Design and Performance Evaluation of Image Processing Algorithms on GPUs

In Kyu Park, *Member*, *IEEE*, Nitin Singhal, *Member*, *IEEE*, Man Hee Lee, *Student Member*, *IEEE*, Sungdae Cho, and Chris W. Kim

Abstract—In this paper, we construe key factors in design and evaluation of image processing algorithms on the massive parallel graphics processing units (GPUs) using the compute unified device architecture (CUDA) programming model. A set of metrics, customized for image processing, is proposed to quantitatively evaluate algorithm characteristics. In addition, we show that a range of image processing algorithms map readily to CUDA using multiview stereo matching, linear feature extraction, JPEG2000 image encoding, and nonphotorealistic rendering (NPR) as our example applications. The algorithms are carefully selected from major domains of image processing, so they inherently contain a variety of subalgorithms with diverse characteristics when implemented on the GPU. Performance is evaluated in terms of execution time and is compared to the fastest host-only version implemented using OpenMP. It is shown that the observed speedup varies extensively depending on the characteristics of each algorithm. Intensive analysis is conducted to show the appropriateness of the proposed metrics in predicting the effectiveness of an application for parallel implementation.

Index Terms—GPU, CUDA, image processing, parallel implementation, GPGPU.

1 Introduction

A long standing challenge in the field of image processing is that intensive computation power is required to achieve high accuracy and real-time performance. Real-time image processing of video frames is difficult to attain even with the most powerful modern CPU. High-resolution video capture devices and increased requirements for accuracy make it even harder to realize real-time performance.

Recently, GPU has evolved into an extremely powerful computation resource. For example, NVIDIA GTX 280 with 240 processing cores at 602 MHz and 1 GB of GDDR3 running through a 512-bit memory bus performs 933 GFLOPS in its peak performance. As a comparison, 3.2 GHz Intel Core2 Extreme (QX9775) operates at roughly 51.2 GFLOPS. In addition, modern GPUs are equipped to support high-level languages. This significantly increases user programmability and facilitates use of GPUs for general purposes, also known as general-purpose computation on GPU (GPGPU) [1], [2], [3]. Most of image processing and computer vision tasks perform the same computation on a number of pixels, a typical data parallel operations. Thus, they can exploit the single instruction multiple data (SIMD) architecture and be effectively parallelized on GPU.

Before the release of compute unified frameworks, such as NVIDIA CUDA [4] and OpenCL [5], the legacy GPGPU architecture provides limited support for GPGPU as a corner case of 3D graphics pipelines. The major bottleneck has been the restricted scatter operation, i.e., the output of the parallel process is limited to a single pixel color in only a few bytes. This limits the flexibility to the design of parallel algorithms for nongraphics experts. However, the recent compute unified framework provides user-controllable parallelism at the thread level and allows unrestricted usage of the global memory space for both gather (read from) and scatter (write to) operations. Moreover, it equips on-chip shared memory for much faster access to data and efficient communication between parallel threads. Consequently, a wider variety of algorithms are now being designed and implemented on the GPU. In this paper, we adopt NVIDIA CUDA [4] for the computing framework.

The purpose of using GPU as an alternative computational platform is to achieve acceleration for computationally intensive tasks beyond the domain of graphics applications. Image processing is selected from a number of areas that involve high computing complexity, since the common problem structure matches GPU's SIMD architecture well. More specifically, GPU is well suited to address massive data parallel processing with high floating-point (FP) arithmetic intensity. Many image processing applications that process large data sets involve highly complex mathematical and logical operations. This particular structure fits very well with the GPU data parallel programming model and facilitates significant acceleration.

However, not all image processing algorithms are ported to GPU with a significant speedup. Modern GPUs have

- *I.K. Park and M.H. Lee are with the School of Information and Communication Engineering, Inha University, 253 Yonghyun-dong, Nam-gu, Incheon 402-751, Korea. E-mail: pik@inha.ac.kr, maninara@hotmail.com.*
- *N. Singhal and S. Cho are with the Telecommunication Module Laboratory, Digital Media and Communication R&D Center, Samsung Electronics Co., Ltd., 416 Maetan3-dong, Yeongtong-gu, Suwon 443-742, Korea. E-mail: nitin.singhal@ieee.org, s-d.cho@samsung.com.*
- *C.W. Kim is with NVIDIA Corporation, 2101 COEX Trade Tower, 159-1 Samsung-dong, Kangnam-gu, Seoul 135-729, Korea. E-mail: chkim@nvidia.com.*

Manuscript received 30 Sept. 2009; revised 31 Jan. 2010; accepted 1 Mar. 2010; published online 27 May 2010.
Recommended for acceptance by D.A. Bader, D. Kaeli, and V. Kindratenko.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-2009-09-0476.
Digital Object Identifier no. 10.1109/TPDS.2010.115.

indigenous architecture and hardware limitations that must be taken into account when the target algorithm is designed and implemented. Although image processing algorithms are well suited to the massive parallel architecture of the GPU in general, a large number of algorithms fail to achieve satisfactory performance gain, since the inherent nature of the algorithms and the GPU platform is uncooperative. Therefore, it is critical to evaluate the algorithm-platform cooperativeness correctly during the design and implementation process and reflect it to obtain the best performance.

In this paper, we deduce key common factors for designing and evaluating image processing algorithms on the massive parallel GPU with the CUDA framework. These are defined as measurable metrics to characterize each algorithm in a quantitative manner. We choose popular algorithms in four major domains (3D imaging, feature extraction, image compression, and computational photography) to experimentally show the effectiveness of the proposed metrics. We implement them on the CUDA programming model, while exploring the relationship between the measured metrics to achieve the significant speedup over CPU (using OpenMP [6]).

This research aims to accomplish an efficient application of image processing algorithms for intensive computation using GPUs. The contributions of this paper are twofold: First, based on the characteristics of image processing, we propose measurable metrics to numerically evaluate the inherent parallelism present in popular image processing algorithms. The resultant numerical data obtained from the metrics allow us to predict the effectiveness of the problem for the parallel implementation. An alternative use for the proposed metrics is to compare the two implementations of the same algorithm on the GPU. Second, we propose design and implementation of algorithms on the CUDA programming model using multiview stereo matching, linear feature extraction, JPEG2000 encoding, and nonphotorealistic rendering (NPR), as our example applications. Preliminary version of this paper was presented in [7].

The remainder of the paper is organized as follows: Section 2 addresses notable previous GPGPU researches. Section 3 provides a brief survey of the CUDA programming model. In Section 4, a set of metrics is proposed to characterize algorithms for parallel implementation. Section 5 describes the algorithms investigated in this work and their mapping on the GPU. Section 6 presents a brief survey on CUDA benchmarking and optimization tools. Experimental results are shown in Section 7. Finally, Section 8 concludes this paper.

2 Related Work

The design and implementation strategy of the GPGPU algorithm on the compute unified device architecture (CUDA) platform is quite different from the legacy shader-based one using *Cg* [8], DirectX high level shading language [9] (HLSL), and OpenGL Shading Language [10] (GLSL). In this section, we introduce notable previous work implemented on different platforms.

2.1 Image Processing on the Legacy GPGPU Platform

Traditionally, general-purpose GPU was accomplished using a shader-based framework [10]. More general use of GPU for nongraphics has become popular over the last five years. A survey of general-purpose computation on graphics hardware is intensively described in [2].

Shen et al. [11] implemented motion compensation and color space conversion of MPEG video encoding on GPU using DirectX to achieve two to three times speedup. However, they focused on the collaboration of CPU and GPU, instead of fully utilizing resources on the GPU. Yang and Pollefeys [12] proposed GPU-based real-time stereo matching with multiple images. They implemented rectification and disparity computing using standard use of texture.

A few interesting projects build an image processing and computer vision library on the GPGPU platform. For example, GpuCV [13] and MinGPU [14] are open-source computer vision libraries that users can use transparently without any knowledge of GPU and GPGPU. GpuCV supports GLSL-based and CUDA-based platforms simultaneously. OpenVIDIA [15] is another open-source library that provides a set of useful computer vision functionalities.

2.2 Image Processing on the Compute Unified Platform

There is a strong desire to use GPU to accelerate computationally intensive tasks in the image processing and computer vision domain, mainly due to recent advances in the development of the compute unified framework, i.e., CUDA by NVIDIA. In [16], several computer vision algorithms are implemented on the GPU. These include scale invariant feature detection, Canny edge detection, Kanade-Lucas-Tomasi (KLT) tracking, optical flow computation, graph cuts, stereo depth, shape from motion, visual hull computation, K-nearest neighbor (KNN) search, and particle filtering. Most of the above works focused on how to map algorithms onto the GPU architecture. Systematic analysis and guidance for application to other domains are lacking.

Ryoo et al. [17] described several useful principles to optimize implementations on the G80 architecture. However, their main goal is efficient use of shared resources. This is difficult to generalize in the image processing domain.

Recently, CUJ2K [18] has been developed as a fast encoder for JPEG2000. For 2D DWT, it adopts the lifting scheme. In the case of Tier-1, its implementation involves parallelism at the code block level, which is the same as in our implementation (see Section 5.4.4). However, their method is not based on the standard JPEG2000 codec, such as [19], and performance evaluation in terms of rate distortion is lacking.

3 GPU with CUDA Framework

Recently, NVIDIA released the *CUDA*, a new GPU programming model, to assist developers in general-purpose computing [4]. All the latest NVIDIA graphics hardware such as GeForce, Quadro-FX, Tesla, and ION are CUDA compliant. In this section, we describe the hardware and software architecture of the CUDA framework.

3.1 Hardware Architecture

A CUDA compliant device is a set of multiprocessor cores, capable of executing a very high number of threads concurrently, that operates as a coprocessor to the CPU or host. Each multiprocessor has a single-instruction multiple thread (SIMT) architecture; that is, each processor of the multiprocessor executes a different thread but all the

TABLE 1
Dependency between the Characteristics of Image Processing Algorithms and the Proposed Metrics

('-' : None, '★' : Low, '★★' : Medium, '★★★' : High)

Characteristics / Metrics	Large Size Memory Buffer	Frequent Access to Memory Buffer	Sequential Access Pattern	Intensive FP Operations	Intensive Logical Operations	A Mixture of Subalgorithms
Parallel Fraction	-	-	-	-	-	★★
FP Computation to Memory Access Ratio	-	★★★	-	★★★	-	-
Per-Pixel FP Computation	-	★	-	★★★	-	-
Per-Pixel Global Memory Access	-	★★★	★	-	-	-
Branching Diversity	-	-	-	-	★★★	-
Task Dependency	★	-	★	-	-	★★★

threads run the same instructions, operating on different data based on its *thread ID*, at any given clock cycle. In concept, it is similar to SIMD architecture, often employed by vector processors in which it processes four clusters of data, but SIMT processes more clusters of data with an array of scalar processors.

G80/G92 has 16 multiprocessors with eight scalar processors in each multiprocessor. The device maintains its own DRAM, referred to as *device memory*. Device memory is divided into three different types: *global memory*, *constant memory*, and *texture memory*. These can all be read from or written to by the host and are persistent through the life of the application. Multiprocessors also have on-chip memory in the form of *registers*, *shared memory*, *constant cache*, and *texture cache*. The *registers* (32-bit) are the fastest available but only support a limited amount of space (32-64 KB). A parallel data cache of *shared memory* is shared by all the processors and is limited to 16 KB. A *constant cache* speeds up reads from the constant memory. Similarly, *texture cache* is used to speed up reads from the texture memory.

3.2 Software Architecture

A parallel application that is executed many times, but independently on different data, is a function that is executed on the device by many threads running on different processors of the multiprocessors. Such a function, called a *kernel*, is compiled to the instruction set. A thread block is a batch of threads that synchronize their execution using shared memory. Each thread block executes on one multiprocessor. The number of threads in a thread block is limited to 512. A group of thread blocks of equal dimensions and size executes the same kernel and is batched together into a *grid of thread blocks*. Threads are organized into *warps* or groups of 32 threads, where each warp executes one kernel instruction at a time. CUDA includes C/C++ software development tools that allow programmers to combine host code with device code. To do so, CUDA programming requires a single program (kernel) written in C/C++ with some extensions [20]. Each source file containing these extensions must be compiled with the CUDA nvcc compiler [21].

4 Metrics to Characterize Parallel Implementation of Image Processing Algorithms on GPU

In this section, we propose novel metrics to characterize the algorithm to judge the effectiveness of the algorithm for parallel implementation.

4.1 Characteristics of Image Processing Algorithms

Image processing algorithms are too diverse to define a general aspect for parallel implementation. However, in practice, image processing involves independent processing of a massive pixel or feature set. This can benefit from SIMD-style GPU architecture. More importantly, image processing involves large memory buffers to store pixel data and needs frequent access to them, where the access pattern is often regular and sequential, as row-major or column-major order. The complexity of operations applied to the pixel data depends on the characteristics of the algorithm. However, the complexity is generally high due to intensive floating-point and logical operations. In addition, image processing algorithms consist of a mixture of subalgorithms, for which the efficiency of implementation on the GPU is affected by the dependency of the task order and data exchange between consequent tasks.

4.2 Proposed Metrics

Based on the characteristics of image processing algorithms, we propose six metrics. These are

1. parallel fraction;
2. the ratio of floating-point computation to global memory access;
3. per-pixel floating-point instructions;
4. per-pixel memory access;
5. branching diversity; and
6. task dependency.

The metrics are customized for image processing tasks. Table 1 shows the relationship between the characteristics of image processing algorithms and each of the above metrics. The fundamental idea when defining each of the metrics is to consider their dependence on the characteristics of image processing tasks. Each of the metrics is numerically evaluated using the serial CPU code before implementation using CUDA. The resultant numerical values obtained from the metrics facilitate efficient analysis of inherent parallelism in a given algorithm and provide information on the bottleneck, if any. The following describes each of the above metrics in detail along with the method of their numerical evaluation.

4.2.1 Parallel Fraction

In parallel computing, Amdahl's law [22] is used to predict the theoretical maximum speedup using multiple processors. For the parallelization scenario, Amdahl's law states that if f is the fraction of a program that can be made

parallel and $(1-f)$ is the fraction that cannot be parallelized, i.e., that must remain serial, then the maximum speedup that can be achieved using N processors is

$$S \leq \frac{1}{1-f+\frac{f}{N}}. \quad (1)$$

In practice, performance falls rapidly once there is even a small component of $(1-f)$. If N is fixed, the maximum speedup that can be achieved using parallel implementation is bottlenecked by the fraction $(1-f)$. In the above context, the numerical value of parallel fraction f helps determine the theoretical maximum speedup.

In the case of image processing algorithms, a problem is more often divided into a mixture of subalgorithms. The parallel fraction is directly proportional to the fraction of these subalgorithms that are parallelized. Note that Amdahl's law presents speedup between implementation on multiple cores and a single core with the same computing capability (CPU or GPU).

4.2.2 Floating-Point Computation to Memory Access Ratio

In the CUDA framework, a memory access instruction includes any instruction that reads from or writes to shared, local, or global memory space. When the threads in a warp are coherent, a multiprocessor takes four clock cycles to issue one memory instruction. However, when accessing local or global memory, there are, in addition, 400-600 clock cycles of memory latency. The latency caused by local or global memory access can be hidden if there are sufficient independent floating-point instructions. This allows the GPU to perform arithmetic operations, while certain threads are waiting for the memory access to be completed. We estimate the ratio between the number of floating-point instructions and the global memory access to numerically evaluate the above property. The higher the ratio is, the better the utilization of the multiprocessor resources; this is reflected in the performance of the parallel implementation.

In the image processing context, most algorithms involve intensive floating-point arithmetic operations along with frequent access to the global memory space, both of which are, respectively, proportional to the memory buffer size. However, the floating-point computation to memory access ratio is independent of the memory buffer size.

4.2.3 Per-Pixel Floating-Point Computation

Modern GPUs outperform CPUs in floating-point operations per second by an approximate factor of $20\times$. The reason behind this discrepancy in floating-point capability is that the GPU is specialized for intensive computing, highly parallel computation, and therefore, designed such that more transistors are devoted to data processing. More specifically, the GPU is well suited to address problems with high floating-point intensity. In the above context, the number of per-pixel floating-point computations provides an estimate of the computational strength of an algorithm and its effectiveness for parallel implementation.

Most image processing algorithms involve intensive floating-point operations that are indirectly correlated with frequent access to memory buffer.

4.2.4 Per-Pixel Device Memory Access

Current GPUs have $10\times$ higher main memory bandwidth and use data parallelism to achieve more operations per second than CPUs. An image processing algorithm having frequent memory access per pixel can exploit the high memory bandwidth to achieve significant speedup. Furthermore, in the CUDA framework, much higher memory access ($100\times$ compared to GPU device memory) is efficiently processed using coherent memory access and on-chip shared memory to achieve substantially higher performance. The number of per-pixel memory accesses determines the frequency of memory access in a given algorithm. It should be noted that the sequential memory access pattern associated with some of the image processing algorithms often limits the peak memory bandwidth due to incoherent memory access.

4.2.5 Branching Diversity

In CUDA architecture, for every instruction instance, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes a single common instruction at a time, so full efficiency is realized when all 32 threads in the warp agree on their execution path. However, flow control instructions such as `if`, `switch`, `do`, `for`, `while` can significantly degrade parallel efficiency by causing threads of the same warp to diverge, i.e., to follow different execution paths that have to be serialized. Many image processing algorithms involve heavy logical operations using control flow instructions other than arithmetic and memory lookup operations. These control flow instructions significantly reduce the parallelism among threads in a warp, forcing the threads to be serialized.

We measure the load balance between threads belonging to the same warp to evaluate the divergence caused by control flow instructions, i.e., branching diversity. We measure the execution time for all tasks (typically in a `for` loop) to be parallelized and normalize it. Then, we compute the variances of task groups consisting of 32 consecutive tasks and take the maximum of these. A greater variance implies that the load is imbalanced, and consequently, the branching diversity is higher.

4.2.6 Task Dependency

CUDA architecture offers three-key abstractions at its core: a hierarchy of thread groups, shared memory, and per-block/global barrier synchronization. A typical CUDA implementation partitions the problem into subproblems (kernels) that can be solved independently. Then, each subproblem is partitioned into finer pieces (threads) that can be solved cooperatively in parallel using thread blocks. The number of global barrier synchronization required by an application defines task dependency. In an image processing context, the task dependency is resolved by the number of kernels that are executed sequentially.

Note that CUDA has no global synchronization, as it becomes expensive to build in hardware for GPUs with a high processor count. Therefore, this necessitates multiple kernel launches to achieve global synchronization.

In addition, large memory buffer size can affect task dependency. For example, in large-resolution video processing or 3D medical image processing, the required memory

TABLE 2
Selected Algorithms and Their Characteristics of Designing and Implementing Parallel Algorithms on the GPU with CUDA Framework

Characteristics / Algorithms	Processing Domain	Degree of Concurrency	Branching Diversity of Parallel Threads	Floating-Point Arithmetic Intensity	Other Features
Multiview Stereo Matching	Pixel	Massive	Low	Low	-
Linear Feature Extraction	Pixel / Feature	Medium	Low (Edge) High (Linking, Fitting)	High	Overcomputation occurs
JPEG2000 Encoding	Pixel / Bitplanes	Massive (DWT) Low (EBCOT)	High	High (DWT) High (EBCOT)	-
Non-Photorealistic Rendering	Pixel	Massive	Low	High	Only pixelwise convolutions

size is often bigger than the amount of device memory. Consequently, multiple kernel calls are used to process the decomposed data.

4.3 Relative Importance of the Metrics

Based on the characteristics of image processing algorithms, we state the relative importance of the proposed metrics as: Parallel Fraction $>$ Branching Diversity $>$ Per-Pixel FP Computation $>$ Per-Pixel Memory Access $>$ FP Computation to Memory Access Ratio $>$ task Dependency.

The parallel fraction determines the intrinsic parallelism in a given application and limits the maximum speedup. Therefore, parallel fraction's influence to the speedup is largest. GPU is well suited to address massive data parallelism, which is the major source of speedup. Hence, maximization of thread parallelism dominates all other factors influencing the speedup. In this context, branching diversity impedes the degree of thread parallelism, which makes it the next most important metric.

Modern GPUs are designed to achieve approximately $10\times$ higher memory bandwidth than CPU. However, in order to achieve peak memory bandwidth, the memory access pattern needs to be coalesced with efficient use of cache and shared memory. Furthermore, general image processing algorithms involve higher per-pixel floating-point computations than memory access. The two reasons stated above make the per-pixel floating-point computation more important than the per-pixel memory access. Floating-point computation to memory access ratio hides the potential stalls caused by high-latency global memory access. However, the use of an efficient cache and shared memory alleviates the latency problem, reducing the importance of this metric. Additionally, floating-point computation to memory access ratio can be computed using per-pixel floating-point computations and per-pixel memory access, which reduces the information entropy.

Task dependency generally shows the ease of implementation and has less influence on the actual speedup. Hence, it stands at the lowest position in the relative importance hierarchy.

5 Test Algorithms and Their Implementation on the GPU

5.1 Criteria for Algorithm Selection

Image processing has a broad spectrum of domains. In this paper, we select four major domains (3D shape reconstruction, feature extraction, image compression, and computational photography) based on the general research interest of the society. In each domain, we select target algorithms (multiview stereo matching, linear feature extraction, JPEG2000 image encoding, and nonphotorealistic rendering) to implement and analyze on the GPU platform. These include a variety of image processing routines such as Gaussian smoothing, window matching, bilateral filtering, Canny edge detection, quantization, arithmetic coding, and wavelet transform. Furthermore, common means of error evaluation, such as sum of the absolute difference (SAD) and normalized cross correlation (NCC), are part of these algorithms.

The inherent structure and the nature of the selected algorithms are quite diverse; hence, several important design and implementation issues on the GPU with CUDA framework can be exploited. For example, multiview stereo matching has massive parallel pixel domain processing with low divergence, while JPEG2000 embedded block coding with optimized truncation (EBCOT) has small-scale parallel threads with high divergence. Linear feature extraction processes a variable number of image features, which makes it difficult to parallelize. JPEG2000 encoding involves multiscale image processing, i.e., discrete wavelet transform (DWT), as well as an encoding stage that has a complex context decision procedure. Finally, nonphotorealistic rendering involves a purely mask convolution operation on the image space and has high floating-point arithmetic intensity. The characteristics of the selected algorithms are summarized in Table 2.

5.2 Multiview Stereo Matching

A multiview stereo (MVS) matching problem is formulated as follows: Given N calibrated images $I = \{I_0, I_1, \ldots, I_{N-1}\}$ and corresponding projection matrix $P = \{P_0, P_1, \ldots, P_{N-1}\}$, find a set of 3D points $X = \{X_0, X_1, \ldots, X_{M-1}\}$ where the projection of X_i to its supporting images (images in which X_i is visible) preserves photoconsistency. That is, the local window around the projected position is well matched in terms of the SAD or NCC metrics.

5.2.1 Algorithm Description

Although there are many different algorithms in this field, including the state-of-the-art one [23], the core computation in MVS is massive local window matching between input images. Individual local window matching is independent and is a type of computation that GPU does very well. In

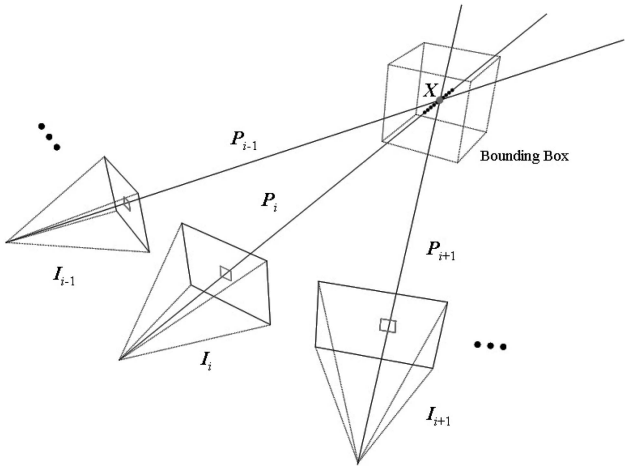


Fig. 1. Multiview stereo reconstruction.

this paper, we are not going to select and implement the whole flow of a particular MVS algorithm but only the initial matching stage that is common.

Given a pixel position (x, y) of a reference image I_i, the depth value is determined as follows: As shown in Fig. 1, the sampled 3D points along the line of sight and inside the bounding volume are projected one-by-one onto one of the neighboring images (for example, I_{i-1} and I_{i+1}) to match. Among them, a 3D point with minimum SAD or NCC is stored as the local best match. The same process is performed for other neighboring images. Finally, a 3D point is determined as the correct depth point if its count of local best match is above the given threshold (MIN_COUNT). The entire process is repeated for every other reference image and all pixel positions.

The total computational complexity is $O(N^2WHL)$, where N, W, H, and L are the number of input images, the horizontal and vertical image resolution, and the size of the (cubic) bounding box, respectively.

5.2.2 Mapping on the GPU

In our implementation on the GPU, $W \times H$ threads are created such that each thread computes the depth of a single pixel. We need $O(N)$ calls to the kernel function to compute the depth map of a single reference image, in which the complexity of each call is $O(L)$. Since GPU can execute a fixed maximum number of threads T_{max} (which is 12,288 for G80 and 30,720 for GX200), actual computational complexity is $O(\frac{NWHL}{T_{max}})$ for large W and H. Therefore, the parallel implementation has reduced complexity $O(\frac{N^2WHL}{T_{max}})$, since the kernel has to be invoked N times to obtain the depth map of all the input images. The input images are copied to the global memory for read/write by threads. The local window coefficients of the reference image are stored in the shared memory for frequent but faster access.

5.3 Linear Feature Extraction

Linear feature (piecewise line fitting of visual edge) plays an important role in many object recognition applications, especially when the target object is composed of linear structures. Appropriate applications include building detection, road lane detection, bin picking in robot assembly, and many other systems that deal with man-made objects.

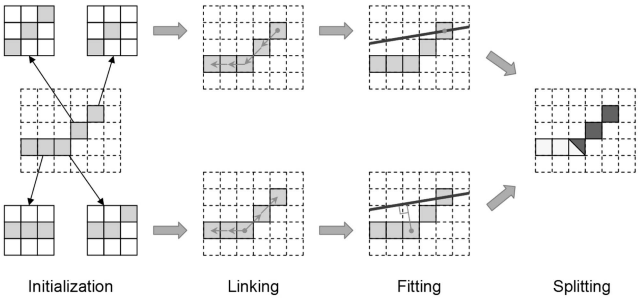


Fig. 2. Parallel edge linking and line fitting procedure. Only two parallel threads are shown as example.

5.3.1 Algorithm Description

We employ Nevatia and Babu's algorithm [24] as a base algorithm and replace edge detection with Canny's method [25]. The basic algorithm starts with edge detection followed by a thinning procedure. Then, edges are grouped into linked edge chains considering eight-neighborhood connectivity. After edge chains are obtained, iterative line fitting is performed on each edge chain.

A chain is first approximated by a line segment connecting the end points. If the fitting error (maximum deviation from the line segment to the chain) is above the threshold, the chain is broken into two separate chains at the edge pixel with maximum deviation. This procedure is performed iteratively until all the chains are fitted with line segments whose fitting errors are below the threshold.

5.3.2 Mapping on the GPU

The input and pattern images are stored in the texture memory to achieve faster access than would be the case using global memory. The parallel implementation on the GPU consists of six different kernel functions that work per-pixel, as follows:

1. Canny edge detection is performed where thread kernel function performs per-pixel filtering.
2. Edge pixels are classified and labeled according to their neighboring (3×3) edge pattern (*Initialization* step in Fig. 2).
3. Edge pixels are traversed to both directions sequentially, until the end is reached, yielding the start position **S** and the end position **E** of the connected edge chain to which the edge pixel belongs (*Linking* step).
4. Per-pixel deviation is computed for the edge pixels by calculating the distance to **SE** (*Fitting* step).
5. Maximum deviation D_{max} is found. If it is above the threshold, then the chain is broken into two subchains and the start and end positions of the edges involved are updated (*Splitting* step).
6. Repeat steps 3-5 until there is no new subchain.

Threads are issued for all the pixels in the image to avoid counting the edge pixels and keep the number of threads unchanged. Consequently, this involves unnecessary thread issues for nonedge pixels. More importantly, there is unavoidable redundant traversing along the edge chain. This causes serious over computation, although all of the procedure in Fig. 2 is implemented and run on the GPU.

5.4 JPEG2000 Encoding

Approved as an International Standard (ISO/IEC 15444-1) [26] in December 2000, JPEG2000 is an emerging image compression standard for next-generation digital imagery. It offers a host of features beyond the scope of conventional JPEG. Better quality at low bit rate, progressive transmission by pixel accuracy and resolution, region of interest (ROI) coding, random code stream access and processing, error resilience, and both lossy and lossless compression are among the most important features of the JPEG2000 standard. JPEG2000 is targeted for rapidly growing diverse imaging applications, e.g., digital photography, printing, mobile applications, medical imagery, and wireless image transmission.

JPEG2000 architecture consists of DWT, scalar quantization, context modeling, arithmetic coding, and rate allocation [26], [27]. It employs the idea of EBCOT Tier-1 [28] for context modeling and arithmetic coding. Although the DWT and EBCOT Tier-1 algorithm offers many benefits for JPEG2000; unfortunately, both algorithms are computation and memory intensive (typically more than 70 percent) in software-based implementations [19]. The intensive complexity of DWT is due to multilevel filtering and downsampling. The EBCOT Tier-1 algorithm adopts fractional bitplane coding using three coding passes; this introduces considerable computation time. In EBCOT Tier-1, each subband is divided into rectangular code blocks and the coding of each code block proceeds by bitplanes. To achieve efficient embedding, the EBCOT Tier-1 block coding algorithm further adopts the fractional bitplane coding ideas, and each bitplane is coded by three coding passes. However, the three coding passes introduce considerable computation time.

Numerous studies have targeted DWT and EBCOT Tier-1 for hardware implementation [29], [30], [31], [32]. This involves replacing the software implementation with dedicated hardware. Although hardware implementation offers a real-time solution, extra cost is needed to fabricate the dedicated hardware. The wavelet transform part is clearly the most demanding part of the algorithm, followed by the encoding stage. Fortunately, both DWT and EBCOT Tier-1 stages can be parallelized. Intrinsically, sequential parts of the algorithm are image and bitstream I/O and R/D allocation. These have relatively low complexity. In this work, we have explored the mapping of the DWT and EBCOT Tier-1 algorithm on modern GPUs using the CUDA programming model.

5.4.1 Algorithm Description: DWT

DWT has traditionally been implemented using two different algorithms, usually known as the filter bank scheme (FBS) [33] and the lifting scheme (LS) [34]. In this work, we adopt the filter bank scheme for parallelizing. The 2D-DWT is obtained by applying a separate 1D transform along each dimension. From an implementation viewpoint, one of the most interesting advantages of LS is that it requires fewer arithmetic operations and consumes less memory. Conversely, FBS uses a pair of quadrature mirror filters. It consumes more memory and requires more computation. However, the LS advantages are at the expense of introducing intermediate value sharing; this becomes a bottleneck on most GPUs. Hence, the parallelizable FBS approach is more favorable.

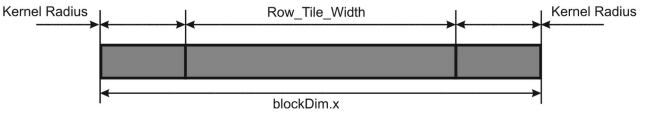


Fig. 3. Layout of the thread block grid for the horizontal filtering.

Let x_n^j be the approximate signal at level j. A low-pass h and high-pass g filter kernels are convolved with the signal x_n^j to produce low- and high-pass subband sequences, given by

$$LP_n^{j-1} = \sum_k h_k x_{2n-k}^j, \tag{2}$$

$$HP_n^{j-1} = \sum_k g_k x_{2n+1-k}^j. \tag{3}$$

5.4.2 Mapping on the GPU : DWT

For 2D DWT, a 2D convolution filter requires $NM(1+K)$ multiplications and additions for each output pixel, where N and M are the width and height of the filter kernel and K is the measure of additional computation required for array indexing. Separable filters are special filters that can be expressed as the composition of two 1D filters, one on the rows on the image, and one on the columns. A separable filter requires only $N + M(1+K)$ multiplies and adds for each output pixel. Generally, $K = 4$ for a 2D convolution filter and $K = 3$ for the separable filter case.

- *Horizontal filter*: Fig. 3 shows the layout of the thread block grid for the horizontal filter. For the shared memory load, each active thread loads one pixel. If the neighboring pixels go beyond the image boundary of the current level, we adopt symmetric periodic extension [35], which mirror pixels across the boundary. This is followed by convolution filtering that involves Row_Tile_Width number of threads with each thread producing one output pixel. Each thread (input base position) is mapped to the output position β. The output pixel position β is given by

$$\beta = \begin{cases} \dfrac{tid-1}{2} + \dfrac{l}{2}, & \text{if } (tid\%2), \\ \dfrac{tid}{2}, & \text{otherwise}, \end{cases} \tag{4}$$

  where l is the length of the input signal at level j and tid is the thread index. The convolution takes place with the input base position at the center and $\pm\lceil\frac{N_0}{2}\rceil$ neighboring pixels, where N_0 is the length of the filter kernel. We determine if the current output pixel belongs to the high-pass or the low-pass region based on the position of β, such that it belongs to the high-pass region if $\beta > \frac{l}{2}$. Fig. 4 describes the mapping operation from output index to input base position.
- *Vertical filter*: Fig. 5 shows the layout of the thread block grid for the vertical filter. Each thread processes more than one pixel to achieve higher efficiency. In a thread block, $Col\_Tile\_Width * (Col\_Tile\_Height + N_0)$ numbers of pixels are prefetched into shared memory. The convolution stage

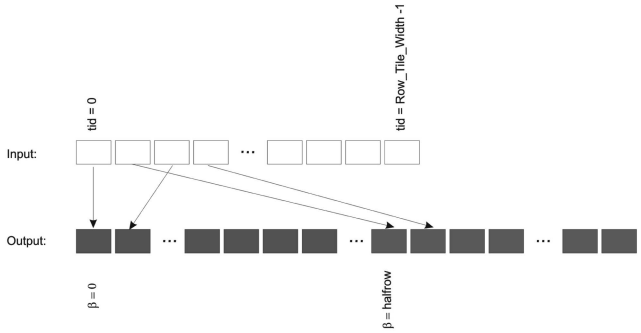


Fig. 4. Mapping to the output position.

processes $(Col\_Tile\_Width * Col\_Tile\_Height)$ number of threads, each producing one output pixel. The convolution filter stage is similar to the horizontal filter described above.

5.4.3 Algorithm Description: EBCOT Tier-1

EBCOT Tier-1 [28] consists of two major parts: context modeling and arithmetic encoder. The quantized transform coefficients are coded by the context modeling and the adaptive binary arithmetic coder to generate the compressed bitstream. The encoding method in the context modeling is bitplane coding. In this module, each wavelet coefficient is divided into one sign bitplane and several magnitude bitplanes. Each bitplane is then coded by three coding passes to generate a context-decision (CX-D) pair. The adaptive-context-based arithmetic encoder, which is also called the MQ-coder, utilizes the probability (CX) to compress the decision (D). In the MQ-coder, symbols in a code stream are classified as either most-probable symbol (MPS) or least-probable symbol (LPS). The basic operation of the MQ-coder is to divide the interval recursively based on the probability of input symbols.

5.4.4 Mapping on the GPU: EBCOT Tier-1

Fig. 6 shows the thread mapping for Tier-1. The approach followed in this work is to change as little as possible the original jasper code for parallelization. We utilize multithread CUDA architecture in the Tier-1 stage. Intrinsically, the context modeling and arithmetic coder are highly recursive and serialized. However, no synchronization is required for encoding code blocks due to the processing of independent code blocks.

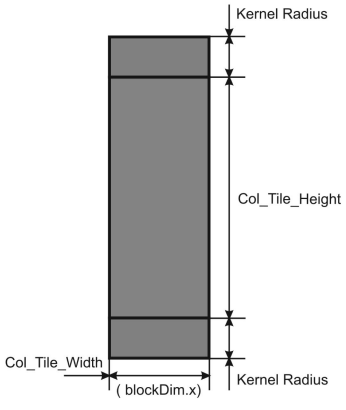


Fig. 5. Layout of the thread block grid for the vertical filtering.

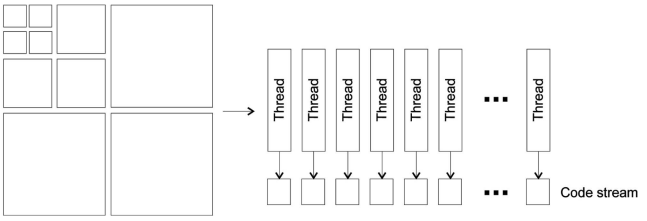


Fig. 6. Thread mapping for Tier-1.

We parallelized context modeling and arithmetic encoding. Each code block is assigned as input to a separate thread inside a CUDA kernel. The encoded bitstream obtained from each thread is then passed to Tier-2 for rate allocation.

5.5 Nonphotorealistic Rendering

As an active example of computational photography, nonphotorealistic rendering has received much attention. Stroke-based rendering for painterly rendering is the main interest in this paper. In this application, images are rerendered as an artistic depiction, like cartoon-style rendering, pen-ink drawing, and water/oily-style painting [36].

5.5.1 Algorithm Description

In this paper, cartoon-style and oily-style rendering are selected and implemented on the GPU. Cartoon-style rendering is a combination of image filtering algorithms such as bilateral filtering [37] and Canny edge detection. In this paper, the goal is not to design a new algorithm but to build common building blocks on the GPU. First, bilateral filtering is applied to several times the input image (10 times in our experiment) to reduce the color level while keeping the boundary unblurred. Next, the edge image obtained by Canny's algorithm is dilated and overlaid on the bilateral filtered image to achieve simple cartoon-style stylization of the input image.

Oily-style rendering is more complex. We employ Hertzmann's algorithm [38] as a base in which multiple brush strokes are sequentially applied from rough to fine (imitating the painters' actual technique). A set of strokes for each brush should be generated from the input image to simulate the brush effect. First, the pixelwise square root difference of the Gaussian blurred input (reference image) and the current canvas is computed, in which the standard deviation is proportional to the brush size. The stroke is generated if the sum of differences on the brush area is above the threshold. The position with local maximal difference is selected as the starting point of the brush. The stroke is extended along the gradient direction and saved if the length is sufficiently long. After strokes are generated, a round-shaped colored brush is swept along the stroke while updating the canvas. Starting from the initial canvas with uniform background, this procedure is iterated from the biggest brush size to the smallest, yielding oily-style painterly rendering.

5.5.2 Mapping on the GPU

Implementation of cartoon-style rendering is straightforward. Since all the computation is per-pixel mask convolution, which is independent from other processes,

we issue the same number of threads as the number of pixels and perform the convolution in a per-pixel/per-thread manner in the kernel function.

Conversely, direct implementation of the oily-style rendering on the GPU is impractical, since the brush touches of different strokes usually overlap. In the proposed parallel implementation, drawing is not done per-stroke but per-pixel, assuming that a brush stroke with a brighter color would be made later than those with darker colors. At each level of brush size, we also issue the same number of threads as the number of pixels. Each thread selects the brightest color from the overlapped brush touches, i.e., the stroke with the brightest color from the nearby strokes, where the distance from the thread's corresponding pixel and the stroke trajectory is within the brush size. The thread finally updates the corresponding pixel's color with the selected color. After repeating the process for all levels of brush size, the final oily-style rendered image is obtained. All input images are stored in the texture memory space to utilize fast access to read-only memory.

6 CUDA Benchmarks and Optimization

This section describes some of the CUDA benchmarking tools. The benchmarks facilitate the determination of the effectiveness of the parallel implementation, while debugging and optimizing the CUDA code. Five different basic principles are considered when benchmarking a CUDA implementation of an image processing algorithm:

1. global memory coalescing;
2. shared memory access to global memory access ratio;
3. global memory transfer;
4. GPU occupancy; and
5. data dependency.

For further details, refer to [20]:

$$\begin{aligned} GM\_coalesced = {} & \{gld\_coalesced + gst\_coalesced\}/ \\ & \{gld\_coalesced + gld\_uncoalesced \\ & + gst\_coalesced + gst\_uncoalesced\}. \end{aligned} \tag{5}$$

6.1 Global Memory Coalescing

Memory coalescing helps conserve bandwidth, while reducing effective latency. Abstractly, it is comparable to loading an entire cache line from memory versus loading one word at a time. Each thread of a half-warp during execution of a single read or write from the global memory is coalesced into a single contiguous and aligned memory access.

We use the CUDA Visual Profiler to evaluate global memory coalescing. We define global memory coalescing $GM\_coalesced$, as shown in (5), where $gld\_coalesced$, $gld\_uncoalesced$, $gst\_coalesced$, and $gst\_uncoalesced$ denote the amount of coalesced global memory load, the amount of uncoalesced global memory load, the amount of coalesced global memory store, and the amount of uncoalesced global memory store, respectively.

6.2 Shared Memory Access to Global Memory Access Ratio

The access to off-chip global memory does not provide a cache mechanism. Therefore, access to the global memory space has high memory latency (400-600 clock cycles). This makes reading from and writing to the global memory extremely expensive and causes a major bottleneck. Conversely, on-chip shared memory space is much faster than the local and global memory space. For threads in a single warp, shared memory access can be as fast as access to a register. In our experiments, we compute the ratio between shared memory and global memory access as a benchmark tool to determine how efficiently memory access is designed.

6.3 Global Memory Transfer

A PC's GPU card is usually connected via a PCI-Express bus. The image data transfer rate from CPU to GPU (HostToDevice) and GPU to CPU (DeviceToHost) is crucial to the performance of a CUDA implementation. The CUDA framework requires programmers to explicitly manage the data exchange between the host and the device. The memory transfer overhead can have a significant impact on the performance of the overall application. The transfer time increases linearly with the amount of data. The memory transfer overhead presents a significant bottleneck in processing algorithms involving large memory buffer and low floating-point computations.

6.4 GPU Occupancy

GPU occupancy is defined in terms of multiprocessor occupancy. The multiprocessor occupancy is the ratio between the active warps and the maximum number of active warps supported on a multiprocessor. We adopt the *CUDA Occupancy Calculator* to determine GPU occupancy. The occupancy is determined by the amount of shared memory and registers used by each thread block. Due to this, programmers need to choose the size of thread blocks carefully to maximize occupancy. GPU occupancy assists in choosing thread block size based on shared memory and register requirements.

Note that higher occupancy does not necessarily give better performance. Higher occupancy will help only if the kernel is bottlenecked by global memory access. Blindly choosing a large number of threads to increase occupancy can create pressure on registers and register spills into local memory can reduce the performance.

6.5 Data Dependency

Parallelism among threads in a thread block is serialized when some threads need to synchronize to share data between each other through memory access. This synchronization among threads is achieved using `__syncthreads()`. The total number of calls to `__syncthreads()` addresses the data dependency of a CUDA kernel.

7 Experimental Results and Discussion

Selected algorithms are implemented on the CPU and are parallelized subsequently using CUDA on the GPU. Our experimental platform is equipped with a quad-core Intel CPU (Q9450 with 42.56 GFLOPS) and an NVIDIA G92 (GeForce 9800 GTX) with 128 cores and 512 MB video memory. G92's peak performance is as high as 648 GFLOPS. We employ the latest GPU (GTX 280) to measure the speedup obtained with respect to GeForce 9800 GTX for the

TABLE 3
Application Performance Analysis in Terms of Preimplementation Metrics (FP: Floating Point)

Characteristics / Algorithms	Parallel Fraction (↑)	FP Computation to Memory Access Ratio (↑)	FP Computation Per Pixel (↑)	Memory Access Per Pixel (↑)	Branching diversity (↓)	Task dependency (↓)
Multiview Stereo Matching	0.994	1.838	4,900	2,665	0.117	1
Linear Feature Extraction	0.706	5.183	1,083	209	0.113	11
JPEG2000 Encoding (DWT)	0.983	3.630	552	152	0.138	12
JPEG2000 Encoding (Tier-1)	0.834	7.630	579	76	0.307	1
Cartoon-Style NPR	0.987	8.043	45,118	5,609	0.156	6
Oily-Style NPR	0.992	13.363	4,972	372	0.121	34

(↑) denotes bigger is better (and vice versa).

GPU scalability test. Note that GTX 280 has 240 cores with peak performance of 933 GFLOPS. CPU code is implemented using OpenMP to maximize CPUs performance so that four cores run in parallel.

7.1 Metrics Evaluation

We performed an application study using the metrics described in Section 4 to determine the effectiveness of the algorithm for parallel implementation. Table 3 lists the numerical results obtained for different algorithms. We wrote debugging codes to estimate the intensity of floating-point operations and frequency of memory access.

The parallel fraction (f) shows that a significant fraction of the original serial code is parallelized. The input/output (I/O) data read and write overhead associated with linear feature extraction and EBCOT Tier-1 encoding lowers the parallel fraction, limiting potential application speedup.

The floating-point computation to memory access ratio shows the extent of memory latency hiding. Significant ratios are obtained for algorithms other than multiview stereo matching. This reflects the algorithms' inherent nature for parallelism. Although multiview stereo matching suffers from a memory latency problem, the huge number of low-latency floating-point operations and efficient use of GPU memory bandwidth are major factors in its speedup. Linear feature extraction shows high floating-point computation intensity. However, this is mainly due to over-computation involved in thread allocation for nonedge pixels and during the line fitting procedure, where all the pixels in an edge link have to be traversed.

The low computational intensity per pixel associated with JPEG2000 encoding serves as a bottleneck and reflects its inability to effectively utilize resources on the GPU, limiting the potential acceleration. Even with low computational intensity, DWT can achieve a respectable performance increase due to the GPU's ability to run a large number of threads simultaneously. With efficient use of cache and shared memory, DWT can exploit the high GPU memory bandwidth to achieve respectable performance. Conversely, the high branching diversity associated with EBCOT Tier-1 encoding is the preliminary reason for its low acceleration. This limits the thread parallelism in a warp forcing the kernel to run threads in serial mode.

Cartoon-style NPR and oil-style NPR involve high computational intensity and frequent access to the memory buffer, while minimizing branching diversity. Both NPR algorithms show encouraging results for the metrics with no major bottlenecks.

The task dependency shows the ease of implementation. High task dependency often requires more modification and significant effort to port to the CUDA framework; the most extreme being the Oil-Style NPR, which involves a number of iterations with updates to the frame buffer in each of those iterations.

From the above analysis, we deduce that multiview stereo matching, cartoon-style NPR, and oil-style NPR with no major bottlenecks are expected to achieve significantly higher speedup for parallel implementation. High thread concurrency and efficient use of cache and shared memory in DWT are expected to reign over low computational intensity to achieve respectable performance. However, EBCOT Tier-1 encoding and linear feature extraction are likely to achieve the lowest performance, being bottlenecked by high branching diversity and low parallel fraction, respectively.

7.2 Evaluation of CUDA Benchmarks

In this section, we discuss some of the CUDA benchmarks that significantly affect the performance. Table 4 shows the results obtained for the CUDA benchmarking tools discussed in Section 6. We obtain the profiling results using CUDA Visual Profiler Version 1.0. Data dependency, source lines, and kernel lines are counted manually from the source code.

In general, memory coalescing is useful to maximize global memory bandwidth. However, most image processing algorithms involve highly random and frequent access to the memory buffer. This causes the memory access pattern to be uncoalesced. Each uncoalesced access is a separate DRAM request, limiting device's memory bandwidth. For algorithms other than JPEG2000 encoding, the problem is alleviated by high floating-point computation to memory access ratio. In JPEG2000 encoding, we ease the problem using efficient on-chip shared memory as reflected by the high SM to GM ratio in Table 4.

The multiview stereo matching algorithm involves a large-size memory buffer (47 images), increasing the global memory transfer overhead. For other applications, global memory transfer overhead is minimal. Applications other than EBCOT Tier-1 encoding achieve 33-83 percent active GPU occupancy. In case of Tier-1 encoding, the number of parallel threads depends on the number of code blocks. This is significantly less than the maximum concurrency in modern GPUs. Efficient tile-based thread clustering in DWT

TABLE 4
Result of CUDA Benchmarking (SM: Shared Memory, GM: Global Memory)

Characteristics / Algorithms	Global memory coalescing (↑)	SM to GM access ratio (↑)	Global memory Transfer (ms) (↓)	GPU occupancy (↑)	Data dependency (↓)	Source lines (Host)	Kernel lines (Device)
Multiview Stereo Matching	0.0013	0.020	107.28	33%	2	184	187
Linear Feature Extraction	0.0256	0.090	3.382	41.65%	36	620	318
JPEG2000 Encoding (DWT)	0.000	5.000	3.069	33.33% (vertical) 83.33% (horizontal)	2	132	112 (vertical) 90 (horizontal)
JPEG2000 Encoding (Tier-1)	0.0167	1.903	0.307	17.00%	2	>1500	406
Cartoon-Style NPR	0.1439	0.000	0.435	40.18%	0	223	232
Oily-Style NPR	0.1517	0.823	0.435	52.19%	4	514	635

significantly reduces the number of registers per thread, increasing the multiprocessor occupancy.

Data dependency refers to the number of per-block barrier synchronizations or, in other words, the number of times an application breaks to synchronize threads in its parallel execution. Linear feature extraction shows high data dependency, indicating high complexity in designing parallel implementation. Additionally, data dependency causes potential kernel halts when threads in a warp suffer from load imbalance, as in Tier-1 encoding.

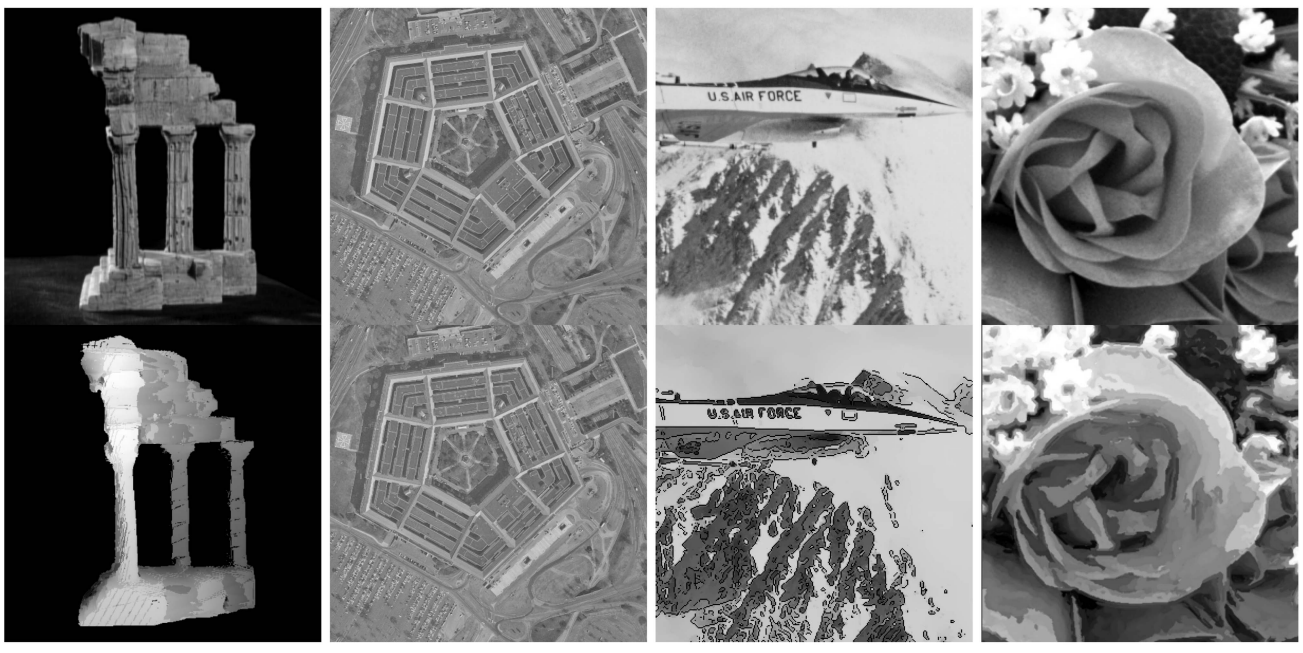
Source and kernel line count shows the complexity of parallel implementation. Larger source code often requires more modification to port to CUDA; the most extreme case was EBCOT Tier-1 encoding, which involved a large-scale code transformation to extract context-decision (CX-D) pair.

7.3 Results

In this section, we present the analysis of speedup for parallel GPU implementation of different algorithms. Fig. 7 shows the subjective evaluation of different parallel algorithms. The approach followed in our parallel implementation is to achieve an image quality identical or nearly identical to that of serial execution.

Table 5 shows the acceleration results of image data with varying resolutions for parallel implementation using the CUDA programming model. The images are shown in Fig. 8. The GPU execution time includes the data transfer (HostToDevice and DeviceToHost) and kernel(s) execution time. It can be inferred from the table that speedup does not vary significantly with the image resolution, since even for smallest image resolution, i.e., 512×512, the degree of concurrency is relatively high. Additionally, execution time depends on the image content when processing the image feature set. This varies for images with the same resolution.

The algorithms in Table 5 with the highest acceleration, namely, multiview stereo matching, cartoon-style NPR, and oil-style NPR, have a high parallel fraction and spend most of their execution time performing computation or accessing low-latency memory. The cartoon-style NPR and oil-style NPR achieve significantly higher speedups and require



(a) (b) (c) (d)

Fig. 7. Visual results of implemented algorithms. (a) Multiview stereo matching. (b) Linear feature extraction. (c) Cartoon-style NPR. (d) Oily-style NPR.

TABLE 5
Runtime Analysis (in Milliseconds) for the Test Images in Fig. 8

Algorithms	Data Resolution	Running Time			Speedup		GPU Scalability
		CPU (Q9450)	GPU (G92)	GPU (GX200)	G92/CPU	GX200/CPU	GX200/G92
Multiview Stereo Matching	TempleRing (47 Images)	18,422	340	110	54.18x	167.47x	3.09x
Linear Feature Extraction	512×512	109	54.77	45.06	1.99x	2.42x	1.22x
	1024×768	422	166.63	145.44	2.53x	2.90x	1.14x
	1280×1024	610	250.42	223.95	2.43x	2.72x	1.11x
	1200×1800	1,250	471.88	387.73	2.65x	3.22x	1.22x
	2288×1712	2,375	1018.98	789.16	2.33x	3.00x	1.29x
JPEG2000 Encoding (DWT)	512×512	31.85	7.84	4.59	4.06x	6.94x	1.71x
	1024×768	150.60	20.72	11.66	7.27x	12.92x	1.78x
	1280×1024	164.50	31.17	18.31	5.28x	8.98x	1.70x
	1200×1800	264.93	51.52	29.01	5.14x	9.13x	1.78x
	2288×1712	471.66	91.08	50.94	5.18x	9.26x	1.79x
	3024×2089	754.95	142.45	80.85	5.30x	9.34x	1.76x
JPEG2000 Encoding (Tier-1)	512×512	94	205	265	0.46x	0.35x	0.77x
	1024×768	234	390	333	0.60x	0.70x	1.17x
	1280×1024	328	484	437	0.68x	0.75x	1.11x
	1200×1800	891	735	796	1.21x	1.12x	0.92x
	2288×1712	1,640	1,468	1,016	1.12x	1.61x	1.44x
	3024×2089	1,500	2,062	1,531	0.73x	0.98x	1.35x
Cartoon-Style NPR	512×512	4,594	49.31	30.66	93.71x	149.84x	1.61x
	1024×768	14,594	149.66	86.63	97.51x	168.46x	1.73x
	1280×1024	18,688	243.35	142.29	76.79x	131.34x	1.71x
	1200×1800	47,688	406.53	236.94	117.30x	201.27x	1.72x
	2288×1712	93,891	741.42	428.67	126.64x	219.03x	1.73x
Oily-Style NPR	512×512	7,172	87.77	55.29	81.71x	129.72x	1.59x
	1024×768	15,609	226.22	138.72	69.00x	112.52x	1.63x
	1280×1024	35,313	334.83	221.86	105.47x	159.17x	1.51x
	1200×1800	49,047	589.78	377.29	83.16x	130.00x	1.56x
	2288×1712	94,406	1,107.12	679.11	85.27x	139.01x	1.63x

Q9450 is Intel Quad Core CPU. G92 and GX200 represent GeForce 9800 GTX and GTX 280, respectively.

additional explanation. One major reason for their performance is that both the algorithms involve sequences of filtering, which are primarily pixelwise operations, involving a massive SIMD-style instruction set. The GPU, with its fitting SIMD architecture, executes these instructions much faster than the CPU. We significantly improved the CPU version of our algorithms using OpenMP [6] multicore architecture. This is approximately $4.0\times$ faster than the original code running on a single core CPU.

As concluded from the metrics results in Section 7.1, DWT achieves respectable performance owing to its efficient cache and shared memory utilization and its high thread concurrency. Despite having low parallel fraction and computation overhead, linear feature extraction achieved an average speedup of $2.38\times$. This is mainly due to the high floating-point computation to memory access ratio, which hides potential halts caused by high-latency memory access.

Major bottlenecks appeared in EBCOT Tier-1 encoding, limiting its acceleration. Bottlenecks appeared in two ways. First, Tier-1 encoding is limited in the number of active threads, since parallelism is exploited at the code block level, compared to the pixel level in other algorithms. In addition, the high register usage per thread (50 in this case) also limits the number of active threads. Second, the high intensity of logical operations causes load imbalance among threads in a warp; this is the primary performance bottleneck. High thread divergence compels the threads in a warp to diverge and process sequentially.

(a) (b) (c) (d) (e) (f)

Fig. 8. Test images for Table 5. The resolution is (a) 512×512, (b) $1{,}024 \times 768$, (c) $1{,}280 \times 1{,}024$, (d) $1{,}200 \times 1{,}800$, (e) $2{,}288 \times 1{,}712$, and (f) $3{,}024 \times 2{,}089$.

It is worth noting that the speedup performance in Table 5 corresponds to the discussion in Section 7.1.

8 Conclusion

In this paper, we explored the design and implementation issues of image processing algorithms on GPUs with the CUDA framework. We selected four major domains 3D shape reconstruction, feature extraction, image compression, and computational photography and implemented multiview stereo matching, linear feature extraction, JPEG2000 image encoding, and nonphotorealistic rendering as example applications. The selected algorithms are parallelized efficiently on the GPU. A set of metrics was proposed to parameterize quantitatively the characteristics of parallel implementation of selected algorithms. In addition, these metrics can be used alternatively to compare the two implementations of the same algorithm on the GPU. Acceleration achieved for individual algorithms is evaluated in terms of the proposed metrics, while intensive analysis is conducted to show the appropriateness of the proposed metrics. These results can be shared and employed by other researchers to predict the appropriateness of their algorithm for parallel implementation.

Acknowledgments

This research was supported by the Ministry of Knowledge Economy (MKE), Korea, under the Information Technology Research Center (ITRC) support program supervised by the National IT Industry Promotion Agency (NIPA) (NIPA-2010-(C1090-1011-0003)). This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2009-0083945).

References

[1] *General Purpose GPU Programming (GPGPU) Website,* http://www.gpgpu.org, 2010.
[2] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A.E. Lefohn, and T.J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum,* vol. 26, no. 1, pp. 80-113, Mar. 2007.
[3] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU Computing," *Proc. IEEE,* vol. 96, no. 5, pp. 879-899, May 2008.
[4] NVIDIA Corporation, *Compute Unified Device Architecture (CUDA),* http://developer.nvidia.com/object/cuda.html, 2010.
[5] Khronos Group, *Open Computing Language (OpenCL),* http://www.khronos.org/opencl/, 2010.
[6] *OpenMP Website,* http://openmp.org/wp/, 2010.
[7] I.K. Park, N. Singhal, M.H. Lee, and S. Cho, "Efficient Design and Implementation of Visual Computing Algorithms on the GPU," *Proc. IEEE Int'l Conf. Image Processing,* pp. 2321-2324, Nov. 2009.
[8] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-Like Language," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 896-907, July 2003.
[9] M. Oneppo, "HLSL Shader Model 4.0," *Proc. ACM SIGGRAPH '07,* Aug. 2007.
[10] R.J. Rost, *OpenGL(R) Shading Language,* second ed. Addison-Wesley Professional, Jan. 2006.
[11] G. Shen, G.-P. Gao, S. Li, H. Shum, and Y. Zhang, "Accelerate Video Decoding with Generic GPU," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 15, no. 5, pp. 685-693, May 2005.
[12] R. Yang and M. Pollefeys, "A Versatile Stereo Implementation on Commodity Graphics Hardware," *Real-Time Imaging,* vol. 11, no. 1, pp. 7-18, Feb. 2005.
[13] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: An Opensource Gpu-Accelerated Framework for Image Processing and Computer Vision," *Proc. ACM Int'l Conf. Multimedia,* pp. 1089-1092, Oct. 2008.
[14] P. Babenko and M. Shah, "MinGPU: A Minimum GPU Library for Computer Vision," *Real-Time Image Processing,* vol. 3, no. 4, pp. 255-268, Dec. 2008.
[15] J. Fung, S. Mann, and C. Aimone, "OpenVIDIA: Parallel GPU Computer Vision," *Proc. ACM Int'l Conf. Multimedia,* pp. 849-852, Nov. 2005.
[16] *Proc. CVPR Workshop Visual Computer Vision on GPUs (CVGPU),* J.-M. Frahm, M. Pollefeys, and M. Shah, eds., June 2008.
[17] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 73-82, Feb. 2008.
[18] *CUJ2K: Jpeg2000 Encoder on Cuda,* http://sourceforge.net/projects/cuj2k/, 2010.
[19] M.D. Adams and F. Kossentini, "JasPer: A Software-Based JPEG-2000 Codec Implementation," *Proc. IEEE Int'l Conf. Image Processing,* pp. 53-56, Sept. 2000.
[20] NVIDIA Corporation, *NVIDIA CUDA Programming Guide 2.3,* 2009.
[21] T.R. Halffill, "Parallel Processing with CUDA," MicroProcessor Report Online, Jan. 2008.
[22] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing,* second ed. Pearson Education Limited, 2003.
[23] Y. Furukawa and J. Ponce, "Accurate, Dense, and Robust Multi-View Stereopsis," *Proc. IEEE Conf. Computer Vision and Pattern Recognition,* pp. 1-8, June 2007.
[24] R. Nevatia and K.R. Babu, "Linear Feature Extraction and Description," *Computer Graphics and Image Processing,* vol. 13, no. 3, pp. 257-269, July 1980.
[25] J.F. Canny, "A Computational Approach to Edge Detection," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. PAMI-8, no. 6, pp. 679-698, Nov. 1986.
[26] *Information Technology—JPEG2000 Image Coding System, ISO/IEC Int'l Standard 15444-1, ITU Recommendation T.800,* 2000.
[27] M. Rabbani and R. Joshi, "An Overview of the JPEG 2000 Still Image Compression Standard," *Signal Processing: Image Comm.,* vol. 17, no. 1, pp. 3-48, Jan. 2002.
[28] D. Taubman, "High Performance Scalable Image Compression with Ebcot," *IEEE Trans. Image Processing,* vol. 9, no. 7, pp. 1158-1170, July 2000.
[29] K. Andra, C. Chakrabarti, and T. Acharya, "A VLSI Architecture for Lifting Based Forward and Inverse Wavelet Transform," *IEEE Trans. Signal Processing,* vol. 50, no. 4, pp. 966-977, Apr. 2002.
[30] C.-T. Huang, P.-C. Tseng, and L.-G. Chen, "Hardware Implementation of Shape-Adaptive Discrete Wavelet Transform with the JPEG Defaulted (9,7) Filter Bank," *Proc. IEEE Int'l Conf. Image Processing,* pp. 571-574, Sept. 2003.
[31] Y.-Z. Zhang, C. Xu, W.-T. Wang, and L.-B. Chen, "Performance Analysis and Architecture Design for Parallel EBCOT Encoder for JPEG2000," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 17, no. 10, pp. 1336-1347, Oct. 2007.
[32] J.-S. Chaing, C.-H. Chang, C.-Y. Hsieh, and C.-H. Hsia, "High Efficiency EBCOT with Parallel Coding Architecture for JPEG2000," *EURASIP J. Applied Signal Processing,* vol. 2006, no. 1, pp. 1-14, Jan. 2006.
[33] S. Mallat, "The Theory for Multiresolution Signal Decomposition: The Wavelet Representation," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 11, no. 7, pp. 674-693, July 1989.
[34] W. Sweldens, "The Lifting Scheme: A Construction of Second Generation Wavelets," *SIAM J. Math. Analysis,* vol. 29, no. 2, pp. 511-546, Mar. 1998.
[35] G. Strang and T. Nguyen, *Wavelets and Filter Banks.* Cambridge Univ. Press, 1996.
[36] A. Hertzmann, "A Survey of Stroke-Based Rendering," *IEEE Computer Graphics and Applications,* vol. 23, no. 4, pp. 70-81, July/Aug. 2003.
[37] C. Tomasi and R. Manduchi, "Bilateral Filtering for Gray and Color Images," *Proc. IEEE Int'l Conf. Computer Vision,* pp. 839-846, Jan. 1998.
[38] A. Hertzmann, "Painterly Rendering with Curved Brush Strokes of Multiple Sizes," *Proc. ACM SIGGRAPH,* pp. 453-460, July 1998.

In Kyu Park received the BS, MS, and PhD degrees in electrical engineering and computer science from Seoul National University (SNU) in 1995, 1997, and 2001, respectively. From September 2001 to March 2004, he was a member of Technical Staff at Samsung Advanced Institute of Technology (SAIT). Since March 2004, he has been with the School of Information and Communication Engineering, Inha University, where he is an associate professor. From January 2007 to February 2008, he was an exchange scholar at Mitsubishi Electric Research Laboratories (MERL). His research interests include the joint area of computer graphics and vision, including 3D shape reconstruction from multiple views, image-based rendering, computational photography, and GPGPU for image processing and computer vision. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Nitin Singhal received the BS degree in electronics and communication engineering from Indian Institute of Technology (IIT), Guwahati, India, in 2006, and the MS degree in electrical engineering and computer science from Seoul National University (SNU), Korea, in 2008. He is presently working at Samsung Electronics Co., Ltd., Suwon, Korea, and is affiliated with the Telecommunication Module Laboratory in Digital Media and Communication R&D Center. He is a member of the IEEE. His research interests include computer vision, computational photography, GPU computing, and digital right management.

Man Hee Lee received the BS degree in computer engineering and the MS degree in information and communication engineering in 2006 and 2008, respectively, from Inha University, where he is currently working toward the PhD degree in information and communication engineering. From April 2007 to February 2008, he was a visiting graduate researcher at Electronics and Telecommunications Research Institute (ETRI). His research interests include image-based modeling and rendering, GPGPU, and 3D game technology. He is a student member of the IEEE and ACM.

Sungdae Cho received the BS degree in computer science from Soongsil University, Seoul, Korea, in 1996, and the MS and PhD degrees from Rensselaer Polytechnic Institute (RPI) in 2000 and 2002, respectively. After two years of postdoctoral research work with the Center for Image Processing Lab in RPI, in 2004, he joined Samsung Electronics Co., Ltd., Suwon, Korea, where he is currently a principal engineer with the Digital Media and Communications R&D Center. His research interests include multimedia signal processing, color processing, and graphics applications on mobile system.

Chris W. Kim received the BS degree in computer science from Carnegie Mellon University in 1994. Currently, he is working as a developer technology engineer at NVIDIA Korea. Before joining NVIDIA, he worked as a game development consultant for various game companies ranging from casual online games to MMORPGs using experiences he learned over 15 years in the industry. His main interest is applying GPU computing to computer graphics, artificial intelligence, social network analysis, and the latest game technologies.