符号包含 48 个映射的复数值、4 个导频信号和 12 个 0 信号，IFFT 与 FFT 均为 64 点运算。为克服符号间干扰（ISI），需要加入循环前缀（GI），GI 为 16 个采样值。OFDM 帧在真正有用的数据（Data）之前，会预先发送短训练字（t_1 至 t_{10}）、长训练字（T_1，T_2）和 SIGNAL 字段，GI 代表循环前缀。其中短训练字用于帧同步、位同步和粗频率同步；长训练字用于精确的频率同步和信道估计；SIGNAL 字段则包含了长度、调制方式等信息。图??是 802.11 的 OFDM 帧结构。

时间同步技术

时间同步分为两个步骤，帧同步和位同步。其中帧同步用于检测到每一帧的帧头部，位同步用于定位到每一个采样数据的具体位置。帧同步与位同步都是采用 OFDM 帧首部的短训练字来实现的。

（1）帧同步

帧同步一般采用延时相关算法，即存在两个滑动窗口 C 和 P，在滑动窗口 C 内计算接受信号和接受信号延时的相关参数，而在滑动窗口 P 内计算互相关窗口内接受信号的能量。滑动窗口 P 用于判决统计的归一化。

帧同步采用的基本公式如式2.1和式2.2所示，y 代表接受到的信号，其括号内数值表示不同的采样时间。D 和 L 是相邻短训练序列之间的间隔。$C(n)$ 为接收信号的延时相关参数，$P(n)$ 为接收信号的能量。

$$C(n) = \sum_{k=0}^{L-1} y(n+k) \cdot y^*(n+k+D) \tag{2.1}$$

$$P(n) = \sum_{k=0}^{L-1} y(n+k+D) \cdot y^*(n+k+D) = \sum_{k=0}^{L-1} |y(n+k+D)|^2 \tag{2.2}$$

最终用于接收端统计判决的参数 $M(n)$ 可以用式2.3来表示：

$$M(n) = \frac{|C(n)|^2}{P^2(n)} \tag{2.3}$$

帧同步算法利用了前导序列中的 10 个重复的短训练序列的重复性，并通过归一化使得统计判决的参数 $M(n)$ 都在 $[0,1]$ 范围之内。由于噪声或者其它信号不具有持续的周期性，所以它们的延时相关值 $C(n)$ 近似为 0。一旦接收端收到 OFDM 帧，延时相关值 $C(n)$ 会迅速变大，从而使 $M(n)$ 迅速跳变为接近于 1 的较大值，并能够持续保持 9 个短训练序列符号的时间。

（2）位同步

在帧未同步时，接收到的数据是不往后续模块输送的。一旦帧同步后，符号定时同步还需进一步精确到抽样点的水平。根据短训练序列良好的自相关特性，可以利用本地已知的一个短训练序列符号 $t(k)$ 生成另一统计判决值 $MF(n)$。$MF(n)$ 由式2.4计算得到：

$$MF(n)=\frac{\left|\sum_{k=0}^{D-1}y(n+k)\cdot t^{*}(k)\right|^{2}}{\sum_{k=0}^{D-1}\left|y(n+k)\right|^{2}\sum_{k=0}^{D-1}\left|t(k)\right|^{2}} \tag{2.4}$$

式2.4中，t_k 与发送的段训练字其中一个符号完全一致。这样，当 OFDM 的短训练字段来到时，统计判决值 $MF(n)$ 会出现 10 个相关峰值。根据 $MF(n)$ 与 $M(n)$ 的当前计算值，就能够准确地找到 OFDM 帧并定位到 OFDM 帧的每一个采样点。

(3) 算法针对 FPGA 的简化分析

根据上述介绍，PHY 层时间同步模块的算法关键需要获取 $M(n)$ 和 $MF(n)$ 两个参数，其中 $M(n)$ 参数用于帧同步的判决，$MF(n)$ 用于位同步的判决。在每个数据采样点到来时，都需要完成以上两个参数的计算。如果考虑到要用 FPGA 对算法进行加速，式2.3、2.4的计算量较大，完成这些计算需要占用大量的 FPGA 资源，同时也很难满足整个系统告诉与实时性的要求，因此有必要对 $M(n)$ 和 $MF(n)$ 的计算进行简化。

从资源占用率的角度出发，式2.3和式2.4有大量乘法与一次除法，这些计算都会在很大程度上增大 FPGA 的资源利用率。

由于 $M(n)$ 和 $MF(n)$ 都属于判决性质的参数，因此，如果计算中有幂运算，可以将其开方来降低其幂次。但是仔细分析可知，即使开方后，$M(n)$ 和 $MF(n)$ 的计算量仍然较大。不过经过观察发现，$M(n+1)$ 和 $M(n)$ 相比，L 次累加中，只有 2 个值不一致，可以根据这个特性简化其累加部分：

$$C(n)=\sum_{k=0}^{L-1}\left|y(n+k)\cdot y^{*}(n+k+D)\right|=C(n-1)+y(n)\cdot y^{*}(n+D)-y(n+L-1)\cdot y^{*}(n+L+D-1) \tag{2.5}$$

$$P(n)=\sum_{k=0}^{L-1}\left|y(n+k+D)\right|^{2}=P(n-1)+\left|y(n+D)\right|^{2}-\left|y(n+D+L-1)\right|^{2} \tag{2.6}$$

$$M(n)=\frac{\left|C(n)\right|}{P(n)}=\frac{\sqrt{C_r^2(n)+C_i^2(n)}}{P(n)}\approx\frac{\left|C_r(n)\right|+\left|C_i(n)\right|}{P(n)} \tag{2.7}$$

完成上述简化后，$C(n)$ 和 $P(n)$ 的计算与未简化前相比，单针对乘法器而言，就节省了很多资源。

最终的判决式采用式2.8完成。

$$M(n)>M_{th} \tag{2.8}$$

根据大量实验结果，判决阈值 M_{th} 取 0.5。

对于位同步判决参数 $MF(n)$，无法用上述同样的方式完成简化。但是由位同步的计算过程可以看出，在位对齐的情况下，$MF(n)$ 会突变为相对非常大的峰值。因此，对于位同步来说，不需要非常高的计算精度。为了简化计算，可以将每个数据的复数值量化为 $2\mathrm{bit}$，即：

$$y(n) = \pm 1 \pm i \tag{2.9}$$

将式2.9代入开方后的2.4，由于 $t(k)$ 是一串固定的序列，所以式??的分母为一常数，由于分子中其中一个待乘复数为量化后的 $y(n)$，所以式??中不需要任何的乘法运算，大量简化了算法的复杂度。

最终位同步的判决式采用式2.10完成：

$$MF(n) > MF_{th} \tag{2.10}$$

其中 MF_{th} 的值可以通过实验得到。

（4）算法结构

算法整体分为两极结构：第一级控制帧结构中字段的界定和处理；第二级为时间同步。其中第二级嵌入在第一级中。

算法整体结构如下：

Algorithm 1 802.11 Time Synchronization Top-level Control

Name:

TryDetectSyncSeq

Input:

InputBuffer /* 从 RX 输入流中截取的一段数据流 */

Length /* InputBuffer 的长度 */

Output:

OutputBuffer /* 同步到一帧以后拿到的数据部分 */

$n\_in \leftarrow 0$ /* 输入缓存数据累加器 */

$staticn\_signal\_output \leftarrow 0$ /* 训练字及 SIGNAL 符号累加器，一共要收集 320 个符号，从第 5 个短训练字开始到 SIGNAL 的最后一个符号 */

$staticn\_data\_output \leftarrow 0$ /* 数据部分符号累加器 */

S0:

/* 进行一次尝试时间同步 */

while $n\_in < Length$ **do**

  if $TimeSync(InputBuffer[n\_in++], OutputTimeSync, num\_th, m\_th, mf\_th) == 1$ **then**

    /* 每次尝试都为时间同步算法输入一个符号和 3 个阈值，同步完成后，OutputTimeSync 中会得到 96 个采样值，即，第一个 GI2 的符号为同步到的边界 */

    $n\_signal\_output \leftarrow 96$ /* "96"包括后 5 个短训练字符号和 GI2 的第一个符号，每个符号包含 16 个采样值 */

    Goto S1

  end if

end while

if $n\_in \geq Length$ **then**

  /* 未同步到，保存当前状态，跳出算法，重新从输入流中截取数据，然后回到算法继续尝试同步 */

end if

S1:

/* 同步到短训练字，继续收集长训练字和 SIGNAL 为后续模块使用 */

if 缓存中剩余的数据长度和已经收集到的符号数目之和尚未达到 320 **then**

  收集完缓存中剩余的符号

else

  收集缓存中剩余的符号，直至 n_signal_output 达到 320

  $n\_data\_output \leftarrow 0$

  Goto S2 /* 开始收集数据 */

end if

S2:

收集数据的过程和收集部分训练字及 SIGNAL 的过程类似，不再赘述

时间同步模块 TimeSync 中定义了三个主要状态：未同步状态、帧同步状态以及位同步状态。模块的状态转移图如图1.1所示。

Algorithm 2 Time Synchronization Alg.

Name:

TimeSync

Input:

One symbol and three threshold values

Output:

Time synchronize sequence containing 96 samplings

1: $staticcount \leftarrow 0$
2: S0:
3: **if** $m > f\_mth$ **then**
4: $count \leftarrow count + 1$ /* 累计互相关峰值的持续时间 */
5: **else**
6: $count \leftarrow 0$ /* 持续时间未到指定下限，状态不变，重新统计 */
7: **end if**
8: **if** $count >= num\_th$ **then** /* 互相关峰值的持续时间达到指定下限，同步到的可行性较大 */
9: $count \leftarrow 0$
10: Goto S1
11: **end if**
12: S1:
13: **if** $m < f\_mth$ **then** /* 累加干扰的持续时间 */
14: $count \leftarrow count + 1$
15: **end if**
16: **if** $count \geq num\_th$ **then** /* 如果干扰持续时间超过指定阈值，同步失败 */
17: $count = 0$
18: Goto S0
19: **end if**
20: **if** $m < f\_mth$ and $mf > f\_mfth$ **then** /* 如果干扰持续时间在阈值范围内的前提下，同时遇到互相关峰值的衰落和自相关的一个峰值，则认为同步成功且已经到达同步序列的末尾 */
21: $count \leftarrow 0$
22: 收集 96 个训练字和 SIGNAL 采样值
23: Goto S0
24: **end if**

该模块开始工作后，首先模块处于未同步状态。在未同步状态下，模块持续计算 $M(n)$ 的大小，并将 $M(n)$ 与阈值 M_{th} 进行比较。由于接收到的噪声在一定几率下也会使 $M(n)$ 的值变大，为防止误判，模块必须保证有至少连续 num_th 个 $M(n)$ 的值都大于 M_{th} 时，才认为同步到了一帧的头部。此时模块跳转至帧同步状态。

在帧同步状态下，计数器 $count$ 计数 $M(n)$ 小于阈值的持续时间，如果在这个持续时间不超过某个阈值（$m < f\_mth$）的情况下，遇到了一个自相关峰值（$mf > f\_mfth$），则代表到了互相关峰值的下降处和最后一个自相关峰值，同步成功。

但是，如果发生一种情况，即在互相关峰值的持续时间内只在最后遇到一个自相关峰值，这种情况也会被上述算法认为同步成功，但显然，这种情况很有可能是噪声，使得同步错误。为了避免这种情况，需要增加一步判断，保证在互相关峰值持续时间内有足够多的自相关峰值数目出现。

（5）时间同步算法的改进

相对于基本的同步算法，下面的算法片段主要针对 S1 时做修改：

Algorithm 3 Improved Time Synchronization Alg.

S1:
if $m < f\_mth$ **then**
    $count \leftarrow count + 1$
else
    $count\_m \leftarrow count\_m + 1$
end if
if $mf > f\_mfth$ **then** /* 遇到一个自相关峰值 */
    if $count\_m > 32$ **then** /* 距离上一个自相关峰值经历的互相关峰值持续时间超过了某个阈值 */
        $count\_mf \leftarrow 0$ /* 遇到假的自相关峰值 */
    else
        $count\_mf \leftarrow count\_mf + 1$ /* 终于遇到真的自相关峰值:-) */
        $count \leftarrow 0$
        $count\_m \leftarrow 0$
    end if
end if
if $count \geq num\_th$ **then**
    同步失败，计数器清零
end if
if $m < f\_mth$ and $mf > f\_mfth$ **then**
    if $count\_mf \geq 5$ **then** /* 这里统计的自相关峰值都是真自相关峰值 */
        同步成功，收集 96 个符号值
    else
        同步失败，计数器清零，相关移位寄存器清零
    end if
end if

改进的同步算法中控制一个计数器 $count\_m$，用来计数两个自相关峰值之间互相关峰值的持续时间。如果这个持续时间超过某个阈值（如 32），则代表同步失败，重新开始同步。算法使用另一个计数器 $count\_mf$ 来计数相邻之间的互相关峰值持续时间小于某个阈值的自相关峰值个数，当 $count\_mf$ 达到一定数目时，才认为有可能同步成功。

频率同步技术

2.11 Usage

当前对于每个模块和流水线的测试使用 4 个参数：

```
./a.out enum_fs mod_type n_tx_ant n_rx_ant
```

其中enum_fs 的类型为LTE_PHY_FS_ENUM，是一个枚举类型，用来选择采样频率。共有 6 个值。根据表1.1，这个值会决定 DFT，FFT 的点数，而且会决定 CP 长度等等参数值.mod_type 也是一个枚举类型，决定调制解调方法，一共有 4 种；n_tx_ant 和n_rx_ant 表示发送天线和接受天线的数目，最大值都为LTE_PHY_N_ANT_MAX，在 lte_phy.h 中定义，目前暂时为 4。这些参数共同决定了各个模块输入输出的长度，这些长度值的具体计算公式在“Benchmark Suite Implementation”一章中已经给出。

需要注意的是，如果要使用 WiBench 的一组测试数据（处理一帧数据时 Turbo encoder 的长度为 2368，FFT 和 DFT 的点数分别为 128 和 75，调制解调方法为 QAM16），则enum_fs 必须选择为 0，使采样频率为 1.92MHz，这样 FFT 和 DFT 的点数分别为 128 和 75；mod_type 必须选择为 2，表示调制解调方法为 QAM16；发送和接收天线数目为 2；再根据 Turbo 的交织长度为离散值，最后根据上述计算公式推算得到 Turbo encoder 的输入长度为 2368，且BLOCK_SIZE 至少也为 2368。

另外，虽然 Turbo 可以作为一个独立的算法使用，这时其输入输出长度不受上述参数的影响。但是当作为 PHY 流水线的一个模块时，比如对于 Encoding，其输出长度也必须由其下游模块的长度决定。

3 Experiment

3.1 Power Measurement

在 MIC 和 K20 上测量 power 和 energy 的总体思路如图3.1所示。要用一个监测 MIC 和 GPU 上温度传感器的进程和线程来对 power 不断的采样与收集，然后设置一个合理的时间窗口，最后累加得到 energy 值。

如果程序的运行时间很短，比如 LTE PHY 层模块运行一帧的时间就很短，甚至都无法达到时间窗口或和时间窗口在同一个数量级，我们就必须增加运行时间使其远大于时间窗口值，以使测得的 energy 值足够准确。

目前检测 K20 的 power 值只能在 host 端进行，所以我们用 host 端的一个进程通过调用 NVIDIA 提供的 API 来获取 device 端的 power 数据。这里有一个 power 值的 threshold，只有 device 上的 power 值达到或超过这个 threshold 才进行采样，并在统计的同时累加采样时间，将这个时间称为 device 活动的时间（active time）。增加程序运行时间我们目前有两种思路：1）调用 kernel 多次；2）在 kernel 中让一个线程重复执行相同任务多次。思路 1 得到的时间是 kernel 运行一次时间的线性倍数，所以这个时间能

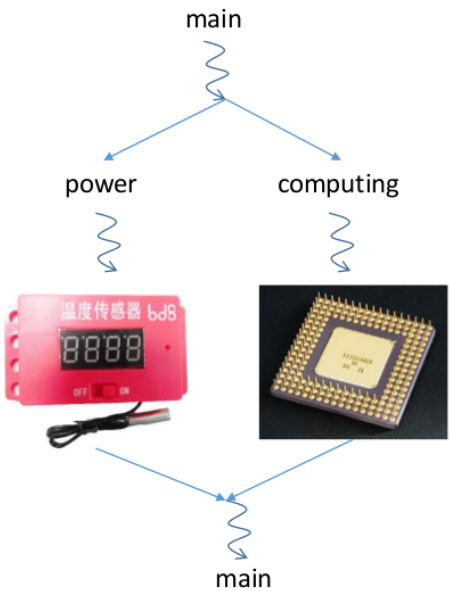

Figure 3.1: Power measurement on MIC and GPU

够理想的模拟线性扩大 kernel 执行时间，但是这样做得到的 active time 要比 kernel 执行多次的时间累加值大，这可能是由于 kernel 的 launch 和 retire 引起的 power 上升时间，也被统计到了 kernel 执行时间中。对于第二种思路，由于 GPU 上线程调度，其实线性增加一个线程的执行时间并不意味着 kernel 执行时间的线性增加，但是由于 kernel 只被 launch 了一次，因此得到的 active time 和 kernel 的运行时间较为吻合；经过实验测试，这种方法测得的 kernel 执行时间与第一种方法通过多次调用 kernel 得到的时间相比得到的误差，要比第一种方法中 active time 和 kernel 执行多次得到的时间相比的差值要小很多。因此目前我们使用第二种思路来测量 energy。另外，我们可以在第二种方法中的 kernel 中，在当一个线程重复执行相同任务的迭代之间加同步操作来进一步减小误差值，使得到的时间更接近 kernel 执行一次的线性倍数。

注意：即使只 launch kernel，而 kernel 里什么也不做，这个 launch 的 overhead 相对于 lte physical 中这些模块的一次运行时间也比较大。

在 MIC 上，我们目前使用 native 模式运行程序，所以直接在 device 端使用一个监测 sensor 的线程，采样和统计的思路和 GPU 上的做法类似。不过说是监测 sensor，其实是不断的读 device 上"/sys/class/micras/power"这个文件来获取其中的 power 数据，这个读文件的操作会给我们采样时间窗口增加误差。也就是说，我们的初衷是要让采样时间窗口的累加值和 MIC 上计算线程的运行时间大致相同，但是由于监测线程中除采样时间之外的其它开销（例如上述的读文件操作），使得线程的运行时间要大于采样时间的累加值，因此我们在最后计算 energy 时要适当修正采样时间窗口值，让每个时间窗口值加上这些其它操作的开销。

在测 dft 和 fft 的 energy 时，当数据集较小时（如 dft 的长度为 75 和 150 时），power 的 active time 要比设计 kernel 的运行时间小不少，当数据集逐渐增大时，二者的差距逐渐减小趋近相等。我想这时因为数据集较小时，活动的线程数据较少，芯片上的能耗也较小；当活动线程数目随着数据集增多，能耗值也随之增大，采样 power 值超过 threshold 的次数越来越多。

3.2 CPU

在用 20 个线程运行 Scrambling 时，使用 RAPL 测量 power 和 energy 时得到的运行时间比不测 power 时测得的运行时间少一个数量级，这时不能用直接测得的时间求 Performance per Watt，要用 RAPL 得到的那个时间，因为 power 是用 RAPL 算出来的。若出现这种情况要注意修正。

3.3 TODO List

```
amplxe-cl -collect knc-general-exploration -knob enable-vpu-metrics=true -knob enable-tlb-metrics=true -kr
```

References

[1] 3GPP. LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding. Technical Report 3GPP TS 36.212 version 11.3.0 Release 11, ETSI, 2013.

[2] 3GPP. LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding. Technical Report 3GPP TS 36.211 version 11.3.0 Release 11, ETSI, 2013.

[3] Eric Dahlman, Stefan Parkvall, Johan Sköld, and et al. *3G Evolution: HSPA and LTE for Mobile Broadhand, 2nd Edition*. Academic Press, 2008.

[4] Jah-Ming Hsu and Chin-Liang Wang. A Parallel Decoding Scheme for Turbo Codes. In *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, ISCAS'98, 1998.

[5] White Paper. LTE in a Nutshell: The Physical Layer. Technical report, Telesystem Innovations, 2010.

[6] Seokhyun Yoon and Yeheskel Bar-Ness. A Parallel MAP Algorithm for Low Latency Turbo Decoding. *Communications Letters, IEEE*, 6:288–290.