AN EFFICIENT PARALLEL MOTION ESTIMATION ALGORITHM AND X264 PARALLELIZATION IN CUDA

Youngsub Ko, and Youngmin Yi, Soonhoi Ha

1 School of EECS, Seoul National University, Seoul, Korea
{kys4464, sha}@iris.snu.ac.kr
2 School of ECE, University of Seoul, Seoul, Korea
ymyi@uos.ac.kr

ABSTRACT
H.264/AVC video encoders have been widely used for its high coding efficiency. Since the computational demand proportional to the frame resolution is constantly increasing, it has been of great interest to accelerate H.264/AVC by parallel processing. Recently, graphics processing units (GPUs) have emerged as a viable target for accelerating general purpose applications by exploiting fine-grain data parallelisms. Despite extensive research effort to use GPUs to accelerate the H.264/AVC algorithm, it has not been successful to achieve any speed-up over the x264 algorithm that is known as the fastest CPU implementation because of significant communication overhead between the host CPU and the GPU and intra-frame dependency in the algorithm. In this paper, we propose a novel motion estimation (ME) algorithm tailored for NVIDIA GPU implementation. It is accompanied by a novel pipelining technique, called sub-frame ME processing, to effectively hide the communication overhead between the host CPU and the GPU. The proposed H.264 encoder achieves more than 20% speed-up compared with x264.

Index Terms— H.264, Motion Estimation, CUDA, GPU

1. INTRODUCTION
Video encoding is to reduce the size of information in a video sequence by exploiting temporal and spatial redundancy. H.264/AVC is the video encoding standard developed by the Joint Video Team (JVT) and has become widely used for its high coding efficiency in digital broadcasting, teleconferencing, multi-channel surveillance cameras, and so on. In particular, as diverse decoding devices with different screen resolution have proliferated over the past few years, the need for video transcoding has been rapidly increased, which first decodes the input video sequence and scales it to the configured resolution, and then encodes the scaled video sequence. The complexity of computation in video encoding increases as the resolution of the input video sequence to encode increases.

The motion estimation (ME) function is the most time-consuming part in H.264 encoding. In the reference H.264 encoder, called JM encoder [1], the ME function takes around 80% of the total encoding time [2]. Thus there has been extensive work to reduce the computational complexity of the ME algorithm. One direction is to replace the full search algorithm of the JM encoder with a faster algorithm: a diamond search algorithm is an example. The x264 [3] that is known as the fastest CPU implementation of the H.264 encoder adopts a diamond search algorithm and achieves significant speed-up (up to about three orders of magnitude, depending on input video sequences) over the JM encoder [4]. In the x264 implementation on the Intel Core2 Quad Q8300, the ME function takes only 20–30% of the total encoding time.

Another direction is to use dedicated hardware acceleration for real time encoding. However, it is not easy to add new functionalities or tailor some features in such hardware acceleration.

Recently, several techniques have been proposed to use graphics processing units (GPU) to accelerate the ME function. GPU programming has spread rapidly after Compute Unified Device Architecture (CUDA) was first introduced to write parallel programs in high-level languages for NVIDIA GPUs. CUDA is a programming framework that can easily describe data-parallelism and enables general purpose computing on GPUs. It is essentially a C/C++ programming language with several extensions.

In this paper, we present a novel H.264 video encoder tailored for GPU processing. The main contribution of our work is to propose an efficient data-parallel motion estimation (ME) algorithm for GPU and a novel pipelining technique to effectively hide the significant communication overhead between a GPU and the host CPU.

The proposed technique is distinguished from the related work as follows. (1) We measure the performance of the overall H.264 algorithm while the related work usually measures the ME performance only. Since the communication overhead between the host CPU and a GPU, the speed-up from GPU processing is likely to be overshadowed by the communication overhead. (2) While
the related work is usually compared with the JM encoder, we use x264 for comparison. There is no published work that uses the best x264 implementation for comparison to our best knowledge. (3) We also consider the coding efficiency and the picture quality as performance indexes while the related work usually ignores one or the other.

The paper is organized as follows. Section 2 reviews the related work that accelerates the ME function with a GPU. Section 3 provides the background of CUDA programming model and the ME algorithm in H.264/AVC. In section 4 our proposed ME algorithm for GPU acceleration is explained and the efficient encoder structure to reduce the communication overhead is presented in section 5. Section 6 provides experimental results, which is followed by concluding remarks in section 7.

2. RELATED WORK

As motion estimation is the most compute intensive function in an H.264 encoder, extensive researches have been carried out to accelerate the motion estimation function in GPU. Since full search algorithm has more parallelism than any other search algorithm, it is widely used to parallelize motion estimation in GPU [5][6]. Although they can obtain huge performance gain compared to the JM encoder, it is not clear if their result is not good enough to be used practically.

In the other search algorithms, motion vector prediction (MVP) is used to increase the compression ratio. In MVP, the initial motion vector (MV) of the current macro block (MB) is predicted from the MVs of the neighbor macro blocks, assuming that they have been already encoded in a sequential raster order. Therefore the motion vector of a block cannot be predicted if MB encoding is performed in parallel. Since MVP limits parallelism, several approaches are proposed to increase the parallelism. If we simply do not use MVP, it is observed that the coding efficiency is significantly degraded, resulting in a larger output size.

Kung [7] has proposed a block-based ME algorithm that uses 4x4 blocks instead of 16x16 blocks. As a frame is divided into finer blocks, the number of blocks that can be executed in parallel is increased. Since dynamic behavior like early termination introduces poor performance in GPU due to load imbalance, they proposed a new search algorithm based on three step search algorithm which has no early termination technique.

Schwalb et al. [8] have proposed a different approach. They ignored MVP to fully exploit macro block (MB) level parallelism. Instead, they use other predictors generated by Forward Dominant Vector Selection (FDVS) [9] and Split and Merge [10] techniques. In FDVS, it predicts the motion vector of \( n \times 16 \) frame by reusing the motion vectors of \( n \) previous frames. In Split and Merge, it exploits correlation of motion vectors between variable block sizes. Their result shows that they can achieve competitive quality and coding efficiency compared to UMHexagons Search algorithm [11] even if they use Diamond Search algorithm [12]. These techniques can be applicable to our approach if we use multiple reference frames. Currently, we only consider one reference frame.

Search algorithm that proposed in [13] is similar to our algorithm in using a square pattern for search points. While their algorithm considers only a 3x3 square pattern, we assume a general square pattern \((mn)\) and the value of \( n \) that optimizes the performance in a target GPU is decided through measurements.

In our approach, we use early termination techniques and avoid the load imbalance by carefully considering the underlying GPU architecture. To fully exploit MB level parallelism, we do not consider MVP that poses dependencies in the sequence of MBs encoding. To circumvent the absence of MVP, we propose a new search algorithm suited for the GPU architecture, where more computations are performed yet in parallel to maintain as much coding efficiency as MVP would provide.

3. BACKGROUND

3.1. Compute Unified Device Architecture (CUDA)

CUDA is a parallel programming framework for utilizing GPU for general purpose computing. Typical GPUs consist of hundreds of highly decoupled processing cores capable of achieving immense parallel computing performance. Since CUDA extends C/C++ language and allows programmers to write parallel programs for GPU, recently many compute-intensive applications have been parallelized with CUDA and show huge performance improvements. However, it is noteworthy that we need to understand the CUDA programming model and the hardware architecture to obtain the potential performance from GPU implementation.

CUDA is based on the SIMD (Single Instruction Multiple Data) architecture and suitable for exploiting various levels of data parallelism. Fine-grain data parallelism can be captured with threads, and more coarse-grain parallelism can be described with thread blocks that are groups of threads. A computational unit that is executed by a number of CUDA threads is called the kernel. When a kernel is invoked, we can configure how many thread blocks with how many threads are launched in parallel. When a kernel executes, each thread block is scheduled on each SM (Stream Multi-Processor) and executed. While a thread block is scheduled and executed independently of each other, the threads in the same thread block are executed in bundles, or warps, and a thread cannot advance to the next instruction if the other threads in the same warp have not yet completed their own execution.

CUDA assumes heterogeneous architectures that consist of different type of processors (i.e., CPUs and GPUs) and separate memory spaces. GPU has its own global memory separated from the host memory; therefore, to execute a kernel in GPU, input data need to be copied to the GPU.
memory from the host memory. Likewise, the result needs to be copied from the GPU memory to the host memory after the kernel is completed.

CUDA programming platform supports both of synchronous and asynchronous communication. With synchronous communication, the CPU task that calls the methods can only proceed after the communication completes. In general, synchronous communication method results in lower throughput than one can get with asynchronous communication.

Kernel launches are by default asynchronous execution so that the function call returns immediately and the CPU thread can proceed during the launched kernel is executed. Communications (i.e., memory copy) performed by CUDA APIs that are suffixed with Async also behave in this manner: instead of blocking the CPU until the memory copy is finished, it returns immediately. Moreover, when streams the communication and kernel execution can be overlapped hiding the communication time. Kernel executions and memory copy with different streams does not have any dependency, therefore can be executed asynchronously overlapping their operations. In our proposed GPU implementation of H.264 encoder, the memory overhead for the copy is significant so that we take advantage of these asynchronous memory operations as will be described in more detail in section 5.

3.2. X264 Encoder

X264[3] is a de facto standard H.264 encoder that is widely used because of its high performance. To optimize the performance, it parallelizes encoding process using SIMD instructions like MMX and SSE. Figure 1 is briefly illustrating the overall execution flow of x264 encoder. First, in the frame analysis task, it determines the slice type of the current frame and a quantization parameter for each MB. After that, the motion estimation task analyzes MBs and it computes motion vectors. Then, these results are transferred to the residual encoding task and are compared with the intra prediction. Based on the comparison results, encoding type is decided for each MB and residual encoding processes are performed through several sub-tasks like quantization and DCT and VLC (Variable Length Coding). Motion Estimation (ME) is a process for deciding Motion Vectors (MVs) that describes the movements of an object in a series of image frames. To determine an MV, it is necessary to find out the MB in previous frames that is the most similar to the MB in the current frame. Sum-of-difference (SAD) is a commonly used metric for evaluating the similarity.

A full Search algorithm calculates SAD for all possible candidates in the search range. Since it checks all candidates, it can find the block that has the smallest SAD and its compression ratio is better than any other search algorithms. However, it requires tremendous SAD calculations as it checks every candidate in the search range.

A full Search algorithm calculates SAD for all possible candidates in the search range. Since it checks all candidates, it can find the block that has the smallest SAD and its compression ratio is better than any other search algorithms. However, it requires tremendous SAD calculations as it checks every candidate in the search range.

4. THE PROPOSED MOTION ESTIMATION ALGORITHM

In this section we explain the proposed motion estimation algorithm that is tailored for GPU computation. We aim to achieve the similar compression quality as the diamond search algorithm but the faster execution by utilizing the massively parallel architecture of the GPU effectively.

4.1. The Overall Flow

The overall flow of the proposed algorithm, called square search (SS) algorithm, is illustrated in Figure 2. Initially, we first decide the starting point of the search process. While the MVP algorithm of x264 predicts the motion vector from the adjacent MBs of the current frame, we use the following motion vectors to decide the starting point: zero vector to choose the center of the current MB as the starting point, the motion vector from the previous frame to utilize the temporal locality, or the motion vector obtained from the larger block. We perform motion estimation for four different sizes of blocks in the order of 16x16, 8x8, 8x16, and 16x8. Therefore we can use the motion vector obtained
for 16x16 block as the candidate motion vector of the smaller block. Among these motion vectors, we choose one to decide the starting point by comparing the SAD values. Since they are all available before the current MB, we can perform SAD computation in parallel. And the minimum SAD value is initialized from this starting point.

After the starting point is decided, the search process begins in an iterative fashion. At each iteration, we compute the SAD values for all the candidate points within a square search region. For example, Figure 3 shows the 5x5 square region that has 24 (5x5 – 1) candidate points (blank point at its center). After all SADs are calculated, we find the smallest SAD and the associated point. Using this value, two conditions for early termination are checked.

It first checks whether the SAD of this point is smaller than the minimum SAD value. If it is smaller, it becomes the minimum SAD value and we proceed to check the next condition. But if it is larger, however, the search process is terminated with the minimum SAD value that is obtained from the previous iteration.

As the second condition, we check whether the selected point is located in the inner space as marked by grey points in Figure 3. If it is the case, the search process is terminated. If the selected point is located in the boundary of the square space as marked by black points in Figure 3, we go to the next iteration after moving the square search region to the direction of the point. The ‘X’ points in Figure 3 represent the next starting points depending on the position of the minimum SAD point. The next starting point is located at the position indicated by a vector two times larger than the vector that goes from the current starting point to the minimum SAD point. And we repeat the search process again until either one of termination condition is met.

4.2. Kernel Implementation

As discussed in section 3, CUDA provides various levels of parallelism in an application with threads and thread blocks. We exploit three levels of parallelisms in the proposed search algorithm as illustrated in Figure 4.

![Figure 2. The overall flow of the proposed square search(SS) algorithm](Image)

Figure 2. The overall flow of the proposed square search(SS) algorithm

![Figure 3. Candidate points and the next starting points in the proposed SS algorithm](Image)

Figure 3. Candidate points and the next starting points in the proposed SS algorithm

- 1: Starting points
- : Candidates in inner space
- : Candidates in boundary
- X: Next Starting Points

![Figure 4. Three level of parallelism in the proposed algorithm](Image)

Figure 4. Three level of parallelism in the proposed algorithm

The top level parallelism is exploited at the MB level. A frame is divided into hundreds or thousands of MBs depending on the image resolution and the search process of Figure 2 is performed for each MB. Since there is no inter-dependency between MBs, the SS algorithm can be applied for all MBs in parallel.

The second level parallelism is point-level parallelism. In the search process for each MB, SAD computations are performed for dozens of candidate points: For a nxn square search region, we compute the SAD values for n² candidate points by iteration. Since all SAD computations are independent of each other, they can be performed in parallel.

The last level parallelism is pixel-level parallelism. For each SAD computation for a 16x16 MB, it calculates the absolute difference of 256 pixels between the candidate MB and the current MB. Although the absolute differences should be summed at the end, the calculations for all the pixels in a MB can be performed in parallel.

Figure 5 shows how three levels of parallelism are configured in the kernel implementation. As each MB is mapped to a thread block, the more Ss GPU has, the more MBs can be processed concurrently. So the top-level parallelism is scalable. At the bottom level, one SAD value for each search point (SP) is calculated by T threads. Since each thread computes 256/T differences, the larger T is, the more differences are calculated concurrently, but the more
overhead is incurred for reduction (summation). So $T$ should be decided carefully. At the middle level, a thread block (TB) consists of $T \times m$ threads where $m$ denotes $n^2$ which determines the number of search points. There is trade-off between compression-ratio and performance depending on the size of the search region.

In order to optimize the performance of kernels, it is important to store each data in the proper type of memory in the GPU. One criterion for deciding which memory should be used for the data is to analyze how much the data are shared. Table 1 summarizes the level of data sharing used in motion estimation. For the current frame, threads in the same thread block share the current MB and threads in different thread blocks have no data to share. Hence it is reasonable to store each MB on the shared memory in GPU.

For the reference frame, the threads in the same thread block refer to the adjacent MBs associated with the candidate points. And the threads in the different thread blocks may refer to the same MBs. Since the reference frame is shared by all threads in the kernel, the texture memory can be used. However, we notice that the recent GPUs are equipped with a hardware cache [14] that has lower latency than the texture cache. As a result we store the reference frame in the global memory, resorting to the hardware cache.

5. ASYNCHRONOUS EXECUTION IN GPU

As only the ME function is performed in the GPU, data copy between the host CPU and the GPU is required. As input data, a reference frame and a current frame must be copied from the CPU to the GPU. In addition, since the output of ME is used in a task that performs residual encoding, the output is copied back from the GPU.

Table 1. The level of data sharing in motion estimation

<table>
<thead>
<tr>
<th></th>
<th>between thread blocks</th>
<th>Between threads in a thread block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current frame</td>
<td>Not Shared</td>
<td>Fully Shared</td>
</tr>
<tr>
<td>Reference frame</td>
<td>Partially Shared</td>
<td>Partially Shared</td>
</tr>
</tbody>
</table>

Figure 5. Kernel configuration of the proposed algorithm

Figure 6. Execution flow before(a) and after(b) applying asynchronous operations

Figure 6 (a) displays the execution flow of the baseline implementation that uses synchronous communication APIs. First, the current frame is copied from the CPU to the GPU(1). Then, ME is executed in the GPU and produces the output data(2). These are copied back to the CPU(3) and used in the residual encoding task. After the residual encoding task completes, the reference frame is copied to the GPU(4). Therefore the memory copy overhead is added to the total execution time. Since the copy between the CPU host memory and the GPU device memory takes place through a PCI-e bus, the overhead may outweigh the benefit from GPU acceleration.

To solve this problem, the proposed algorithm uses asynchronous operations in order not to block the CPU while memory copy is being processed. As illustrated in Figure 6 (b), the frame analysis task and the residual encoding task executed in the CPU hide the ME kernel execution time and the memory copy overhead completely.

In Figure 6 (b), however, there is a problem, a feedback dependency between the ME and the residual encoding task. The ME task requires a reference frame as input data, which is produced by the residual encoding task; the current ME task cannot be executed before the previous residual encoding task has been completed. The backward arrows in Figure 6 (b) imply this dependency. To solve this problem we must break the feedback dependency between the ME task and the residual encoding task, which is accomplished by a novel pipelining technique based on sub-frame execution.

5.1. Sub-frame Asynchronous Execution

The proposed sub-frame execution is based on the observation that the ME task requires only the data within the search range in the reference frame, not the whole frame. Therefore even if not all data in the reference frame have
been produced by the residual encoding task, the ME task may start its execution. We divide a frame into several subframes and execute the ME task and the encoding task in the unit of sub-frames. These tasks do not execute the same subframe but execute different sub-frames in the pipelined fashion to break the feedback dependency.

Figure 7 shows the proposed sub-frame decomposition where a frame is divided into 4 sub-frames. The sub-frame size should not be less than the search region. In the figure, the current frame is shown in the right side and the previous frame in the left side. The sub-frames in the current frame are classified into two groups. The first group (group 1) includes the sub-frames whose previous sub-frame exists in the current frame, such as sub-frames 1, 2, and 3. In case of the first group, there is no dependency between the ME task and the encoding task since the ME task executes the current sub-frame referring to the previous frame while the encoding task execute the previous sub-frame which lies in the current frame. Hence the ME and the encoding tasks can be executed simultaneously.

The second group (Group 2) contains the first sub-frame only whose previous sub-frame exists in the previous frame. Since both the ME task and the encoding task access the same frame, we need to check whether the dependency between the two tasks exists or not. As shown in Figure 7, the ME task that processes the current sub-frame 0 accesses the first three sub-frames of the previous frame while the encoding task updates the fourth sub-frame.

Figure 8 shows the execution flow when we execute the ME and the encoding tasks in the sub-frame unit while the frame analysis task is executed per frame. The arrows indicate dependencies posed by the reference frame. Unlike Figure 6 (b), the arrows go forward following the execution order.

5.2. Kernel Merging

In the proposed SS algorithm, the total execution time is determined by the CPU time to execute the frame analysis task and the encoding task plus the kernel call overhead. Since a kernel call has a constant overhead and we launch kernels for each sub-frame, we need to reduce kernel call overhead for further optimization. In our implementation, the ME task consists of five kernels that computes the SAD value for 16x16, 16x8, 8x16, 8x8 size block and sub-pixel processing. Since their thread configurations are the same, they can be merged into one kernel to reduce the number of kernel calls.

By merging kernels, we reduce the execution time of the kernel as well as the kernel call overhead. Before kernel merging, the output data of each original kernel need to be stored in the global memory. In the merged kernel, however, the kernels can share the result using the shared memory which is much faster than global memory. Hence kernel merging may actually reduce the kernel execution time.

But caution is needed since kernel merging results in more pressure on the registers and the size of shared memory. It may limit the number thread blocks that can be executed simultaneously in GPU so that the kernel execution time can be even increased. In our case, the merged kernel execution time has been reduced as discussed in the next section.

6. EXPERIMENT

We have implemented the proposed SS algorithm on a target host that consists of a NVIDIA GTX 480 GPU and Intel Core2 Quad Q8300 running Windows XP. GTX 480 has 480 cores with 15 SMs and 1535 MB memory and supports PCI 2.0. The toolkit and SDK version for CUDA is 3.2 and the driver version is 263.06. We use three video sequences: city (4CIF, 704x576), shields (720p, 1280x720), Rush Hour (1080p, 1920x1080). Encoding options used in the experiments are as follow: one reference frame, no b-frames, no CABAC (Context Adaptive Binary Arithmetic coding) and 16x16, 16x8, 8x16, 8x8 block motion estimation.

To begin with, we have measured the execution time of the motion estimation kernel and the output file size varying n, the number of search points in the n nxn square search region. Figure 9 and Figure 10 show the kernel execution time and output file size respectively. The shortest execution time is obtained with 8 search points, or 3x3 square region. However, since the encoded file size with 8 points is considerably increased compared with 24 points, we choose to use 5x5 square region that contains 24 points in the current implementation. Another reason is to make the number of threads to be a multiple of 32 threads, which is a warp size in GTX480. If 24 points are used, the total number of threads in a thread block becomes 96 = 4·24 (T·m) as mentioned in section 4.2, which is a multiple of 32.
number threads for computing latency than texture memory reference frame execution global memory res reference frame are assigned to this experiment. The result is shown in texture memory.

Next we compare the 16x16 kernel implementation (second).

Figure 9. Kernel execution time with varying search region size

Figure 10. Output file size with varying search region size

Next we compare various memory configurations for 16x16 kernel implementation. For the current frame, it is assigned to the global memory, a shared memory, and the texture memory, denoting as G, S, T respectively. For the reference frame, it is assigned to the global memory or the texture memory. So, 6 combinations in total are tested in this experiment. The result is shown in Figure 11. As discussed in section 4.2, when the current frame and the reference frame are assigned to the shared memory and the global memory respectively, it shows the shortest kernel execution time. The reason that global memory is better for reference frame is that the hardware cache which has lower latency than texture memory is utilized.

We also measure 16x16 kernel execution time varying the number threads for computing one SAD. The best result is obtained when 4 threads are used as shown in Figure 12.

![Kernel Time (second) vs # of points](image1)

![Output File Size (Kbyte) vs # of points](image2)

Table 2. Kernel execution time and the kernel call overhead for separate kernels and a merged kernel

<table>
<thead>
<tr>
<th>Kernel Time(s)</th>
<th>City</th>
<th>Shields</th>
<th>RushHour</th>
<th>Reduction ratio(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate kernels(s)</td>
<td>0.982</td>
<td>2.481</td>
<td>6.018</td>
<td>17.19%</td>
</tr>
<tr>
<td>Merged kernel(s)</td>
<td>0.813</td>
<td>2.111</td>
<td>5.078</td>
<td>14.90%</td>
</tr>
<tr>
<td>Kernel call overhead(s)</td>
<td>0.425</td>
<td>0.545</td>
<td>0.256</td>
<td>57.10%</td>
</tr>
</tbody>
</table>

Table 2 shows the execution time of the kernel and the kernel call overhead to examine the performance improvement by using a merged kernel rather than separate kernels. The kernel execution time is reduced by 15% to 17% and the kernel call overhead is reduced by 53% to 57%. It proves that the kernel merging is quite effective in our implementation.

Next, we compare our GPU implementation with an x264 single threaded version which is fully optimized with MMX and SSE instructions. Figure 13 shows the execution time of ME in x264 and our implementation with the merged kernel. Compared with x264, the execution time of the proposed ME in GPU gets halved resulting in 2x speedup. However, the overall execution time for ME is almost the same since the memory copy overhead and the additional CUDA overhead is significant as shown in the figure.

Figure 14 shows the total execution time of x264 encoder and our GPU implementation with synchronous communication APIs and asynchronous APIs. As shown in Figure 13, the execution time of x264 and the SS algorithm with synchronous APIs is almost the same. However, compared with x264, the execution time of the proposed algorithm with asynchronous APIs is reduced by 17-20%, which is shown in Table 3. Note that the reduction ratio is close to the ideal case that Amdahl’s law implies after ME execution time is reduced to none by parallel execution and some kernel call overhead is added. It is because the ME kernel execution time and the memory copy overhead are completely hidden by asynchronous execution.

Table 4 shows the output file size and PSNR between x264 and the proposed encoder. The PSNR values of the two are almost the same for all cases, implying there is no loss in quality. The output file size is slightly increased by 3% to 9% depending on the video sequence. This decreased coding efficiency is mainly caused by the replacement of MVP with a simpler scheme used in the proposed technique. Nonetheless we believe that the loss is within the acceptable level.

Table 4 shows that in most previous work that has accelerated the ME task in CUDA, JM reference implementation is used for comparison, which is about three orders of magnitude slower than x264 [4]. No comparison result with x264 has been reported to our best knowledge while we actually accelerated the x264 with acceptable loss in coding efficiency.
Figure 13. Execution time of ME in x264 and GPU

Figure 14. Execution time of three H.264 encoder implementations: x264, our GPU implementation with synchronous APIs, and asynchronous APIs.

Table 3. ME Ratio in an x264 encoder and the reduction ratio through the proposed algorithm with asynchronous APIs

<table>
<thead>
<tr>
<th></th>
<th>ME Ratio</th>
<th>Reduction ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>City</td>
<td>26.00 %</td>
<td>17.69 %</td>
</tr>
<tr>
<td>Shields</td>
<td>23.37 %</td>
<td>19.06 %</td>
</tr>
<tr>
<td>RushHour</td>
<td>25.21 %</td>
<td>19.70 %</td>
</tr>
</tbody>
</table>

Table 4. Comparison of the output file size and PSNR

<table>
<thead>
<tr>
<th></th>
<th>X264</th>
<th>Proposed GPU Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>City</td>
<td>File Size</td>
<td>4829152</td>
</tr>
<tr>
<td></td>
<td>PSNR</td>
<td>37.812</td>
</tr>
<tr>
<td>Shields</td>
<td>File Size</td>
<td>13219547</td>
</tr>
<tr>
<td></td>
<td>PSNR</td>
<td>36.565</td>
</tr>
<tr>
<td>RushHour</td>
<td>File Size</td>
<td>11326260</td>
</tr>
<tr>
<td></td>
<td>PSNR</td>
<td>41.621</td>
</tr>
</tbody>
</table>

7. CONCLUSION

In this paper, we propose a novel motion estimation (ME) algorithm, called square search (SS) algorithm, tailored for NVIDIA GPU implementation. It is accompanied by a novel pipelining technique, called sub-frame ME processing, to effectively hide the communication overhead between the host CPU and the GPU. The proposed H.264 encoder achieves 2x speedup for the ME task and more than 20% speed-up for the entire encoding time.

In this paper, we have only considered parallelizing motion estimation in H.264 encoder. However, to get more performance gain, we need to consider parallelizing the other tasks such as the frame analysis task and the residual encoding task, as they are now the bottleneck of the current implementation.

8. ACKNOWLEDGEMENT

This work was supported by BK21 project, Seoul R&BD Program (JP090955), the 2010 Research Fund of the University of Seoul, and the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2011-C1090-1100-0010). The ICT at Seoul National University provided research facilities for this study.

9. REFERENCES