ABSTRACT

Being developed by Joint Video Exploration Team (JVET), Future Video Coding (FVC) aims at higher resolutions and higher compression performance than the state-of-the-art HEVC standard, undoubtedly at the cost of further computing increases. As an efficient computing platform, Graphics Processing Unit (GPU) is often used to accelerate encoding. But with the adoption of instruction set acceleration in the reference software of FVC, previous methods often become less efficient or even lead to a lower speed. In this paper, based on the comparative analysis of the time consumption between HEVC and FVC, we propose a GPU based acceleration method for the most computation-intensive step – frame interpolation of FVC, where frame caching strategy and a multi-stream mechanism is designed to make the best of GPU resources. Experimental results show that compared with the instruction set accelerated reference software of FVC, our method could achieve average 67.12% speed-up gains on the interpolation module and average 6.35% speed-up gains on overall encoding with exactly the same performance as before.

Index Terms— FVC, Frame Interpolation, GPU

1. INTRODUCTION

In recent years, the next generation of video coding standard — Future Video Coding (FVC) [1] is prepared based on the state-of-the-art HEVC [2] with different encoding methods for higher encoding efficiency. Unfortunately, these methods often lead to significant increase of computational complexity of FVC, especially at the encoder side, where inter-frame prediction, including sum of absolute differences (SAD) calculation and motion-compensated interpolation (ITPL), usually accounts for half of the encoding time [3]. Nevertheless, due to their simple but massively repetitive computations, both SAD calculation and interpolation suit the Graphics Processing Unit (GPU) based acceleration, since GPU platforms, i.e. Compute Unified Device Architecture (CUDA) [4], can perform large-scale computation in parallel manners.

Many algorithms of GPU acceleration have previously been studied on HEVC. Lin et al. [5] proposed a parallel block-matching algorithm, where SAD of varied block sizes are calculated on GPU on a basis of full-search. Kim et al. [6] used adaptive search range to speed up the GPU-based motion estimation (ME) on HEVC. Kao et al. [7] proposed a new search pattern for better cooperation of CPU and GPU. Most of these acceleration methods mainly are focused on ME and SAD calculation, which require intensive computation and thus usually get assigned to GPU for acceleration in HEVC [8][9][10].

Despite the good results achieved by the above methods in HEVC, note that changes have taken place when it comes to FVC. Firstly, in contrast to the HEVC reference software HM [11], the FVC reference software JEM [12] has already adopted instruction set acceleration for ME and SAD calculation, which leaves much less margin of acceleration for existing methods. Secondly, existing methods of HEVC often necessitate adaptations of encoding configurations, which however may not be applicable to FVC. For example, references [5][6][7] all build on full-search but not advanced search methods, and both [6] and [13] work without asymmetric motion partition (AMP) and limit the largest coding tree unit (CTU) size to 32x32. Such adaptations could cause much performance loss of FVC and thus affect the effectiveness of conventional methods.

To our knowledge, few methods have been developed specially for GPU acceleration of FVC. Considering the long encoding time of JEM, efficient acceleration by means of GPU without any performance loss is still highly desired. In this paper, we first examine carefully each encoding module in terms of execution time and find out the most time-consuming step of FVC – motion-compensated (MC) frame interpolation. Then GPU based accelerated interpolation is...
proposed based on frame caching strategy and multi-stream mechanism. Experiment results show that 67.12% and 6.35% average speed-up gain can be achieved on the MC frame interpolation module and the whole encoding process respectively without any performance loss.

### 2. COMPUTATIONAL COMPLEXITY ANALYSIS OF JEM AND HM

Considering the differences between FVC and HEVC as well as the instruction set acceleration in JEM, it’s necessary to find the time-consuming parts of the reference softwares in preparation for acceleration of JEM. The tested reference softwares of HEVC and FVC are HM 16.15 and JEM 6.0 respectively and run under the “lowdelay_P_iter10” configuration with QP=37. We choose one representative sequence from each class of standard sequences and the test results are given in Table 1, where the time distribution of integer-pixel search, fractional-pixel search and advanced motion vector prediction (AMVP) is listed since inter-frame prediction usually accounts for nearly half of the total encoding time.

From Table 1 we can see that the time distribution of various modules varies dramatically from HM to JEM. Specifically, with instruction set acceleration enabled, both integer-pixel and fractional-pixel search account for less in JEM, however, AMVP accounts for more in JEM. Interpolation, which is utilized to generate the prediction samples for fractional sampling positions [2], is performed much more in JEM. The complexity contributed by interpolation in FVC can be explained as follows. Firstly, motion vectors of increased accuracy have been adopted. Take YUV420 video for example. The accuracy of the luma component increases from 1/4 pixel to 1/16 pixel and that of the chroma component increases from 1/8 pixel to 1/32 pixel. Secondly, FVC adopts more interpolation filters [1], which introduces extra complexity as well. So as the most time-consuming step interpolation desires acceleration highly. A GPU based method of interpolation acceleration is proposed in the next section.

### 3. FAST INTERPOLATION METHOD ON GPU

In our method, four types of interpolation in FVC, including both horizontal and vertical interpolation for luma and chroma, are performed on GPU. In order to combine the interpolation calculation intensively, we perform interpolation on the entire reference frame rather than CU by CU. Frame caching strategy is used to avoid repeating interpolation, and multi-stream mechanism and shared memory of kernel functions are designed for high efficiency.

For the purpose of efficient interpolation of the entire reference frame, the sizes of thread block (TB), which have a close relationship with GPU calculation performance, have to vary with different sizes of YUV sequences to take full advantage of the computational resources. Obviously, efforts should be taken to minimize the elapsed time of interpolation on GPU, which can be expressed as

\[
\min_{t_x, t_y} \text{Kernel}(t_x, t_y)
\]

s.t.

\[
\begin{align*}
W & \mod (\text{Step}_x \cdot t_x) = 0 \\
H & \mod (\text{Step}_y \cdot t_y) = 0
\end{align*}
\]

where \(\text{Kernel}(t_x, t_y)\) denotes the elapsed time of kernel functions; \(t_x\) and \(t_y\) denote the width and height of TB respectively; \(W\) and \(H\) denote the width and height of the image respectively; and \(\text{Step}_x\) and \(\text{Step}_y\) are the integer parameters respectively in the horizontal and vertical direction. As the key step in our method, the kernel function has to be carefully designed to obtain the optimal solution of (1).

#### 3.1. Design of Kernel Function

As illustrated in Figure 1, the kernel function builds upon the shared memory of GPU, which helps high memory access speed. Every TB transfers a part of the image, denoted by “Image Block (IB)” in the following, to its corresponding shared memory. An IB contains a data block (DB) and two apron blocks (AB). Each pixel in the DB corresponds to one pixel in the output results. Considering the limited capacity of the shared memory, the size of DB is an integer multiple of the size of TB, i.e. the parameter \(\text{Step}\). For the sake of the interpolation at the edges, AB has the same size of TB. Fig. 1

### Table 1. Time distribution for each test sequence.

<table>
<thead>
<tr>
<th>Modules</th>
<th>Kimono 1920x1080</th>
<th>BasketballDrill 832x480</th>
<th>BasketballPass 416x240</th>
<th>FourPeople 1280x720</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int-pel</td>
<td>12.19% 3.85%</td>
<td>9.34% 3.30%</td>
<td>9.21% 3.53%</td>
<td>5.41% 1.68%</td>
<td>9.04% 3.09%</td>
</tr>
<tr>
<td>Sub-pel</td>
<td>14.24% 5.28%</td>
<td>15.61% 6.25%</td>
<td>15.28% 5.86%</td>
<td>19.34% 6.38%</td>
<td>16.12% 5.95%</td>
</tr>
<tr>
<td>AMVP</td>
<td>0.28% 2.21%</td>
<td>0.27% 2.95%</td>
<td>0.22% 2.83%</td>
<td>0.24% 0.09%</td>
<td>0.25% 2.02%</td>
</tr>
<tr>
<td>ITPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-pel</td>
<td>16.01% 4.00%</td>
<td>17.60% 4.57%</td>
<td>17.51% 4.52%</td>
<td>22.61% 5.64%</td>
<td>18.43% 4.68%</td>
</tr>
<tr>
<td>AMVP</td>
<td>2.16% 8.57%</td>
<td>1.35% 8.13%</td>
<td>1.73% 8.15%</td>
<td>1.01% 3.22%</td>
<td>1.56% 7.02%</td>
</tr>
</tbody>
</table>

Fig. 1
takes row interpolation for example to demonstrate the GPU kernel function design. Algorithm 1 shows the pseudo code of the kernel function.

In (1), Step influences the size of the shared memory in each TB and is set to 4 empirically in this paper. By examining different configurations in terms of theoretical occupancy (TO), achieved occupancy (AO) and the theoretical number of resident TBs in a streaming multiprocessor (TB per SM), the optimal TB sizes of (1) are given in Table 2. Since 66% AO is enough to saturate the bandwidth [14], we can see that all the proposed kernel functions work at high efficiency except those for the chroma component in class C and D, where the AO is limited by the total number of TBs. The reason of that is the size of the sequences in class C and D, especially 416x240 sequences in class D, is too small to saturate the bandwidth. And all the TB per SM is more than 10 except for the luma component in Class C, which results from a trade-off between the occupancy and the TB per SM. Generally, 16x8 or 8x8 are common sizes of TBs [5], but the combinations for the luma component in Class C, 35x8 and 70x4, contain the odd factors 5 and 7, so it's hard to maximize both the occupancy rate and the number of activated TBs per SM at this time. However, the trade-off can still ensure good acceleration effects at these configurations.

3.2. Frame Caching Strategy

In inter-frame prediction, each frame may be referenced by multiple frames. Take the frequently used “lowdelay_P_jvet10” configuration for example. Each P frame may reference 4 frames and each frame may be referenced 13 times, which may lead to repeated interpolation. Fortunately, two adjacent P frames only have one different reference frame, which means interpolated images can be reused. Therefore, we use a frame caching strategy to control the interpolation procedure as shown in Fig. 2. When encoding each P frame, GPU only performs interpolation for the new reference frame with the interpolated images of the other three reference frames cached in the host memory. In this way, both I/O bandwidth and computational resources can be...

![Fig. 1. An illustration of kernel function design and thread organization.](image)

**Table 2. Optimal configurations of kernel functions.**

<table>
<thead>
<tr>
<th>Kernel configuration</th>
<th>Lx</th>
<th>Ly</th>
<th>TO</th>
<th>AO</th>
<th>TB per SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y horz</td>
<td>24</td>
<td>8</td>
<td>94%</td>
<td>86.1%</td>
<td>10</td>
</tr>
<tr>
<td>Y vert</td>
<td>16</td>
<td>9</td>
<td>94%</td>
<td>85.8%</td>
<td>12</td>
</tr>
<tr>
<td>U/V horz</td>
<td>12</td>
<td>9</td>
<td>100%</td>
<td>85.7%</td>
<td>16</td>
</tr>
<tr>
<td>U/V vert</td>
<td>16</td>
<td>9</td>
<td>94%</td>
<td>82.6%</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y horz</td>
<td>35</td>
<td>8</td>
<td>98%</td>
<td>86.0%</td>
<td>7</td>
</tr>
<tr>
<td>Y vert</td>
<td>70</td>
<td>4</td>
<td>98%</td>
<td>88.9%</td>
<td>7</td>
</tr>
<tr>
<td>U/V horz</td>
<td>4</td>
<td>16</td>
<td>100%</td>
<td>63.0%</td>
<td>32</td>
</tr>
<tr>
<td>U/V vert</td>
<td>16</td>
<td>4</td>
<td>100%</td>
<td>65.8%</td>
<td>32</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y horz</td>
<td>16</td>
<td>8</td>
<td>100%</td>
<td>78.5%</td>
<td>16</td>
</tr>
<tr>
<td>Y vert</td>
<td>16</td>
<td>11</td>
<td>94%</td>
<td>76.8%</td>
<td>10</td>
</tr>
<tr>
<td>U/V horz</td>
<td>8</td>
<td>12</td>
<td>98%</td>
<td>29.2%</td>
<td>21</td>
</tr>
<tr>
<td>U/V vert</td>
<td>16</td>
<td>11</td>
<td>94%</td>
<td>42.5%</td>
<td>10</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y horz</td>
<td>8</td>
<td>16</td>
<td>100%</td>
<td>87.1%</td>
<td>16</td>
</tr>
<tr>
<td>Y vert</td>
<td>16</td>
<td>9</td>
<td>94%</td>
<td>84.3%</td>
<td>12</td>
</tr>
<tr>
<td>U/V horz</td>
<td>14</td>
<td>9</td>
<td>100%</td>
<td>74.5%</td>
<td>12</td>
</tr>
<tr>
<td>U/V vert</td>
<td>16</td>
<td>9</td>
<td>94%</td>
<td>80.2%</td>
<td>16</td>
</tr>
</tbody>
</table>

**Algorithm 1: Kernel function for interpolation**

*Input:* Pointer to the reference frame: src
Width of the reference frame: imageW
Stride of the reference frame: stride_src
Stride of the interpolated reference frame: stride_dst
Interpolation filter: filter,
Span of the interpolation filter: span,
Step: step,
Pointer to the interpolated reference frame: dst

*Output:* Void

```c
void __shared__ sh_mem[blockDim.y][(step + 2) * blockDim.x];
baseX = (blockIdx.x * step - 1) * blockDim.x + threadIdx.x;
baseY = blockDim.y * blockIdx.y + threadIdx.y;
dst += baseY * stride_dst + baseX;
src += baseX * stride_src + baseY;

for i in range(0, 1)
for j in range(−radius, radius) + 2)
do
sum += filter[−radius + j] ×
sh_mem[threadIdx.y][threadIdx.x + i * blockDim.x] × src[i × blockDim.x];
dst[i × blockDim.x] = sum/(radius × 2);
```

In (1), Step influences the size of the shared memory in each TB and is set to 4 empirically in this paper. By examining different configurations in terms of theoretical occupancy (TO), achieved occupancy (AO) and the theoretical number of resident TBs in a streaming multiprocessor (TB per SM), the optimal TB sizes of (1) are given in Table 2. Since 66% AO is enough to saturate the bandwidth [14], we can see that all the proposed kernel functions work at high efficiency except those for the chroma component in class C and D, where the AO is limited by the total number of TBs. The reason of that is the size of the sequences in class C and D, especially 416x240 sequences in class D, is too small to saturate the bandwidth. And all the TB per SM is more than 10 except for the luma component in Class C, which results from a trade-off between the occupancy and the TB per SM. Generally, 16x8 or 8x8 are common sizes of TBs [5], but the combinations for the luma component in Class C, 35x8 and 70x4, contain the odd factors 5 and 7, so it’s hard to maximize both the occupancy rate and the number of activated TBs per SM at this time. However, the trade-off can still ensure good acceleration effects at these configurations.

### 3.2. Frame Caching Strategy

In inter-frame prediction, each frame may be referenced by multiple frames. Take the frequently used “lowdelay_P_jvet10” configuration for example. Each P frame may reference 4 frames and each frame may be referenced 13 times, which may lead to repeated interpolation. Fortunately, two adjacent P frames only have one different reference frame, which means interpolated images can be reused. Therefore, we use a frame caching strategy to control the interpolation procedure as shown in Fig. 2. When encoding each P frame, GPU only performs interpolation for the new reference frame with the interpolated images of the other three reference frames cached in the host memory. In this way, both I/O bandwidth and computational resources can be...
saved greatly.

![Frame caching strategy. GPU only interpolates the new reference frame.](image)

### 3.3. Multi-Stream in GPU Programming

In GPU programming, the CUDA stream, which can bind a sequence of commands that execute in order, is used to execute asynchronous commands in concurrency [15]. With the increase of image sizes, GPU interpolation involves not only massive computation but also time-consuming I/O transfer. Thus by means of CUDA streams it is significant to hide I/O transfer in a concurrent way for high efficiency.

In our method, data transfer parallels kernel computation to hide I/O occupancy as shown in Fig. 3. Because different types of interpolation in FVC are totally independent of each other, we design all the kernel functions and I/O transfer in an asynchronous way. When one type of interpolation kernel functions finish, the data transfer of interpolated images will start simultaneously with the next type of interpolation calculation.

### 4. EXPERIMENTAL RESULTS

The proposed method is implemented on JEM-6.0 with existing instruction set acceleration enabled. The experiment is conducted on Linux 64-bit OS platform with Intel Xeon E5-2643 v4 12-Core at 3.40GHz and 128GB RAM. The GPU device is NVIDIA GeForce GTX 1080 with CUDA 8.0 Toolkit. The encoding configuration of JEM is the default “lowdelay_P_jvct10” with QP=22, 27, 32, 37. The speed-up gain is defined as

\[
\text{SpeedupGain} = \frac{T_{\text{anchor}} - T_{\text{test}}}{T_{\text{test}}} \times 100\% \quad (2)
\]

where \(T_{\text{anchor}}\) and \(T_{\text{test}}\) denote respectively the elapsed time before and after GPU acceleration. Table 3 shows the speed-up results of interpolation module and the overall encoding. The average speed-up gain of interpolation module alone is 67.12% and the overall speed-up gain is 6.35%.

### 5. CONCLUSION

In this paper, based on a comparative analysis of the elapsed time distribution of inter-frame prediction in HM and JEM, a GPU based interpolation acceleration method is proposed with frame caching strategy and multi-stream mechanism. Our method achieves further speed-up compared with instruction set accelerated JEM without changing its original encoding algorithm.

### 6. ACKNOWLEDGMENTS

This work was supported by NSFC (61671296, U1611461, 61521062 and 61601282), the 111 Project (B07022 and Sheitc No.150633) and the Shanghai Key Laboratory of Digital Media Processing and Transmissions.
7. REFERENCES


