ABR 算法

刘小杰（QQ：472249968)

（主要针对 P，B 帧和连续 I 帧）

1 首先图像半精度的残差 STAD 之和,利用 J=SATD+λR 进行模式选择

2 利用当前帧的 SATD 计算图像的模糊复杂度(Blurred Complexity)，设当前帧的 SATD 为 SATD(i),累积复杂度为 cplx_sum(i),

cplx_sum(i)=0.5cplx_sum(i-1)+SATD(i),

当前帧 i 的模糊复杂度为 cplx_blur(i),

$$\text{cplx\_blur(i)}=\frac{cplx\_sum(i)}{cplx\_count(i)},$$

其中 cplx_count(i)表示累积加权帧数，

cplx_count(i)=0.5cplx_count(i-1)+1，

原始量化参数为 qscale_raw,

$$\text{qscale\_raw(i)} = cplx\_blur(i)^{1-qc},$$

qc 为压缩控制参数，用来调控 qscale_raw 的幅度。

qscale_raw 需要两次修正。

第一次修正利用 rate_factor 修正

$$\text{qscale\_adjust(i)}=\frac{qscale\_raw(i)}{rate\_factor(i)},$$

其中

$$\text{rate\_factor(i)}=\frac{wanted\_bits\_window(i)}{cplxr\_sum(i)},$$

其中 wanted_bits_window 表示到当前编码帧为止所有目标比特累计值。cplxr_sum(i) 为根据前一帧的量化等级参数求取情况估计出的当前帧复杂度,是一个迭代量,初始值为.01 * pow(7.0e5, m_qCompress) * pow(m_ncu, 0.5) * tuneCplxFactor;

`double tuneCplxFactor = (m_ncu > 3600 && m_param->rc.cuTree) ? 2.5 : m_isGrainEnabled ? 1.9 : 1;` 它可以反映经过码率比率因子 rate_factor 修正后的复杂度情况,计算公式如下:

$$\text{cplxr\_sum(i)= cplxr\_sum(i-1)+bits(i-1)*}\frac{qscale\_adjust(i-1)}{qscale\_raw(i-1)}$$

其中,bits(i 一 l)是上一帧编码得出的实际比特数,qscale_raw(i 一 1)为前一帧的初始量化等级,qscale_adjust(i 一 l)是前一帧经过 rate_factor 因子修正的量化等级参数.

X265 中

```
{
    rateControlUpdateStats
    m_cplxrSum += rce->rowCplxrSum;
    rateControlEnd函数中
        if (rce->sliceType != B_SLICE)
        {
            /* The factor 1.5 is to tune up the actual bits, otherwise the cplxrSum is scaled too low
             * to improve short term compensation for next frame. */
            m_cplxrSum += (bits * x265_qp2qScale(rce->qpaRc) / rce->qRceq) - (rce->rowCplxrSum);
        }
        else
        {
            /* Depends on the fact that B-frame's QP is an offset from the following P-frame's.
             * Not perfectly accurate with B-refs, but good enough. */
            m_cplxrSum += (bits * x265_qp2qScale(rce->qpaRc) / (rce->qRceq * fabs(m_param->rc.pbFactor))) - (rce->rowCplxrSum);
        }
}
```

第二次修正,利用溢出判断因子 overflow 来修正,它可以表示出总目标比特和实际产生的总比特的之间的偏差,修正公式如下:

qscale(i)= qscale_adjust(i)*overflow(i).

overflow 限定在 0.5 到 2 之间。

$$\text{overflow}(i) = 1 + \frac{total_{bits(i-1)} - wanted\_bits(i-1)}{abr\_buffer(i)}$$

其中,total_bits(i 一 l)为到前一帧为止编码所产生的实际比特数之和;wanted_bits(i-l)
为到前一帧为止累计的目标比特数之和,和上文所提到的 wanted_bits_window(i)相比
差值为当前帧的目标比特。abr_buffer(i)称为平均比特率缓冲区,初始值是两倍的平均目标比特和瞬时码率容忍度(默认为 1.0)的乘积,是根据当前帧数和编码帧率而增长的,理论上没有上限。

$QP = \alpha + \beta \log2(\frac{qscale}{\gamma})$, 其中 α =12， β =6， γ =0.85.

预估 bits 的计算

在RateControl类中有

```
Predictor m_pred[4];       /* Slice predictors to preidct bits for each Slice type - I,P,Bref and B */ 用于估计当前帧的bits
```

在RateControlEntry结构体中有

```
    Predictor  rowPreds[3][2];
    Predictor* rowPred[2];
 rce->rowPred[0] = &rce->rowPreds[m_sliceType][0];
 rce->rowPred[1] = &rce->rowPreds[m_sliceType][1];

struct Predictor
{
    double coeffMin;
    double coeff;
    double count;
    double decay;
    double offset;
};
```

Bit=（coeff*satd*+offset）/(qScale*count);

```
updatePredictor(rce->rowPred[0], qScaleVbv, (double)rowSatdCost, encodedBits);
updatePredictor(Predictor *p, double q, double var, double bits)
{
    if (var < 10)
         return;
     const double range = 2;
     double old_coeff = p->coeff / p->count;
     double old_offset = p->offset / p->count;
     double new_coeff = X265_MAX((bits * q - old_offset) / var, p->coeffMin );
     double new_coeff_clipped = x265_clip3(old_coeff / range, old_coeff * range, new_coeff);
     double new_offset = bits * q - new_coeff_clipped * var;
     if (new_offset >= 0)
          new_coeff = new_coeff_clipped;
     else
          new_offset = 0;
     p->count    *= p->decay;
     p->coeff    *= p->decay;
     p->offset *= p->decay;
     p->count++;
     p->coeff    += new_coeff;
     p->offset += new_offset;
}
```

Predictor 的更新

1：在rowVbvRateControl()更新

{

所有帧情况下

```
updatePredictor(rce->rowPred[0], qScaleVbv, (double)rowSatdCost, encodedBits);
```

在非I帧情况下

当前行的qp小于参考qp

```
predictSize(rce->rowPred[1], qScale, intraCostForPendingCus);
```

}

2：在updateVbv（）更新（rateControlEnd调用updateVbv）

updatePredictor(&m_pred[predType], x265_qp2qScale(rce->qpaRc), (double)rce->lastSatd, (double)bits);

其他参数的更新 1

I 帧 qp 调整

1 m_accumPNorm 初始化 m_accumPNorm = .01; //init 函数调用

2 m_accumPQp 初始化 (m_param->rc.rateControlMode == X265_RC_CRF ? CRF_INIT_QP : ABR_INIT_QP_MIN) * m_accumPNorm;//init 函数调用

这两个参数的更新在函数中 accumPQpUpdate()（在 rateControlStart 函数中调用）

```
void RateControl::accumPQpUpdate()
{
    m_accumPQp   *= .95;
    m_accumPNorm *= .95;
    m_accumPNorm += 1;
    if (m_sliceType == I_SLICE)
        m_accumPQp += m_qp + m_ipOffset;
    else
        m_accumPQp += m_qp;
}
```

在影响I帧qp的调整（在函数rateEstimateQscale调用中）

```
if ((m_sliceType == I_SLICE && m_param->keyframeMax > 1
    && m_lastNonBPictType != I_SLICE && !m_isAbrReset) || (m_isNextGop && !m_framesDone))
{
    if (!m_param->rc.bStrictCbr)
        q = x265_qp2qScale(m_accumPQp / m_accumPNorm);
         q /= fabs(m_param->rc.ipFactor);
    m_avgPFrameQp = 0;
}
```

其他参数的更新 2

m_bufferFillFinal, m_bufferFillActual 和　m_bufferExcess

1 初始化

m_bufferFillFinal = m_bufferSize * m_param->rc.vbvBufferInit;

m_bufferFillActual = m_bufferFillFinal;

m_bufferExcess = 0;

2 更新 updateVbv 函数中（在 rateControlEnd 函数中调用）

```
updateVbv(int64_t bits, RateControlEntry* rce)
 {
      m_bufferFillFinal -= bits;
      m_bufferFillFinal = X265_MAX(m_bufferFillFinal, 0);
      m_bufferFillFinal += m_bufferRate;

      if (m_param->rc.bStrictCbr)
      {
          if (m_bufferFillFinal > m_bufferSize)
          {
              filler = (int)(m_bufferFillFinal - m_bufferSize);
              filler += FILLER_OVERHEAD * 8;
          }
          m_bufferFillFinal -= filler;
          bufferBits = X265_MIN(bits + filler + m_bufferExcess, m_bufferRate);
          m_bufferExcess = X265_MAX(m_bufferExcess - bufferBits + bits + filler, 0);
          m_bufferFillActual += bufferBits - bits - filler;
      }
      else
      {
          m_bufferFillFinal = X265_MIN(m_bufferFillFinal, m_bufferSize);
          bufferBits = X265_MIN(bits + m_bufferExcess, m_bufferRate);
          m_bufferExcess = X265_MAX(m_bufferExcess - bufferBits + bits, 0);
          m_bufferFillActual += bufferBits - bits;
          m_bufferFillActual = X265_MIN(m_bufferFillActual, m_bufferSize);
      }

  }
```

3　在updateVbvPlan(Encoder* enc)中更新（由rateControlStart调用）

```
{
   m_bufferFill = m_bufferFillFinal;
   for (int i = 0; i < m_param->frameNumThreads; i++)
    {
        FrameEncoder *encoder = m_frameEncoder[i];
        if (encoder->m_rce.isActive && encoder->m_rce.poc != rc->m_curSlice->m_poc)
        {
             int64_t bits = m_param->rc.bEnableConstVbv ? (int64_t)encoder->m_rce.frameSizePlanned :
                        (int64_t)X265_MAX(encoder->m_rce.frameSizeEstimated, encoder->m_rce.frameSizePlanned);
            rc->m_bufferFill -= bits;
            rc->m_bufferFill = X265_MAX(rc->m_bufferFill, 0);
            rc->m_bufferFill += encoder->m_rce.bufferRate;
            rc->m_bufferFill = X265_MIN(rc->m_bufferFill, rc->m_bufferSize);
            if (rc->m_2pass)
                rc->m_predictedBits += bits;
        }
    }
```

```
}
```

Two Pass

Bits 估计 qScale2bits(rce, x265_qp2qScale(rce->qpNoVbv));

```
inline double qScale2bits(RateControlEntry *rce, double qScale)
{
    if (qScale < 0.1)
        qScale = 0.1;
    return (rce->coeffBits + .1) * pow(rce->qScale / qScale, 1.1)
           + rce->mvBits * pow(X265_MAX(rce->qScale, 1) / X265_MAX(qScale, 1), 0.5)
           + rce->miscBits;
}
```