H.264/AVC MOTION ESTIMATION IMPLEMENTATION ON COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

Wei-Nien Chen\textsuperscript{1} and Hsueh-Ming Hang\textsuperscript{1,2}

\textsuperscript{1}Department of Electronics Engineering, National Chiao-Tung University, Taiwan  
\textsuperscript{2}Dept. of Computer Sci. and Inform. Technology, National Taipei University of Technology, Taipei, Taiwan  
Email: wnchen.ee95g@ntcu.edu.tw, hmhang@ntut.edu.tw

\section*{ABSTRACT}

Due to the rapid growth of graphics processing unit (GPU) processing capability, using GPU as a coprocessor to assist the central processing unit (CPU) in computing massive data becomes essential. In this paper, we present an efficient block-level parallel algorithm for the variable block size motion estimation (ME) in H.264/AVC with fractional pixel refinement on a computer unified device architecture (CUDA) platform, developed by NVIDIA in 2007. The CUDA enhances the programmability and flexibility for general-purpose computation on GPU. We decompose the H.264 ME algorithm into 5 steps so that we can achieve highly parallel computation with low external memory transfer rate. Experimental results show that, with the assistance of GPU, the processing time is 12 times faster than that of using CPU only.\textsuperscript{*}

\textbf{Index Terms—}H.264/AVC, Motion Estimation, Parallel Processing, Compute Unified Device Architecture (CUDA), Graphics Processing Unit (GPU).

\section{1. INTRODUCTION}

H.264/AVC is a recent video compression standard that can achieve much higher compression efficiency than the other existing video codecs [1]. To achieve the high efficiency, its encoding process has a very high complexity. Therefore, it becomes very difficult to implement the H.264 encoder on a single ordinary CPU operating in real time.

In the past few years, the progress of the graphics processing units (GPU) is tremendous. With multiple cores and large memory bandwidth, the computational capability of GPU today is much higher than that of the CPU. Due to its high computational capability, the GPU nowadays serves not only for accelerating the graphics display but also for speeding up non-graphics applications, such as linear algebra computation or scientific simulations. This type of applications is thus called “General-purpose computation on GPUs” (GPGPU) [2].

NVIDIA recently announced a powerful GPU architecture called “Compute Unified Device Architecture” (CUDA) [3]. It is basically a single program multiple data (SPMD) computing device. However, this new architecture integrates the vertex shader and the pixel shader, which were originally designed for different applications, into a unified device and it provides an internal shared memory to reduce the external DRAM access. These modifications largely improve the flexibility and programmability of GPGPU. Therefore, we implement the most time-consuming H.264 coding process – the motion estimation unit – on this device.

Although a few GPU-based motion estimation methods have been proposed [4-7], the new CUDA architecture needs a new algorithm to fully utilize its features [8]. In this paper, a highly parallel variable block size full-search ME algorithm with fractional pixel refinement is proposed. Our ME algorithm is optimized for CUDA architecture by using a large number of threads that can execute on GPU in parallel and can make good use of the shared memory to reduce DRAM access. The implementation details are presented in the following sections and are evaluated on the NVIDIA GeForce 8800GTX GPU platform.

The rest of the paper is organized as follows. Section 2 contains a brief overview of CUDA. In section 3, the algorithm and implementation details are presented. And in section 4, the performance of our implementation is evaluated. A short conclusion is given in section 5.

\section{2. COMPUTE UNIFIED DEVICE ARCHITECTURE}

\subsection{2.1. Hardware Architecture}

Unlike the previous GPU architecture, the CUDA integrates the vertex shader and the pixel shader into a unified computing unit called “streaming processor (SP).” For example, the NVIDIA 8800GTX contains 128 SPs. Every 8 streaming processors constitute a “streaming multiprocessors (SM)”. The SM has the signal instruction, multiple data (SIMD) architecture that executes the same instructions on different data.

Four types of on-chip memory are associated with each SM: (1) a set of local 32-bits general propose registers per processor, (2) a 16-bank shared memory that is shared by all SPs within a single SM, which allows SPs to communicate with each other without passing data outside the chip, (3) a read-only constant cache that maps to the constant memory of the device DRAM accessible by all SPs, and (4) a read-only texture cache that maps to the texture memory of the device DRAM accessible by all SPs. In addition to the constant memory and texture memory that are read-only memory for SPs, a global on-device memory is both readable and writable by SPs. However, because the global memory lacks cache, the data access to the global memory is much slower than that to the constant memory and the texture memory [3].

\subsection{2.2. Programming and Execution Model}

In CUDA programming, the original program is first compiled to conform to the CUDA device instruction set and it becomes a pa-
rallleled new program, which is called “kernel.” The kernel is downloaded to the GPU device that acts as a coprocessor to the host (CPU). It is executed by the mechanism of “threads” that are organized in thread block. The threads within a thread block can co-work with each other through the shared memory and can synchronize their execution to coordinate their memory access. The maximum number of threads within a thread block is limited; however, thread blocks that execute the same kernel can be batched together to form a grid of blocks. Therefore, the total number of threads that execute a single kernel can be much larger. Nevertheless, threads in different thread blocks are unable to access the same shared memory and thus, they run independently.

The SM may execute one or more thread blocks concurrently depending on the shared memory and register occupancy. Each thread block is split into SIMD groups of threads called “warps”; these warps contain 32 successive threads that are executed by the SM in a SIMD fashion. As mentioned above, the shared memory has only 16 banks. Therefore, the shared memory requests in a warp are split into two halves. A thread scheduler periodically switches from one warp to another to maximize the SM’s computational resources [3].

3. MOTION ESTIMATION ON CUDA

3.1. H.264 Motion Estimation Algorithm

The basic unit in the H.264/AVC motion compensation process is the 16x16 macroblock. Each macroblock of an encoding frame can be split into 16x8, 8x16, and 8x8 blocks. The 8x8 block can be further partitioned into 8x4, 4x8 and 4x4 blocks. There are two steps involved in deciding the final encoding mode. In the first step, the best motion vectors (MV) of each possible mode (block sizes) are calculated. The matching criterion is typically the sum of the absolute difference (SAD). In the second step, the rate-distortion performance of each mode is evaluated. The best one is selected as the final mode [1]. Due to the numerous candidate modes, the H.264/AVC motion estimation algorithm is generally extremely complex and time-consuming. Although many of the fast motion estimation algorithms have been proposed [1], [9], most of them require a large amount of branch instructions that may become a strong bottleneck in CUDA. Therefore, in this paper, we focus on the integer pixel full search with quarter pixel refinement.

In our implementation, a macroblock is divided into sixteen 4x4 blocks, and the SAD value of each 4x4 block is calculated in parallel for all candidate motion vectors (positions) within the search range on the reference frame. We then merge these 4x4 block SADs to form the 4x8, 8x4, 8x8, 8x16, 16x8, and 16x16 block SADs, respectively. For each block size, we compare the SADs of all candidate MVs and the one with the least SAD is the integer-pixel motion vector (IMV), MV with integer pixel accuracy. Next, in order to obtain the fractional pixel motion vectors, the reference frame is interpolated using a six-tap filter and a bilinear filter defined in the H.264/AVC standard [1]. We calculate the SADs at 24 fractional pixel positions that are adjacent to the best integer MV, and then choose the least SAD position as the fraction-pixel motion vector (FMV).

The aforementioned process is repeated for all block sizes in a macroblock, and the total calculation is thus tremendous. Fortunately, this algorithm is block-based and can easily be parallelized. Therefore, we propose a fully parallelized ME algorithm that fits into the CUDA’s architecture. As discussed in Section 2, the performance of CUDA highly depends on the parallelization hierarchy, which is controlled by the number of thread blocks in a grid and the number of threads in a thread block. The thread assignment has a strong impact on the data distribution, which decides the amount of memory transfer. Increasing the usage of shared memory can reduce the memory transfer between the device DRAM and GPU, and the latter is very time-consuming.

In our design, we optimize the memory usage and threads allocation/synchronization to achieve a high-performance ME implementation. The reference frame and the encoding frame are first loaded into the texture memory on the device DRAM and then a parallelized ME algorithm is performed on CUDA. After calculation, MVs of all block sizes in a macroblock are derived and transferred back to the host. In the following discussions, we use 4CIF picture size as an example, and the search range is assumed to be 32x32.

3.2. 4x4 Block-Size SAD Calculation

We first calculate the SAD values for the basic 4x4 blocks. There are 176x144 blocks in a frame. The 32x32 search range leads to 1024 candidate positions (MV) for each block. Each candidate SAD is computed by one thread and 256 threads are executed in a thread block, as illustrated by Fig. 1. Thus, the total number of thread blocks is:

\[
\text{Block\_num} = \frac{\text{Frame\_Width}}{4} \times \frac{\text{Frame\_Height}}{4} \times \text{Search\_Range} \times \frac{1}{256} \quad (1)
\]

Every 256 candidates (gray circles) of one 4x4 block SAD calculations is assigned to a thread block (B1 to B101376 in total) so that the 4x4 encoding block can be put into the shared memory in advance and shared by all 256 threads without re-fetching from the device DRAM. The amount of memory access is considerably reduced. After this process, all 4x4 block SADs are generated and stored back to the device DRAM.

3.3. Variable Block-Size SAD Generation

The SADs of block sizes 4x8, 8x4, 8x8, 8x16, 16x8, and 16x16 are the combinations of 4x4 SADs as shown in Fig. 2 (a). Thus, we merge the 4x4 SADs obtained in the previous step to derive the SADs of all different block sizes. As shown in Fig. 2 (b), each thread fetches sixteen 4x4 SADs of one macroblock at a candidate position and combines them in different ways to form the SADs of six other block sizes. The total number of thread blocks is:

\[
\text{Block\_num} = \frac{\text{Frame\_Width}}{16} \times \frac{\text{Frame\_Height}}{16} \times \text{Search\_Range}^2 \times \frac{1}{256} \quad (2)
\]

After this operation, SADs of different block sizes (25 values for all block-mode combinations) are generated by each thread. These SADs then transferred back to the device DRAM.

3.4. Integer Pixel SAD Comparison

After variable block-size SADs are generated, all 1024 SADs of one block are compared and the one with the least SAD is chosen as the integer-pixel motion vector (IMV). Each block size (from 4x4 to 16x16) has its own kernels for SAD comparison; therefore, seven kernels are implemented and executed sequentially. Take the 4x4 block size as an example, as illustrated by Fig. 3, each block has 1024 SADs stored in the device DRAM and the smallest SAD is identified as the final output. The comparison of 1024 SADs is done in one thread block, in which 256 threads are activated. In other words, the total number of thread blocks equals to the block number of a frame. First, each thread reads in 4 SADs from the
device DRAM and produces a least SAD. These temporary SAD values and their indexes are stored in the shared memory. Then, we activate 128 threads and each of them compares two SADs. The smaller one is stored back to the shared memory. In the next iteration, the thread number is halved. This process is repeated until the final winner (best MV) is obtained.

During the thread reduction process, two problems may occur: (1) shared memory bank conflict and (2) highly divergent warps. Either of them results in kernel inefficiency. Therefore, a sequential addressing with non-divergent branching strategy is adopted [10], as illustrated by Fig. 4. In each iteration, the activated thread IDs are arranged in the successive order in order to keep the warps from diverging. In addition, the data fetches are addressed by the striped index, of which the strip equals to the activated thread numbers.

After several iterations, the smallest SAD is identified, and its index, which is the motion vector of that block, is stored back to the device DRAM. In this process, with the help of shared memory, the amount of device DRAM access is minimized.

The H.264 standard supports the quarter-pixel precision fractional ME (FME). In order to perform this task, the reference frame is first interpolated by a 6-tap filter to generate the half-pixel image values, and then a bilinear filter is applied to generate the quarter-pixel image values. These two filters are sequentially applied due to data dependency and are thus implemented in two kernels. Every integer pixel is assigned to one thread of the first kernel to generate three half-pixels, which are then stored back to the device DRAM. The second kernel performs the quarter-pixel interpolation by fetching the half-pixel values. Then, three quarter-pixel values in each thread are produced and stored back to the device DRAM.

3.5. Fractional Pixel Interpolation

The H.264 standard supports the quarter-pixel precision fractional ME (FME). In order to perform this task, the reference frame is first interpolated by a 6-tap filter to generate the half-pixel image values, and then a bilinear filter is applied to generate the quarter-pixel image values. These two filters are sequentially applied due to data dependency and are thus implemented in two kernels. Every integer pixel is assigned to one thread of the first kernel to...
One kernel is assigned for doing both SAD calculations and comparisons in FME as shown by Fig. 5 (b). Thus, 24 threads are bounded together as a group to produce one FMV. Each thread block contains multiple groups that execute independently to fully utilize the GPU capacity. First, an encoding block is loaded into the shared memory in advance, which is to be accessed by all threads in a group. And each thread fetches the reference data from the device DRAM at different candidate position. Threads then calculate the 24 SADs in parallel and temporarily store them in the shared memory. Then, we activate 12 threads and each thread does the comparison of two SADs and stores the smaller one back to shared memory. After four iterations, the smallest SAD survives, which is the FMV of that block. Both SAD and FMV are transferred back to the device DRAM. Note that in the last iteration, the thread compares three candidates. At the end, the best MVs are produced by GPU and stored back to the host PC.

4. EXPERIMENTAL RESULTS

To evaluate the performance of the proposed algorithm on CUDA, the following development environment is used: (1) AMD Athlon 64 X2 Dual Core 2.1GHz with 2048 MB memory, (2) NVIDIA GeForce 8800GTX with 768MB DRAM, (3) Microsoft Windows XP sp2, (4) Microsoft Visual Studio 2005, (5) CUDA Toolkit and SDK 1.1, and (6) NVIDIA Driver for Microsoft Windows XP with CUDA Support (169.09). Our proposed algorithm is the full-search ME algorithm. Therefore, its computation time is picture-independent. A popular MPEG test sequence, City (4CIF, 704x576), is examined with a 32x32 search range. The execution time of each of the steps described in Section 3 is shown in Table 1.

The 4x4 block-size SAD calculation and the variable block-size SAD generation together take about 59.4% of the total execution time, which is the largest in all steps. It is because that in these two steps, the amount of device DRAM access and the SAD calculations are much higher than those in the other three steps. Note that the execution time of "others" also takes about 15.2% of the total execution time, including the memory transfer and texture binding. It indicates that the memory allocation and transfer issues remain a bottleneck in the CUDA applications.

Next, we compare two cases. Case 1: We run the proposed ME algorithm on the host CPU only. Case 2: The H.264 ME is mainly executed on GPU as described in Section 3. Two test sequences with different resolutions, Stefan (CIF, 352x288) and City (4CIF, 704x576), are examined with the same 32x32 search range. The results are shown in Table 2. For the CIF-size test sequence, the GPU/CPU configuration achieves 31.54 fps, and the CPU only configuration produces 3.04 fps. The speed-up ratio is nearly 11 times. For the 4CIF-size test sequence, the GPU/CPU configuration is about 9.19 fps, which is about 12 times faster than using CPU only. With the assistance of GPU, the ME executing time is significantly reduced.

5. CONCLUSION

In this paper, we present an efficient block-level parallelized algorithm for variable block-size motion estimation with fractional pixel refinement using the CUDA GPU. We demonstrate that the GPU acting as a coprocessor can effectively accelerate massive data computation. Experimental results show that with GPU, the ME process can be 12 times faster than the CPU only configuration. The concepts behind our proposed parallelized algorithm can be used by all CUDA-based GPUs. Our future work is to port the other parts of the H.264/AVC encoder on the CUDA GPU.

Table. 1. The average execution time in milliseconds (ms) for processing one frame using the proposed algorithm

<table>
<thead>
<tr>
<th>Steps</th>
<th>Frame rate (fps)</th>
<th>Speed-up (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step1. 4x4 Block Size SADs Calculation</td>
<td>33.98</td>
<td>31.24</td>
</tr>
<tr>
<td>Step2. Variable Block size SADs Generation</td>
<td>30.64</td>
<td>28.16</td>
</tr>
<tr>
<td>Step3. Integer Pixel SAD Comparison</td>
<td>9.69</td>
<td>8.90</td>
</tr>
<tr>
<td>Step4. Fractional Pixel Interpolation</td>
<td>7.10</td>
<td>6.52</td>
</tr>
<tr>
<td>Step5. Fractional Pixel ME Refinement</td>
<td>10.87</td>
<td>9.99</td>
</tr>
<tr>
<td>Others</td>
<td>16.49</td>
<td>15.16</td>
</tr>
<tr>
<td>Total</td>
<td>108.77</td>
<td>100</td>
</tr>
</tbody>
</table>

Table. 2. The performance comparison between CPU and GPU

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Frame rate (fps) using AMD CPU</th>
<th>Frame rate (fps) using GPU</th>
<th>Speed-up (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stefan(CIF)</td>
<td>3.04</td>
<td>31.54</td>
<td>10.38</td>
</tr>
<tr>
<td>City(4CIF)</td>
<td>0.78</td>
<td>9.19</td>
<td>11.78</td>
</tr>
</tbody>
</table>

6. ACKNOWLEDGEMENTS

The authors would like to thank Prof Wen-Mei Hwu and his research group at University of Illinois, Urbana-Champaign, for providing the background knowledge and parallel processing ideas on CUDA. Particularly, we thank Mr. C.I. Rodrigues (UIUC) and Mr. H.C. Lin (NCTU) for their valuable suggestions and comments. This international collaboration is sponsored by NSC, Taiwan.

7. REFERENCES