Optimizing H265 Kernels

Ganesh Vernekar, Guide: Dr. Ramakrishna Upadrasta

Indian Institute of Technology, Hyderabad

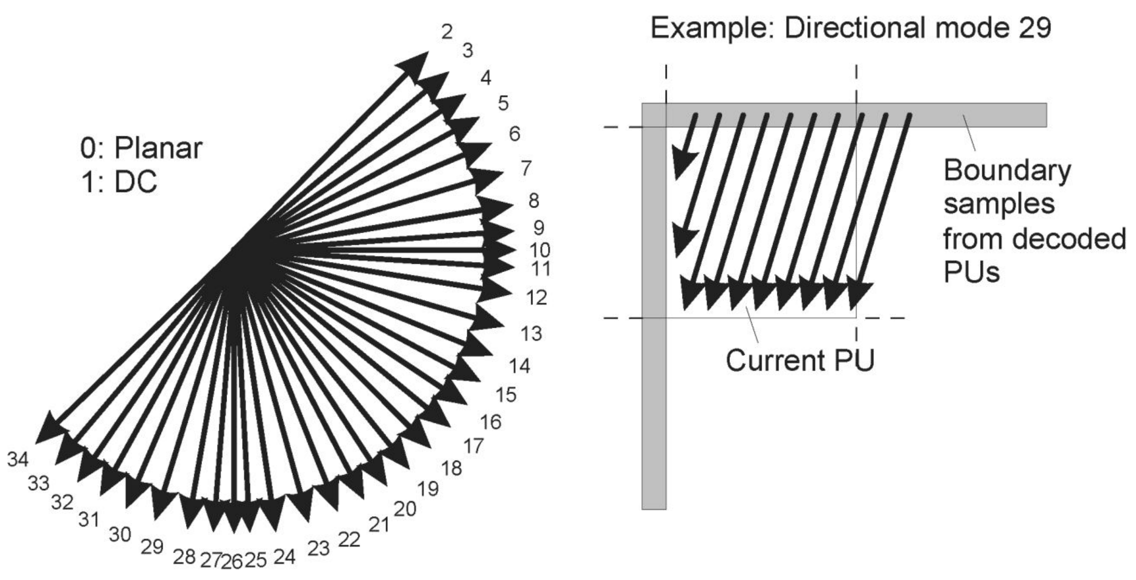
Introduction

H.265 is a block-oriented motion-compensation-based (predicting frames) video compression standard, developed as a successor to the very successful H264.

Compression is based on predicting pixels within the frame (**intra** prediction), or pixels across the frames (**inter** prediction).

Intra Prediction

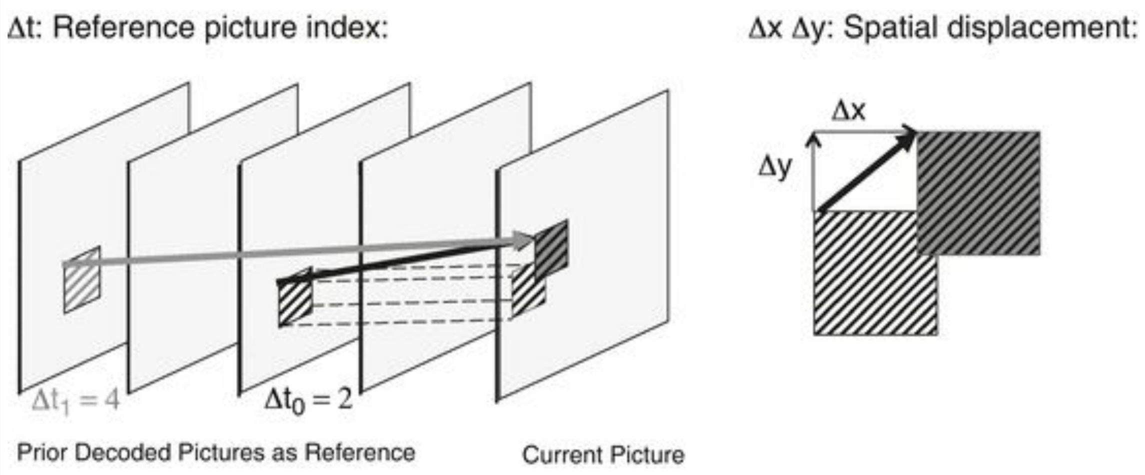
- A block of pixels within a frame is predicted using pixels from above and left side of the block.
- H265 has **35** intra prediction **kernels**. 33 are directional.
  - In contrast, H264 has 9 kernels with 8 being directional.
- In H265, Block sizes can be 4x4,8x8,16x16,32x32,64x64.
  - Only 4x4, 8x8, 16x16 in H264.



Inter Prediction

This is a predominant type of prediction (or) compressions method used.

In this the relative displacement of pixels from previous or future frame is encoded to get more compression.



Types of Frames

I-frames (intra coded frames) use only **intra** predictions.

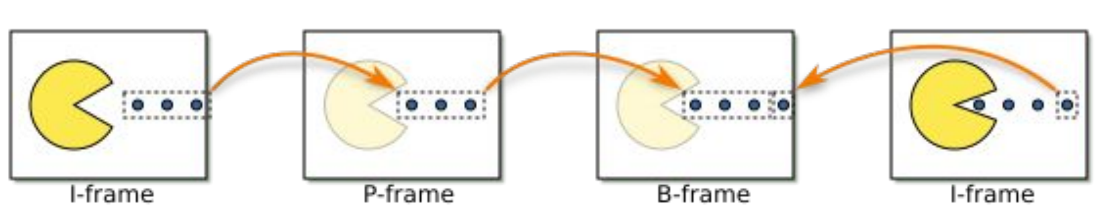
- Less frequent in a video. Used as reference frames. Least compressible, but highest quality.

P-frames (predicted picture) can use data from previous frames to decompress.

- More frequent compared to I-frames. Uses both **inter** and **intra** prediction.

B-frames (bidirectional predicted picture) can use both previous and forward frames for data reference.

- Most frequent. Absent in some settings. Gives highest compression. Mostly **inter** prediction.



Source of Code

Method: We took the industry standard Multicoreware's (**MCW**) H265 open source code (x265.org) as our base and work on top of that.

Two sets of kernels. Each set has implementation of all kernels separately or a single function called **`all_angle_pred`** which has all kernels combined, and they are used as required.

1. First set is written purely in **C++** to be compiled in the platform required.
   - A **single templated function** which takes care of all block sizes and direction modes.
   - `all_angle_pred` just calls the single kernels one after another, no optimizations.
2. Second set is written **assembly** (ASM) for **x86** and **ARM** architectures.
   - Individual implementation of direction modes for all sizes and `all_angle_pred` for all sizes with hand tuned optimization.

Our focus: "**Optimize the C++ kernels** using compiler optimization techniques".

We took the C++ kernels from MCW source code as the base and generated our own modified kernels and optimized it to get improvement in encoding performance.

Last Semester

- Studied H264 (found in SPEC-2006) and played with its kernels.
- Tried modifying H264 kernels and testing the modifications.
- Started studying H265 and how much more compute intensive it was when compared to H264 (~4x more kernels).
- Studied MCW C++ and ASM kernels.

This Semester

- Did a minor study of VP8/VP9 (encoding standard by Google).
- Generated individual H265 kernels from MCW C++ kernel templates.
- Tried and tested many different modifications in the generated kernels and many different optimization sequence.

Modifications in the kernels:

Following flow is the sequence of modification as a result of lots of different trials and tests.

Most of the work was on `all_angle_pred` function which happens to be most challenging.

1. Generated individual kernels using the MCW C++ kernel template as a base (per direction mode, per block size, i.e. 4 * 33 = 132 kernels) and removed some calculations and memory accesses which would else be present in the templated functions. [Last semester]
2. Fuse all single kernels (per block size) to get `all_angle_pred` for all block sizes.
   - Similar loops were fused and similar instructions were kept together, instead of having them one after another, while still maintaining isolation between the data.

```
// Kernel 1
stmts1_1;
for(...) { stmts2_1; }
stmts3_1;
for(...) { stmts4_1; }
....
```

```
// Kernel 2
stmts1_2;
for(...) { stmts2_2; }
stmts3_2;
for(...) { stmts4_2; }
....
```

```
// Fused Kernel 1 and 2
stmts1_1; stmts1_2;
for(...) { stmts2_1; stmts2_2; }
stmts3_1; stmts3_2;
for(...) { stmts4_1; stmts4_2; }
....
```

3. Manually combine common data between the kernels in the C++ kernel generator.
4. Combined random looking constants inside the half unrolled loop into constant arrays for better vectorization. Inspired from study of MCW assembly code.

```
for(int x=0; x<32; x++) {
    ...
    dst1[1*32 + x] = (19*ref_1[0+0] + 13*ref_1[0+1] + 16) >> 5;
    dst1[1*32 + x] = (6*ref_1[0+1] + 26*ref_1[0+2] + 16) >> 5;
    dst1[1*32 + x] = (25*ref_1[1+2] + 7*ref_1[1+3] + 16) >> 5;
    ...
    dst2[2*32 + x] = (17*ref_2[-1+0] + 15*ref_2[-1+1] + 16) >> 5;
    dst2[2*32 + x] = (2*ref_2[-2+1] + 30*ref_2[-2+2] + 16) >> 5;
    dst2[2*32 + x] = (19*ref_2[-2+2] + 13*ref_2[-2+3] + 16) >> 5;
    ...
}
```

```
const int fracp_1[] = {19,6,25,...};
const int fracq_1[] = {13,26,7,...};
const int off_1[] = {0,0,1,...};
const int fracp_2[] = {17,2,19,...};
const int fracq_2[] = {15,30,13,...};
const int off_2[] = {-1,-2,-2,...};
...

for(int y=0; y<32; y++) {
    for(int x=0; x<32; x++) {
        dst[y*32 + x] = (fracp_1[y]*ref_1[off_1[y]+x]
                         + fracq_1[y]*ref_1[off_1[y]+x+1] + 16) >> 5;
    }
}
dst += 32*32;
for(int y=0; y<32; y++) {
    for(int x=0; x<32; x++) {
        dst[y*32 + x] = (fracp_2[y]*ref_2[off_2[y]+x]
                         + fracq_2[y]*ref_2[off_2[y]+x+1] + 16) >> 5;
    }
}
...
```

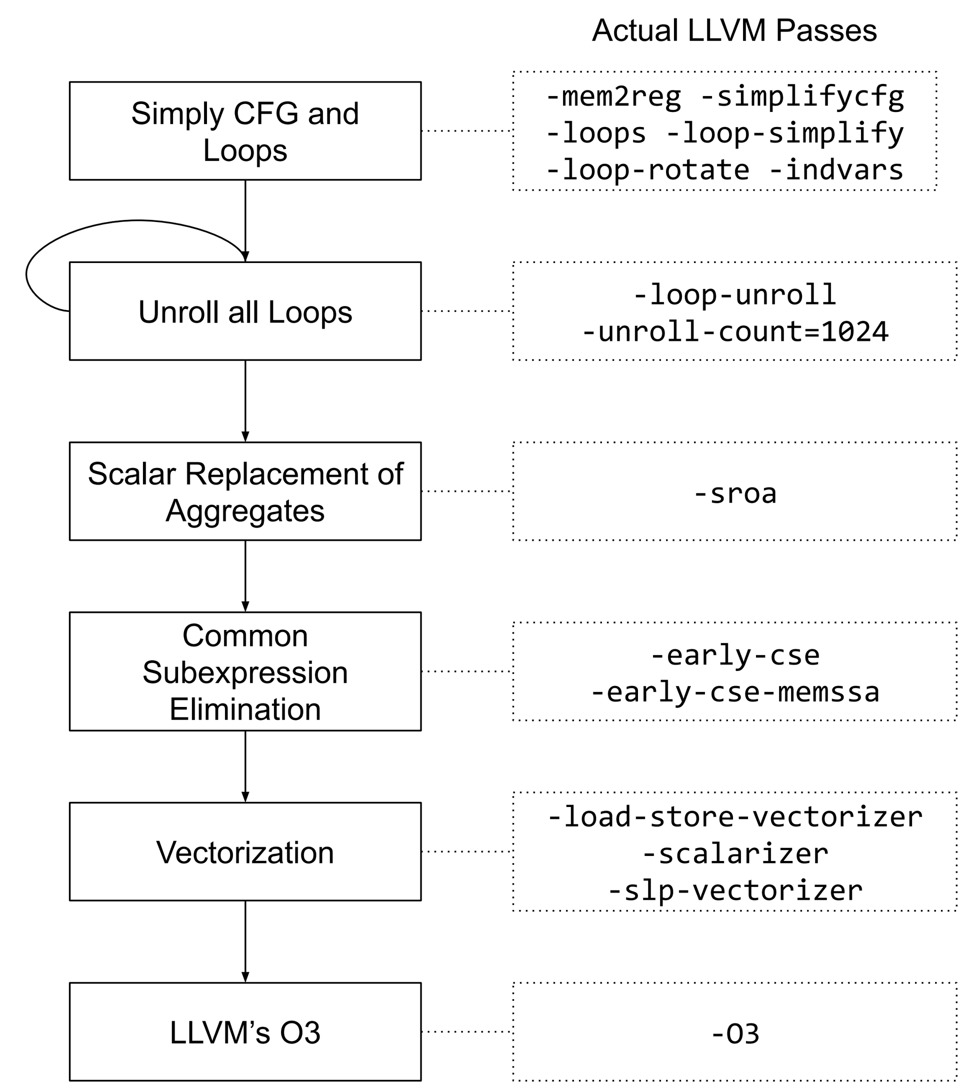
5. Converted doubly nested loops into single loops by unrolling the inner loop.
6. Broke down large expressions into multiple smaller expressions inside the half unrolled loop, so that similar expressions are together after total unrolling (optimizations discussed later).

```
// a1 ~ fracp_1, a2 ~ ref_1, a3 ~ off_1, a4 ~ fracq_1
for(int y=0; y<32; y++) {
    dst[y*32+0] = ((a1[y]*a2[a3[y]+0])+(a4[y]*a2[a3[y]+1])+16)>>5;
    dst[y*32+1] = ((a1[y]*a2[a3[y]+1])+(a4[y]*a2[a3[y]+2])+16)>>5;
    ...
}
```

```
for(int y=0; y<32; y++) {
    tmp_dst[0] = a1[y]*a2[a3[y]+0];
    tmp_dst[1] = a1[y]*a2[a3[y]+1];
    ...
    tmp_dst[0]+ = a4[y]*a5[a3[y]+1];
    tmp_dst[1]+ = a4[y]*a5[a3[y]+2];
    ...
    dst[y*32+0] = (tmp_dst[0]+16)>>5
    dst[y*32+1] = (tmp_dst[1]+16)>>5
    ...
}
```

Compiler Optimization

We used **LLVM** compiler toolchain to perform the following optimizations.



Initial Tests Results

Machine: 2 × Intel(R) Xeon(R) X5675 @3.07GHz (Total 12 cores, 24 HW threads), 157 GiB RAM

Videos:

- Generated 6000x4000 15 fps raw video from 941 Farewell pics.
- Generated 2160p10 video from same 941 pics.
- 5 2160p50 videos from https://media.xiph.org/video/derf/

Settings:

- B-Frames disabled.
- At max 5 P-frames between 2 consecutive I-frame (5:1 ratio).

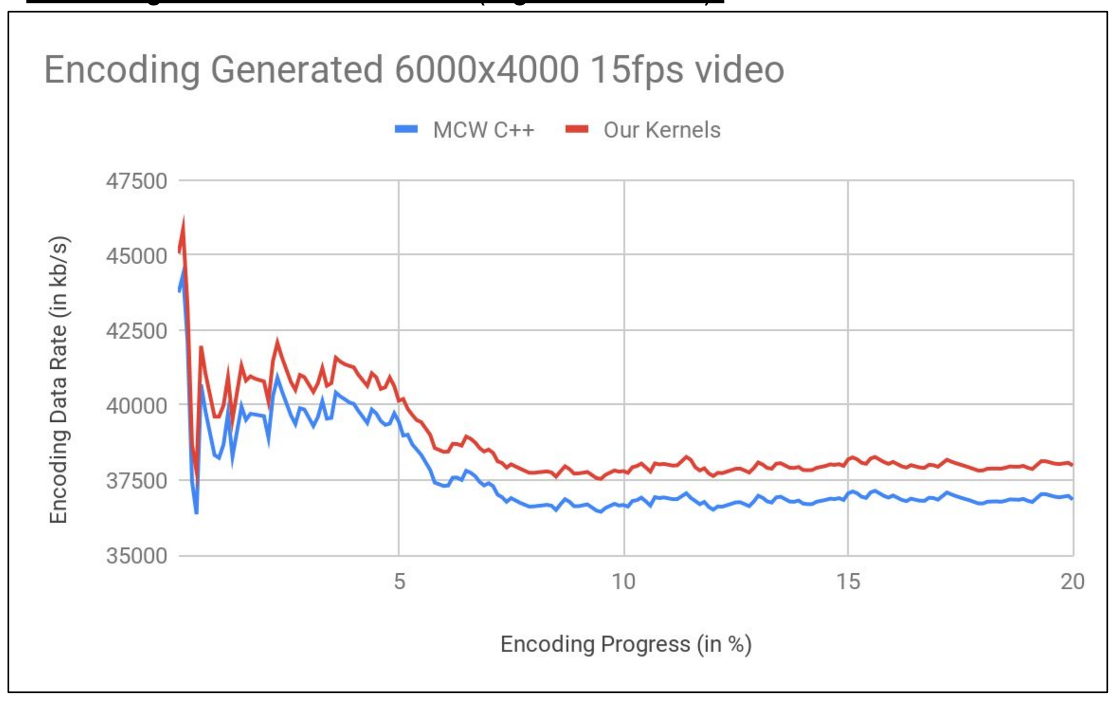
Average data processing speed for I and P frames (higher is better):

Videos	I-frame (kb/s data processed)		P-frame (kb/s data processed)	
	MCW C++	Our Kernels	MCW C++	Our Kernels
generated_6kx4k_15fps	50285	50771.64 (+486.64)	34149.56	35430.77 (+1281.21)
generated_2160p10	17618.55	17809.07 (+190.52)	10908.58	11330.01 (+421.43)
crowd_run_2160p50	125933.75	127436.21 (+1502.46)	22630.15	22824.63 (+194.48)
ducks_take_off_2160p50	112553.07	112499.99 (-53.08)	37257.64	37246.89 (-10.75)
in_to_tree_2160p50	96882.99	96788.12 (-94.87)	6513.32	6534.2 (+20.88)
old_town_cross_2160p50	74279.62	74427.94 (+148.32)	2027.83	2049.24 (+21.41)
park_joy_2160p50	168585.14	168730.73 (+145.59)	29506.8	29604.15 (+97.35)

Encoding time (lower is better):

Videos	Encoding time (seconds)	
	MCW C++	Our Kernels
generated_6kx4k_15fps	3199.39	3095.61 (-103.78)
generated_2160p10	1162.09	1123.68 (-38.41)
crowd_run_2160p50	331.14	322.45 (-8.69)
ducks_take_off_2160p50	432.38	419.94 (-12.44)
in_to_tree_2160p50	314.46	305.87 (-8.59)
old_town_cross_2160p50	255.87	249.83 (-6.04)
park_joy_2160p50	354.89	348.13 (-6.76)

Encoding Data Rate over time (higher is better):



Conclusion and Future work

- Though the code size bloats with these changes, we could see good improvements in encoding speed wherever intra prediction was used.
- More effective for higher resolutions.
- There is still room for improvement in C++ code.
  - MCW ASM is 2-4x faster than MCW C++.

Future Work:

- Reduce the size of generated kernels via modification and compiler optimizations.
- Inter prediction forms a large part of encoding process (P and B frames). Explore Inter predictions and optimize that.
- Explore other parts written in ASM, like video filters and copying blocks and other memory operations, and see how it can be made better in C++ using compiler optimizations.

Image Credits: http://iphome.hhi.de/wiegand/assets/pdfs/2012_12_IEEE-HEVC-Overview.pdf, https://en.wikipedia.org/wiki/Video_compression_picture_types