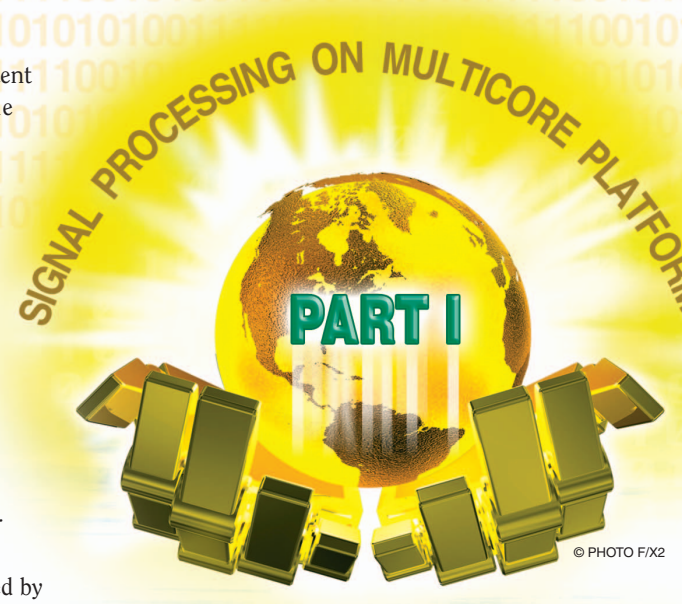
Dennis Lin, Xiaohuang (Victor) Huang,
Quang Nguyen, Joshua Blackburn, Christopher Rodrigues,
Thomas Huang, Minh N. Do, Sanjay J. Patel, and Wen-Mei W. Hwu

The Parallelization of Video Processing

From programming models to applications



The explosive growth of digital video content from commodity devices and on the Internet has precipitated a renewed interest in video processing technology, which broadly encompasses the compression, enhancement, analysis, and synthesis of digital video. Video processing is computationally intensive and often has accompanying real-time or super-real-time requirements. For example, surveillance and monitoring systems need to robustly analyze video from multiple cameras in real time to automatically detect and signal unusual events. Beyond today's known applications, the continued growth of functionality and speed of video processing systems will likely further enable novel applications.

Due to the strong computational locality exhibited by video algorithms, video processing is highly amenable to parallel processing. Video tends to exhibit high degrees of locality in time: what appears on the tenth frame of a video sequence does not strongly affect the contents of the 1,000th frame, and in space: an object on the left side of single frame does not strongly influence the pixel values of on the right. Such locality makes it possible to divide video processing tasks into smaller, weakly interacting pieces amenable to parallel processing. Furthermore, these pieces can share data to economize on memory bandwidth.

This article is based on our experiences in the research and development of massively parallel architectures and programming technology, in construction of parallel video processing components, and in development of video processing applications. We describe several program transformations necessary to realize the performance benefits of today's multi- and many-core architectures on video processing. We describe program optimizations using three-dimensional (3-D) convolution as a pedagogical example. We compare the relative importance of these transformations on multicore CPUs versus many-core graphics processing units (GPUs). In addition, we relate our efforts in accelerating applications in major areas of video processing using many-core GPUs.

Digital Object Identifier 10.1109/MSP.2009.934116

MULTICORE AND MANY-CORE TECHNOLOGIES

The semiconductor industry has shifted from increasing clock speeds to a strategy of growth through increasing core counts.

Dual-core and quad-core CPU systems are now commonplace. For example, Intel's Core i7 processor has four cores, and a high-end configuration has a peak computation rate of 140 single precision giga floating-point operations per second (GFLOPS) and a peak off-chip memory bandwidth of 32 GB/s. This change presents a major challenge to application developers who must now work to expose sufficient parallelism in performance-sensitive applications. In the extreme, many-core systems such as the programmable GPU platforms developed by NVIDIA, AMD, and Intel, feature more than tens of cores. For developers that can find sufficient parallelism, these supply dramatic gains in computing power that can potentially revolutionize what is possible within a particular domain. For example, the NVIDIA Tesla S1070 contains four GT200 processors, each of which provides about one teraflop (TFLOP) of single-precision computational throughput and over 100 GB/s of memory bandwidth.

THE SEMICONDUCTOR INDUSTRY HAS SHIFTED FROM INCREASING CLOCK SPEEDS TO A STRATEGY OF GROWTH THROUGH INCREASING CORE COUNTS.

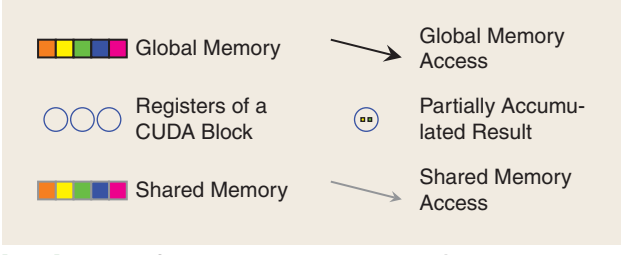
PARALLEL PROGRAMMING MODELS

Mapping a set of algorithms onto a multi- or many-core platform requires the use of a parallel programming model, which describes and controls the concurrency, communication, and synchronization of the parallel components within the application. Parallel programming models and development tools have evolved alongside multicore and many-core architectures. Such models are typically provided as extensions of existing languages, such as through application programming interfaces (APIs) added to C/C++, rather than as entirely new parallel programming languages. Rather than provide a comprehensive list of all programming models, we discuss the most popular and best supported. We separate these models into three categories: disjoint, task based, and thread based.

Disjoint models have the property that concurrent entities are separated by different memory address spaces. Models such as the message passing interface (MPI) and unified parallel C (UPC) are useful for large-scale distributed systems with separate address spaces. These are usually not found on commodity hardware and may not be of general interest to video processing researchers.

On smaller systems where all the cores have access to a common address space, programmers often develop applications using the threading support of the underlying operating system. These include POSIX threads and the Windows thread interface. Because all the threads can access each other's data, the threading libraries provide sophisticated synchronization and locking mechanisms. This model is the least restrictive of the three discussed, and its expressiveness permits applications that are irregularly parallel to be parallelized. However, the correct use of these synchronization and locking mechanisms have proven to be challenging for even the most experienced parallel programmers and thus will unlikely be used by the majority of video processing application developers.

Task-based models, such as Intel's Threaded Building Blocks (TBB) and NVIDIA's Compute Unified Device Architecture (CUDA), represent the finest grain of parallelism. Other examples of this model include OpenMP, Cilk, and OpenCL. These trade the flexibility of general threading for increased ease of scheduling. Under these models, the programmer divides the application into small independent tasks, which are enqueued into the task system. The task system then performs the scheduling, satisfying dependencies and maximizing throughput. Task systems that run on central processing units (CPUs), such as TBB, are generally more dynamic, allowing the spontaneous creation of new tasks. Task systems designed for execution on GPUs, on the other hand, can perform well given even very small tasks—those on the order of a few hundreds of instructions—at the cost of less dynamic task generation and termination support.



[FIG1] Legend for the algorithm diagrams (shown in Figures 2–4). Global memory is the off-chip (but on the GPU card) memory. Each global memory location holds a different video pixel value, represented by different colors. The registers are a per-task resource. In our examples, they are used to accumulate the partial sums of convolutions, indicated by the small boxes. Shared memory is a small on-chip scratchpad memory.

CUDA AND MCUDA

In this article, we focus on the task-based model CUDA, which is by far the most mature of the GPU-based programming models. NVIDIA designed this extension of the C programming language to support parallel processing on its GPUs. Since CUDA is geared towards fine-grained parallelism, it often works well for the highly data parallel applications we often find in video processing. Also, using Multicore CUDA (MCUDA) [1)], developers can use the CUDA programming model on multi-core CPUs. As such, it serves as a reasonable basis for the comparisons between optimizations on multicore CPUs and many-core GPUs discussed in this article.

A fundamental building block of CUDA programs is the CUDA kernel function. When launching a CUDA kernel function, a developer specifies how many copies of it to run. We call each of these copies a task. Because of the hardware support of the GPU, each of these tasks can be small, and the developer can queue hundreds of thousands of them for execution at once.

These tasks are organized in a two-level hierarchy, block and grid. Small sets of tightly coupled tasks are grouped into blocks. In a given execution of a CUDA kernel function, all blocks contain the same number of tasks. The tasks in a block run concurrently and can easily communicate with each other, which enables useful optimizations such as those of the section "Shared Memory." A GPU's hardware keeps multiple blocks in flight at once, with no guarantees about their relative execution order. As a result, synchronization between blocks is difficult. The set of all blocks run during the execution of a CUDA kernel function is called a grid.

IT APPEARS THAT THE CPU ARCHITECTURE IS BETTER SUITED TO THE RELATIVELY RANDOM MEMORY ACCESSES OF AN FFT WORKLOAD.

A CUDA-enabled GPU has enough resources to keep many tasks (more than 10,000) active on the chip and to switch between these tasks without time penalty. At each clock cycle, the GPU scheduling hardware identifies the tasks whose next instructions are not waiting for a long-latency instruction such as global memory access and switches to these tasks for execution. This ability to swiftly switch among many tasks allows the GPU to hide the latency of slow instructions.

The CUDA tasks, by themselves, are usually too fine grained to map well to multicore CPUs. However, CUDA blocks, which contains tens to hundreds of tasks, are closer to the size of the tasks found in CPU-based task systems. Thus, MCUDA uses a CUDA block as its fundamental unit of execution. It does this by transforming the CUDA block into a large CPU loop, applying code transformations to preserve the semantics of local variables and synchronization. Since CUDA has very weak guarantees about synchronization between blocks, MCUDA can then run the different blocks of a grid on different CPUs without worrying about race conditions. MCUDA supports OpenMP and Pthreads as underlying threading systems.

The generated MCUDA code is relatively efficient. The original CUDA task probably fits well in the instruction cache of the CPU, leading to a tight inner loop. The CPU's ability to execute instructions out of order and support for single instruction, multiple data (SIMD) instructions helps to make up for the absence of specialized task scheduling hardware. The optimizations in the next section that deal with reducing memory bandwidth usage can also map to CPU code.

CASE STUDY: 3-D CONVOLUTION

In this section, we provide archetypal examples of optimizations that a programmer should consider while parallelizing a video processing application. We choose the 3-D convolution as our sample application component. This is a common signal processing computation, given by

$$(\mathrm{V} * \mathrm{K})[x, y, t] = \sum_m \sum_n \sum_l \mathrm{V}[x-m, y-n, t-l]\mathrm{K}[m, n, l]. \quad (1)$$

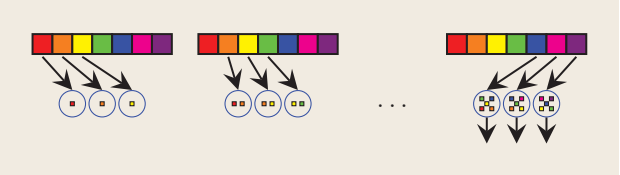
In the case where K is separable, that is, when K may be written in the form of $\mathrm{K}[x, y, t] = f[x]g[y]h[t]$, the problem may be reduced to a series of one-dimensional (1-D) convolutions. However, we will focus on the more general non-separable case as it is more taxing on the hardware. Similarly, we choose the 3-D version of the convolution task because the amount of work increases quickly with the size of the convolution kernel. We consider convolution kernels of size $k \times k \times k$, for $5 \le k \le 15$. Our results generalize to rectangular kernels of comparable sizes.

Convolution is a local operation, with each output value a function of a region of nearby input values. As such, it serves as a model for other more sophisticated scanning window algorithms. As a simple algorithm with little computation per pixel, it is an extreme example of the pressures put on the memory subsystem. The remainder of this section starts with a straightforward implementation of (1) and applies optimizations that improve memory utilization. For completeness, we also consider the use of Fourier transforms in the section "Fourier."

REFERENCE IMPLEMENTATION

The first step in parallelizing the application is to determine how to decompose it into tasks. For maximum parallelization, one might consider assigning a block of tasks to compute one output value. However, even with large k, there is not enough work to overcome the overhead of managing these tasks. Therefore, we begin by assigning a single task to compute one output value.

Figure 2 illustrates this BASELINE algorithm, a straightforward implementation of (1). It depicts a one-dimensional convolution for clarity; the overall logic remains the same for three dimensions. Each task has a register, depicted as a blue circle, in which it accumulates partial results. As the computation runs, the task loops over the elements of the convolution kernel. For each convolution kernel element, it reads the corresponding video pixel value, multiplies the two together, and accumulates the product in its register. (The task does not have to read the convolution kernel element from global memory—that value is presumed to be stored in a special cacheable, read-only region called constant memory.) Figure 2 shows snapshots of the first, second, and last loop iterations of three tasks. Note that the same video pixel value can be read multiple times, each time by a different task. After looping through all five values of the convolution kernel, each task writes its output to the appropriate location and terminates.



[FIG2] An illustration of the BASELINE algorithm. (See Figure 1 for a legend.) This simplified diagram shows the convolution of a 1-D kernel of length five against a video of length seven, producing three output values. Time flows from left to right.

This is a very straightforward implementation of our application; the CUDA code is about two dozen lines long. The MCUDA translation of this code is also roughly equivalent to a naïve implementation with nested loops. Their performance may be found in Table 1 and Table 2 in the section "Timing Results," and will serve as the reference point for our improvements.

THE PURPOSE OF THE FEATURE EXTRACTION COMPONENT IS TO TAKE WINDOWS FROM THE VIDEO AND CALCULATE A CHARACTERISTIC SIGNATURE THAT CAN BE USED FOR ANALYSIS.

SHARED MEMORY

Although a GPU can tolerate memory latency by interleaving execution of many tasks, the BASELINE algorithm runs into memory bandwidth limitations by performing too many redundant accesses. To produce one output value for a size-k convolution, the algorithm loads k^3 video pixels, performing only one multiply and add per load. That is, it performs $2k^3$ mathematical operations and k^3 memory reads, giving a ratio of compute instructions to memory accesses is two. A quick calculation shows that this is too low: a GT200 processor in an NVIDIA Tesla S1070 has a peak performance of about 1 TFLOPS, while its peak memory bandwidth is about 100 GB/s; to prevent memory bandwidth from capping performance, it must perform at least 40 arithmetic operations for every 4-B floating-point value that is loaded or stored.

The SHARED MEMORY algorithm, shown in Figure 3, improves upon the baseline's bandwidth consumption by amortizing the cost of loading image pixels. In this algorithm, each CUDA block preloads an image tile, consisting of all pixels needed to compute a $4 \times 4 \times 4$ tile of the output image, into a programmer-managed scratchpad area known as shared memory. It then runs the same loop that was used for the BASELINE algorithm, reading from shared memory instead of global memory. Shared memory, which is located on the GPU chip, has lower latency and much higher bandwidth than global memory. As a result, shifting the majority of data accesses to shared memory results in large speed improvements, as shown in Table 1.

Note that the number of pixels that need to be preloaded into the shared memory is $(k + o - 1)^3$, where k is the dimension size of the convolution kernel and o is that of the output tile calculated by each block. Also, to compute an $o \times o \times o$ block of output, it must perform o^3k^3 multiplications and o^3k^3 additions, for a total of $2o^3k^3$ mathematical operations. So, as an example, when $k = 9$ and $o = 4$, the compute-to-memory ratio is 54. The ratio improves as k and o increases. Unfortunately, all currently available CUDA hardware can provide a maximum of 16K of share memory per block, and we need to use small amounts of it for argument passing and other operations. This means, that with $o = 4$, we cannot support cases where $k \geq 13$.

A similar analysis may be applied to the MCUDA implementation of the algorithm. However, the CPU's L1 cache can hold all of input data elements when a small convolution kernel is used. As a result, any algorithm will work well. The SHARED MEMORY algorithm shows more improvement as k increases, with a 2.4 times speedup for $k = 15$.

Since our example is extremely light in computation, there are possibilities for further optimization, as shown in the section "Streaming." However, more compute-intensive algorithms such as adaptive filtering (similar to those in the section "Enhancement Example: Spatial Interpolation") or bilateral

[TABLE 1] TIMING RESULTS ON A SINGLE GPU OF AN NVIDIA TESLA S1070 FOR DIFFERENT CONVOLUTION ALGORITHMS. THE NUMBERS GIVEN ARE THE MILLISECONDS REQUIRED TO COMPUTE FOUR FRAMES OF A SIZE-*k* CONVOLUTION ON A 720 × 560 VIDEO SEQUENCE.

k	BASELINE	SHARED MEMORY	STREAMING	3-D FOURIER	HYBRID FOURIER
5	16	11	4	24	15
7	44	15	8	34	17
9	96	48	16	39	20
11	180	77	27	44	23
13	295		45	74	24
15	454		75	56	26

[TABLE 2] TIMING RESULTS USING A DUAL SOCKET DUAL CORE 2.4 GHZ OPTERON FOR DIFFERENT CONVOLUTION ALGORITHMS. THE NUMBERS GIVEN ARE THE MILLISECONDS REQUIRED TO COMPUTE FOUR FRAMES OF A SIZE-*k* CONVOLUTION ON A 720 × 560 VIDEO SEQUENCE.

k	BASELINE	SHARED MEMORY	STREAMING	3-D FOURIER	HYBRID FOURIER
5	136	117	140	128	133
7	362	289	317	235	152
9	1,018	597	614	208	213
11	1,954	1,065	1,135	238	237
13	3,590	1,733	1,771	267	271
15	6,453	2,676	2,633	338	356

filtering (discussed in the section "Synthesis Example: Depth Image-Based Rendering") may have enough work to need no further memory optimization. Furthermore, one key advantage of this implementation is that the entire (x, y, t) window of the video needed to produce the output is available in shared memory at the same time. This should make implementing these sophisticated algorithms much easier.

IN THIS ARTICLE, WE FOCUS ON THE TASK-BASED MODEL CUDA, WHICH IS BY FAR THE MOST MATURE OF THE GPU-BASED PROGRAMMING MODELS.

STREAMING

The limited size of shared memory on a Tesla S1070 GPU restricts the kernel size that can be processed with the SHARED MEMORY algorithm. Applying an optimization similar to loop skewing changes the kernel's memory accesses to a STREAMING pattern that uses less image data at any given time, overcoming this limit. Note that in Figure 3, SHARED MEMORY tasks have similar access patterns modulo a time shift: the first task accesses image pixels at positions $1, 2, 3, \ldots$ while the second task accesses pixels $2, 3, 4, \ldots$, and so forth. By delaying the nth task by n time steps, we arrange for all tasks to read the same value at the same time. Instead of preloading a shared array, the necessary value is loaded in every step, as depicted in the first three steps of Figure 4.

Rather than processing a small block of data, our 1-D STREAMING algorithm processes the entire image. After a few initial steps, the algorithm reaches a steady state where there are exactly k running tasks, one of which completes and writes an output value in each iteration. Instead of terminating, the task resets its accumulator to 0 and begins to compute a new output. The steady state is depicted in the last two steps of Figure 4.

Extending to the 3-D case, the STREAMING algorithm uses one less dimension of shared memory compared to SHARED MEMORY. For an $o \times o$ output tile, every step of computation reads in a two-dimensional (2-D) slice of video $[(k + o - 1)^2$ pixels$]$, accumulates partial results for a 3-D volume $(o \times k \times o)$ of output, and writes out a completed 2-D slice $(o \times o)$ of output. Each CUDA block generates a $o \times n \times o$ sliver of output, where n is the height of the video. We further tune the performance of the STREAMING algorithm by using a non-square 16×1 output tile, for which GPU hardware more efficiently coalesces outputs into vector stores to the global memory.

Because the STREAMING algorithm works along the entire length of one of the dimensions, it achieves a higher compute-to-memory ratio. During each step, the algorithm reads $(k + 15)k$ values. It also writes 16 output values, each of which represents $2k^3$ mathematical operations. Thus, for large n, the compute-to-memory ratio approaches $32k^2/k + 15$. For $k = 9$, this is 108, twice that of the SHARED MEMORY algorithm. This allows STREAMING to run approximately twice as fast, as shown in Table 1.

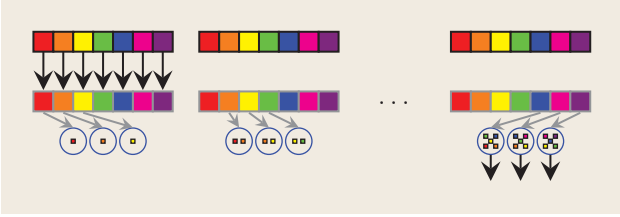
While this algorithm illustrates what can be done to increase throughput, it is probably of interest only to those specifically seeking maximum performance of the convolution task on GPUs. With this method, an individual task no longer sees the entire window of input data elements at once. Instead, it has to accumulate the result as the input values are streamed in. While this is a straightforward process for a convolution kernel, it is likely to pose challenges for other algorithms. Also, as shown in Table 2, this method grants no additional advantage on CPUs, whose hardware caches can discover the locality in the SHARED MEMORY algorithm.

FOURIER

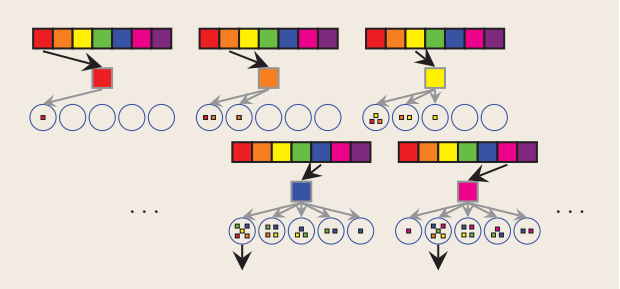
All of the previous methods involve directly implementing (1), but there is a well-known shortcut that involves moving to the Fourier domain. It can be shown that

$$\mathrm{V} * \mathrm{K} = \mathcal{F}^{-1}(\mathcal{F}(\mathrm{V}) \times \mathcal{F}(\mathrm{K})), \tag{2}$$

where $\mathcal{F}$, $\mathcal{F}^{-1}$, and $\times$ are the Fourier transform, the inverse Fourier transform, and pointwise multiplication, respectively. The straight multiplication method requires $O(n_\mathrm{K} n_\mathrm{V})$ operations, where n_K is the number of pixels in K and n_V is



[FIG3] The SHARED MEMORY algorithm. (See Figure 1 for a legend.) This diagram is similar to Figure 2 except for the addition of shared memory. After loading the video pixel values from global memory, all reads are satisfied from shared memory. Otherwise, the algorithm is identical to the BASELINE algorithm.



[FIG4] The STREAMING algorithm. (See Figure 1 for a legend.) Whereas the SHARED MEMORY algorithm loaded all the necessary data at the beginning, the STREAMING algorithm performs the loads incrementally. At each step, we only load a single value of the video. We then distribute that value to all tasks that can use it; ignoring the boundary conditions, this means multiplying that value with five kernel values and accumulating the results in different tasks.

the number of pixels in V. Fast Fourier transform (FFT) algorithms perform (2) in $O(n_V \log n_V)$ steps. This implies a win when $n_K > \log n_V$. Since K is 3-D, n_K grows with the cube of k, and the Fourier technique should quickly win over straight multiplication.

AN IMPORTANT TASK IN DIBR IS DEPTH-MAP GENERATION, WHICH IS USUALLY COMPUTATIONALLY INTENSIVE.

The effect of implementing (2) with 3-D Fourier transforms depends heavily on the performance of the underlying Fourier transform implementation. On an NVIDIA GPU, using the vendor-supplied CUDA FFT (CUFFT) library often results in a slowdown compared to the STREAMING direct multiplication technique, as shown in Table 1. CUFFT makes multiple passes through the data, and as a result, this method takes more time even though it is performing fewer operations. The Fourier technique is much more successful on the CPU, as shown in the last two columns of Table 2. These results were generated using the Fastest Fourier Transform in the West (FFTW) library, and they are almost always faster than the multicore CUDA (MCUDA)-generated code. FFTW is much more heavily tuned than MCUDA, extensively utilizing locality optimizations and making effective use of streaming SIMD extensions (SSE). Also, it appears that the CPU architecture is better suited to the relatively random memory accesses of an FFT workload, allowing the algorithm that is more computationally efficient to actually attain higher performance. On either platform, this method has the disadvantage of requiring all the frames to be resident in memory at the same time to perform the transform.

The issues with 3-D Fourier transforms on the GPU can be partially alleviated by transforming each frame individually. The HYBRID FOURIER algorithm performs a 2-D transform on each frame and then a (complex) 1-D convolution in the time direction. This utilizes the ability to separate a 3-D Fourier transform into three 1-D Fourier transforms, one in each dimension. On the GPU, this method scales better than the 3-D FOURIER technique, and begins to win out over STREAMING for kernels larger than size 11. On the CPU, this approach runs at about the same speed as the direct 3-D FOURIER method. However, since it treats the frames separately, it is easier to process long video sequences where it is not possible to squeeze all the frames in memory.

[TABLE 3] TIMING COMPARISONS OF DIFFERENT IMPLEMENTATIONS OF A VIDEO ANALYSIS PIPELINE. THE TIMES ARE THE NUMBER OF MILLISECONDS NECESSARY TO ANALYZE ONE 720 × 560 FRAME OF VIDEO. THE CPU TIMES ARE FROM SINGLE CORE OF A DUAL CORE 2.4 GHZ OPTERON. THE GPU TIMES ARE FROM ONE GT200 OF AN NVIDIA TESLA S1070. THE CPU VERSION OF THE FEATURE EXTRACTION STEP CONSISTS OF HIGHLY UNOPTIMIZED LEGACY CODE.

	FETCH FRAME	OPTICAL FLOW	CPU TO GPU TRANSFER	FEATURE EXTRACTION	PAIRWISE DISTANCE
CPU	59	90	N/A	2,993	5,036
GPU			14	249	176

TIMING RESULTS

In Table 1, we record timings on an NVIDIA Tesla S1070 with a GT200 processor. We used CUDA for the BASELINE, SHARED MEMORY, and STREAMING algorithms, and we used NVIDIA's CUFFT for 3-D FOURIER and HYBRID FOURIER. In Table 2, we show the timings for the same algorithm on a dual socket dual core 2.4 GHz Opteron. For the BASELINE, SHARED MEMORY, and STREAMING algorithms, we used MCUDA to translate the CUDA code to C++. For 3-D FOURIER and HYBRID FOURIER, we relied on the FFTW library. In all cases, we operate on $k + 3$ frames of 720×560 video to produce exactly four frames of output. The video is preloaded in (GPU) memory to avoid timing the overhead, and the numbers given are the best of five trials.

EXAMPLES OF PARALLELIZATION OF VIDEO PROCESSING

We now move from focusing on one single component to descriptions of more holistic applications. This section provides an overview of some of our work in the analysis, enhancement, and synthesis of video.

ANALYSIS EXAMPLE: VIDEO EVENT DETECTION

As the hardware becomes more affordable and mass deployment of surveillance cameras becomes more common, we see increasing interest in automated methods for human activity recognition. Still, the flood of video data threatens to overwhelm even automatic systems. The TREC Video (TRECVID) Event Detection Evaluation sponsored by the National Institute of Standards and Technology illustrates the scope of the problem. In 2008, the evaluation set consisted of 50 h of data. Our sequential algorithm [2] required about 30 s/frame on a single CPU core. This meant that even running on a 16-node, 64-core cluster and processing only five frames from each second of video, we needed five days to complete the computations. This long latency posed a logistical challenge that limited our ability to experiment and tune our algorithm, and it degraded our results.

After we submitted the results for 2008, we began laying the framework for the 2009 evaluation. During the refactoring, we decomposed our video event detection algorithm into several discrete steps, as shown in Table 3. At the same time, we optimized the CPU version of the pairwise distance computation, and the sequential CPU version now runs in 8 s/frame. Profiling indicated that the feature extraction and the pairwise distance computation combined represent 98% of the total computation time, so we decided to shift these portions to the GPU.

The purpose of the feature extraction component is to take windows from the video and calculate a characteristic signature that can be used for analysis. The CPU feature extraction code is a legacy of the original system, and it is somewhat poorly optimized. The GPU code is simple port of the CPU version, with

one CUDA block assigned for each feature. Unfortunately, we do not take advantage of the fact that the windows overlap to reduce memory bandwidth. We believe that with the proper effort, both the CPU and GPU versions can see an order of magnitude increase in performance. However, we have found that the feature detector performs poorly in overall detector accuracy. As a result, we intend to replace this code with component with the biologically inspired features of [3]. These features consists of banks of Gabor filters, each of which is a 3-D convolution. It was partially the desire to implement these features that inspired the analysis of our case study.

AS THE HARDWARE BECOMES MORE AFFORDABLE AND MASS DEPLOYMENT OF SURVEILLANCE CAMERAS BECOMES MORE COMMON, WE SEE INCREASING INTEREST IN AUTOMATED METHODS FOR HUMAN ACTIVITY RECOGNITION.

We use the pairwise distance component to decide if the features correspond to the actions we are looking for. This operation involves comparing the m feature vectors (each of size l) we extracted from the video with n features from our database. Given a matrix $\mathbf{F}_{m\times l}$ of extracted features and a matrix $\mathbf{D}_{n\times l}$ of database features, we wish to compute a matrix $\mathbf{P}_{m\times n}$ where

$$\mathbf{P}[i][j] = \sum_{k=1}^{l} (\mathbf{F}[i][k] - \mathbf{D}[j][k])^2. \tag{3}$$

In parallelizing this component, we ran into the same memory bandwidth issue as discussed in the section "Case Study: 3-D Convolution." The naïve implementation performs one subtraction, one multiplication, and one addition for two memory reads, leading to a compute-to-memory ratio of 1.5. To achieve higher performance, we use loop tiling. The insight here is in reducing the problem so that it fits in shared memory. If $m = n = l = 16$, we can easily load $\mathbf{F}$ and $\mathbf{D}$ into shared memory. We would perform $2\times16\times16$ global loads and perform $3\times16\times16\times16$ operations. This would give us a compute-to-memory ratio of 24, a 16-fold improvement. Since m and n are larger, we divide the problem into 16×16 chunks, and assign a separate CUDA block to each chunk. For $l > 16$, we can loop over l in each task, loading 16 values at once. This gives us an order of magnitude improvement over the original code.

Overall, we were able to obtain a 13x speedup in the computation time. Using a 64-GPU cluster, we can evaluate our 2008 TRECVID algorithm in super-realtime speeds, processing 50 h of video in three hours. More importantly, we have divided the video analysis task into self-contained, reusable modules. We are collecting these into the Vision for Video (ViVid) [4] toolbox. In addition, we wish to provide a coherent interface to other video and image processing libraries. We have already incorporated the GPU-based optic flow library (OFLib) [5]. Switching to OFLib will reduce the optic flow computation time to about 40 ms while increasing the quality of the output. We also would like to incorporate GPU-accelerated code [6] for the scale invariant feature transform (SIFT). We plan to include a more advanced learning algorithm such as the support vector machine (SVM). There is a GPU-based implementation [7] of SVM that can achieve 30–50 times for training and 120–150 times speedup over a sequential CPU. Finally, we would like to be able to integrate seamlessly with other GPU-based vision libraries like GpuCV and OpenVidia. In the end, we hope to assemble a fast and flexible system for video analysis to facilitate research in this growing field.

ENHANCEMENT EXAMPLE: SPATIAL INTERPOLATION

Video upconversion is a computationally demanding and increasingly relevant video processing application. Many videos, such as those recorded by Web cams or stored on YouTube, have inherently low spatial resolution. At the same time, most display devices are increasing in resolution, such as HDTVs and high-resolution computer monitors. Because video upconversion is computationally localized both in time and space, parallel implementations can produce significant performance improvements.

Most current implementations use either nearest neighbor or bilinear interpolation for each frame of the video. The advantage of these techniques is their speed. Omitting address calculations, nearest neighbor has no calculations and bilinear has approximately four calculations per output pixel. The disadvantages of these techniques are image artifacts, including aliasing and zig-zagging edges. On the other hand, bicubic B-spline interpolation provides better image quality at the expense of more computation [8].

Bicubic B-spline interpolation consists of two main components: a prefilter and an interpolation filter [9]. The prefilter calculates the B-spline coefficients needed in the interpolation. This filter can be decomposed into two single-tap infinite impulse response (IIR) filters in each direction. As each scanline is an independent filter, this can be parallelized with one thread per scanline.

The interpolation filter consists of a single separable finite impulse response (FIR) filter with a 4×4 kernel. As FIR filters can be implemented as 2-D convolutions, all of the results of the previous discussion are applicable. In particular, our implementation uses the SHARED MEMORY algorithm from the section "Shared Memory." This is faster than the BASELINE version, and the additional complexity involved the STREAMING algorithm is unwarranted in the 2-D case with small kernels. The additional challenge with this filter is that the impulse response is not shift invariant. If the upconversion rate u is an integer, the impulse response is periodically shift invariant with period u. Because of the small size of the kernel, each impulse response can be precomputed, stored in memory, and accessed appropriately.

We tested the performance of both the parallel and sequential algorithms by interpolating the test "Claire" video sequence from quarter common intermediate format (QCIF) (176×144) by a factor of seven, to $1{,}232\times1{,}008$.

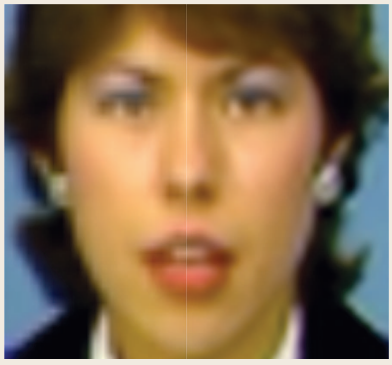
This is approximately the resolution of HDTV with black bars on the sides. Specific timing numbers are provided in Table 4. The parallel implementation demonstrates a speedup of approximately 350 times. This speedup includes gains both due to parallelization and due to optimizations. Back-of-the-envelope calculations suggest that a fully optimized CPU version would run in 250 ms. Therefore, we estimate that the GPU version runs about 50 times faster than a single core CPU for the FIR filter. More significantly, the GPU accelerated time of 5 ms is now fast enough for inclusion in a real-time video system. In addition, the speed is fast enough to further improve the quality of the algorithm while maintaining real-time computation. As an alternate comparison, Figure 5 compares the quality of the upconversion for bilinear and bicubic interpolation. The bicubic is much smoother, particularly along edges such as the jawline and lips.

Ruijters et al. perform bicubic interpolation using a concatenation of bilinear texture fetches [10]. This technique reduces the number of floating point operations by exploiting the dedicated bilinear filtering hardware attached to the GPU's texture memory. On the other hand, reading from texture memory has longer latencies on every memory read while shared memory algorithms only have long latencies on the original load from global memory to shared memory. These two considerations seem to offset for similar performance. Our algorithm can perform the FIR filter on a 256×256 image with an interpolation factor of two in approximately 800 frames/s. Ruijters' algorithm can perform the FIR filter on this image size at 846 frames/s [11].

VIDEO SYNTHESIS HAS AN INCREASINGLY WIDE RANGE OF APPLICATIONS, INCLUDING VIRTUAL VIEW SYNTHESIS FOR 3-D TELEVISION, FREE VIEWPOINT TV, ENTERTAINMENT, AND REMOTE EDUCATION.

[TABLE 4] COMPARISON OF SEQUENTIAL AND PARALLEL IMPLEMENTATIONS FOR ONE FRAME OF CUBIC B-SPLINE INTERPOLATION FROM 176×144 (QCIF) TO 1232×1008 (NEARLY HDTV). ALL TIMES ARE IN MILLISECONDS.

	HARDWARE	IIR TIME	FIR TIME
CPU	INTEL PENTIUM D 2.80 GHZ	5	1,689
GPU	NVIDIA GEFORCE 8800 GTX 1.35 GHZ	1	4



(a) (b)

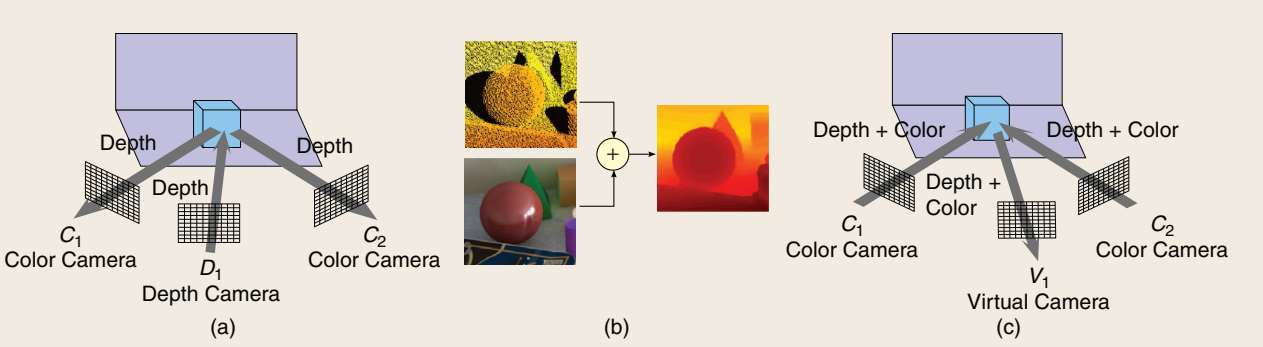
[FIG5] Quality comparison of bilinear (a) and bicubic (b) B-spline interpolation. Note the jagged edges along the jawline and lips when using bilinear interpolation.

SYNTHESIS EXAMPLE: DEPTH IMAGE-BASED RENDERING

Video synthesis has an increasingly wide range of applications, including virtual view synthesis for 3-D television (3-DTV), free viewpoint TV (FTV), entertainment, and remote education [12]. One recently popular approach is depth image-based rendering (DIBR), an emerging technology that synthesizes novel realistic images of a scene at virtual viewpoints, using a set of multiple images and per-pixel depth information. An important task in DIBR is depth-map generation, which is usually computationally intensive. Several efforts have investigated ways to parallelize and speed up the process. In the plane sweeping algorithm proposed in [13], GPU texture mapping functionality is used to project the input images onto a series of depth planes and then perform per-pixel comparisons. In another approach [14], views are synthesized by first applying the plane sweeping algorithm through 3-D space on the GPU to generate a noisy virtual view and a crude depth map. The result is then ameliorated by a CPU-based min-cut/max-flow algorithm. This approach combines the strengths of CPU and GPU by balancing the computational load between them and enabling pipeline processing.

Recently, we proposed a 3-D propagation algorithm for DIBR that combines color and depth images from multiple color and range cameras at arbitrary positions in 3-D space to synthesize arbitrary novel virtual views [15]. The algorithm includes three main steps described in Figure 6. In developing the algorithm, we chose image processing techniques that have a high degree of locality, such as bilateral filtering, median filtering, and the Sobel operator, to take advantage of massive parallelism. The rest of this section shows the GPU-based parallelization of two main bottlenecks in the algorithm.

The first bottleneck for GPU-based implementation is the depth propagation step. In this step, the depth value from each pixel in the depth view is copied to the corresponding pixel in a different color view [Figure 6(a)]. Even though there is not too much computation, this copying causes two problems. First, the algorithm is not trivially parallel because several depth pixels can map to the same pixel in the color view, which must be detected and resolved by selecting the closest pixel. Second, it tends to produce irregular memory accesses (both reading and writing). This is a typical problem whenever there is a 3-D-to-2-D projection in the video synthesis, which shows that not all video processing techniques are naturally amenable to parallelism. Our implementation uses atomic



[FIG6] Three main steps in the 3-D propagation algorithm. (a) Depth propagation step: depth information is propagated from depth cameras to color cameras. (b) Color-based depth filling and enhancement step performed at each color camera. (c) Rendering step: depth and color information are propagated from the color cameras to the virtual view.

hardware primitives, which guarantee the read-modify-write sequence of operations is performed without interference from other threads. Since atomic primitives temporarily lock a part of the memory, the GPU-based implementation suffers a significant slowdown. Nonetheless, it is still slightly faster than the CPU-based implementation (1.6 times). We are investigating a prefix-sum based parallel algorithm for a higher-level speedup.

After the depth propagation step, at each color view, we have one color image and one incomplete depth map (with unknown depth pixels). Then at the second step, a few more processing techniques are needed to fill in the depth map based on known depth and color pixels. Of these, the depth-color bilateral filter (DCBF) becomes the second bottleneck due to its heavy computational requirement. The DCBF is defined as follows:

$$d_A = \frac{1}{W_A} \sum_{B \in S_A} G_{\sigma_s^2}(|\dot{x}_A - \dot{x}_B|) \cdot G_{\sigma_r^2}(|I_A - I_B|) \cdot d_B$$
$$W_A = \sum_{B \in S_A} G_{\sigma_s^2}(|\dot{x}_A - \dot{x}_B|) \cdot G_{\sigma_r^2}(|I_A - I_B|),$$

where d_A, I_A, and $\dot{x}_A$ are the depth value, the color value, and the 2-D coordinate of a pixel A. S_A is a set of neighboring pixels of A. $G_\sigma(|\dot{x}|) = exp(-|\dot{x}|^2/(2\sigma^2))$ is the Gaussian kernel with variance σ^2 and W_A is the normalizing term.

The DCBF combines a spatial Gaussian kernel in depth-domain $G_{\sigma_s^2}$, whose weights depend on the Euclidean distance between depth samples, with a range Gaussian kernel in color domain $G_{\sigma_r^2}$, whose weights depend on the distance in values of corresponding color samples. The computational complexity of the DCBF is approximately $O(N^2K^2)$ for an $N \times N$ propagated depth map and a $K \times K$ kernel size. The filter is similar to the 2-D convolution with variant impulse responses, and the optimization principles of the STREAMING algorithm described in the section "Streaming" can be applied here. Each task calculates a depth value for one pixel based on neighboring pixels. The Gaussian kernel weights for both spatial and range filters are precomputed and stored as lookup tables. Table 5 shows the comparison result, with a speedup of 74.4 times compared to the sequential implementation on CPU.

[TABLE 5] TIMING COMPARISON (IN MILLISECONDS) OF SEQUENTIAL CPU-BASED AND GPU-BASED IMPLEMENTATIONS FOR TWO MAIN BOTTLENECKS: DEPTH PROPAGATION AND DEPTH-COLOR BILATERAL FILTERING (DCBF). THE IMAGE RESOLUTION IS 800×600 AND THE FILTER KERNEL SIZE IS 11×11.

	HARDWARE	DEPTH PROP.	DCBF
CPU	INTEL CORE 2 DUO E8400 3.0 GHZ	38	1041
GPU	NVIDIA GEFORCE 9800 GT	24	14
SPEEDUP		1.6X	74.4X

CONCLUSIONS

In this article, we focus on the applicability of parallel computing architectures to video processing applications. We demonstrate different optimization strategies in detail using the 3-D convolution problem as an example, and show how they affect performance on both many-core GPUs and symmetric multiprocessor CPUs. Applying these strategies to case studies from three video processing domains brings out some trends. The highly uniform, abundant parallelism in many video processing kernels means that they are well suited to a simple, massively parallel task-based model such as CUDA. As a result, we often see ten times or greater performances increases running on many-core hardware. Some kernels, however, push the limits of CUDA, because their memory accesses cannot be shaped into regular, vectorizable patterns or because they cannot be efficiently decomposed into small independent tasks. Such kernels, like the depth propagation kernel in the section "Synthesis Example: Depth Image-Based Rendering" may achieve a modest speedup, but they are probably better suited to a more flexible parallel programming model. We look forward to additional advances, as more researchers learn to harness the processing capabilities of the latest generation of computation hardware.

ACKNOWLEDGMENTS

We acknowledge the support of the FCRP Gigascale Systems Research Center, Intel/Microsoft Universal Parallel Computing Research Center, and VACE Program. Experiments were made

possible by generous donations of hardware from NVIDIA and Intel and by NSF CNS grant 05-51665.

AUTHORS

Dennis Lin (djlin@illinois.edu) is a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His research interests include gesture recognition and acceleration architectures for vision processing.

Xiaohuang (Victor) Huang (xhuang22@illinois.edu) is a Ph.D. student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He currently works with the IMPACT Group. His research interests are GPGPU programming and optimization.

Quang Nguyen (qnguyen2@illinois.edu) is an M.Sc. student in the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. His research interest includes image and video processing and depth image-based rendering.

Joshua Blackburn (jblackb2@illinois.edu) received the B.S.E.E. degree with highest honors from Cedarville University, Ohio, in 2007. He is currently pursuing his Ph.D. degree in electrical engineering at the University of Illinois at Urbana-Champaign. His research interests include multiscale geometric analysis and visual information representation. He received the 2007 Faculty Scholar Award from Cedarville University. He is a member of Tau Beta Pi and is a Student Member of IEEE.

Christopher Rodrigues (cirodrig@crhc.illinois.edu) is a graduate research assistant at the University of Illinois, where he earned his M.S. degree in electrical engineering in 2008. His research interests include language and compiler support for parallelization and memory safety.

Thomas Huang (huang@ifp.uiuc.edu) received his Sc.D. degree from Massachusetts Institute of Technology (MIT) in electrical engineering. He joined the University of Illinois at Urbana-Champaign in 1980 and is currently the William L. Everitt Distinguished Professor of Electrical and Computer Engineering, research professor of the Coordinated Science Laboratory, professor of the Center for Advanced Study, and cochair of the human computer intelligent interaction major research theme of the Beckman Institute for Advanced Science and Technology. He is a member of the National Academy of Engineering and received the IEEE Jack S. Kilby Signal Processing Medal and the King-Sun Fu Prize of the International Association of Pattern Recognition. He has published 21 books and more than 600 technical papers in network theory, digital holography, image and video compression, multimodal human computer interfaces, and multimedia databases.

Minh N. Do (minhdo@illinois.edu) is an associate professor of electrical and computer engineering at the University of Illinois, Urbana-Champaign (UIUC). He received a Ph.D. degree in communication systems from the Swiss Federal Institute of Technology Lausanne (EPFL). His research interests include image and multidimensional signal processing, computational imaging, wavelets and multiscale geometric analysis, and visual information representation. He received the EPFL Best Doctoral Thesis Award, a National Science Foundation CAREER Award, a Xerox Award for Faculty Research from UIUC, and the IEEE Signal Processing Society Young Author Best Paper Award.

Sanjay J. Patel (sjp@illinois.edu) is an associate professor of electrical and computer engineering and Willett Faculty Scholar at the University of Illinois at Urbana-Champaign. From 2004 to 2008, he served as the chief architect and chief technology officer at AGEIA Technologies, prior to its acquisition by Nvidia Corp. He earned his bachelor's degree (1990), M.Sc. degree (1992), and Ph.D. degree (1999) in computer science and engineering from the University of Michigan, Ann Arbor.

Wen-mei W. Hwu (w-hwu@illinois.edu) is a professor and holds the Sanders-AMD Endowed Chair in ECE from the University of Illinois at Urbana-Champaign. He directs the IMPACT research group. He has received numerous awards including the ACM SigArch Maurice Wilkes Award, the ACM Grace Murray Hopper Award, and the ISCA Most Influential Paper Award. He is a Fellow of the IEEE and ACM.

REFERENCES

[1] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Proc. 21st Int. Workshop Languages and Compilers for Parallel Computing* (Lecture Notes in Computer Science). New York: Springer-Verlag, 2008, pp. 16–30.

[2] L. Cao, V. Le, S-F Tsai, K-H Lin, Z. Li, Ji. Yang, T. S. Huang, F. Lv, W. Xu, M. Yang, K. Yu, Z. Zhao, G. Zhu, and Y. Gong , "Surveillance event detection," in *Proc. TRECVID Video Evaluation Workshop*. Gaithersburg, MD: NIST, Nov. 2008.

[3] H. Ning, T. X. Han, D. B. Walther, M. Liu, and T. S. Huang, "Hierarchical space-time model enabling efficient search for human actions," *IEEE Trans. Circuits Syst. Video Technol.*, to be published.

[4] D. Lin and M. Dikmen, Vivid library [Online]. Available: http://libvivid.sourceforge.net

[5] C. Zach, T. Pock, and H. Bischof, "A duality based approach for realtime TV-L1 optical flow," in *Proc. DAGM Pattern Recognition*, Heidelberg, Germany, 2007, pp. 214–223.

[6] C. Wu. (2007). SiftGPU: A GPU implementation of scale invariant feature transform *(SIFT)* [Online]. Available: http://cs.unc.edu/~ccwu/siftgpu

[7] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *Proc. 25th Int. Conf. Machine Learning ICML '08*. New York: ACM, 2008, pp. 104–111.

[8] P. Thévenaz, T. Blu, and M. Unser, "Interpolation revisited," *IEEE Trans. Med. Imag.*, vol. 19, no. 7, pp. 739–758, July 2000.

[9] M. Unser, "Splines: A perfect fit for signal and image processing," *IEEE Signal Processing Mag.*, vol. 16, no. 5, pp. 22–38, 1999.

[10] C. Sigg and M. Hadwiger, "Fast third-order texture filtering," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Reading, MA: Addison-Wesley, 2005, no. 20, pp. 313–329.

[11] D. Ruijters, B. M. ter H. Romeny, and P. Suetens, "Efficient GPU-based texture interpolation using uniform B-splines," *J. Graph. Tools*, vol. 13, no. 4, pp. 61–69, 2008.

[12] S. C. Chan, H. Y. Shum, and K. T. Ng, "Image-based rendering and synthesis," *IEEE Signal Processing Mag.*, vol. 24, no. 6, pp. 22–33, Nov. 2007.

[13] R. Yang and M. Pollefeys, "Multi-resolution real-time stereo on commodity graphics hardware," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, June 2003.

[14] I. Geys and L. V. Gool, "View synthesis by the parallel use of GPU and CPU," *Image Vis. Comput.*, July 2007.

[15] Q. H. Nguyen, M. N. Do, and S. J. Patel, "Depth image-based rendering from multiple cameras with 3D propagation algorithm," in *Proc. Int. Conf. Immersive Telecommunications*, May 2009, pp. 1–6.

[SP]