An Optimized Parallel IDCT on Graphics Processing Units

An Optimized Parallel IDCT on Graphics Processing Units

Biao Wang[1], Mauricio Alvarez-Mesa[1,2], Chi Ching Chi[1], and Ben Juurlink[1]

[1] Embedded Systems Architecture, Technische Universität Berlin, Berlin, Germany.
biaowang@mailbox.tu-berlin.de,
{mauricio.alvarezmesa,cchi,b.juurlink}@tu-berlin.de,
home page: http://www.aes.tu-berlin.de/
[2] Multimedia Communications, Fraunhofer HHI, Berlin, Germany

Abstract. In this paper we present an implementation of the H.264/AVC Inverse Discrete Cosine Transform (IDCT) optimized for Graphics Processing Units (GPUs) using OpenCL. By exploiting that most of the input data of the IDCT for real videos are zero valued coefficients a new compacted data representation is created that allows for several optimizations. Experimental evaluations conducted on different GPUs show average speedups from 1.7× to 7.4× compared to an optimized single-threaded SIMD CPU version.

Keywords: IDCT, GPU, H.264, OpenCL, parallel programming

1 Introduction

Currently H.264/AVC is one of the most widely used video codecs in the world [1]. It achieves significant improvements in coding performance compared to previous video codecs at the cost of higher computational complexity. Single-threaded performance, however, is no longer increasing at the same rate and now performance scalability is determined by the ability of applications to exploit thread-level parallelism on parallel architectures.

Among different parallel processors available today Graphics Processing Units (GPUs) have become popular for general-purpose computing because of its high computational capabilities and the availability of general purpose GPU programming models such as CUDA [2] and OpenCL [3]. GPUs can accelerate applications to a great extent as long as they feature massive and regular parallelism. Video decoding applications, however, do not meet these requirements completely because of different block sizes and multiple prediction modes.

H.264 decoding consists of 2 main stages, namely: entropy decoding and macroblock reconstruction. The latter, in turn, includes the inverse transform, coefficient rescaling, intra- and inter-prediction and the deblocking filter. Among them, the inverse transform is a good candidate for GPU acceleration because it has only two block sizes and the blocks in frame can be processed independently.

The H.264 transform is an integer approximation of the well known Discret Cosine Transform (DCT) [4]. The main transform is applied to 4×4 blocks and, the H.264 High Profile, currently the most widely used profile, allows another transform for 8×8 blocks [5].
In H.264 High Profile (that uses the 4:2:0 color format) each picture is divided in macroblocks, each one consisting of one block of 16×16 luma samples and two blocks of 8×8 chroma samples. For the luma component the encoder is allowed to select between the 4×4 and 8×8 transforms on a macroblock by macroblock basis. For the chroma components only the 4×4 transform is allowed. The general form of the Inverse DCT (IDCT), that is applied in the H.264 decoder, is defined in Equation 1:

$$\mathbf{X} = \mathbf{C^T Y C} \tag{1}$$

where Y is a matrix of input coefficients, X is a matrix of output residual data and C is the transform matrix. In H.264, C is defined in such a way that the transform can be performed only using integer operations without multiplications. Typically, the IDCT is not implemented using a matrix multiplication algorithm. Instead, the 2D-IDCT is implemented as two 1D-IDCT using a row-column decomposition [6]. Figure 1 shows a flow diagram of 4×4 1D-IDCT.

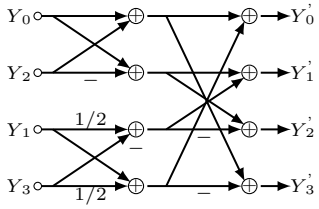

Fig. 1: 4×4 1D-IDCT

In this paper we present an H.264 IDCT implementation optimized for GPUs using OpenCL with portability purpose. We exploit the fact that a significant part of the input data consists of zero data to create an efficient data representation. This simplifies the computation on the GPU and reduces the amount of data that has to be transferred between CPU and GPU memories. On the GPU kernel itself, additional optimizations are applied such as synchronization removal, workgroup enlargement, data granularity enlargement and coalesced write back.
Some previous work has parallelized the IDCT on GPUs. Fang et. al. implemented an 8×8 JPEG IDCT using single precision floating point arithmetic on GPUs using Microsoft DirecX9.0 [7]. The paper shows that their optimal GPU kernel is faster than a CPU optimized kernel with MMX (64-bit SIMD) but slower than SSE2 (128-bit SIMD).
As part of the NVIDIA CUDA Software Development Kit there are two implementations of the JPEG 8×8 DCT/IDCT, one using single precision floating point and the other one using 16-bit integer arithmetic [8]. However, no performance analysis or a comparison with CPU optimized codes is performed. The algorithm presented in this paper uses a similar

thread mapping strategy but with some enhancements for improving the data access efficiency. Furthermore, the IDCT in H.264 is more complex as it contains two different transforms: 4×4 and 8×8.
The rest of paper is organized as follows: The parallelization strategy and the main optimization techniques are presented in Section 2. Experimental setup and results are presented in Section 3. Finally, in Section 4 conclusions are drawn.

2 Implementation of IDCT on GPU

Our GPU implementation is based on an optimized CPU version of the H.264 decoder that, in turn, is based on FFmpeg [9]. In our base code, entropy decoding and macroblock reconstruction have been decoupled into several frames passes. In order to offload the IDCT kernel to the GPU, we further decouple the IDCT from the macroblock reconstruction loop, and create a separated frame pass for it as well. Decoupling requires intermediate frame buffers for input and output data.
The general concept for offloading the IDCT to the GPU is as follows. First the CPU performs entropy decoding on the whole frame and produces the input buffer for the IDCT. This buffer is transferred from the CPU to the GPU, termed as host and device in OpenCL, respectively. Then, the GPU performs the IDCT for the whole frame. When finished, the results are transferred back to the host memory. Finally, the CPU performs the remaining macroblock reconstruction stages.

2.1 Compaction and Separation

A baseline computation of GPU IDCT includes detecting the type of IDCT 4×4 and 8×8, as well as detecting and skipping blocks with zero coefficients. However, skipping zero blocks has no benefit when it results in branch divergence.
We investigate the input of IDCT kernel. Figure 2 shows the average non-zero blocks ratio for sequences in different resolutions (1080p and 2160p) and 3 different encoding modes (labeled as CRF37, CRF22, and Intra). The details of the video sequences and encoding modes are presented in Section 3. According to the Figure 2, a considerable portion of the input are zero blocks.

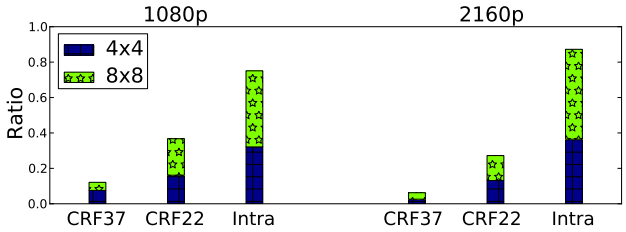

Fig. 2: Non-zero blocks ratio for different encoding modes.

To avoid the computation and memory transfer for the non-zero blocks, we propose two preprocessing steps done by CPU based on the high ratio of zero blocks within input, referred as compaction and separation. With compaction, all zero blocks are squeezed out before they are transferred to GPU, relieving memory transfer overhead to a great extent. With separation, input data are separated into two different buffers according to their IDCT type.
In order to reduce the overhead of these preprocessing steps, the entropy decoding stage has been adapted for producing dense buffers, separated for 4×4 and 8×8 blocks. Introducing these steps has no noticeable effect in the execution time.

2.2 GPU Kernel Implementation

The computation of the IDCT is performed in an OpenCL kernel, a function executed on the GPU. OpenCL uses a hierarchical data-parallel programming model with two-levels. At the higher level, the global thread index space, termed as *NDRange*, is divided into *workgroups*. At the lower level each workgroup is further divided into *workitems*. Each workitem is an instance of a kernel execution, i.e a thread [3].
For the IDCT, three main options are possible for assigning data elements to threads: sample, line and block. Sample to thread mapping, in which each output sample computation is assigned to a thread, leads to divergence as calculation for each sample is different. Line to thread mapping is chosen over block to thread mapping for our implementation because it supplies a larger amount of parallelism. Block to thread mapping, in which each 4×4 or 8×8 block computation is assigned to a thread, leads to insufficient parallelism. Then, line to thread mapping, in which each thread processes one line (row and then column) of coefficients, represents a good solution and it is chosen for our implementation.
In the baseline kernel, the size of workgroup is chosen as 96 threads to match the size of macroblock. For better comparison, the compacted and separated kernel has the same size of workgroup, with a configuration of 32 and 3 in dimension of x and y, respectively. We will investigate the size of workgroup in the future.
The main operations of both 4×4 and 8×8 kernels are shown in Listing 1. First, the index of threads and workgroups are calculated to locate appropriate portion of the input. Second, input coefficients are loaded in rows per thread from the global memory into the private memory and row transform (1D-IDCT) is performed. However, when proceeding to the column transform, the coefficients in the same column are spread across different threads. Therefore, a store to local memory is required first for sharing the data. We transpose the coefficients by storing them in column order. Then, a synchronization is applied to ensure transposed data is written to the local memory. Next, the transposed input is loaded into private memory and column transform is performed by each thread. Finally, each thread stores its results back to global memory in their original positions.
Figure 3 exemplifies the data flow across different memory layers for the 4×4 IDCT in the GPU. Coefficients in the same rows are labeled with the

same color. Numbers within each box indicates their original positions in global memory. Arrows between different memory layers represent the threads within the x dimension of the workgroup. The 8×8 IDCT works in a similar fashion, but has a different implementation of the transform (1D-IDCT) and has a size of N=8 instead of 4. We will refer to this version of kernels as "compacted".

```
__kernel void Idct_NxN(__global short *InOut){
    //workgroup dimension: 3*32 (Dimy*Dimx)
    __local short Shared[3*32*N];
    short Private[N];
    int Tx = get_local_id(0), Ty = get_local_id(1);
    int Dimx = get_local_size(0);
    int Dimy = get_local_size(1);
    int TyOffset = Ty*Dimx*N;
    int TxOffset = Tx*N;
    int WGx = get_group_id(0), WGy = get_group_id(1);
    int WGsInWidth = get_num_groups(0);
    int WGIdx = WGy*WGsInWidth+WGx;
    int WGOffset = WGIdx*Dimy*Dimx*N;

    __global short *TSrc = &InOut[WGOffset+TyOffset+TxOffset];
    Private[0:1:N] = TSrc[0:1:N];
    1D_IDCT(Private);
    //write in column order
    __local short *TShared = &Shared[TyOffset+Tx];
    TShared[0:Dimx:N*Dimx] = Private[0:1:N];
    barrier(CLK_LOCAL_MEM_FENCE); //synchronization

    TShared = &Shared[TyOffset+Tx*N];
    Private[0:1:N] = TShared[0:1:N];
    1D_IDCT(Private);

    int DstBlock = Tx%(N*N);
    int DstCol   = Tx/(Dimx/N);
    int Dst      = DstBlock*N*N+DstCol;
    __global short *TDst = &InOut[WGOffset+TyOffset+Dst];
    TDst[0:N:N*N] = Private[0:1:N];
}
```

Listing 1: Pseudo-code of IDCT kernel

2.3 Further Optimizations for Compacted Kernel

Several optimizations are applied on top of the compacted kernel. In Nvidia GPU, instructions are scheduled in groups of 32 threads, termed as warps [2]. Therefore, the synchronization can be removed, as the warp size is greater than 4 or 8. Second, we increase the size of workgroup to 192. This leads to more warps feeding the compute units, improving the utilization of GPU. Third, we increase the data granularity processed per thread by 4 and 2 times for 4x4 and 8x8 kernel, respectively. By doing this, the index calculation overhead is reduced, leading to less instructions executed overall. However, the granularity can not be increased too much, as the size of input transferred to GPU is rounded up to multiples of unit data mapped to one workgroup. Increased data granularity will introduce more unnecessary computation. Finally, in the compacted

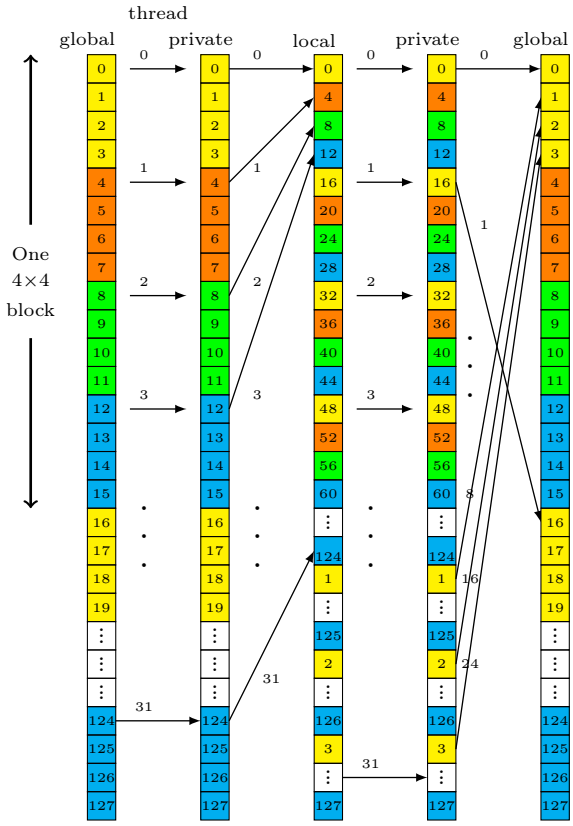

Fig. 3: Data flow for 4×4 IDCT in GPU

kernel, the results are not written back coalescingly to the global memory [2], each thread accesses addresses with a stride of 16 elements, as shown in Figure 3. To solve this problem, we rearrange the result first in local memory, with a padded stride to avoid the bank conflict in local memory, and then write them back to global memory in a coalesced way. This reduces the number of write requests to global memory.

3 Experimental Results and Discussion

We carry out our experiments on several hardware platforms consisting of different NVIDIA GPUs based on the Fermi Architecture [10]. Performance comparison is made against a highly optimized single-threaded CPU version executed on a Intel Sandybridge processor with SSE SIMD instructions, evaluated by the average time of five executions. Table 1 lists our hardware and software configuration. All results are obtained from GT430 except subsection 3.3.

System		GPU	
CPU	i5-2500K	GPU Architecture	Fermi
Frequency	3.3GHz	Compute capability	2.1
ISA	X86-64	GT430	Bandwidth:25.6GB/s
Operating system	Ubuntu 11.10	Shader cores:96	Frequency:1.4GHz
Linux kernel	3.0.0-13-generic	GTS450	Bandwidth:28.8GB/s
H.264 encoder	x264 0.115.1937	Shader cores:192	Frequency:1.56GHz
Compiler	GCC-4.4.6	GTX560Ti	Bandwidth:131.33GB/s
Optimization	-O2	Shader cores:384	Frequency:1.64GHz
Nvidia driver	280.13	OpenCL verison	1.1
CUDA toolkit	4.0	OpenCL build options	-cl-mad-enable

Table 1: Experimental setup

In order to cover videos with different characteristics we selected four (1920×)1080p videos (*blue_ sky, park_ joy, pedestrian_ area and riverbed*) and two (3840×)2160p videos(*crowd_ run and park_ joy*).
The performance of the optimized IDCT depends heavily on the non-zero block ratio, and this, in turn depends on the encoding options. To cover different application scenarios we encoded all videos using three different encoding modes. The first two modes use a constant quality encoding mode (referred to as CRF) with different quality settings, CRF22 and CRF37. CRF22 generates higher quality videos at the cost of higher bitrate, and it is representative of high quality video applications. CRF37 increases the compression level at the cost of quality losses, and is representative of low to medium quality Internet video. The third mode is based on constant bitrate encoding with Intra-only prediction in which no samples from other frames will be used. The encoding options are listed in Table 2

Option	Value	Brief Description
Rate control	CRF-22, CRF-37	Constant quality encoding
	Intra-100	Constant bitrate at 100mbps for 1080p
	Intra-400	Constant bitrate at 400mbps for 2160p
Reference pictures	16	Number of reference pictures
Motion estimation	UMH	Uneven multi-hexagon
Motion search range	24	Max range of the motion search in pixels
Macroblock Partition	all	All macroblock partitions allowed

Table 2: Encoding parameters

3.1 Effect of Compaction and Separation

Table 3 shows the effect of the compaction and separation optimization for 1080p and 2160p sequences with different encoding modes. Both the baseline and the compacted solutions are divided into three parts: kernel execution (Kernel), Host to Device (H2D) transfer, and Device to Host (D2H) transfer.
The kernel execution time is greatly reduced, proportional to the zero block ratio of different encoding modes. Along with the reduction of the kernel time, transfer time between CPU and GPU are reduced as well. In fact, they contribute more to overall speedup.

Res	Mode	Baseline			Compacted			Speedup		
		Kernel	H2D	D2H	Kernel	H2D	D2H	Kernel	H2D	D2H
1080p	CRF37	1.84	2.05	1.91	0.14	0.12	0.12	12.82	16.99	15.75
	CRF22	2.15	2.05	1.91	0.42	0.38	0.37	5.17	5.38	5.13
	Intra	2.59	2.07	1.91	0.84	0.79	0.78	3.08	2.63	2.46
2160p	CRF37	7.22	8.14	7.60	0.28	0.25	0.24	25.53	32.69	31.07
	CRF22	7.99	8.18	7.57	1.21	1.19	1.11	6.62	6.88	6.83
	Intra	10.85	8.23	7.58	3.84	3.94	3.57	2.82	2.09	2.12

Table 3: Execution time (in ms) for baseline and compacted kernels

3.2 Effect of Further Optimizations

Table 4 shows the effect of each of the optimization in section 2.3, in speedups over the previous one starting from the compacted kernel. Accumulated speedups of 1.51× and 1.08× are gained for 4×4 and 8×8 kernels, respectively. For *sync*hronization removal, 1.05× speedup is gained for 4×4 and 1.01× for 8×8 kernel. The difference in the speedups are caused by the different computation granularities of the two kernels. For *workgroup* enlargement, 1.14× is gained for 4×4, compared to 1.01× for 8×8 kernel. More instructions in 8×8 kernel can be executed in parallel, making it running efficiently on compute units even with less number of active warps. By enlarging the data *granularity*, relatively less instructions are reduced for the 8×8 kernel, compared to 4×4 kernel. Thus less speedup is gained. Finally, *coalescing* optimization contributes speedups of 1.02× for 8×8 kernel and 1.07× for 4×4.

3.3 Performance of Optimized IDCT kernel

The GPU IDCT kernels with all the optimizations enabled are compared against two CPU implementations: scalar and SIMD. For the SIMD implementation MMX and SSE are used to accelerate the 4×4 and 8×8 IDCT respectively. The performance of the SIMD and GPU kernel are presented in speedups over the scalar code. For scalability analysis purposes, the IDCT kernels are executed on three GPUs with numbers of shader cores 96, 192, and 384, respectively. Figure 4 shows the performance of 4×4 and 8×8 kernels as expected according to their computational capability. Maximum speedups over 25× are observed for both kernels in GTX560Ti in Intra mode, because of its high bandwidth and large number of shader cores. The 4×4 kernel running on the GTS450 achieves a limited speedup compared to the speedup gained on the GT430. The low increase in memory bandwidth of the GTS450 over the GT430 limits the scalability for 4×4 kernel because of its high ratio of global memory access. GPU implementation is faster than SIMD in all cases except the 8×8 kernel for CRF37 mode in the GT430. This results from the small data of the input, which also explains the low speedups gained in this mode. By contrast, maximum speedup is achieved in Intra mode, as kernel launch overhead is relatively small when computing a large amount of input data.

Kernel	Sync	Workgroup	Granularity	Coalescing	Total
4x4	1.05	1.14	1.19	1.07	1.51
8x8	1.01	1.01	1.04	1.02	1.08

Table 4: Speedups of further optimizations

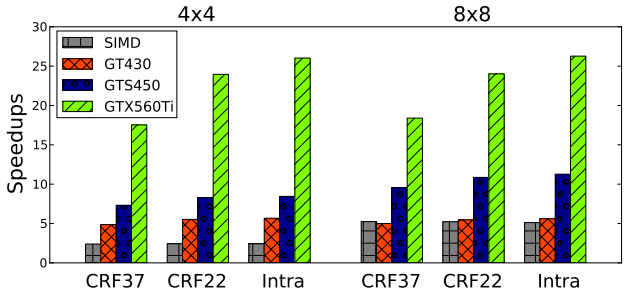

Fig. 4: Optimized kernel speedups over scalar code

3.4 Performance of Complete IDCT

The complete IDCT execution time, consisting of kernel execution time as well as memory transfer times between CPU and GPU, is presented in Table 3. Since modern GPUs are capable of concurrent memory copy and computation, data transfers can be overlapped with kernel computation. We perform overlapped execution between the 4×4 and 8×8 kernels. Speedups gained over scalar code for sequences encoded in different modes for 1080p and 2160p are presented in Figure 5. Overlapped execution gains speedups ranging from 1.2× to 1.3× over non-overlapped execution. However, compared to SIMD code, the complete IDCT performance is slower because memory transfers still are the bottleneck for the overall performance.

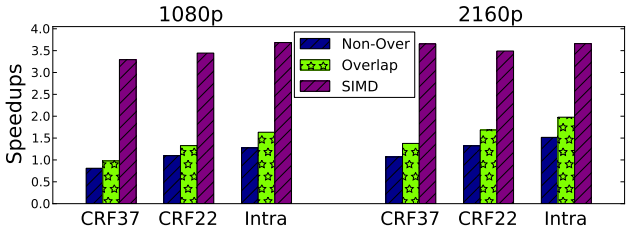

Fig. 5: Complete IDCT performance compared to scalar code

4 Conclusions

In this paper we have exploited the fact that the input of the IDCT in H.264 contains a large number of zero coefficients and propose input compaction and separation to improve the GPU computation. Furthermore, additional optimizations are applied such as enlargement of data granularity and coalesced write back. The optimized GPU kernel shows a significant speedup compared to SIMD execution on the CPU, but the performance of complete IDCT is slower than the CPU SIMD version because of CPU-GPU memory transfer overheads. Our results do suggest, however, that kernels like H.264 IDCT could benefit from architectures with integrated CPUs and GPUs in which the cost of memory transfers is low.

References

1. T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Trans. on Circuits and Sys. for Video Technol.*, vol. 13, no. 7, pp. 560–576, July 2003.
2. NVIDIA, "NVIDIA CUDA C Programming Guide 4.2," `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.
3. Khronos OpenCL Working Group, "The OpenCL Specification 1.1," `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.
4. HS Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-Complexity Transform and Quantization in H.264/AVC," *IEEE Trans. on Circuits and Sys. for Video Technol.*, vol. 13, no. 7, pp. 598–603, 2003.
5. G. J. Sullivan, P. Topiwala, and A. Luthra, "The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions," in *SPIE Conf. on Applications of Digital Image Processing XXVII*, 2004, pp. 454–474.
6. W. H. Chen, C. Smith, and S. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Transactions on Communications*, vol. 25, no. 9, pp. 1004 – 1009, sep 1977.
7. B. Fang, G. Shen, S. Li, and H Chen, "Techniques for Efficient DCT/IDCT Implementation on Generic GPU," in *Proc. of the IEEE Int. Symp. on Circuits and Sys.*, May 2005.
8. A. Obukhov and A. Kharlamov, "Discrete Cosine Transform for 8x8 Blocks with CUDA," `http://www.nvidia.com/content/cudazone/cuda_sdk/Image_Video_Processing_and_Data_Compression.html`, October 2008.
9. "FFmpeg," `http://ffmpeg.org/`, A H.264/AVC decoder.
10. C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, pp. 50–59, 2011.