Performance of HEVC Discrete Cosine and Sine Transforms

GPU using CUDA

Moustafa Masoumi and HamidReza Ahmadifar

University of Guilan, Engineering department, Rasht, Iran

Phone Number: +98-935-9301109

*Corresponding Author's E-mail: mostafa.masoumi.vamlagh@gmail.com

Abstract

High Efficiency Video Coding (HEVC) is the most recent video encoding standard that achieves much higher compression while maintaining the same quality as its predecessor x264. This is done by a variety of upgrades for example a more diverse transform block size while some fundamental methods remain unchanged. Discrete Cosine Transform (DCT) along with an alternate Discrete Sine Transform (DST) is used by HEVC before quantization to achieve a lossless encoding of transform blocks. Both of these transforms are implemented in HEVC encoder using CPU. In this paper we implement these transforms on GPU using CUDA and analyze their performance compared to the default implementation. Using a batched kernel call 48% and 13% improvement is achieved on Inverse 32x32 and 16x16 DCT transform respectively compared to SIMD while our GPU implementation outperforms other CPU implementations on every transform. We also present a benchmark software for this very purpose which we will use to measure their performance real time.

Keywords: video coding; High Efficiency Video Coding (HEVC); Discrete Sine Transform; Discrete Cosine Transform; Graphics Processor unit (GPU), Computed unified device architecture (CUDA)

1. Introduction

As popularity of streaming services increases at an exponential rate while providers struggle to deliver higher resolution videos over limited bandwidth, better encoding software that can achieve a good compression, without sacrificing quality becomes a necessity. High Efficiency Video Coding (HEVC) [1] also called x265 by ISO/IEC MPEG and ITU-T VCEG joint collaboration team (JCT-VC), proposed in January 2010 and reviewed by JCT-VC meeting in April 2010 [2]. It was released in 2013 and is the latest coding standard that significantly improves upon Advanced Video Coding (AVC) or x264 encoding, realizing more than 50% bit reduction over the previous standard [3] better compression while maintaining the same quality. This is achieved by upgrades on the previous coding standard such as varying the transform block sizes for more compression per block. In HEVC a quad tree structure is used, a picture is partitioned into square Coding Tree Blocks (CTB) no smaller than 16x16 (macroblock size in previous coding standard) and no bigger than 64x64 (increase in macroblock size helps in encoding higher resolution images). CTB is the root of Coding Blocks (CB) which can be as small as 8x8, which is further split into prediction blocks and another quad tree structure called transform tree (or residual quadtree (RQT)). The leaves of this tree are square transform blocks (TBs),

Article History:
Received Date: Feb. 11, 2018
Accepted Date: Jun. 18, 2019
Available Online: Jul. 01, 2019
transform blocks define the size of residual transforms which start from 4x4 (smallest TB) to 32x32 (a maximum sized 64x64 TB which must be partitioned to 4, 32x32 transform blocks) [4].

After Prediction Blocks (PBs) go through inter (between frames) and intra (inside a frame) predictions to reduce redundant or predictable data, transform block further go through a series of simplifications or transforms using Cosine and Sine Transforms and quantization on the result [4]. Here we dedicate the bulk of our endeavor to transforms which will be expanded on in later sections.

2. DCT and DST estimation

HEVC uses finite precision approximation of two dimensional Discrete Cosine Transform of sizes 8x8 to 32x32 and an alternate Discrete Sine Transform of size 4x4 which is used for luma intra prediction residual blocks. A Number of considerations were taken during design for example preservation of symmetry properties and limited bit depth for easier implementation [5]. The standard specifies matrices for core transform of every block size (4x4, 8x8, 16x16, 32x32) along with a matrix for alternate transform of 4x4 luma intra prediction blocks which is an integer transform based on the DST.

Each matrix coefficient is represented with 8bits (including sign bit) and scaled to a factor of $2^{6+M/2}$ compared to orthonormal DCT. The matrices are designed to preserve (anti-) symmetry properties which enables us to get the right half of the matrix from the left half. Furthermore, coefficients of smaller matrices (4x4, 8x8, 16x16) can be derived from 32x32 matrix, also the inverse transform matrix is defined as transpose forward transform matrices.
Figure 1: Left half of 32x32 forward transform matrix. The highlighted areas demonstrate 16x16, 8x8 and 4x4 matrices which can be derived from 32x32 matrix [5].

Transform coefficients are then obtained by applying two-stage 1D transform for columns and rows respectively which entails matrix multiplication of resulting predicted coefficients with their respective matrices. Finally transform coefficients go through Quantization using a Q-step scale. Inverse of this procedure is De-quantization followed by another two-stage transform which is multiplication of transpose of mentioned matrices. These stages are outlined in the following figure:
As mentioned previously matrices are scaled by a $2^{6+M/2}$ so in order to preserve the norm of the residual block additional scaling in the form of $S_{T1}, S_{T2}, S_{R1}, S_{R2}$ is required. These factors are outlined in the standard and are designed in a way to be applicable using logical shift operations. Let $B$ be bit-depth while $M = \log_2 N$ where $N$ is transform size, then Scaling factors are as outlined below:

$$S_{T1} = 2^{-(B+M-9)}, S_{T2} = 2^{-(M+6)}, S_{T1} = 2^{-7}, S_{T1} = 2^{-(20-B)}$$

A similar procedure is applied for alternate DST which provides 1% bit-rate reduction on intra pictures [6]. The DST matrix is demonstrated below:

$$A_4 = \begin{bmatrix} 29 & 55 & 74 & 84 \\ 74 & 74 & 0 & -74 \\ 84 & -29 & -74 & 55 \\ 55 & -84 & 74 & -29 \end{bmatrix}$$

If implemented as simple matrix multiplication, each 1D transform requires $N^2$ multiplications and $N(N - 1)$ additions, using (anti-) symmetry properties there is another method implemented in HEVC test model called partial butterfly [7] which reduces the number of multiplications at the cost of more additions (for more details refer to [8]).

### 3. GPU Implementation

Using CUDA we provide 4 implementations of DCT and DST transforms mentioned in the last section along with a 5th implementation discussed in the next section. These are tested real time and finally compared to HEVC model implementations used in the reference software [9].
A. Plain
The first simple implementation which we call GPUPlain constitutes N*N kernels, where N is transform size, each kernel is taxed with providing one element of the resulting matrix which will result in an N loop. The kernel is called twice per transform for each 1D transform, outlined in Code 1 below.

B. Shared
The second implementation is a modified version of the first but fetches both matrices to shared memory. pre-fetching includes one thread synchronization, but further queries should prove faster. So in cases of one slot being read multiple times this should help decrease latency. Code 2

C. OneStep
Number Device synchronization between calls and one instance copying from shared to device memory can be eliminated by combining both calls into one, this is possible because the maximum space required to keep 3 matrices on shared memory is 6144 bytes (32*32*3*2(sizeof(short)). By removing device synchronization and adding a thread synchronization in kernel we can calculate the transform in one kernel call, see Code 3.

D. Butterfly
Last implementation is HEVC test model’s partial butterfly using CUDA. Since this method relies heavily on intermediate variables, the parallelization is limited and multiple thread synchronizations are necessary. Nevertheless, all loops can be eliminated by using shared memory between synchronizations. To this end kernels for transforms larger than 4x4 utilize N blocks so these variables are shared per element column calculation and N threads are used to calculate each variable in a block separately. Since 4x4 size transformations (DCT4, DST and their inverse), only require one set of variables they can be obtained in 1 block of N threads. Code 4.

```
1. transformPlain<DCTN1><<<1,(n,n)>>>(src,src2,n,shiftSt1);
2. cudaDeviceSynchronize();
3. transformPlain<DCTN2><<<1,(n,n)>>>(src2,dst,n,shiftSt2);
4. 
Code 1 example kernel call of plain transform implementation
```

```
1. transformShared <DCTN1> << 1, (n, n), 2 * n*n*sizeof(int16) >>>(src,src2,n,shiftSt1);
2. cudaDeviceSynchronize();
3. transformShared <DCTN2> << 1, (n, n),2*n*n*sizeof(int16) >>>(src2, dst, n, shiftSt2);
4. 
Code 2 example kernel call of shared transform implementation
```

```
1. transformOneStep<DCTN><<<1,(n,n),3*n*n*sizeof(int16)>>>(src,src2,n,shift1,shift2);
2. 
Code 3 example kernel call of one step transform implementation
```
1. `partialButterfly4 < <<1,n >> > (src, src2, shiftSt1, n);`
2. `cudaDeviceSynchronize();`
3. `partialButterfly4 < << 1, n >> > (src2, dst, shiftSt2, n);`
4. `partialButterfly8 < << n, n >> > (src, src2, shiftSt1, n);`
5. `cudaDeviceSynchronize();`
6. `partialButterfly8 < << n, n >> > (src2, dst, shiftSt2, n);`
7. `cudaDeviceSynchronize();`
8. `partialButterfly8 < << n, n >> > (src, src2, shiftSt1, n);`

**Code 4 example kernel call of butterfly transform implementation**

To benchmark all 4 implementations we provide our project which is accessible on github [10]. On our test system with Nvidia GTX 860m and Intel i4710HQ, Results are as follows (in nanoseconds):

<table>
<thead>
<tr>
<th></th>
<th>DST</th>
<th>DCT4</th>
<th>DCT8</th>
<th>DCT16</th>
<th>DCT32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td>121485</td>
<td>120034</td>
<td>129385</td>
<td>126719</td>
<td>153053</td>
</tr>
<tr>
<td>Shared</td>
<td>122414</td>
<td>119754</td>
<td>119248</td>
<td>127908</td>
<td>143104</td>
</tr>
<tr>
<td>OneStep</td>
<td>72751</td>
<td>73034</td>
<td>72281</td>
<td>77793</td>
<td>92993</td>
</tr>
<tr>
<td>Butterfly</td>
<td>120892</td>
<td>121528</td>
<td>122239</td>
<td>125900</td>
<td>132049</td>
</tr>
</tbody>
</table>

**Table 1: Forward Transform benchmark results of GPU implementations**

<table>
<thead>
<tr>
<th></th>
<th>Inverse DST</th>
<th>Inverse DCT4</th>
<th>Inverse DCT8</th>
<th>Inverse DCT16</th>
<th>Inverse DCT32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td>119978</td>
<td>118844</td>
<td>119499</td>
<td>125099</td>
<td>136438</td>
</tr>
<tr>
<td>Shared</td>
<td>121551</td>
<td>124455</td>
<td>121041</td>
<td>122941</td>
<td>128115</td>
</tr>
<tr>
<td>OneStep</td>
<td>72932</td>
<td>72435</td>
<td>70836</td>
<td>73847</td>
<td>78615</td>
</tr>
<tr>
<td>Butterfly</td>
<td>119300</td>
<td>117771</td>
<td>121257</td>
<td>136438</td>
<td>128094</td>
</tr>
</tbody>
</table>

While Plain is slightly slower than Shared and Butterfly methods, it is evident that the overhead of starting up a kernel exceeds other costs and the fastest implementation is OneStep where one kernel solves the transform in its entirety. Also it appears that minimizing multiplications has minimal effect on execution time as Butterfly performs roughly the same as Shared. There is a significant cost to copying data over to GPU device memory which cannot be averted, in the next section we discuss minimizing this cost so that GPU implementation is viable compared to CPU.

### 4. Comparison and Analysis

HEVC reference software uses several implementations of transform methods. The butterfly algorithm in C++ and several assembly implementations are used [9], of the assembly implementations we use avx2 version which utilizes the most recent SIMD instructions. We also use a basic $O(n^3)$ implementation of matrix multiplication for demonstration purposes. The results are as follows (nanoseconds):
Table 3: Forward Transform benchmark results comparison to CPU implementations

<table>
<thead>
<tr>
<th></th>
<th>DST</th>
<th>DCT4</th>
<th>DCT8</th>
<th>DCT16</th>
<th>DCT32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM</td>
<td>13</td>
<td>10</td>
<td>56</td>
<td>278</td>
<td>2043</td>
</tr>
<tr>
<td>CButterfly</td>
<td>54</td>
<td>41</td>
<td>217</td>
<td>1647</td>
<td>11280</td>
</tr>
<tr>
<td>C Basic</td>
<td>208</td>
<td>199</td>
<td>1130</td>
<td>7890</td>
<td>57500</td>
</tr>
<tr>
<td>GPU oneStep</td>
<td>72751</td>
<td>73034</td>
<td>72281</td>
<td>77793</td>
<td>92993</td>
</tr>
</tbody>
</table>

Table 4: Inverse Transform benchmark results comparison to CPU implementations

<table>
<thead>
<tr>
<th></th>
<th>Inverse DST</th>
<th>Inverse DCT4</th>
<th>Inverse DCT8</th>
<th>Inverse DCT16</th>
<th>Inverse DCT32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM</td>
<td>14</td>
<td>13</td>
<td>36</td>
<td>473</td>
<td>3685</td>
</tr>
<tr>
<td>CPU Butterfly</td>
<td>88</td>
<td>79</td>
<td>392</td>
<td>2371</td>
<td>15601</td>
</tr>
<tr>
<td>CPU Basic</td>
<td>196</td>
<td>194</td>
<td>1027</td>
<td>7968</td>
<td>54202</td>
</tr>
<tr>
<td>GPU oneStep</td>
<td>72932</td>
<td>72435</td>
<td>70836</td>
<td>73847</td>
<td>78615</td>
</tr>
</tbody>
</table>

Although OneStep was fastest of the 4 GPU implementations, the overload of transfer to GPU is too much for small HEVC matrix transformations. To eliminate this, we propose a 5th implementation where we bundle transformations to pass to GPU in one transfer, afterwards we can assign maximum number of possible threads per block as long as there is an element to process.

Using this method which we name OneStepBatch, a batch of matrices will be solved at once. We can batch matrices smaller than 32x32 together to run 1024 threads per block and run as many blocks as necessary to solve any batch greater than 32x32. With this method we will have \( M \times N \times N / 1024 \) blocks, each running 1024 threads, This will greatly improve GPU occupancy.

Table 5: Forward Transform benchmark results for batched GPU implementation

<table>
<thead>
<tr>
<th></th>
<th>DST</th>
<th>DCT4</th>
<th>DCT8</th>
<th>DCT16</th>
<th>DCT32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>287382</td>
<td>286144</td>
<td>1053519</td>
<td>4839693</td>
<td>47002692</td>
</tr>
<tr>
<td>Batched</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>Each</td>
<td>28</td>
<td>28</td>
<td>105</td>
<td>483</td>
<td>4700</td>
</tr>
</tbody>
</table>

Table 6: Inverse Transform benchmark results for batched GPU implementation

<table>
<thead>
<tr>
<th></th>
<th>IDST</th>
<th>IDCT4</th>
<th>IDCT8</th>
<th>IDCT16</th>
<th>IDCT32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>287613</td>
<td>280341</td>
<td>1038692</td>
<td>4160001</td>
<td>19258243</td>
</tr>
<tr>
<td>Batched</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>Each</td>
<td>28</td>
<td>28</td>
<td>103</td>
<td>416</td>
<td>1925</td>
</tr>
</tbody>
</table>

Batched GPU call clearly outperforms pure CPU implementations although SIMD seems to outperform GPU on Inverse transforms smaller than 16x16 and on DCT transforms by a small margin. We can see 13% and 48% improvement over assembly implementation on Inverse DCT16 and Inverse DCT32 respectively.
CONCLUSION

As the latest coding standard, HEVC is designed to support high resolution videos while keeping bandwidth usage and coding latency as low as possible to support live streaming. In this paper we analyzed the implementation of Discrete Sine and Cosine transforms on GPU using CUDA. While memory transfer overhead is surely a hindrance, using batched calls to minimize this overhead we proved that we could gain an increase compared to CPU methods used in the reference x265 coding software. A faster and more parallel transform implementation could further help the next stage (quantization) where works like [11] and [12] are conducted.

References


