[Ngai-Man Cheung, Xiaopeng Fan, Oscar C. Au, and Man-Cheung Kung]

Video Coding on Multicore Graphics Processors

[The challenges and advantages of the GPU implementation]

Today, video coding [1]–[5] has become the central technology in a wide range of applications. Some of these include digital TV, DVD, Internet streaming video, video conferencing, distance learning, and surveillance and security [6]. A variety of video coding standards and algorithms have been developed (e.g., H.264/AVC [5], VC-1 [7], MPEG-2 [8], AVS [9]) to address the requirements and operating characteristics of different applications. With the prevalent applications of video coding technologies, it is important to investigate efficient implementation of video coding systems on different computing platforms and processors [10], [11].

Recently, graphics processing units (GPUs) have emerged as coprocessing units for central processing units (CPUs) to accelerate various numerical and signal processing applications [10], [12]–[14]. Modern GPUs may consist of hundreds of highly decoupled processing cores capable of achieving immense parallel computing performance. For example, the NVIDIA GeForce 8800 GTS processor has 96 individual stream processors each running at 1.2 GHz [15]. The stream processors can be grouped together to perform single instruction multiple data (SIMD) operations suitable for arithmetic intensive applications. With the advances in the GPU programing tools such as thread computing and C programming interface [16], [17], GPUs can be efficiently utilized to perform a variety of processing tasks in addition to conventional vertex and pixel operations.

With many personal computers (PCs) or game consoles equipped with multicore GPUs capable of performing general purpose computing, it is important to study how the GPU can be utilized to assist the main CPU in computation-intensive tasks such as video compression/decompression [18]. In fact, as high-definition (HD) contents are getting more popular, video coding

Digital Object Identifier 10.1109/MSP.2009.935416

would require more and more computing power. Therefore, leveraging the computing power of GPUs could be a cost-effective approach to meet the requirements of these applications. Note that with dozens of available video coding standards (H.264, MPEG-2, AVS, VC-1, WMV, DivX) it is advantageous to pursue a flexible solution based on software.

LEVERAGING THE COMPUTING POWER OF GPUs COULD BE A COST-EFFECTIVE APPROACH TO MEET THE REQUIREMENTS OF VIDEO CODING.

INTRODUCTION

Focusing on software-based video coding applications running on PCs or game consoles equipped with both CPUs and GPUs, this article investigates how GPUs can be utilized to accelerate video encoding/decoding. Recent work has proposed applying multicore GPUs/CPUs for various video/image processing applications. Table 1 summarizes some of them. In this article, we survey prior work on video encoding and decoding to illustrate the challenges and advantages of the GPU implementation. Specifically, we discuss previous work on GPU-based motion estimation (ME), motion compensation (MC), and intraprediction. Our focus is on how the algorithms can be designed to harness the massive parallel processing capability of the GPU. In addition, we discuss previous work on partitioning the decoding flow between CPUs and GPUs (for completeness, we also report the speedup results in previous work. However, since the GPU/multicore software/hardware technologies have evolved dramatically over the last few years, some of the results could be outdated). After that, we investigate a GPU-based fast ME. We discuss some strategies to break dependency between different data units and examine the tradeoff between speedup and coding efficiency.

BACKGROUND

VIDEO CODING

The latest video coding standards have achieved state-of-the-art coding performance. For example, H.264/AVC, which is the latest international video coding standard approved by the International Telecommunications Union (ITU-T) and the International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), typically requires 60% or less of the bit rate compared to previous standards to achieve the same reconstruction quality [5]. Other advanced video coding algorithms, such as AVS-Video developed by the Audio and Video Coding Standard Working Group of China [9], or VC-1 initially developed by Microsoft [7], have also achieved competitive compression performance. Next, we provide an overview of the H.264 video coding standard.

The H.264 video coding standard is designed based on the block-based hybrid video coding approach [2], [5], which has been used since earlier video coding standards. The coding algorithm exploits spatial correlation between neighboring pixels of the same picture. In addition, it also exploits temporal correlation between neighboring pictures in the input video sequence to achieve compression. Figure 1 depicts the encoder block diagram. The input picture is partitioned into different blocks, and each block may undergo intraprediction using neighboring reconstructed pixels in the same frame as predictor. H.264 supports intraprediction block sizes of 16×16, 8×8, and 4×4, and it allows different ways to construct the prediction samples from the adjacent reconstructed pixels. Alternatively, the input block may undergo interprediction using the reconstructed blocks in the reference frames as predictor. Interprediction can be based on partition size of 16×16, 16×8, 8×16, 8×8, 8×4, 4×8, or 4×4. Displacement between the current block and the reference block can be up to quarter-pel accuracy and is signaled by the motion vector and the reference picture index [2].

The prediction residue signal from intraprediction or interprediction would then undergo transformation to decorrelate the data. In H.264, a 4×4 separable integer transform is used, which is similar to 4×4 DCT but avoids the mismatch between forward and inverse transform. Then, the transform coefficients would be scalar quantized and zig-zag scanned. The context-adaptive variable length coding (CAVLC) may then be employed to entropy code the scanned transform coefficients. CAVLC is an adaptive coding scheme, and it may switch between different codeword tables during encoding depending on the values of the already-coded elements. Alternatively, the transform coefficients may be coded by context-adaptive binary arithmetic coding (CABAC). To mitigate blocking artifacts, an adaptive in-loop deblocking filter would be applied to the reconstruction from the feedback loop.

GPUs

Originally designed as specialized hardware for three-dimensional (3-D) graphics, GPUs have recently emerged as coprocessing units to accelerate arithmetic intensive applications in

[TABLE 1] VIDEO AND IMAGE PROCESSING APPLICATIONS ON MULTICORE PROCESSORS.

APPLICATIONS	EXAMPLES
VIDEO ENCODING	MOTION ESTIMATION [19]–[23], INTRAPREDICTION [24]–[27], TRANSFORM [28]
VIDEO DECODING	MOTION COMPENSATION [10], [29], DECODER DESIGN [10], [30]–[32]
HIGH DYNAMIC RANGE IMAGES	TEXTURE COMPRESSION [33]
VIDEO WATERMARKING	REAL-TIME VIDEO WATERMARKING SYSTEM [14]
SIGNAL PROCESSING KERNELS	MATRIX AND VECTOR COMPUTATIONS [12], FFT, AND CONVOLUTION [13]
IMAGE ANALYSIS	HOUGH TRANSFORM [34], RADON TRANSFORM [35], [36], CHIRPLET TRANSFORM [35], FEATURE EXTRACTION [37]

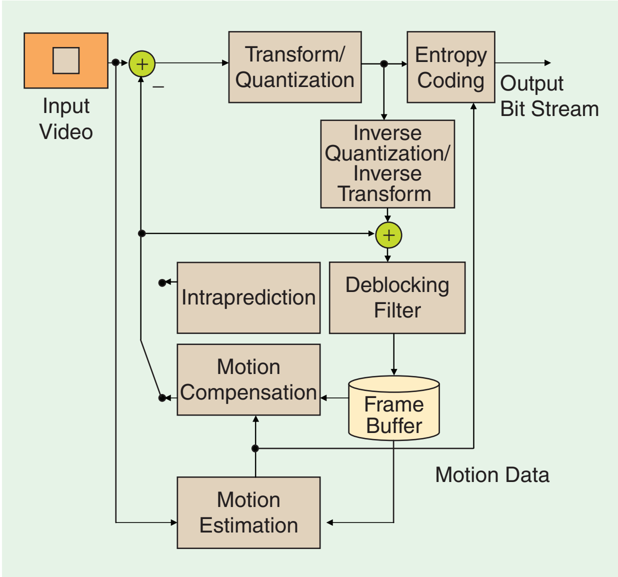
PCs or game consoles. A key feature of modern GPUs is that they offer massive parallel computation capability through hundreds of highly decoupled processing cores [38]. For example, NVIDIA GeForce 8800 GTS processor consists of 96 stream processors each running at 1.2 GHz [15].

The design philosophy of GPUs is quite different from that of general-purpose CPUs. Throughout the years, GPUs have been designed with an objective to support the massive number of calculations and huge amount of data transfer required in advanced video games [38], [39]. In addition, they need to meet the stringent cost requirement of consumer applications. Therefore, GPUs have become very cost effective for arithmetic computation. Furthermore, the peak computation capability of GPUs is increasing at a faster pace than general-purpose CPUs (Figure 2).

Besides arithmetic computation capability, there are other fundamental differences between CPUs and GPUs. First, to address a wide range of applications, general-purpose CPUs would use many transistors to implement sophisticated control hardware that can support some advanced control functions such as branch prediction [40]. On the contrary, GPUs would instead devote chip area to arithmetic computation. As a consequence, GPUs may not perform well for programs with many conditional statements. Second, CPUs use a lot of chip area to implement cache memory to reduce instruction and data access latencies. GPUs, on the other hand, use much simpler memory models but rely on the high degree of parallelism in an application to hide the memory access latency. Thus, it is central to expose a large amount of data parallelism in the GPU programs.

GPU-ASSISTED VIDEO CODING: CHALLENGES

Following from the previous discussion, it is clear that only certain types of computation are suitable for the GPU execution. In particular, to fully harness the computational power in the GPU, one would need to design the algorithm to utilize the massive number of processing cores in parallel. As an example, a good application may run up to thousands of threads simultaneously on a high-end GPU so as to keep all the processing cores working continuously [38]. Therefore, one of the main challenges to utilize the GPU for video coding is how to structure a certain module to expose as much data parallelism as possible. Note that this may not be trivial for some video coding modules since dependency may exist between different data units in the computation, as pointed out by previous work [22], [24], [25], [27]. Moreover, flow control instructions (`if`, `switch`, `do`, `for`, `while`) can significantly degrade the performance of the GPU execution, since such instructions may cause different threads to follow different execution paths and the execution would need to be serialized [39]. Therefore, using GPUs for entropy coding such as CAVLC could be challenging. Furthermore, an implementation should try to avoid as much as possible off-chip data access, which may incur considerable latency (recall that the GPU is not optimized for memory access latency). For example, some GPUs may require from 400 to 600 cycles latency for off-chip memory access (while they can

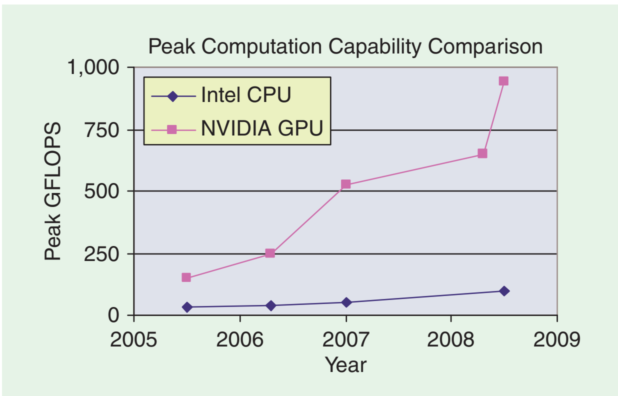


[FIG1] H.264/AVC encoding algorithm [5].

perform single-precision floating-point multiply-add in a single cycle in each core) [39]. Note that it is possible to hide such memory access latency if there are enough independent arithmetic computations. Therefore, if possible, a video coding module should be implemented with high *arithmetic intensity* (which is defined as the number of mathematical operations per memory access operation). In some situations, it could be more efficient to recalculate some variables rather than loading them from the off-chip memory.

PREVIOUS WORK

In this section, we review previous work on applying GPUs for video coding. Previous work has proposed to utilize GPUs to undertake ME [19]–[22], intraprediction [24]–[27], and MC [10], [29]. Note that ME, intraprediction, and MC are some of the most computation-intensive modules in interframe encoding, intraframe encoding, and decoding, respectively. Therefore, it is important to understand how these modules can be efficiently implemented on GPUs. In addition to these



[FIG2] Peak computation capability of GPUs and CPUs [39], [41].

modules, GPU-based discrete cosine transform (DCT) has been discussed in [28]. There seems to be no previous work on the GPU-based deblocking filter. Since the deblocking filter involves some conditional statements to determine the strength of the filter at each block boundary, some study may be necessary to determine its performance on GPUs.

WE DISCUSS SOME STRATEGIES TO BREAK DEPENDENCY BETWEEN DIFFERENT DATA UNITS AND EXAMINE THE TRADEOFF BETWEEN SPEEDUP AND CODING EFFICIENCY.

MOTION ESTIMATION ON GPUs

ME is one of the most computation-intensive modules in video encoding and there has been a lot of interest to offload it to GPUs to improve the overall encoding performance. Earlier work in this area focuses on ME algorithms where sum of absolute differences (SAD) is used in block matching to determine the best candidate. SAD computation can be easily parallelized as each individual pixel in the current block is compared independently with the corresponding pixel in the candidate reference block. Note that SAD-based ME is commonly used in MPEG-1/2 and H.263.

More recent video encoding algorithms, on the other hand, may employ rate-distortion (RD)-optimized ME that considers both the rate and distortion in selecting the best candidate. For example, one common metric is the weighted sum of the SAD (between the current block and the candidate block) and the encoding rate of the motion vectors (MVs). In the H.264 standard, predictive coding is used to encode the MV of the current block, and the predictor is the median of the MVs in the adjacent left, top and top-right blocks. Therefore, in RD-optimized ME, the MVs of the neighboring blocks would need to be first determined. Then, based on the median of the neighboring MVs, the encoding rate of the current MV can be determined and the cost of the current block can be computed in the block matching. Such dependency makes it difficult to utilize GPUs for RD-optimized ME. We will discuss example designs to address this issue.

GPU-BASED ME BASED ON LOOP UNROLLING

To increase the degree of parallelism, [20] proposed to unroll the computation loop in SAD-based full search ME. The ME computation loop is shown in Figure 3, and loop unrolling is possible since there is no dependency between individual macroblocks (MBs) when SAD is used as metric for matching. Due to resource constraint in earlier GPUs, the algorithm in [20] needs to be partitioned into two separate passes so that the GPU memory can accommodate the instructions. The experiments in [20] compared full search ME on an INTEL Pentium 4 3.0 GHz CPU and on a NVIDIA GeForce 6800 GT GPU, and the results suggest the GPU-based ME can achieve up to two times and 14 times of speedup for integer-pel and half-pel ME, respectively. The considerable improvement in the half-pel ME is due to the fact that [20] utilizes the built-in hardware support in the GPU for interpolation.

```
Loop (rows of macroblocks) {
   Loop (columns of macroblocks) {
      Loop (rows of search range) {
         Loop (columns of search range) {
            SAD computation;
            SAD comparison;
         }
      }
   }
}
```

[FIG3] Pseudocode of conventional integer-pel ME based on SAD.

Note that with loop unrolling it is possible to schedule a massive number of parallel threads (subject to device's constraint). Consider an example to assign one thread to compute one SAD between an MB and a candidate block in the search window. Then, in the case of full search, the number of independent threads could be as large as the number of MBs times the number of candidate blocks per MB (search window size). For HD 720p videos ($1,280 \times 720$, 3,600 MBs per frame), and a search range of 64 (129×129 search window size), the number of threads could be as many as $3,600 \times 129 \times 129 = 59,907,600$.

Although full search is highly parallel, it may have only little practical interest because of the prohibitive computational requirement, especially for HD video contents. Moreover, when MBs are processed independently and MVs are computed concurrently in different threads, it becomes difficult to use motion vector prediction, where MVs of neighboring blocks are used to initialize the search of current MB, and this may affect ME performance when the search window is small. In the next section, we will discuss the GPU implementation of fast ME, which can (in general) achieve comparable coding performance as full search with a much smaller number of computations [42].

GPU-BASED ME BASED ON REARRANGING THE ENCODING ORDER

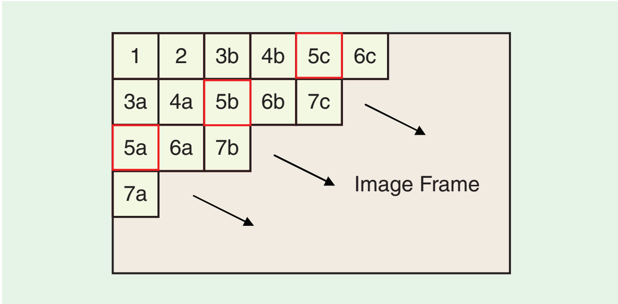
Due to the dependency between adjacent blocks as discussed, RD-optimized ME commonly employed in recent video coding standards cannot be parallelized simply by loop unrolling. In [21] and [22], rearrangement of the encoding order is proposed to increase the degree of parallelism. In these algorithms, instead of processing the blocks in the conventional raster-scan order, the blocks are processed along the diagonal direction to address the dependency issue. This is shown in Figure 4 for the case of 4×4 ME. In their proposed encoding order, at each iteration, the ME will process all the blocks of which the neighboring blocks (left, top, and top-right) have been processed. That is, the ME processes at each iteration all the blocks of which neighboring MVs have been computed and median predictors are available. By processing blocks along the diagonal direction the proposed rearrangement can substantially increase the degree of parallelism. For example, [22] reported that the maximum degree of parallelism can be up to 44, 160, and 240 for CIF (352×288), 720p, and 1080p video, respectively. Note that for each 4×4 block, individual search points in the search window can be examined in parallel (in the cases of full search or some fast search with regular sampling of search window). Therefore, with block-level parallelism of 240 (i.e., 240 4×4 blocks in the current frame can be processed in parallel) and a search range of 64 (129×129 search window size), $240 \times 129 \times 129 = 3{,}993{,}840$ independent threads can be launched simultaneously in principle. Pixel level parallelism can also be implemented, e.g., by decomposing the SAD calculation into several threads. The results in [22] suggest that over 40 times of speedup can be achieved in a system with an INTEL Pentium 4 3.2 GHz CPU and a NVIDIA GeForce 8800 GTS graphics processor. Note that Pentium 4 CPUs are relatively slow compared with more recent CPUs. Also, the program code on Pentium might not have been well optimized. Thus the reported speedups in [22] could be higher than those w.r.t. more efficient CPU implementation. Nonetheless, the results still suggest RD-optimized ME can be implemented efficiently on GPUs.

RD-OPTIMIZED INTRAMODE DECISION ON GPUs

Recent video encoding algorithms use RD-optimized intramode selections to determine the optimal intraprediction direction. In these methods, the encoder would compute the Lagrangian costs of all the candidate prediction modes and select the prediction mode that minimizes the cost. The Lagrangian cost can be the weighted sum of the sum of square differences (SSD) between the original and reconstructed block and the encoding rate for header and quantized residue block. To calculate the cost for a candidate mode, it may involve computing the intraprediction residue, transformation, and quantization on the prediction residue, inverse quantization and inverse transformation, and entropy coding of the quantized transform coefficients. Therefore, the computational complexity of RD-optimized intramode selections could be very significant [43]–[45].

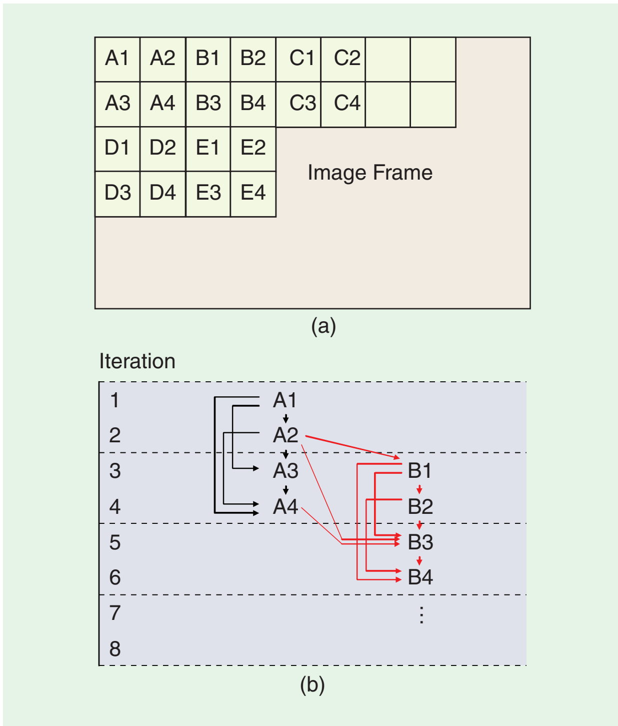
Achieving massive parallelization of RD-optimized intradecision can be challenging. It is because, in intraprediction, the reconstructed pixels of the neighboring blocks are used to compute the reference samples. Therefore, the intraprediction modes of the neighboring blocks would need to be first determined, and these blocks would be encoded and reconstructed accordingly. Then, different candidate modes of the current block can be evaluated based on the reconstructed pixels in the neighboring blocks. Such a dependency hinders the parallelization of the RD-optimized intradecision for GPU implementation.

To address the dependency issue, previous work has proposed different strategies to modify the block processing order [26], [27], [46]. In particular, [27] analyzes the dependency constraint and proposes to process the blocks following a greedy strategy: in each iteration, the encoder would process all the blocks of which parent blocks have been encoded (in the dependency graph, Block A is the parent block of block B if block B requires the reconstructed pixels from block A under various candidate prediction modes). Also, in the greedy strategy, a video block will



[FIG4] Block encoding order proposed in [22] for H.264 4×4 RD-optimized ME. Each square represents a 4×4 block. Blocks with the same number (e.g., 5a, 5b, 5c) are to be processed in parallel.

be scheduled for processing immediately after all its parent blocks have been processed. Figure 5 depicts the dependency constraint in H.264 4×4 intraprediction and the scheduling under the greedy strategy. [27] argues that the greedy strategy is optimal for H.264 and AVS encoding: under the specific constraints imposed by H.264/AVS, and among all encoding orders obeying the constraints, the greedy-based encoding order



[FIG5] (a) Notations for dependency graph: each block corresponds to a 4×4 block. (b) Dependency graph when processing an image frame in H.264 RD-optimized intramode selection. Each node represents a 4×4 block (see (a) for notations). A directed edge going from block *A* (parent node) to block *B* (child node) indicates that block *B* requires the reconstructed pixels from block *A* to determine the RD costs of various candidate prediction modes. The graph is processed following the greedy strategy proposed in [27], and the figure shows the iteration at which each block is processed.

requires the minimum number of iterations to process all the blocks. Simulation results suggest that, using the greedy strategy, GPU-based intramode decision can achieve about up to two times speedup in a system with an INTEL Pentium 4 3.2 GHz CPU and a NVIDIA GeForce 8800 GTS graphics processor (Table 2). According to [27], the average parallelism is about 127 for 1080p videos, and a two times speedup seems to agree with our results given in the section "Case Study: GPU-Based Fast Motion Estimation."

TO FULLY HARNESS THE COMPUTATIONAL POWER IN THE GPU, ONE WOULD NEED TO DESIGN THE ALGORITHM TO UTILIZE THE MASSIVE NUMBER OF PROCESSING CORES IN PARALLEL.

MOTION COMPENSATION ON GPUs

GPU-based MC has been proposed by [10] and [29] for Windows media video (WMV) and H.264 video decoding, respectively. MC requires a lot of computations, since video coding standards allow MVs to point to subpixel locations (e.g., half-pel or quarter-pel) and intensive pixel interpolation would be necessary to generate the prediction samples for motion displacements with fractional values. For example, in H.264, a half-pel sample is generated from six other samples using a six-tap interpolation filter. And to generate a quarter-pel sample it may require an additional linear interpolation.

The work in [10] discusses techniques to address the overflow and rounding problem in interpolation that arose in MC. Note that MC can be parallelized since each block can be processed independently using its motion vector information, and this is implemented by a pipeline of vertex/pixel shader procedures in [10]. In their GPU implementation, they use a multipass technique that handles the residuals and rounding control parameter in a separate pass to avoid overflow while preserving the precision. In addition, [10] discusses how different modules in video decoding can be partitioned between the CPU and the GPU, and how the CPU computation can be maximally overlapped with the GPU computation (this will be further discussed). Simulation results suggest that, in a system with an INTEL Pentium III 667 MHz GPU and a NVIDIA GeForce3 Ti200 GPU, by leveraging the GPU the system can achieve more than three times of speedup, and it is possible to achieve real-time WMV (version 8) decoding of HD video of resolution up to 1280×720 [10].

[TABLE 2] COMPARISON BETWEEN THE PARALLEL H.264 INTRAPREDICTION ON GPU PROPOSED IN [27] AND CONVENTIONAL H.264 INTRAPREDICTION ON THE CPU. THE NUMBERS ARE THE RATIOS OF THE CPU RUNNING TIME TO THE GPU RUNNING TIME. NOTE THAT THE GPU RUNNING TIME INCLUDES ALL THE DATA TRANSFER OVERHEAD.

	QP = 28	QP = 36	QP = 44
CIF:			
FLOWER_CIF	1.14	1.12	1.14
PARIS_CIF	1.12	1.14	1.12
MOBILE_CIF	1.14	1.12	1.12
AVERAGE (CIF)	1.13	1.13	1.13
1,280 × 720:			
CREW	1.38	1.40	1.37
NIGHT	1.49	1.42	1.39
CITY	1.48	1.47	1.43
AVERAGE (1,280 × 720)	1.45	1.43	1.39
1,920 × 1,080:			
BLUE_SKY	1.90	1.82	1.73
RIVERBED	1.93	1.82	1.76
STATION	1.89	1.81	1.80
AVERAGE (1,920 × 1,080)	1.91	1.82	1.76

TASK PARTITION BETWEEN THE CPU AND THE GPU

To obtain competitive system performance, the CPU and the GPU need to be considered together for encoding/decoding. Investigating the optimal partition of computation tasks between the CPU and the GPU, however, could be very involved, and it requires serious evaluation on many issues. For example:

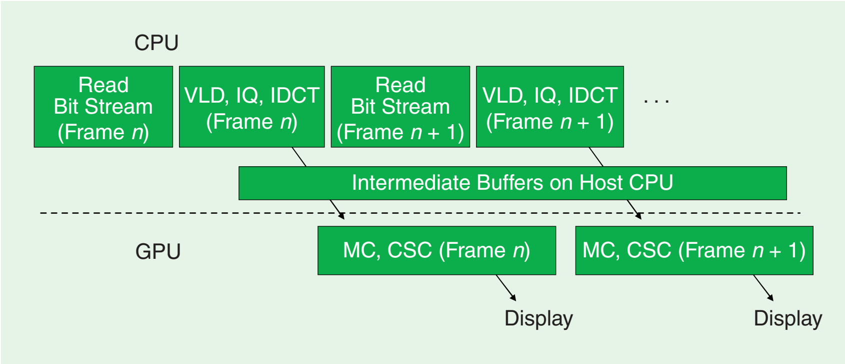
- It is necessary to investigate how to allocate the tasks such that the GPU computation can overlap with the CPU computation as much as possible, thereby achieving maximal parallel processing.
- Since the bandwidth between the GPU memory and main memory could be slow, it is important to investigate how to minimize the data transfer between main memory and the GPU memory.
- It is also important to study and evaluate which modules in the encoding/decoding flow can be efficiently offloaded to the GPU, while others would be executed on CPU.

Focusing on WMV decoding, [10] proposes a partition strategy where the whole feedback loop, including MC and color space conversion (CSC), is offloaded to the GPU. By doing so, they can avoid transferring the data back from the GPU to the CPU. Since read-backs from the GPU memory to main memory could be slow due to common asymmetric implementation of the memory bus [10], such read-backs should be minimized. Figure 6 depicts the partition strategy. Note that while the GPU is performing MC and CSC of frame n, CPU would be performing variable-length decoding (VLD), inverse quantization (IQ), and inverse DCT (IDCT) of the frame $n + 1$. Note also that intermediate memory buffer is used between the CPU and the GPU to absorb the jitters in CPU/GPU processing time. Simulation results in [10] suggest intermediate buffer size of four frames can considerably improve the overall decoding speed.

While [11] has discussed some issues (e.g., bandwidth requirement) on offloading ME to the GPU, there seems to be no prior work on rigorous investigation on how video encoding may be partitioned between the CPU and the GPU. We remark that the GPU implementation of several important encoding modules (including ME, intramode decision, MC, and transform) have been investigated in the past, while that of deblocking filter and entropy coding need further research.

CASE STUDY: GPU-BASED FAST MOTION ESTIMATION

To illustrate some design considerations in using GPUs for video coding, we discuss in detail in this section a GPU-based fast ME (the GPU ME code was developed by the authors based on the H.264 JM 14.2 reference software). The focuses are on how to address the data dependency in the algorithm to harness the parallel processing capability of GPUs, and on how to tradeoff the speedup with RD performance.



[FIG6] Task partitioning in WMV decoding proposed by [10].

FAST MOTION ESTIMATION

Our GPU implementation of fast ME is based on *simplified unsymmetrical multihexagon search (smpUMHexagonS)* [42], which is one of the fast ME algorithms adopted by the H.264 JM reference software. We select smpUMHexagonS because it can achieve very good tradeoff between computational complexity and coding efficiency. For example, on a Pentium 4 CPU it was reported smpUMHexagonS can achieve up to 94% reduction in ME execution time with comparable RD efficiency, when compared with the fast full search in the JM software [42]. In addition, smpUMHexagonS is quite compact, so it could meet the memory constraint of the GPU. In our implementation, all the GPU kernels that deal with integer-pel estimation have about 600 lines of code.

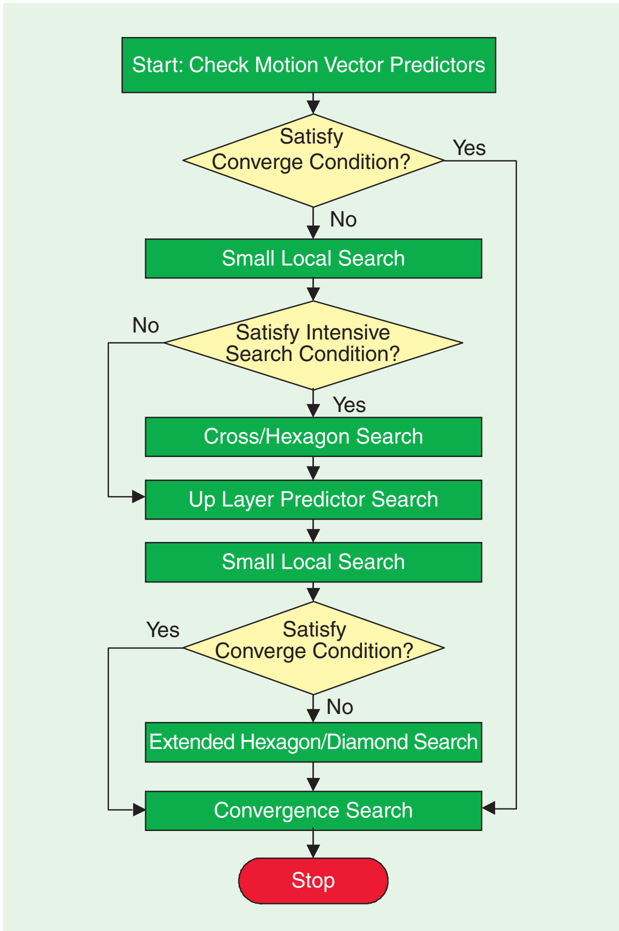
Figure 7 depicts the flow chart of smpUMHexagonS. For each MB, smpUMHexagonS computes the MVs for all the MB partitions (16×16, 16×8, $\ldots$ 4×4). MVs are selected by minimizing the Lagrangian cost $D + \lambda R$, where D is the SAD between the current block and the candidate, and R is the bitrate to encode the MV. In smpUMHexagonS, computation reduction is achieved mainly by sampling the search space judiciously, using several techniques including motion vector prediction, different search patterns (cross, hexagon, and diamond) and early termination. In particular, MVs from spatially adjacent blocks and from other MB partitions are used to initialize the search for the current partition. Notice that as depicted in Figure 7, smpUMHexagonS uses several tests to determine if the search (of the current partition) can be terminated based on the minimum cost computed so far. As a result, different MBs with different contents may undergo different processing paths (which is typical in many fast ME algorithms [47]), and this may affect the performance of the GPU implementation.

THE GPU IMPLEMENTATION USING TILING

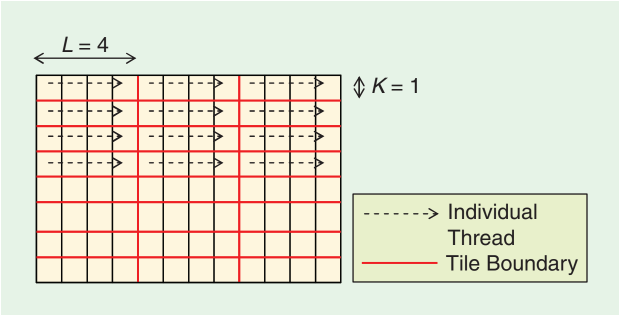
To utilize the parallelism in the GPU, we partition the current frame into multiple tiles, and each tile contains K (height) $\times$ L (width) MBs. For example, Figure 8 depicts the case with $K = 1$, $L = 4$. Each tile is processed by a single GPU thread, i.e., each thread processes $K \times L$ MBs in a tile sequentially, and different tiles are processed by different independent threads concurrently on the GPU.

Following from the discussion in the section "Fast Motion Estimation," individual MBs are not independent under smpUMHexagonS. In particular, MBs depend on their neighbors in the following ways:

- First, to compute the rate term R in the Lagrangian cost the MVs of the neighboring MBs are required. If a neighboring MB belongs to another tile, we assume its motion vector is equal to zero in computing R. Therefore, with tiling, the



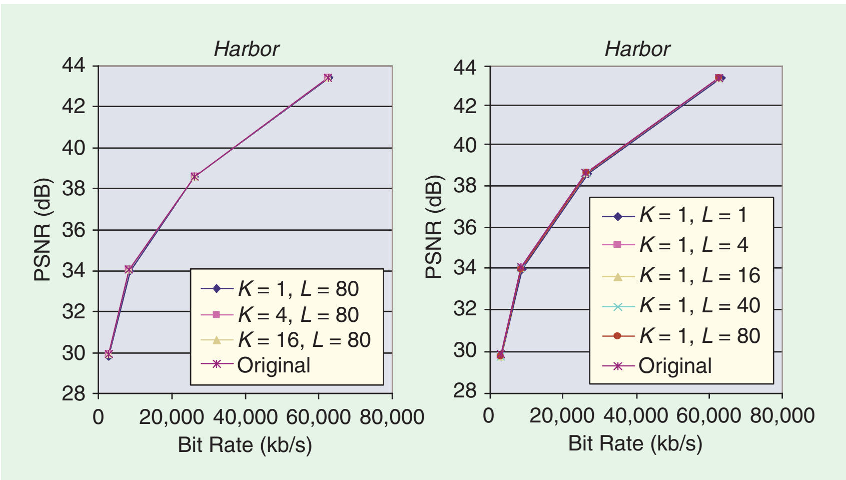
[FIG7] Fast ME using smpUMHexagonS [42]. The figure depicts the steps for integer-pel search for an MB partition.



[FIG8] GPU-based fast ME: the current frame is divided into multiple tiles to facilitate parallel processing in ME. Here each square represents an MB.

computed Lagrangian cost may not be very accurate and suboptimal MVs may be chosen by smpUMHexagonS as a result. The impact of tiling in this case depends on the value of λ and hence the target bit rate. For low bit-rate applications (rate constrained), encoders would focus more on rate efficiency, and large λ would be chosen and the rate term would dominate the Lagrangian cost [48]. Tiling therefore shall have a more pronounced negative impact on the performance of smpUMHexagonS for low bit-rate applications (since tiling affects the rate term).

- Second, smpUMHexagonS (and many other fast ME [47]) uses motion vector prediction, i.e., MVs of the neighboring MBs are used to initialize the search. Under tiling, some information about neighboring MVs is not available, and this may result in poor-quality initial search points, and suboptimal MVs may get selected at the end of the search (hence the RD performance is compromised). Moreover, since smpUMHexagonS employs early termination, poor initial points may also result in longer processing time, as more search points would need to be examined until the cost is small enough to terminate the search (e.g., we observe about 4% increase in the ME processing time when encoding the HD 720p sequence *Harbor* using tiling $K = 1, L = 1$ in the sequential smpUMHexagonS).



[FIG9] RD performance of *Harbor* with different tile sizes in fast ME. Here "original" refers to the reference software (that is, without tiling).

The above discussions are also applicable to many other fast ME algorithms. Note that in our simulation, tiling is used only in ME to facilitate the GPU computation, and the rest of the encoding proceeds in the same manner as in the reference software. Therefore, our tiling is different from other partitioning ideas such as slice [47], where individual partitions are treated independently in most of the encoding.

EXPERIMENTS

To examine the performance of the GPU-based fast ME using tiling, we conduct experiments on PCs equipped with one GeForce 8800 GTS PCIe graphics card with 96 stream processors [15], and an Intel Core 2 Quad Q9400 2.66 GHz CPU with 3.23 GB of RAM. We use NVIDIA's Compute Unified Device Architecture (CUDA) [39] to implement the GPU code. We choose CUDA solely because of the availability of the NVIDIA device in our laboratory, and we remark that there are other well-designed GPU programming models such as ATI CTM [49], Stream Computing SDK, and Brook+ [50].

We first evaluate how tiling may affect the RD performance. We use JM 14.2 to encode HD 720p sequences (1280×720, 60 frame per second) *Crew*, *City*, *Harbor* and *Night* (We focus on encoding HD videos because of its high computational requirement, and because of the growing interest on HD contents.) We use H.264 high profile with search range of 64. All the pictures are encoded as P-frames except the initial I-frame. Figure 9 depicts the RD performance with different tile sizes for the *Harbor* sequence. As shown in the figure the impact of tiling is small in this case until tile size is down to $K = 1, L = 1$, when the degradation is about 0.2 dB compared to the original reference software (with smpUMHexagonS). Table 3 shows the average peak signal-to-noise ratio (PSNR) degradation and the average increase in bit rate using different tile sizes, measured by Bjontegaard Delta PSNR (BDPSNR) and Bjontegaard Delta bit rate (BDBR), respectively. Note that BDPSNR and BDBR are used frequently in the video standardization community [51]. The results suggest tiling may lead to average degradation between 0.08 dB to 0.4 dB for these sequences with tile size $K = 1, L = 1$.

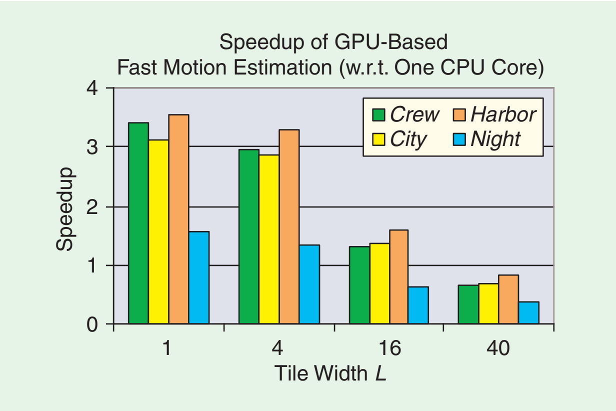
We then discuss how tiling may affect the speedup. Table 4 shows the GPU execution time (in integer-pel ME) with different tile sizes, and Figure 10 shows the speedup between the GPU implementation (with tiling and using parallel processing on multicore) and the sequential CPU implementation (without tiling and using sequential processing on a single core). Comparison with parallel program code on multiple CPU cores will be discussed next. The GPU execution time includes the overhead to transfer the video frames from system memory to the GPU memory. Compiler optimization is

[TABLE 3] TRADEOFF BETWEEN TILE SIZE AND RD PERFORMANCE. AVERAGE INCREASE IN BIT RATE AND AVERAGE PSNR DEGRADATION ARE COMPUTED WITH RESPECT TO THE REFERENCE SOFTWARE (I.E., WITHOUT TILING).

		CREW		*CITY*		*HARBOR*		*NIGHT*	
TILE SIZE	NUMBER OF TILES	BDBR (%)	BDPSNR (DB)	BDBR (%)	BDPSNR (DB)	BDBR (%)	BDPSNR (DB)	BDBR (%)	BDPSNR (DB)
$K = 1,\ L = 1$	3,600	3.135	−0.082	12.933	−0.407	5.578	−0.221	4.636	−0.17
$K = 1,\ L = 4$	900	3.081	−0.079	11.115	−0.352	2.385	−0.094	3.546	−0.13
$K = 1,\ L = 16$	225	3.116	−0.08	11.171	−0.35	2.246	−0.089	3.415	−0.125
$K = 1,\ L = 40$	90	3.224	−0.083	10.821	−0.339	2.205	−0.087	3.4	−0.124
$K = 4,\ L = 80$	12	0.63	−0.016	1.412	−0.044	0.57	−0.022	1.19	−0.043
$K = 16, L = 80$	3	0.094	−0.003	0.261	−0.008	0.07	−0.003	0.161	−0.006

applied to both the GPU program and the CPU program. However, both the GPU/CPU code have rooms for further speed improvement. In particular, the GPU code stores pixel data in global memory (off-chip memory), which has considerable access latency [39]. As ME is fairly memory access intensive (SAD calculation performs only three mathematical operations per two memory loads, giving an arithmetic intensity of 1.5, which is rather small for the GPU computation [37]), such latency may impact the GPU code performance. Therefore, the code can be improved by judicious use of shared memory (on-chip memory) [37]. As shown in Figure 10 speedup increases with smaller tile size, as more independent threads can be scheduled. This is particularly important in the current GPU code to hide the memory access latency. Note also that different sequences have different GPU execution time and speedups, as different video contents may lead to different execution paths in smpUMHexagonS and different amount of penalty incurred by execution serialization. Figure 10 suggests speedups of 1.5–3.5 can be achieved in integer-pel smpUMHexagonS in these sequences using tile size $K = 1, L = 1$.

Figure 11 shows the speedup between the GPU implementation and a parallel CPU implementation using the four CPU cores on the Intel Core 2 Quad. To achieve parallel CPU processing, the current frame is partitioned into four tiles of equal number of MB rows (i.e., L = width of the video frame in MB, K = height of the video frame in MB/4), and each tile is processed by an independent thread running on a CPU core. We use OpenMP to implement the parallel CPU program [52]. We observe the parallelization reduces the CPU running time by a factor of three approximately. Note that the theoretical maximum speedup of four cannot be achieved by this parallelization strategy, as smpUMHexagonS may spend different execution time on each MB and optimal load balancing cannot be achieved by simple tiling. Figure 11 suggests the running time of the GPU implementation



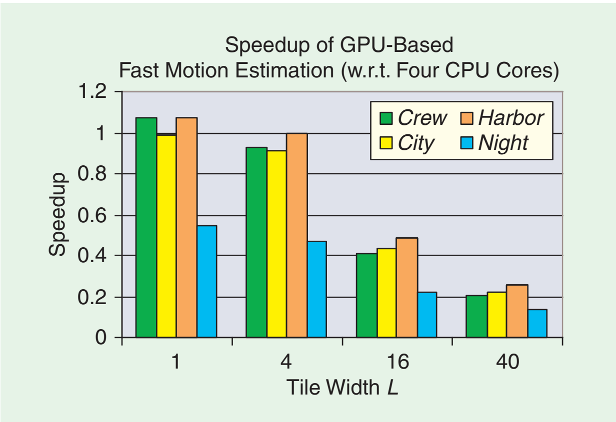
[FIG10] **Tradeoff between tile width and speedup (tile height K is equal to one). Speedup is the ratio of the CPU running time (sequential program code on one CPU core) to the GPU running time (including data transfer overhead).**

and the parallel CPU implementation can be comparable in some cases (while the GPU implementation incurs some RD performance degradation as depicted in Table 3).

In the experiment, we observe the overhead to transfer a frame from the CPU to the GPU is about 1.6 ms, and this is about 0.1–0.2% of the running time of integer pel ME (see Table 4). In general, data transfer overhead could be a less serious issue in interframe encoding compared with decoding and

[TABLE 4] GPU EXECUTION TIME FOR FAST INTEGER-PEL MOTION ESTIMATION WITH DIFFERENT TILE WIDTH (TILE HEIGHT K IS EQUAL TO ONE). DATA TRANSFER OVERHEADS ARE INCLUDED.

		GPU EXECUTION TIME (MS)			
TILE WIDTH	NUMBER OF THREADS	*CREW*	*CITY*	*HARBOR*	*NIGHT*
$L = 1$	3,600	835.05	927.32	1,248.95	1,688.50
$L = 4$	900	959.16	1,005.55	1,341.45	1,975.95
$L = 16$	225	2,169.25	2,108.71	2,763.79	4,175.44
$L = 40$	90	4,373.63	4,165.28	5,318.38	6,920.73



[FIG11] **Tradeoff between tile width and speedup (tile height K is equal to one). Speedup is the ratio of the CPU running time (parallel program code on four CPU cores) to the GPU running time (including data transfer overhead).**

intraframe encoding, since interframe encoding requires a significantly larger amount of execution time in general.

Finally, we would like to remark that both the CPU and the GPU implementations can be further optimized. Our discussion has suggested that it is nontrivial to achieve the peak performance offered by these multicore devices in video coding, and more algorithm research and instruction level optimization would be needed.

[IT IS NONTRIVIAL TO ACHIEVE THE PEAK PERFORMANCE OFFERED BY THESE MULTICORE DEVICES IN VIDEO CODING, AND MORE ALGORITHM RESEARCH WOULD BE NEEDED.]

CONCLUSIONS AND DISCUSSION

We have reviewed previous work on using GPUs for video encoding and decoding. In particular, we have discussed how some video coding modules can be implemented in certain ways to expose as much data parallelism as possible, so that the massive parallel-processing capability of GPUs can be fully utilized. Simulation results in previous work suggest GPU-based implementations can achieve considerable speedups for some of the most computation-intensive modules in video coding. Therefore, it could be a cost-effective approach to leverage the computing power of GPUs to meet the data processing requirement in video coding. We have also discussed an example to partition the video decoding flow between CPUs and GPUs to achieve maximum overlapping of computation. In addition, we have discussed a GPU-based fast ME and examined the tradeoff between speedup and RD performance.

There are several related research issues. First, there seem to be no studies on partitioning the encoding flow between CPUs and GPUs. Second, with the availability of many different video formats (e.g., SD and HD) and coding standards there is a growing need to transcode one encoded video format to another [53]–[55]. However, while there are a few commercial transcoding applications available [56], [57], there seems to be no prior work on investigating the optimal usage of GPUs for transcoding. Note that unlike video encoding/decoding, there is no standard algorithm for video transcoding, and there are many previously proposed approaches that achieve a wide range of transcoding quality with different complexity requirements [53]. This complicates the study of GPU-based transcoding.

ACKNOWLEDGMENTS

This work has been supported in part by the Innovation and Technology Commission (GHP/048/08) and the Research Grants Council (RPC07/08.EG22 and 610109) of the Hong Kong Special Administrative Region, China. The authors would also like to thank the editor and the anonymous reviewers for their helpful comments, which significantly improved the article.

AUTHORS

Ngai-Man Cheung (ncheung@stanford.edu) received the Ph.D. degree from the University of Southern California (USC), Los Angeles, in 2008. He is currently a postdoctoral researcher with the Information Systems Laboratory, Stanford University, Stanford, California. He was a research associate with Hong Kong University of Science and Technology (HKUST) from 2008 to 2009. His research interests include multimedia signal processing and compression, multimedia retrieval, and multicore and GPU applications. He received paper awards from *EURASIP Journal of Advances in Signal Processing*, 2007 IEEE International Workshop on Multimedia Signal Processing, IS&T/SPIE VCIP 2008, and the USC's Department of Electrical Engineering. He has five granted U.S. patents with others pending.

Xiaopeng Fan (eexp@ust.hk) received the B.S. and M.S. degrees in 2001 and 2003, respectively, in computer science from the Harbin Institute of Technology, China. He is currently pursuing the Ph.D. degree in the Department of Electrical and Computer Engineering, HKUST. He worked with Intel China Software Laboratory, Shanghai, from 2003 to 2005. His current research interests are in image/video coding and processing, video streaming, and wireless communication.

Oscar C. Au (eeau@ust.hk) received the Ph.D. from Princeton University in 1991. He is with the Department of Electrical and Computer Engineering of HKUST. Some of his research interests include video coding, ME, rate control, denoising, deinterlacing, multiview coding, scalable video coding, distributed video coding, and the GPU. He has over 260 papers and more than 50 patents, with some accepted into MPEG-4 and AVS standards. He is an associate editor for six journals including *IEEE Transactions on Circuits and Systems for Video Technology, IEEE Transactions on Image Processing,* and *IEEE Transactions on Circuits and Systems–I: Fundamental Theory and Applications*. He is involved with many technical and steering committees. He is the chair of the IEEE International Conference on Multimedia and Expo 2010 and the general cochair of the 2010 Packet Video Workshop. He won the 2007 SiPS and PCM Best Paper Awards.

Man-Cheung Kung (mckung@ust.hk) received the B.Eng. degree in computer engineering and the M.Phil. degree in electronic and computer engineering from HKUST, in 2006 and 2008, respectively. From 2005 to 2006, he was with the Applied Science and Technology Research Institute Company Ltd., Shatin, Hong Kong. He is currently the research and development engineer of the Visual Perception Dynamics Labs (Mobile) Ltd., Shatin, Hong Kong. His research interests include general-purpose computation on GPUs, fast ME, intraprediction, image demosaicking, and subpixel rendering.

REFERENCES

[1] Y. Wang, J. Ostermann, and Y.-Q. Zhang, *Video Processing and Communications*. Englewood Cliffs, NJ: Prentice-Hall, 2002.

[2] T. Wiegand, "Joint final committee draft for joint video specification H.264," *Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG,* Tech. Rep. JVT-D157, 2002.

[3] AVS Video Expert Group, Information Technology—Advanced Audio Video Coding Standard Part 2: Video, in Audio Video Coding Standard Group of China (AVS), Doc. AVS-N1063, Dec. 2003.

[4] L. Yu, F. Yi, J. Dong, and C. Zhang, "Overview of AVS-Video: Tools, performance and complexity," in *Proc. Visual Communications and Image Processing (VCIP)*, 2005, pp. 679–690.

[5] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 560–576, July 2003.

[6] G. Sullivan, J.-R. Ohm, A. Ortega, E. Delp, A. Vetro, and M. Barni, "dsp Forum—Future of video coding and transmission," *IEEE Signal Processing Mag.*, vol. 23, no. 6, pp. 76–82, Nov. 2006.

[7] J. Loomis and M. Wasson. (2007). *VC-1 technical overview* [Online]. Available: http://www.microsoft.com/windows/windowsmedia/

[8] B. G. Haskell, A. Puri, and A. N. Netravali, *Digital Video: An Introduction to MPEG-2*. Berlin: Springer-Verlag, 1996.

[9] L. Fan, S. Ma, and F. Wu, "An overview of AVS video standard," in *Proc. IEEE Int. Conf. Multimedia and Expo*, 2004, pp. 423–426.

[10] G. Shen, G. Gao, S. Li, H. Shum, and Y. Zhang, "Accelerate video decoding with generic GPU," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 5, pp. 685–693, 2005.

[11] A. Mather, "GPU-accelerated video encoding," in *Proc. SIGGRAPH Tech Talks*, 2008.

[12] J. Kruger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," in *Proc. Int. Conf. Computer Graphics and Interactive Techniques*, Los Angeles, CA, 2005, p. 234.

[13] O. Fialka and M. Cadik, "FFT and convolution performance in image filtering on GPU," in *Proc. Conf. Information Visualization*, 2006, pp. 609–614.

[14] A. Brunton and J. Zhao, "Real-time video watermarking on programmable graphics hardware," in *Proc. Canadian Conf. Electrical and Computer Engineering*, 2005, pp. 1312–1315.

[15] NVIDIA, "NVIDIA GeForce 8800 architecture technical brief," NVIDIA, Tech. Rep., 2006.

[16] NVIDIA. (2009). *CUDA—Compute unified device architecture* [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[17] Khronos Group. (2009). OpenCL—The open standard for parallel programming of heterogeneous systems [Online]. Available: http://www.khronos.org/opencl/

[18] M. Macedonia, "The GPU enters computing's mainstream," *Computer,* vol. 36, no. 10, pp. 106–108, Oct. 2003.

[19] F. Kelly and A. Kokaram, "General purpose graphics hardware for accelerating motion estimation," in *Proc. Irish Machine Vision and Image Processing Conf.*, 2003.

[20] Y. Lin, P. Li, C. Chang, C. Wu, Y. Tsao, and S. Chien, "Multi-pass algorithm of motion estimation in video encoding for generic GPU," in *Proc. IEEE Int. Symp. Circuits and Systems*, 2006.

[21] C.-W. Ho, O. Au, G. Chan, S.-K. Yip, and H.-M. Wong, "Motion estimation for H.264/AVC using programmable graphics hardware," in *Proc. IEEE Int. Conf. Multimedia and Expo*, 2006, pp. 2049–2052.

[22] M. Kung, O. Au, P. Wong, and C. Liu, "Block based parallel motion estimation using programmable graphics hardware," in *Proc. Int. Conf. Audio, Language and Image Processing*, 2008, pp. 599–603.

[23] M. L. Schmit, R. M. Rawther, and R. Giduthuri, "Software video encoder with GPU acceleration," U.S. Patent 20 090 016 430, 2009.

[24] G. Jin and H.-J. Lee, "A parallel and pipelined execution of H.264/AVC intra prediction," in *Proc. IEEE Int. Conf. Computer and Information Technology*, 2006, p. 246.

[25] W. Lee, S. Lee, and J. Kim, "Pipelined intra prediction using shuffled encoding order for H.264/AVC," in *Proc. IEEE Region 10 Conf. (TENCON)*, 2006, pp. 1–4.

[26] M. Kung, O. Au, P. Wong, and C. Liu, "Intra frame encoding using programmable graphics hardware," in *Proc. Pacific Rim Conf. Multimedia (PCM)*, 2007, pp. 609–618.

[27] N.-M. Cheung, O. Au, M. Kung, H. Wong, and C. Liu, "Highly parallel rate-distortion optimized intra mode decision on multi-core graphics processors," *IEEE Trans. Circuits Syst. Video Technol. (Special Issue on Algorithm/Architecture Co-Exploration of Visual Computing)*, vol. 19, no. 11, pp. 1692–1703, Nov. 2009.

[28] A. Obukhov and A. Kharlamov, "Discrete cosine transform for 8×8 blocks with CUDA," NVIDIA, Santa Clara, CA, Tech. Rep., Oct. 2008.

[29] B. Pieters, D. Van Rijsselbergen, W. De Neve, and R. Van de Walle, "Motion compensation and reconstruction of H.264/AVC video bitstreams using the GPU," in *WIAMIS '07: Proc. 8th Int. Workshop Image Analysis for Multimedia Interactive Services*. Washington, DC: IEEE Comput. Soc., 2007, p. 69.

[30] A. Hirvonen and T. Leppanen, "H.263 video decoding on programmable graphics hardware," in *Proc. 5th IEEE Int. Symp. Signal Processing and Information Technology*, Dec. 2005, pp. 902–907.

[31] B. Han and B. Zhou, "Efficient video decoding on GPUs by point based rendering," in *GH '06: Proc. 21st ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware*. New York, NY: ACM, 2006, pp. 79–86.

[32] NVIDIA. (2009). *PureVideo overview* [Online]. Available: http://www.nvidia.com/object/purevideo_overview.html

[33] J. Munkberg, P. Clarberg, J. Hasselgren, and T. Akenine-Moller, "High dynamic range texture compression for graphics hardware," *ACM Trans. Graph.*, vol. 25, no. 3, 2006, pp. 698–706.

[34] Y.-K. Chen, W. Li, J. Li, and T. Wang, "Novel parallel Hough transform on multi-core processors," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, 2008, pp. 1457–1460.

[35] J. Fung, S. Mann, and C. Aimone, "OpenVIDIA: Parallel GPU computer vision," in *Proc. ACM Multimedia*, 2005, pp. 849–852.

[36] P. M. Novotny, J. A. Stoll, N. V. Vasilyev, P. J. del Nido, P. E. Dupont, T. E. Zickler, and R. D. Howe, "GPU based real-time instrument tracking with three-dimensional ultrasound," *Med. Image Anal. (Special Issue on the 9th Int. Conf. on Medical Image Computing and Computer-Assisted Interventions—MICCAI 2006)*, vol. 11, no. 5, pp. 458–464, 2007.

[37] D. Lin, V. Huang, Q. Nguyen, J. Blackburn, C. Rodrigues, T. Huang, M. N. Do, S. J. Patel, and W.-M. W. Hwu, "The parallelization of video processing," *IEEE Signal Processing Mag.*, vol. 26, no. 6, pp. 103–112, Nov. 2009.

[38] D. Kirk and W. Hwu, "Textbook for UIUC ECE 498 AL: Programming massively parallel processors," *Draft*, 2009.

[39] NVIDIA, "CUDA programming guide," NVIDIA, Tech. Rep., 2009.

[40] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. Boston: McGraw-Hill, 2005.

[41] M. Houston. (2007). *SIGGRAPH 2007 GPGPU course* [Online]. Available: http://gpgpu.org/s2007

[42] X. Yi, J. Zhang, N. Ling, and W. Shang, "Improved and simplified fast motion estimation for JM," in *Proc. JVT Meeting, Poznan, Poland,* Tech. Rep. JVT-P021, July 2005.

[43] C.-L. Yang, L.-M. Po, and W.-H. Lam, "A fast H.264 intra prediction algorithm using macroblock properties," in *Proc. IEEE Int. Conf. Image Processing (ICIP)*, 2004, pp. 461–464.

[44] R. Su, G. Liu, and T. Zhang, "Fast mode decision algorithm for intra prediction in H.264/AVC," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, 2006.

[45] M. Liu and Z.-Q. Wei, "A fast mode decision algorithm for intra prediction in AVS-M video coding," in *Proc. Int. Conf. Wavelet Analysis and Pattern Recognition*, Nov. 2007, pp. 326–331.

[46] K.-W. Yoo and H.-H. Kim, "Intra prediction method and apparatus," U.S. Patent 2005/0 089 094, 2005.

[47] I. E. Richardson, *H.264 and MPEG-4 Video Compression, Video Coding for Next-Generation Multimedia*. England: Wiley, Nov. 2003.

[48] T. Wiegand and B. Girod, "Lagrange multiplier selection in hybrid video coder control," in *Proc. IEEE Int. Conf. Image Processing (ICIP)*, 2001, pp. 542–545.

[49] Advanced Micro Devices Inc. (2006). *ATI CTM guide* [Online]. Available: http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[50] Advanced Micro Devices Inc. (2009). *ATI stream software development kit (SDK)* [Online]. Available: http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx

[51] G. Bjontegaard, "Calculation of average PSNR differences between RD-curves," in *Proc. JVT Meeting,* Tech. Rep. VCEG-M33, Apr. 2001.

[52] OpenMP. (2009). *The OpenMP API specification for parallel programming* [Online]. Available: http://openmp.org/

[53] A. Vetro, C. Christopoulos, and H. Sun, "Video transcoding architectures and techniques: An overview," *IEEE Signal Processing Mag.*, vol. 20, no. 2, pp. 18–29, Mar. 2003.

[54] H. Sun, X. Chen, and T. Chiang, *Digital Video Transcoding for Transmission and Storage*. New York: CRC, 2005.

[55] I. Ahmad, X. Wei, Y. Sun, and Y.-Q. Zhang, "Video transcoding: An overview of various techniques and research issues," *IEEE Trans. Multimedia*, vol. 7, no. 5, pp. 793–804, Oct. 2005.

[56] Techreport. (2009). *Badaboom 1.0 uses Nvidia GPUs to transcode video* [Online]. Available: http://techreport.com/discussions.x/15763

[57] AnandTech. (2009). *AVIVO video converter Redux and ATI stream quick look* [Online]. Available: http://www.anandtech.com/video/showdoc.aspx?i=3578